# OpenShift Container Platform 4.7

## Operator

在 OpenShift Container Platform 中使用 Operator

Last Updated: 2023-03-24

# OpenShift Container Platform 4.7 Operator

在 OpenShift Container Platform 中使用 Operator

Enter your first name here. Enter your surname here.
Enter your organisation's name here. Enter your organisational division here.
Enter your email address here.

## 法律通告



## 摘要

本文档提供有关在 OpenShift Container Platform 中使用 Operator 的信息。文中为集群管理员提供了 Operator 的安装和管理说明，为开发人员提供了如何通过所安装的 Operator 创建应用程序的信息。另外还提供了一些使用 Operator SDK 构建自用 Operator 的指南。

# 目录

# 第 1 章 OPERATOR 概述

Operator 是 OpenShift Container Platform 中最重要的组件。Operator 是 control plane 上打包、部署和管理服务的首选方法。它们还可以为用户运行的应用程序提供优势。

Operator 与 Kubernetes API 和 CLI 工具（如 **kubectl** 和 **oc** 命令）集成。它们提供了监控应用程序、执行健康检查、管理无线(OTA)更新的方法，并确保应用程序保持在指定的状态。

虽然这两个操作都遵循类似的 Operator 概念和目标，但 OpenShift Container Platform 中的 Operator 由两个不同的系统管理，具体取决于其用途：

- 由 Cluster Version Operator(CVO)管理的 Cluster Operator 被默认安装来执行集群功能。
- 可选的附加组件 Operator 由 Operator Lifecycle Manager(OLM)管理，供用户在其应用程序中运行。

使用 Operator，您可以创建应用程序来监控集群中运行的服务。Operator 是专为您的应用程序而设计的。Operator 实施并自动执行常见的第 1 天操作，如安装和配置以及第 2 天操作，如自动缩放和缩减并创建备份。所有这些活动均位于集群中运行的一个软件中。

## 1.1. 对于开发人员

作为开发人员，您可以执行以下 Operator 任务：

- 安装 Operator SDK CLI。
- 创建基于 Go 的 Operator、基于 Ansible 的 Operator 和基于 Helm 的 Operator。
- 使用 Operator SDK 来构建、测试并部署 Operator。
- 安装 Operator 并订阅命名空间。
- 通过 Web 控制台 从已安装的 Operator 创建应用程序。

## 1.2. 对于管理员

作为集群管理员，您可以执行以下 Operator 任务：

- 管理自定义目录
- 允许非集群管理员安装 Operator
- 从 OperatorHub 安装 Operator
- 查看 Operator 状态。
- 管理 Operator 条件
- 升级安装的 Operator
- 删除安装的 Operator
- 配置代理支持
- 在受限网络中使用 Operator Lifecycle Manager

要了解红帽提供的集群 Operator 的信息，请参阅 Cluster Operators 参考。

## 1.3. 后续步骤

要了解更多有关 Operator 的信息，请参阅 Operator 是什么？

# 第 2 章 了解 OPERATOR

## 2.1. 什么是 OPERATOR?

从概念上讲，*Operator* 会收集人类操作知识，并将其编码成更容易分享给消费者的软件。

Operator 是一组软件，可用于降低运行其他软件的操作复杂程度。它可以被看作是软件厂商的工程团队的扩展，可以在 Kubernetes 环境中（如 OpenShift Container Platform）监控软件的运行情况，并根据软件的当前状态实时做出决策。Advanced Operator 被设计为用来无缝地处理升级过程，并对出现的错误自动进行响应，而且不会采取"捷径"（如跳过软件备份过程来节省时间）。

从技术上讲，Operator 是一种打包、部署和管理 Kubernetes 应用程序的方法。

Kubernetes 应用程序是一款 app，可在 Kubernetes 上部署，也可使用 Kubernetes API 和 **kubectl** 或 **oc** 工具进行管理。要想充分利用 Kubernetes，您需要一组统一的 API 进行扩展，以便服务和管理 Kubernetes 上运行的应用程序。可将 Operator 看成管理 Kubernetes 中这类应用程序的运行时。

### 2.1.1. 为什么要使用 Operator？

Operator 可以：

- 重复安装和升级。
- 持续对每个系统组件执行运行状况检查。
- 无线 (OTA) 更新 OpenShift 组件和 ISV 内容。
- 汇总现场工程师了解的情况并将其传输给所有用户，而非一两个用户。

**为什么在 Kubernetes 上部署？**

Kubernetes（通过扩展），OpenShift Container Platform 包含构建复杂分布式系统所需的所有原语 - secret 处理、负载均衡、服务发现、自动扩展 - 在内部和云供应商中工作。

**为什么使用 Kubernetes API 和 kubectl 工具来管理您的应用程序？**

这些 API 功能丰富，所有平台均有对应的客户端，并可插入到集群的访问控制/审核中。Operator 会使用 Kubernetes 的扩展机制"自定义资源定义 (CRD)"支持您的自定义对象，如 **MongoDB**，它类似于内置的原生 Kubernetes 对象。

**Operator 与 Service Broker 的比较？**

服务代理（service broker）是实现应用程序的编程发现和部署的一个步骤。但它并非一个长时间运行的进程，所以无法执行第 2 天操作，如升级、故障转移或扩展。它在安装时提供对可调参数的自定义和参数化，而 Operator 则可持续监控集群的当前状态。非集群服务仍非常适合于 Service Broker，但也存在合适于这些服务的 Operator。

### 2.1.2. Operator Framework

Operator Framework 是基于上述客户体验提供的一系列工具和功能。不仅仅是编写代码；测试、交付和更新 Operator 也同样重要。Operator Framework 组件包含用于解决这些问题的开源工具：

**Operator SDK**

Operator SDK 辅助 Operator 作者根据自身专业知识，引导、构建、测试和包装其 Operator，而无需了解 Kubernetes API 的复杂性。

**Operator Lifecycle Manager**

Operator Lifecycle Manager (OLM) 能够控制集群中 Operator 的安装、升级和基于角色的访问控制 (RBAC)。在 OpenShift Container Platform 4.7 中默认部署。

**Operator Registry**

Operator Registry 存储 ClusterServiceVersions (CSV) 和自定义资源定义 (CRD) 以便在集群中创建，并存储有关软件包和频道的 Operator 元数据。它运行在 Kubernetes 或 OpenShift 集群中，向 OLM 提供这些 Operator 目录数据。

**OperatorHub**

OperatorHub 是一个 web 控制台，供集群管理员用来发现并选择要在其集群上安装的 Operator。它在 OpenShift Container Platform 中默认部署。

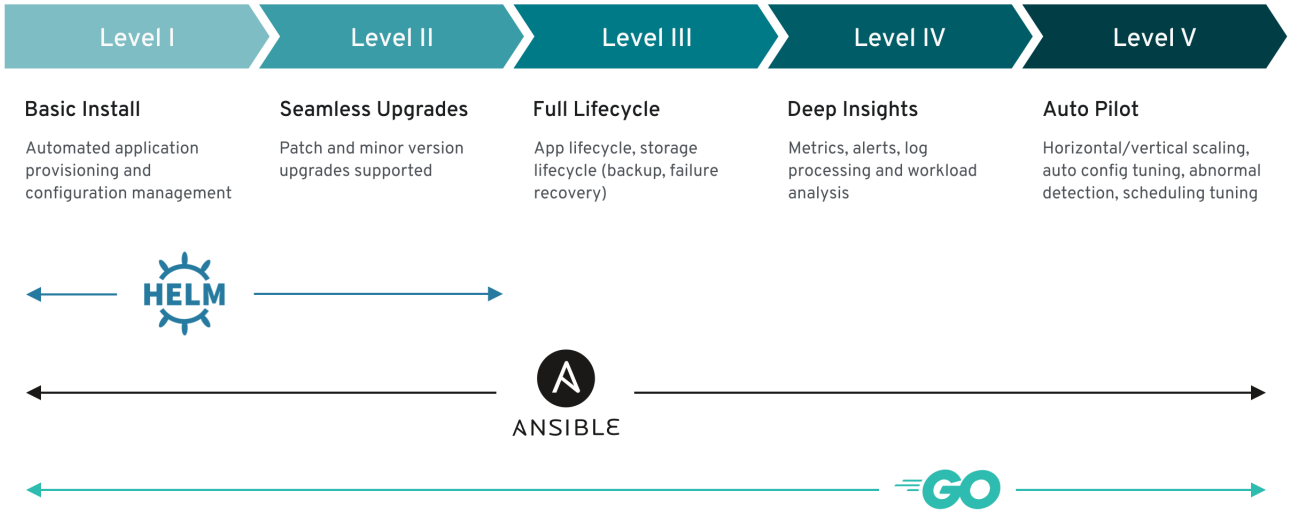
这些工具可组合使用，因此您可自由选择对您有用的工具。

### 2.1.3. Operator 成熟度模型

Operator 内部封装的管理逻辑的复杂程度各有不同。该逻辑通常还高度依赖于 Operator 所代表的服务类型。

对于大部分 Operator 可能包含的特定功能集来说，可以大致推断出 Operator 封装操作的成熟度等级。就此而言，以下 Operator 成熟度模型针对 Operator 的第 2 天通用操作定义了五个成熟度阶段。

**图 2.1. Operator 成熟度模型**




以上模型还显示了如何通过 Operator SDK 的 Helm、Go 和 Ansible 功能更好地开发这些功能。

## 2.2. OPERATOR FRAMEWORK 打包格式

本指南概述了 OpenShift Container Platform 中 Operator Lifecycle Manager (OLM) 所支持的 Operator 打包格式。

### 2.2.1. Bundle Format

Operator 的 *Bundle Format* 是 Operator Framework 引入的新打包格式。为提高可伸缩性并为自行托管目录的上游用户提供更好地支持，Bundle Format 规格简化了 Operator 元数据的发布。

Operator 捆绑包代表 Operator 的单一版本。磁盘上的*捆绑包清单*是容器化的，并作为*捆绑包镜像*提供，该镜像是一个不可运行的容器镜像，其中存储了 Kubernetes 清单和 Operator 元数据。然后，使用现有容器工具（如 **podman** 和 **docker**）和容器 registry（如 Quay）来管理捆绑包镜像的存储和发布。

Operator 元数据可以包括：

- 标识 Operator 的信息，如名称和版本。
- 驱动 UI 的额外信息，例如其图标和一些示例自定义资源 (CR)。
- 所需的和所提供的 API。
- 相关镜像。

将清单加载到 Operator Registry 数据库中时，会验证以下要求：

- 该捆绑包必须在注解中至少定义一个频道。
- 每个捆绑包都只有一个集群服务版本（CSV）。
- 如果 CSV 拥有自定义资源定义（CRD），则该 CRD 必须存在于捆绑包中。

#### 2.2.1.1. 清单

捆绑包清单指的是一组 Kubernetes 清单，用于定义 Operator 的部署和 RBAC 模型。

捆绑包包括每个目录的一个 CSV，一般情况下，用来定义 CRD 所拥有的 API 的 CRD 位于 **/manifest** 目录中。

**Bundle Format 布局示例**

```
etcd
├── manifests
│   ├── etcdcluster.crd.yaml
│   └── etcdoperator.clusterserviceversion.yaml
│   └── secret.yaml
│   └── configmap.yaml
└── metadata
    └── annotations.yaml
    └── dependencies.yaml
```

**额外支持的对象**
以下对象类型也可以包括在捆绑包的 **/manifests** 目录中：

**支持的可选对象类型**

- **ClusterRole**
- **ClusterRoleBinding**
- **ConfigMap**
- **ConsoleYamlSample**
- **PodDisruptionBudget**

- **PriorityClass**
- **PrometheusRule**
- **角色**
- **RoleBinding**
- **Secret**
- **服务**
- **ServiceAccount**
- **ServiceMonitor**
- **VerticalPodAutoscaler**

当捆绑包中包含这些可选对象时，Operator Lifecycle Manager（OLM）可以从捆绑包创建对象，并随 CSV 一起管理其生命周期：

**可选对象的生命周期**

- 删除 CSV 后，OLM 会删除可选对象。
- 当 CSV 被升级时：
  - 如果可选对象的名称相同，OLM 会更新它。
  - 如果可选对象的名称在版本间有所变化，OLM 会删除并重新创建它。

#### 2.2.1.2. 注解

捆绑包还在其 **/metadata** 文件夹中包含 **annotations.yaml** 文件。此文件定义了更高级别的聚合数据，以帮助描述有关如何将捆绑包添加到捆绑包索引中的格式和软件包信息：

**annotations.yaml 示例**

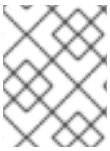
```
annotations:
  operators.operatorframework.io.bundle.mediatype.v1: "registry+v1" 1
  operators.operatorframework.io.bundle.manifests.v1: "manifests/" 2
  operators.operatorframework.io.bundle.metadata.v1: "metadata/" 3
  operators.operatorframework.io.bundle.package.v1: "test-operator" 4
  operators.operatorframework.io.bundle.channels.v1: "beta,stable" 5
  operators.operatorframework.io.bundle.channel.default.v1: "stable" 6
```

1. Operator 捆绑包的介质类型或格式。**registry+v1** 格式表示它包含 CSV 及其关联的 Kubernetes 对象。
2. 镜像中的该路径指向含有 Operator 清单的目录。该标签保留给以后使用，当前默认为 **manifests/**。**manifests.v1** 值表示捆绑包包含 Operator 清单。
3. 镜像中的该路径指向包含捆绑包元数据文件的目录。该标签保留给以后使用，当前默认为 **metadata/**。**metadata.v1** 值表示这个捆绑包包含 Operator 元数据。

4 捆绑包的软件包名称。

5 捆绑包添加到 Operator Registry 时订阅的频道列表。

6 从 registry 安装时，Operator 应该订阅到的默认频道。



**注意**

如果出现不匹配的情况，则以 **annotations.yaml** 文件为准，因为依赖这些注解的集群 Operator Registry 只能访问此文件。

#### 2.2.1.3. 依赖项文件

Operator 的依赖项列在捆绑包的 **metadata/** 目录中的 **dependencies.yaml** 文件中。此文件是可选的，目前仅用于指明 Operator-version 依赖项。

依赖项列表中，每个项目包含一个 **type** 字段，用于指定这一依赖项的类型。受支持的 Operator 依赖项有两种：

- **olm.package**：此类型表示特定 Operator 版本的依赖项。依赖项信息必须包含软件包名称以及软件包的版本，格式为 semver。例如，您可以指定具体版本，如 **0.5.2**，也可指定一系列版本，如 **>0.5.1**。
- **olm.gvk**：使用 **gvk** 类型，作者可以使用 group/version/kind（GVK）信息指定依赖项，类似于 CSV 中现有 CRD 和基于 API 的使用。该路径使 Operator 作者可以合并所有依赖项、API 或显式版本，使它们处于同一位置。

在以下示例中，为 Prometheus Operator 和 etcd CRD 指定依赖项：

**dependencies.yaml 文件示例**

```
dependencies:
  - type: olm.package
    value:
      packageName: prometheus
      version: ">0.27.0"
  - type: olm.gvk
    value:
      group: etcd.database.coreos.com
      kind: EtcdCluster
      version: v1beta2
```

**其他资源**

- Operator Lifecycle Manager 依赖项解析

#### 2.2.1.4. 关于 opm

**opm** CLI 工具由 Operator Framework 提供，用于 Operator Bundle Format。您可以通过一个名为 *index* 的捆绑包列表创建和维护 Operator 目录，类似于软件存储库。其结果是一个名为 *index image（索引镜像）* 的容器镜像，它可以存储在容器的 registry 中，然后安装到集群中。

索引包含一个指向 Operator 清单内容的指针数据库，可通过在运行容器镜像时提供的已包含 API 进行查询。在 OpenShift Container Platform 中，Operator Lifecycle Manager（OLM）可通过在

**CatalogSource** 中引用索引镜像来使它作为目录一个（catalog），它会定期轮询镜像，以对集群上安装的 Operator 进行更新。

- 有关安装 **opm** CLI 的步骤，请参阅 CLI 工具。

### 2.2.2. Package Manifest Format

Operator 的 *Package Manifest Format* 是 Operator Framework 中原先引入的旧版打包格式。虽然 OpenShift Container Platform 4.5 中已弃用此格式，但它仍受支持，红帽提供的 Operator 目前仍使用此方法提供。

在这个格式中，Operator 的版本由单个集群服务版本（CSV）代表，自定义资源定义（CRD）用来定义 CSV 所拥有的 API，另外也可能包含其他对象。

Operator 的所有版本都嵌套到单个目录中：

**Package Manifest Format 布局示例**

```
etcd
├── 0.6.1
│   ├── etcdcluster.crd.yaml
│   └── etcdoperator.clusterserviceversion.yaml
├── 0.9.0
│   ├── etcdbackup.crd.yaml
│   ├── etcdcluster.crd.yaml
│   ├── etcdoperator.v0.9.0.clusterserviceversion.yaml
│   └── etcdrestore.crd.yaml
├── 0.9.2
│   ├── etcdbackup.crd.yaml
│   ├── etcdcluster.crd.yaml
│   ├── etcdoperator.v0.9.2.clusterserviceversion.yaml
│   └── etcdrestore.crd.yaml
└── etcd.package.yaml
```

它还包含一个 **<name>.package.yaml** 文件，它是定义软件包名称和频道详情的*软件包清单*：

**软件包清单示例**

```
packageName: etcd
channels:
- name: alpha
  currentCSV: etcdoperator.v0.9.2
- name: beta
  currentCSV: etcdoperator.v0.9.0
- name: stable
  currentCSV: etcdoperator.v0.9.2
defaultChannel: alpha
```

将软件包清单加载到 Operator Registry 数据库中时，会验证以下要求：

- 每个软件包至少有一个频道。
- 软件包中频道指向的每个 CSV 都存在。
- Operator 的每个版本都只有一个 CSV。

- 如果 CSV 拥有 CRD，则该 CRD 必须存在于 Operator 版本的目录中。
- 如果 CSV 替换了另一个 CSV，则软件包中必须同时存在旧的和新的 CSV。

## 2.3. OPERATOR FRAMEWORK 常用术语表

本主题提供了与 Operator Framework 相关的常用术语表，其中包括 Operator Lifecycle Manager (OLM) 和 Operator SDK，适用于两种打包格式： Package Manifest Format 和 Bundle Format。

### 2.3.1. 常见 Operator Framework 术语

#### 2.3.1.1. 捆绑包（Bundle）

在 Bundle Format 中，*捆绑包*是 Operator CSV、清单和元数据的集合。它们一起构成了可在集群中安装的 Operator 的唯一版本。

#### 2.3.1.2. 捆绑包镜像

在 Bundle Format 中， *捆绑包镜像*是一个从 Operator 清单中构建的容器镜像，其中包含一个捆绑包。捆绑包镜像由 Open Container Initiative (OCI) spec 容器 registry 存储和发布，如 Quay.io 或 DockerHub。

#### 2.3.1.3. 目录源

*目录源（catalog source）* 是定义应用程序的 CSV、CRD 和软件包的仓库。

#### 2.3.1.4. 目录镜像

在 Package Manifest Format 中， *目录镜像*是一个容器化数据存储，用于描述一组可使用 OLM 在集群中安装的 Operator 元数据和更新元数据。

#### 2.3.1.5. 频道

*频道*为 Operator 定义更新流，用于为订阅者推出更新。频道头指向该频道的最新版本。例如， **stable** 频道中会包含 Operator 的所有稳定版本，按由旧到新的顺序排列。

Operator 可以有几个频道，与特定频道绑定的订阅只会在该频道中查找更新。

#### 2.3.1.6. 频道头

*频道头*是指特定频道中最新已知的更新。

#### 2.3.1.7. 集群服务版本

*集群服务版本（cluster service version，简称 CSV* 是一个利用 Operator 元数据创建的 YAML 清单，可辅助 OLM 在集群中运行 Operator。它是 Operator 容器镜像附带的元数据，用于在用户界面填充徽标、描述和版本等信息。

此外，CSV 还是运行 Operator 所需的技术信息来源，类似于其需要的 RBAC 规则及其管理或依赖的自定义资源 (CR)。

#### 2.3.1.8. 依赖项

Operator 可能会依赖于*集群*中存在的另一个 Operator。例如，Vault Operator 依赖于 etcd Operator 的数据持久性层。

OLM 通过确保在安装过程中在集群中安装 Operator 和 CRD 的所有指定版本来解决依赖关系。通过在目录中查找并安装满足所需 CRD API 且与软件包或捆绑包不相关的 Operator，解决这个依赖关系。

#### 2.3.1.9. 索引镜像

在 Bundle Format 中，*索引镜像*是一种数据库（数据库快照）镜像，其中包含关于 Operator 捆绑包（包括所有版本的 CSV 和 CRD）的信息。此索引可以托管集群中 Operator 的历史记录，并可使用 **opm** CLI 工具添加或删除 Operator 来加以维护。

#### 2.3.1.10. 安装计划

*安装计划（install plan）*是一个列出了为自动安装或升级 CSV 而需创建的资源的计算列表。

#### 2.3.1.11. operator 组

*Operator 组*将部署在同一命名空间中的所有 Operator 配置为 **OperatorGroup** 对象，以便在一系列命名空间或集群范围内监视其 CR。

#### 2.3.1.12. 软件包

在 Bundle Format 中，*软件包*是一个目录，其中包含每个版本的 Operator 的发布历史记录。CSV 清单中描述了发布的 Operator 版本和 CRD。

#### 2.3.1.13. 容器镜像仓库（Registry）

Registry *是*一个存储了 Operator 捆绑包镜像的数据库，每个都包含所有频道的最新和历史版本。

#### 2.3.1.14. Subscription

*订阅（subscription）* 通过跟踪软件包中的频道来保持 CSV 最新。

#### 2.3.1.15. 更新图表

*更新图表*将 CSV 的版本关联到一起，与其他打包软件的更新图表类似。可以依次安装 Operator，也可以跳过某些版本。只有在添加新版本时，更新图表才会在频道头上扩大。

## 2.4. OPERATOR LIFECYCLE MANAGER (OLM)

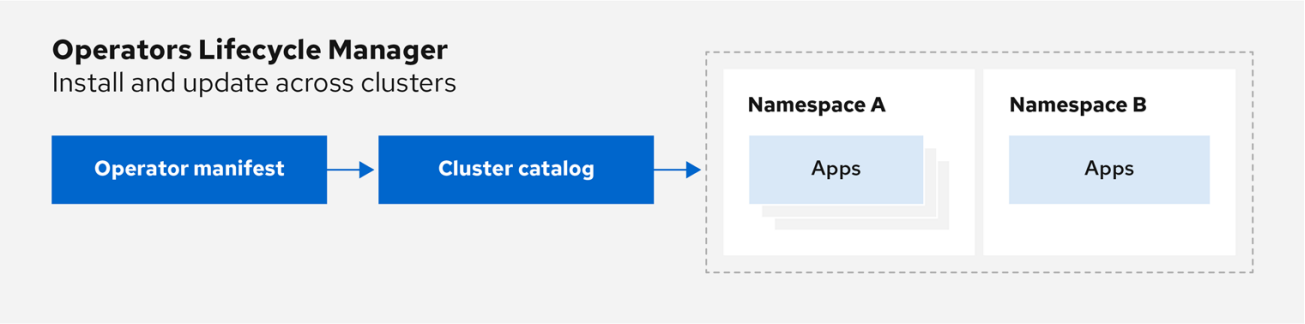
### 2.4.1. Operator Lifecycle Manager 概念和资源

本指南概述了 OpenShift Container Platform 中 Operator Lifecycle Manager (OLM) 背后的概念。

#### 2.4.1.1. Operator Lifecycle Manager 是什么？

*Operator Lifecycle Manager* （OLM）可帮助用户安装、更新和管理所有 Kubernetes 原生应用程序（Operator）以及在 OpenShift Container Platform 集群中运行的关联服务的生命周期。它是 Operator Framework 的一部分，后者是一个开源工具包，用于以有效、自动化且可扩展的方式管理 Operator。

图 2.2. Operator Lifecycle Manager 工作流




OLM 默认在 OpenShift Container Platform 4.7 中运行，辅助集群管理员对集群上运行的 Operator 进行安装、升级和授予访问权。OpenShift Container Platform Web 控制台提供一些管理界面，供集群管理员安装 Operator，以及为特定项目授权以便使用集群上的可用 Operator 目录。

开发人员通过自助服务体验，无需成为相关问题的专家也可自由置备和配置数据库、监控和大数据服务的实例，因为 Operator 已将相关知识融入其中。

#### 2.4.1.2. OLM 资源

以下自定义资源定义 (CRD) 由 Operator Lifecycle Manager (OLM) 定义和管理：

**表 2.1. 由 OLM 和 Catalog Operator 管理的 CRD**

| 资源 | 短名称 | 描述 |
|---|---|---|
| **ClusterServiceVersion** (CSV) | **csv** | 应用程序元数据。例如：名称、版本、图标、所需资源。 |
| **CatalogSource** | **catsrc** | 定义应用程序的 CSV、CRD 和软件包存储库。 |
| **Subscription** | **sub** | 通过跟踪软件包中的频道来保持 CSV 最新 |
| **InstallPlan** | **ip** | 为自动安装或升级 CSV 而需创建的资源的计算列表。 |
| **OperatorGroup** | **og** | 将部署在同一命名空间中的所有 Operator 配置为 **OperatorGroup** 对象，以便在一系列命名空间或集群范围内监视其自定义资源 ( CR)。 |
| **OperatorConditions** | - | 在 OLM 和它管理的 Operator 之间创建通信频道。操作员可以写入 **Status.Conditions** 数组，以向 OLM 通报复杂状态。 |

##### 2.4.1.2.1. 集群服务版本

*集群服务版本* (CSV) 代表 OpenShift Container Platform 集群中运行的 Operator 的特定版本。它是一个利用 Operator 元数据创建的 YAML 清单，可辅助 Operator Lifecycle Manager (OLM) 在集群中运行 Operator。

OLM 需要与 Operator 相关的元数据，以确保它可以在集群中安全运行，并在发布新版 Operator 时提供有关如何应用更新的信息。这和传统的操作系统的打包软件相似；可将 OLM 的打包步骤认为是制作 **rpm**、**deb** 或 **apk** 捆绑包的阶段。

CSV 中包含 Operator 容器镜像附带的元数据，用于在用户界面填充名称、版本、描述、标签、存储库链接和徽标等信息。

此外，CSV 还是运行 Operator 所需的技术信息来源，例如其管理或依赖的自定义资源 (CR)、RBAC 规则、集群要求和安装策略。此信息告诉 OLM 如何创建所需资源并将 Operator 设置为部署。

#### 2.4.1.2.2. 目录源

*catalog source* 代表元数据存储，通常通过引用存储在容器 registry 中的 *index image*。Operator Lifecycle Manager (OLM) 查询目录源来发现和安装 Operator 及其依赖项。OpenShift Container Platform Web 控制台中的 OperatorHub 也会显示由目录源提供的 Operator。

### 提示

集群管理员可以使用 web 控制台中的 **Administration → Cluster Settings → Global Configuration → OperatorHub** 页面查看集群中已启用的目录源提供的 Operator 的完整列表。

**CatalogSource** 的 **spec** 指明了如何构造 pod，以及如何与服务于 Operator Registry gRPC API 的服务进行通信。

**例 2.1. CatalogSource 对象示例**

```
apiVersion: operators.coreos.com/v1alpha1
kind: CatalogSource
metadata:
  generation: 1
  name: example-catalog (1)
  namespace: openshift-marketplace (2)
spec:
  displayName: Example Catalog (3)
  image: quay.io/example-org/example-catalog:v1 (4)
  priority: -400 (5)
  publisher: Example Org
  sourceType: grpc (6)
  updateStrategy:
    registryPoll: (7)
      interval: 30m0s
status:
  connectionState:
    address: example-catalog.openshift-marketplace.svc:50051
    lastConnect: 2021-08-26T18:14:31Z
    lastObservedState: READY (8)
  latestImageRegistryPoll: 2021-08-26T18:46:25Z (9)
  registryService: (10)
    createdAt: 2021-08-26T16:16:37Z
    port: 50051
    protocol: grpc
    serviceName: example-catalog
    serviceNamespace: openshift-marketplace
```

1. **CatalogSource** 对象的名称。此值也用作在请求的命名空间中创建相关 pod 的名称的一部分。
2. 创建可用目录的命名空间。要使目录在所有命名空间中都可用，请将此值设置为 **openshift-marketplace**。默认红帽提供的目录源也使用 **openshift-marketplace** 命名空间。否则，将值设置为特定命名空间，使 Operator 仅在该命名空间中可用。
3. 在 Web 控制台和 CLI 中显示目录的名称。
4. 目录的索引镜像。
5. 目录源的权重。OLM 在依赖项解析过程中使用权重进行优先级排序。权重越高，表示目录优先于轻量级目录。
6. 源类型包括以下内容：
   - 带有**镜像**引用的 **grpc**：OLM 拉取镜像并运行 pod，为兼容的 API 服务。
   - 带有**地址**字段的 **grpc**：OLM 会尝试联系给定地址的 gRPC API。在大多数情况下不应该使用这种类型。
   - **configmap**：OLM 解析配置映射数据，并运行一个可以为其提供 gRPC API 的 pod。
7. 在指定的时间段内自动检查新版本以保持最新。
8. 目录连接的最后观察到状态。例如：
   - **READY** ：成功建立连接。
   - **CONNECTING** ：连接正在尝试建立。
   - **TRANSIENT_FAILURE** ：尝试建立连接（如超时）时发生了临时问题。该状态最终将切回到 **CONNECTING**，然后重试。

   如需了解更多详细信息，请参阅 gRPC 文档中的连接状态。
9. 存储目录镜像的容器注册表最近轮询的时间，以确保镜像为最新版本。
10. 目录的 Operator Registry 服务的状态信息。

在订阅中引用 **CatalogSource** 对象的**名称**会指示 OLM 搜索查找请求的 Operator 的位置：

**例 2.2. 引用目录源的 Subscription 对象示例**

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: example-operator
  namespace: example-namespace
spec:
  channel: stable
  name: example-operator
  source: example-catalog
  sourceNamespace: openshift-marketplace
```

**其它资源**

- 了解 OperatorHub
- 红帽提供的 Operator 目录
- 目录优先级
- 使用 CLI 查看 Operator 目录源状态

#### 2.4.1.2.3. 订阅

*订阅*（由一个 **Subscription** 对象定义）代表安装 Operator 的意图。它是将 Operator 与目录源关联的自定义资源。

Subscription 描述了要订阅 Operator 软件包的哪个频道，以及是自动还是手动执行更新。如果设置为自动，订阅可确保 Operator Lifecycle Manager（OLM）自动管理并升级 Operator，以确保集群中始终运行最新版本。

**Subscription 对象示例**

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: example-operator
  namespace: example-namespace
spec:
  channel: stable
  name: example-operator
  source: example-catalog
  sourceNamespace: openshift-marketplace
```

此 **Subscription** 对象定义了 Operator 的名称和命名空间，以及从中查找 Operator 数据的目录。频道（如 **alpha**、**beta** 或 **stable**）可帮助确定应从目录源安装哪些 Operator 流。

订阅中的频道名称可能会因 Operator 而异，但应遵守给定 Operator 中的常规约定。例如，频道名称可能会遵循 Operator 提供的应用程序的次发行版本更新流（**1.2**、**1.3**）或发行的频率（**stable**、**fast**）。

除了可从 OpenShift Container Platform Web 控制台轻松查看外，还可以通过检查相关订阅的状态来识别是否有较新版本的 Operator 可用。与 **currentCSV** 字段关联的值是 OLM 已知的最新版本，而 **installedCSV** 是集群中安装的版本。

**其它资源**

- 使用 CLI 查看 Operator 订阅状态

#### 2.4.1.2.4. 安装计划

*安装计划*（由一个 **InstallPlan** 对象定义） 描述了 Operator Lifecycle Manager (OLM) 为安装或升级到 Operator 的特定版本而创建的一组资源。该版本由集群服务版本 (CSV) 定义。

要安装 Operator、集群管理员或被授予 Operator 安装权限的用户，必须首先创建一个 **Subscription** 对象。订阅代表了从目录源订阅 Operator 可用版本流的意图。然后，订阅会创建一个 **InstallPlan** 对象来方便为 Operator 安装资源。

然后，根据以下批准策略之一批准安装计划：

- 如果订阅的 **spec.installPlanApproval** 字段被设置为 **Automatic**，则会自动批准安装计划。
- 如果订阅的 **spec.installPlanApproval** 字段被设置为 **Manual**，则安装计划必须由集群管理员或具有适当权限的用户手动批准。

批准安装计划后，OLM 会创建指定的资源，并在订阅指定的命名空间中安装 Operator。

**例 2.3. InstallPlan 对象示例**

```
apiVersion: operators.coreos.com/v1alpha1
kind: InstallPlan
metadata:
  name: install-abcde
  namespace: operators
spec:
  approval: Automatic
  approved: true
  clusterServiceVersionNames:
    - my-operator.v1.0.1
  generation: 1
status:
  ...
  catalogSources: []
  conditions:
    - lastTransitionTime: '2021-01-01T20:17:27Z'
      lastUpdateTime: '2021-01-01T20:17:27Z'
      status: 'True'
      type: Installed
  phase: Complete
  plan:
    - resolving: my-operator.v1.0.1
      resource:
        group: operators.coreos.com
        kind: ClusterServiceVersion
        manifest: >-
        ...
        name: my-operator.v1.0.1
        sourceName: redhat-operators
        sourceNamespace: openshift-marketplace
        version: v1alpha1
      status: Created
    - resolving: my-operator.v1.0.1
      resource:
        group: apiextensions.k8s.io
        kind: CustomResourceDefinition
        manifest: >-
        ...
        name: webservers.web.servers.org
        sourceName: redhat-operators
        sourceNamespace: openshift-marketplace
        version: v1beta1
      status: Created
    - resolving: my-operator.v1.0.1
      resource:
        group: ''
        kind: ServiceAccount
```

```
      manifest: >-
      ...
      name: my-operator
      sourceName: redhat-operators
      sourceNamespace: openshift-marketplace
      version: v1
    status: Created
  - resolving: my-operator.v1.0.1
    resource:
      group: rbac.authorization.k8s.io
      kind: Role
      manifest: >-
      ...
      name: my-operator.v1.0.1-my-operator-6d7cbc6f57
      sourceName: redhat-operators
      sourceNamespace: openshift-marketplace
      version: v1
    status: Created
  - resolving: my-operator.v1.0.1
    resource:
      group: rbac.authorization.k8s.io
      kind: RoleBinding
      manifest: >-
      ...
      name: my-operator.v1.0.1-my-operator-6d7cbc6f57
      sourceName: redhat-operators
      sourceNamespace: openshift-marketplace
      version: v1
    status: Created
    ...
```

**其它资源**

- 允许非集群管理员安装 Operator

#### 2.4.1.2.5. operator 组

由 **OperatorGroup** 资源定义的 *Operator 组*，为 OLM 安装的 Operator 提供多租户配置。Operator 组选择目标命名空间，在其中为其成员 Operator 生成所需的 RBAC 访问权限。

这一组目标命名空间通过存储在 CSV 的 **olm.targetNamespaces** 注解中的以逗号分隔的字符串来提供。该注解应用于成员 Operator 的 CSV 实例，并注入它们的部署中。

**其它资源**

- Operator 组。

#### 2.4.1.2.6. Operator 条件

作为管理 Operator 生命周期的角色的一部分，Operator Lifecycle Manager（OLM）从定义 Operator 的 Kubernetes 资源状态中推断 Operator 状态。虽然此方法提供了一定程度的保证来确定 Operator 处于给定状态，但在有些情况下，Operator 可能需要直接向 OLM 提供信息，而这些信息不能被推断出来。这些信息可以被 OLM 使用来更好地管理 Operator 的生命周期。

OLM 提供了一个名为 **OperatorCondition** 的自定义资源定义（CRD），它允许 Operator 与 OLM 相互通信条件信息。当在一个 **OperatorCondition** 资源的 **Status.Conditions** 数组中存在时，则代表存在一组会影响 OLM 管理 Operator 的支持条件。

**其它资源**

- Operator 条件。

### 2.4.2. Operator Lifecycle Manager 架构

本指南概述了 OpenShift Container Platform 中 Operator Lifecycle Manager (OLM) 的组件架构。

#### 2.4.2.1. 组件职责

Operator Lifecycle Manager (OLM) 由两个 Operator 组成，分别为：OLM Operator 和 Catalog Operator。

每个 Operator 均负责管理 CRD，而 CRD 是 OLM 的框架基础：

**表 2.2. 由 OLM 和 Catalog Operator 管理的 CRD**

| 资源 | 短名称 | 所有者 | 描述 |
|---|---|---|---|
| **ClusterServiceVersion** (CSV) | **csv** | OLM | 应用程序元数据：名称、版本、图标、所需资源、安装等。 |
| **InstallPlan** | **ip** | Catalog | 为自动安装或升级 CSV 而需创建的资源的计算列表。 |
| **CatalogSource** | **catsrc** | Catalog | 定义应用程序的 CSV、CRD 和软件包存储库。 |
| **Subscription** | **sub** | Catalog | 用于通过跟踪软件包中的频道来保持 CSV 最新。 |
| **OperatorGroup** | **og** | OLM | 将部署在同一命名空间中的所有 Operator 配置为 **OperatorGroup** 对象，以便在一系列命名空间或集群范围内监视其自定义资源 ( CR)。 |

每个 Operator 还负责创建以下资源：

**表 2.3. 由 OLM 和 Catalog Operator 创建的资源**

| 资源 | 所有者 |
|---|---|
| **部署** | OLM |
| **ServiceAccounts** | |
| **(Cluster)Roles** | |

| 资源 | 所有者 |
| --- | --- |
| **(Cluster)RoleBindings** | |
| **CustomResourceDefinitions** (CRD) | Catalog |
| **ClusterServiceVersions** | |

#### 2.4.2.2. OLM Operator

集群中存在 CSV 中指定需要的资源后，OLM Operator 将负责部署由 CSV 资源定义的应用程序。

OLM Operator 不负责创建所需资源；用户可选择使用 CLI 手动创建这些资源，也可选择使用 Catalog Operator 来创建这些资源。这种关注点分离的机制可以使得用户逐渐增加他们选择用于其应用程序的 OLM 框架量。

OLM Operator 使用以下工作流：

1. 观察命名空间中的集群服务版本（CSV），并检查是否满足要求。
2. 如果满足要求，请运行 CSV 的安装策略。

**注意**

CSV 必须是 Operator 组的活跃成员，才可运行该安装策略。

#### 2.4.2.3. Catalog Operator

Catalog Operator 负责解析和安装集群服务版本（CSV）以及它们指定的所需资源。另外还负责监视频道中的目录源中是否有软件包更新，并将其升级（可选择自动）至最新可用版本。

要跟踪频道中的软件包，您可以创建一个 **Subscription** 对象来配置所需的软件包、频道和 **CatalogSource** 对象，以便拉取更新。在找到更新后，便会代表用户将一个适当的 **InstallPlan** 对象写入命名空间。

Catalog Operator 使用以下工作流：

1. 连接到集群中的每个目录源。
2. 监视是否有用户创建的未解析安装计划，如果有：
   a. 查找与请求名称相匹配的 CSV，并将此 CSC 添加为已解析的资源。
   b. 对于每个受管或所需 CRD，将其添加为已解析的资源。
   c. 对于每个所需 CRD，找到管理相应 CRD 的 CSV。
3. 监视是否有已解析的安装计划并为其创建已发现的所有资源（用户批准或自动）。
4. 观察目录源和订阅并根据它们创建安装计划。

#### 2.4.2.4. Catalog Registry

Catalog Registry 存储 CSV 和 CRD 以便在集群中创建，并存储有关软件包和频道的元数据。

*package manifest* 是 Catalog Registry 中的一个条目，用于将软件包标识与 CSV 集相关联。在软件包中，频道指向特定 CSV。因为 CSV 明确引用了所替换的 CSV，软件包清单向 Catalog Operator 提供了将 CSV 更新至频道中最新版本所需的信息，逐步安装和替换每个中间版本。

### 2.4.3. Operator Lifecycle Manager 工作流

本指南概述了 OpenShift Container Platform 中 Operator Lifecycle Manager (OLM) 的工作流。

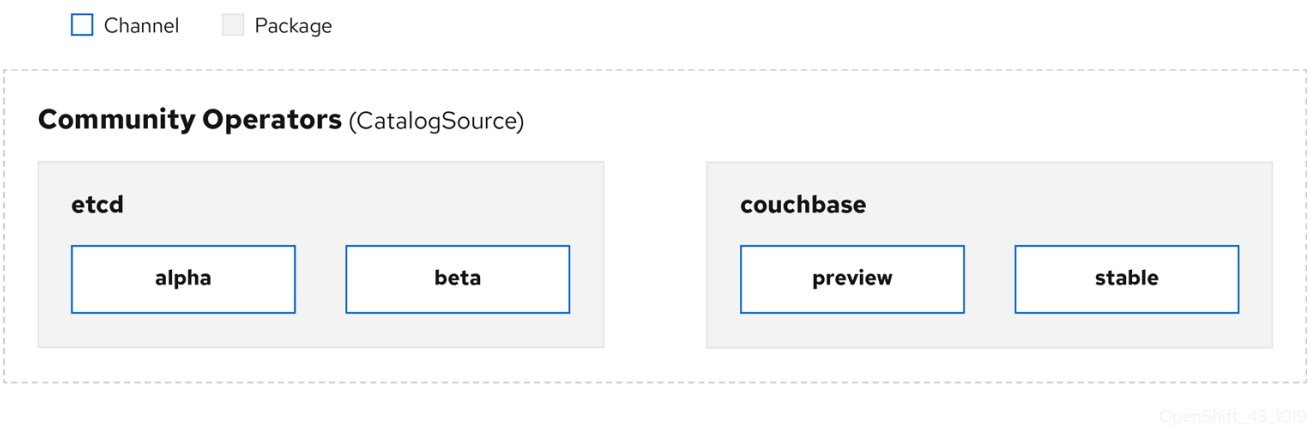
#### 2.4.3.1. OLM 中的 Operator 安装和升级工作流

在 Operator Lifecycle Manager (OLM) 生态系统中，以下资源用于解决 Operator 的安装和升级问题：

- **ClusterServiceVersion** (CSV)
- **CatalogSource**
- **Subscription**

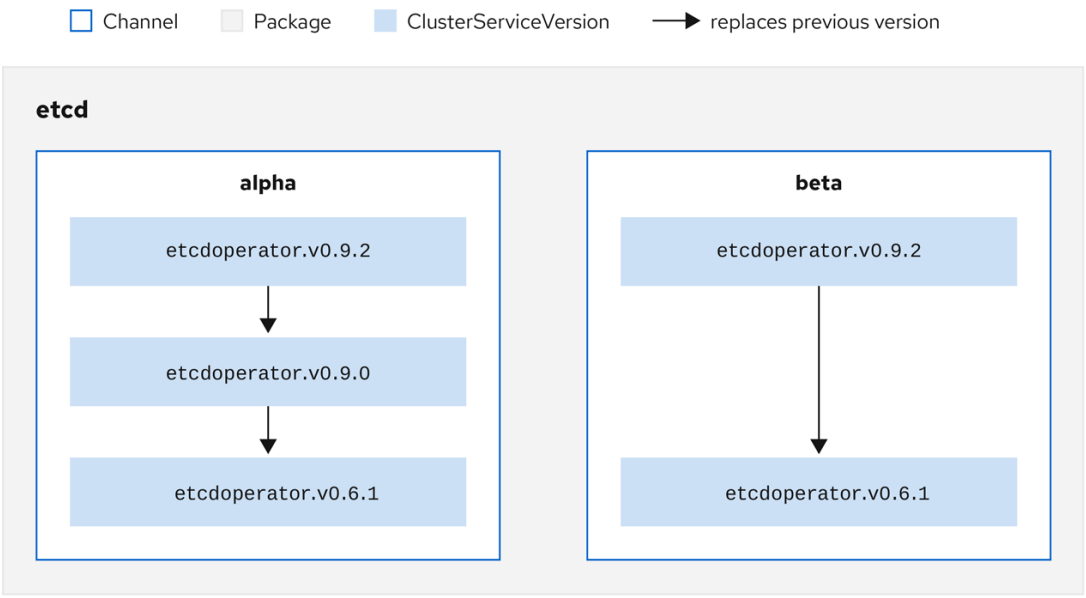
CSV 中定义的 Operator 元数据可保存在一个称为目录源的集合中。目录源使用 Operator Registry API，OLM 又使用目录源来查询是否有可用的 Operator 及已安装 Operator 是否有升级版本。

**图 2.3. 目录源概述**

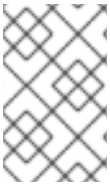



在目录源中，Operator 被整合为更新*软件包*和更新流，我们称为*频道*，这应是 OpenShift Container Platform 或其他软件（如 Web 浏览器）在持续发行周期中的常见更新模式。

图 2.4. 目录源中的软件包和频道




用户在*订阅*中的特定目录源中指示特定软件包和频道，如 **etcd** 包及其 **alpha** 频道。如果订阅了命名空间中尚未安装的软件包，则会安装该软件包的最新 Operator。

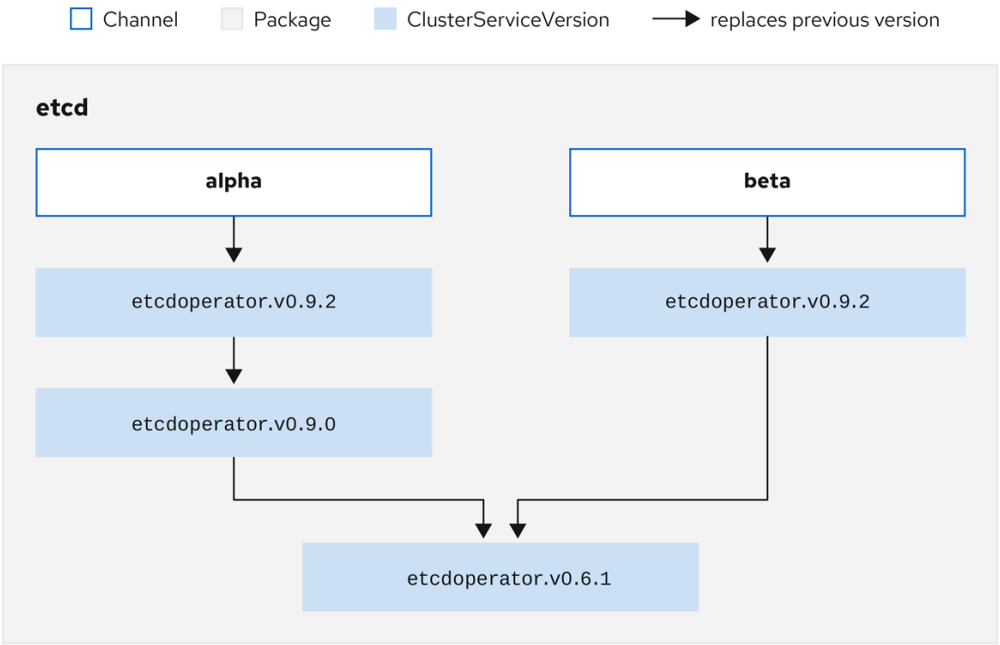


**注意**

OLM 会刻意避免版本比较，因此给定 *catalog → channel → package* 路径提供的"latest"或"newest"Operator 不一定是最高版本号。更应将其视为频道的 *head* 引用，类似 Git 存储库。

每个 CSV 均有一个 **replaces** 参数，指明所替换的是哪个 Operator。这样便构建了一个可通过 OLM 查询的 CSV 图，且不同频道之间可共享更新。可将频道视为更新图表的入口点：

图 2.5. OLM 的可用频道更新图表




软件包中的频道示例

```
packageName: example
channels:
- name: alpha
  currentCSV: example.v0.1.2
- name: beta
  currentCSV: example.v0.1.3
defaultChannel: alpha
```

为了让 OLM 成功查询更新、给定一个目录源、软件包、频道和 CSV，目录必须能够明确无误地返回**替换**输入 CSV 的单个 CSV。

#### 2.4.3.1.1. 升级路径示例

对于示例升级场景，假设安装的 Operator 对应于 **0.1.1** 版 CSV。OLM 查询目录源，并在订阅的频道中检测升级，新的 **0.1.3** 版 CSV 替换了旧的但未安装的 **0.1.2** 版 CSV，后者又取代了较早且已安装的 **0.1.1** 版 CSV。

OLM 通过 CSV 中指定的 **replaces** 字段从频道头倒退至之前的版本，以确定升级路径为 **0.1.3 → 0.1.2 → 0.1.1**，其中箭头代表前者取代后者。OLM 一次仅升级一个 Operator 版本，直至到达频道头。

对于该给定场景，OLM 会安装 **0.1.2** 版 Operator 来取代现有的 **0.1.1** 版 Operator。然后再安装 **0.1.3** 版 Operator 来取代之前安装的 **0.1.2** 版 Operator。至此，所安装的 **0.1.3** 版 Operator 与频道头相匹配，意味着升级已完成。

#### 2.4.3.1.2. 跳过升级

OLM 中升级的基本路径是：

- 通过对 Operator 的一个或多个更新来更新目录源。
- OLM 会遍历 Operator 的所有版本，直到到达目录源包含的最新版本。

但有时这不是一种安全操作。某些情况下，已发布但尚未就绪的 Operator 版本不可安装至集群中，如版本中存在严重漏洞。

这种情况下，OLM 必须考虑两个集群状态，并提供支持这两个状态的更新图：

- 集群发现并安装了“不良”中间 Operator。
- “不良”中间 Operator 尚未安装至集群中。

通过发送新目录并添加*跳过的*发行版本，可保证无论集群状态如何以及是否发现了不良更新，OLM 总能获得单个唯一更新。

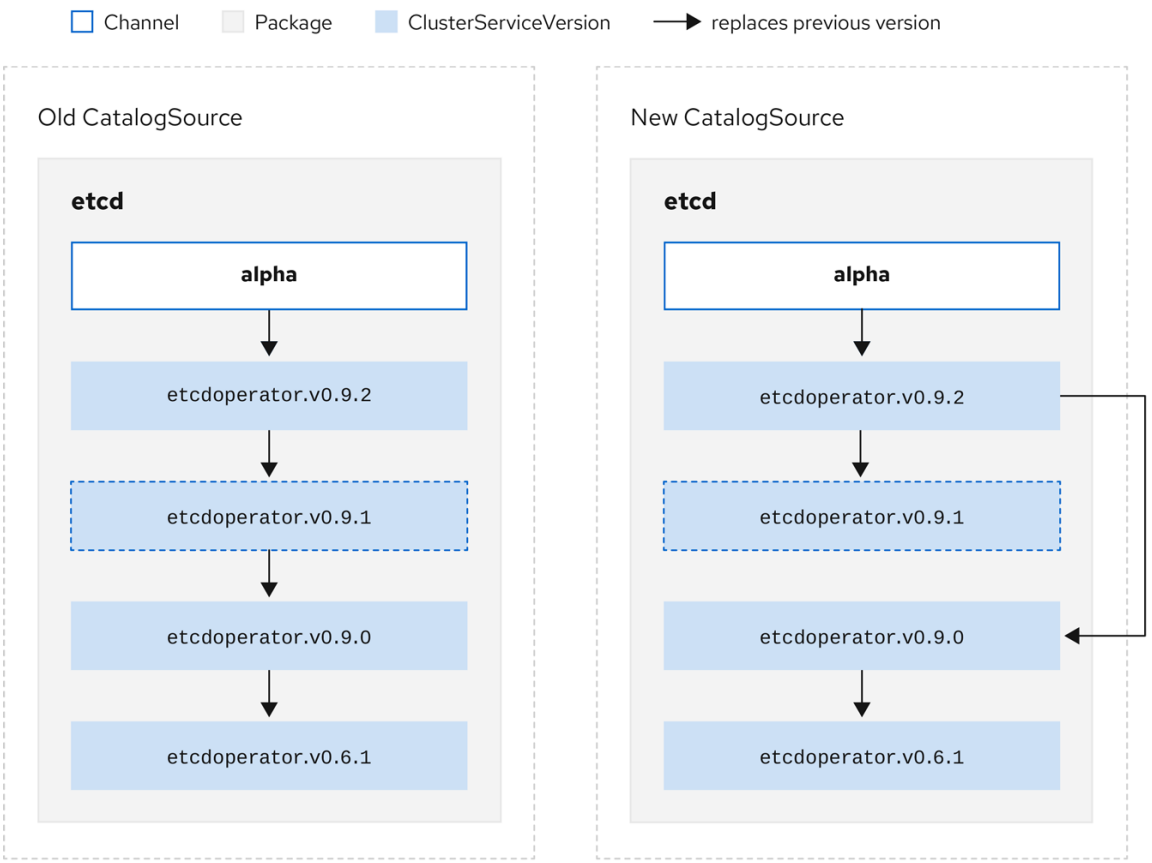
**带有跳过发行版本的 CSV 示例**

```
apiVersion: operators.coreos.com/v1alpha1
kind: ClusterServiceVersion
metadata:
  name: etcdoperator.v0.9.2
  namespace: placeholder
  annotations:
spec:
    displayName: etcd
    description: Etcd Operator
```

```
replaces: etcdoperator.v0.9.0
skips:
- etcdoperator.v0.9.1
```

考虑以下 **Old CatalogSource** 和 **New CatalogSource** 示例。

**图 2.6. 跳过更新**




OpenShift_43_1019

该图表明：

- **Old CatalogSource** 中的任何 Operator 在 **New CatalogSource** 中均有单一替换项。
- **New CatalogSource** 中的任何 Operator 在 **New CatalogSource** 中均有单一替换项。
- 如果未曾安装不良更新，将来也绝不会安装。

#### 2.4.3.1.3. 替换多个 Operator

按照描述创建 **New CatalogSource** 需要发布 CSV 来**替换**单个 Operator，但可**跳过**多个。该操作可通过 **skipRange** 注解来完成：

```
olm.skipRange: <semver_range>
```

其中 **<semver_range>** 具有 semver library 所支持的版本范围格式。

当在目录中搜索更新时，如果某个频道头提供一个 **skipRange** 注解，且当前安装的 Operator 的版本字段在该范围内，则 OLM 会更新至该频道中的最新条目。

先后顺序：

1. Subscription 上由 **sourceName** 指定的源中的频道头（满足其他跳过条件的情况下）。
2. 在 **sourceName** 指定的源中替换当前 Operator 的下一 Operator。
3. 对 Subscription 可见的另一个源中的频道头（满足其他跳过条件的情况下）。
4. 在对 Subscription 可见的任何源中替换当前 Operator 的下一 Operator。

**带有 skipRange 的 CSV 示例**

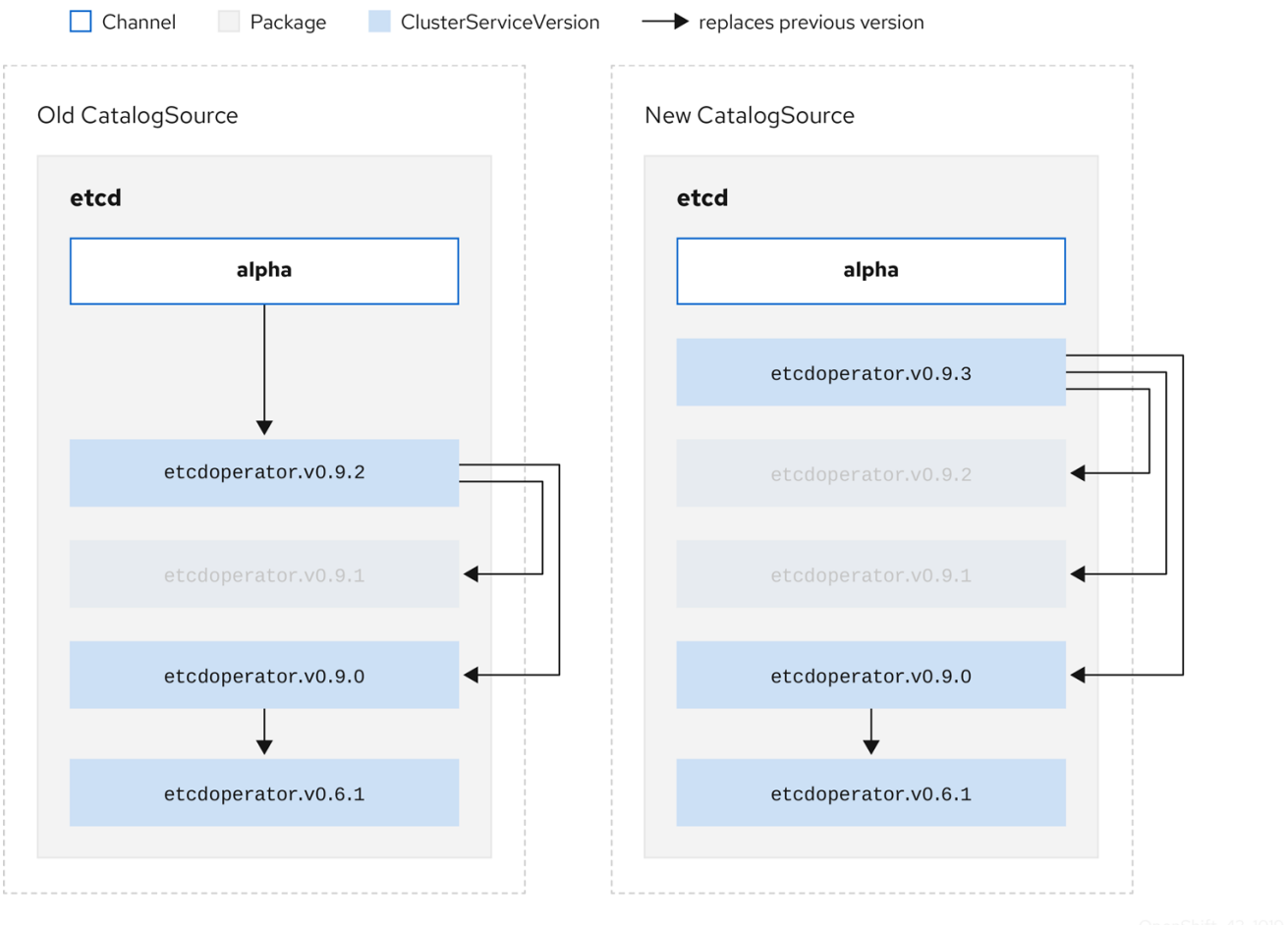
```
apiVersion: operators.coreos.com/v1alpha1
kind: ClusterServiceVersion
metadata:
  name: elasticsearch-operator.v4.1.2
  namespace: <namespace>
  annotations:
    olm.skipRange: '>=4.1.0 <4.1.2'
```

#### 2.4.3.1.4. Z-stream 支持

对于相同从版本，*z-stream* 或补丁版本必须取代所有先前 z-stream 版本。OLM 不考虑主版本、次版本或补丁版本，只需要在目录中构建正确的图表。

换句话说，OLM 必须能够像在 **Old CatalogSource** 中一样获取一个图表，像在 **New CatalogSource** 中一样生成一个图表：

**图 2.7. 替换多个 Operator**



该图表明：

- **Old CatalogSource** 中的任何 Operator 在 **New CatalogSource** 中均有单一替换项。
- **New CatalogSource** 中的任何 Operator 在 **New CatalogSource** 中均有单一替换项。
- **Old CatalogSource** 中的所有 z-stream 版本均会更新至 **New CatalogSource** 中最新 z-stream 版本。
- 不可用版本可被视为"虚拟"图表节点；它们的内容无需存在，注册表只需像图表看上去这样响应即可。

### 2.4.4. Operator Lifecycle Manager 依赖项解析

本指南概述了 OpenShift Container Platform 中 Operator Lifecycle Manager (OLM) 内的依赖项解析和自定义资源定义 (CRD) 升级生命周期。

#### 2.4.4.1. 关于依赖项解析

OLM 管理运行 Operator 的依赖项解析和升级生命周期。在很多方面，OLM 的问题与其他操作系统软件包管理程序类似，比如 **yum** 和 **rpm**。

但其中有一个限制是相似系统一般不存在而 OLM 存在的，那就是：因为 Operator 始终在运行，所以 OLM 会努力确保您所接触的 Operator 组始终相互兼容。

这意味着 OLM 不得执行以下操作：

- 安装一组需要无法提供的 API 的 Operator。
- 更新某个 Operator 之时导致依赖该 Operator 的另一 Operator 中断。

#### 2.4.4.2. 依赖项文件

Operator 的依赖项列在捆绑包的 **metadata/** 目录中的 **dependencies.yaml** 文件中。此文件是可选的，目前仅用于指明 Operator-version 依赖项。

依赖项列表中，每个项目包含一个 **type** 字段，用于指定这一依赖项的类型。受支持的 Operator 依赖项有两种：

- **olm.package**：此类型表示特定 Operator 版本的依赖项。依赖项信息必须包含软件包名称以及软件包的版本，格式为 semver。例如，您可以指定具体版本，如 **0.5.2**，也可指定一系列版本，如 **>0.5.1**。
- **olm.gvk**：使用 **gvk** 类型，作者可以使用 group/version/kind（GVK）信息指定依赖项，类似于 CSV 中现有 CRD 和基于 API 的使用。该路径使 Operator 作者可以合并所有依赖项、API 或显式版本，使它们处于同一位置。

在以下示例中，为 Prometheus Operator 和 etcd CRD 指定依赖项：

**dependencies.yaml 文件示例**

```
dependencies:
  - type: olm.package
    value:
      packageName: prometheus
      version: ">0.27.0"
```

```
- type: olm.gvk
  value:
    group: etcd.database.coreos.com
    kind: EtcdCluster
    version: v1beta2
```

#### 2.4.4.3. 依赖项首选项

有很多选项同样可以满足 Operator 的依赖性。Operator Lifecycle Manager（OLM）中的依赖项解析器决定哪个选项最适合所请求 Operator 的要求。作为 Operator 作者或用户，了解这些选择非常重要，以便明确依赖项解析。

##### 2.4.4.3.1. 目录优先级

在 OpenShift Container Platform 集群中，OLM 会读取目录源以了解哪些 Operator 可用于安装。

**CatalogSource** 对象示例

```
apiVersion: "operators.coreos.com/v1alpha1"
kind: "CatalogSource"
metadata:
  name: "my-operators"
  namespace: "operators"
spec:
  sourceType: grpc
  image: example.com/my/operator-index:v1
  displayName: "My Operators"
  priority: 100
```

**CatalogSource** 有一个 **priority** 字段，解析器使用它来知道如何为依赖关系设置首选选项。

目录首选项有两个规则：

- 优先级较高目录中的选项优先于较低优先级目录的选项。
- 与依赖项相同的目录里的选项优先于其它目录。

##### 2.4.4.3.2. 频道排序

目录中的 Operator 软件包是用户可在 OpenShift Container Platform 集群中订阅的更新频道集合。可使用频道为次发行版本（**1.2**, **1.3**）或者发行的频率（**stable**, **fast**）提供特定的更新流。

同一软件包中的 Operator 可能会满足依赖项，但可能会在不同的频道。例如，Operator 版本 **1.2** 可能存在于 **stable** 和 **fast** 频道中。

每个软件包都有一个默认频道，该频道总是首选非默认频道。如果默认频道中没有选项可以满足依赖关系，则会在剩余的频道中按频道名称的字母顺序考虑这些选项。

##### 2.4.4.3.3. 频道中的顺序

一般情况下，总会有多个选项来满足单一频道中的依赖关系。例如，一个软件包和频道中的 Operator 提供了相同的 API 集。

当用户创建订阅时，它们会指示要从哪个频道接收更新。这会立即把搜索范围限制在那个频道。但是在频道中，可以会有许多 Operator 可以满足依赖项。

在频道中，应该首选考虑使用更新图中位置较高的较新的 Operator。如果某个频道的头满足依赖关系，它将会被首先尝试。

#### 2.4.4.3.4. 其他限制

除了软件包依赖关系的限制外，OLM 还添加了其他限制来代表所需用户状态和强制实施解析变量。

##### 2.4.4.3.4.1. 订阅约束

一个订阅（Subscription）约束会过滤可满足订阅的 Operator 集合。订阅是对依赖项解析程序用户提供的限制。它们会声明安装一个新的 Operator（如果还没有在集群中安装），或对现有 Operator 进行更新。

##### 2.4.4.3.4.2. 软件包约束

在命名空间中，不同的两个 Operator 不能来自于同一软件包。

### 2.4.4.4. CRD 升级

如果自定义资源定义（CRD）属于单一集群服务版本（CSV），OLM 会立即对其升级。如果某个 CRD 被多个 CSV 拥有，则当该 CRD 满足以下所有向后兼容条件时才会升级：

- 所有已存在于当前 CRD 中的服务版本都包括在新 CRD 中。
- 在根据新 CRD 的验证模式（schema）进行验证后，与 CRD 的服务版本关联的所有现有实例或自定义资源均有效。

**其他资源**

- 添加新版 CRD
- 弃用或删除 CRD 版本

### 2.4.4.5. 依赖项最佳实践

在指定依赖项时应该考虑的最佳实践。

**依赖于 API 或 Operator 的特定版本范围**

操作员可以随时添加或删除 API；始终针对 Operator 所需的任何 API 指定 **olm.gvk** 依赖项。例外情况是，指定 **olm.package** 约束来替代。

**设置最小版本**

Kubernetes 文档中与 API 的改变相关的部分描述了 Kubernetes 风格的 Operator 允许进行哪些更改。只要 API 向后兼容，Operator 就允许 Operator 对 API 进行更新，而不需要更改 API 的版本。对于 Operator 依赖项，这意味着了解依赖的 API 版本可能不足以确保依赖的 Operator 正常工作。

例如：

- TestOperator v1.0.0 提供 **MyObject** 资源的 v1alpha1 API 版本。
- TestOperator v1.0.1 为 **MyObject** 添加了一个新的 **spec.newfield** 字段，但仍是 v1alpha1。

您的 Operator 可能需要将 **spec.newfield** 写入 **MyObject** 资源。仅使用 **olm.gvk** 约束还不足以让 OLM 决定您需要 TestOperator v1.0.1 而不是 TestOperator v1.0.0。

如果事先知道提供 API 的特定 Operator，则指定额外的 **olm.package** 约束来设置最小值。

**省略一个最大版本，或允许一个广泛的范围**

因为 Operator 提供了集群范围的资源，如 API 服务和 CRD，所以如果一个 Operator 为依赖项指定了一个小的窗口，则可能会对依赖项的其他用户的更新产生不必要的约束。
在可能的情况下，尽量不要设置最大版本。或者，设置一个非常宽松的语义范围，以防止与其他 Operator 冲突。例如：**>1.0.0 <2.0.0**。

与传统的软件包管理器不同，Operator 作者显性地对更新通过 OLM 中的频道进行编码。如果现有订阅有可用更新，则假定 Operator 作者表示它可以从上一版本更新。为依赖项设置最大版本会绕过作者的更新流，即不必要的将它截断到特定的上限。

**注意**

集群管理员无法覆盖 Operator 作者设置的依赖项。

但是，如果已知有需要避免的不兼容问题，就应该设置最大版本。通过使用版本范围语法，可以省略特定的版本，如 **> 1.0.0 !1.2.1**。

**其他资源**

- Kubernetes 文档： 更改 API

### 2.4.4.6. 依赖项注意事项

当指定依赖项时，需要考虑一些注意事项。

**没有捆绑包约束（AND）**

目前还没有方法指定约束间的 AND 关系。换句话说，无法指定一个 Operator，它依赖于另外一个 Operator，它提供一个给定的 API 且版本是 **>1.1.0**。
这意味着，在指定依赖项时，如：

```
dependencies:
- type: olm.package
  value:
    packageName: etcd
    version: ">3.1.0"
- type: olm.gvk
  value:
    group: etcd.database.coreos.com
    kind: EtcdCluster
    version: v1beta2
```

OLM 可以通过两个 Operator 来满足这个要求：一个提供 EtcdCluster，另一个有版本 **>3.1.0**。是否发生了这种情况，或者选择某个 Operator 是否满足这两个限制，这取决于是否准备了潜在的选项。依赖项偏好和排序选项被明确定义并可以指定原因，但为了谨慎起见，Operator 应该遵循一种机制或其他机制。

**跨命名空间兼容性**

OLM 在命名空间范围内执行依赖项解析。如果更新某个命名空间中的 Operator 会对另一个命名空间中的 Operator 造成问题，则可能会造成更新死锁。

### 2.4.4.7. 依赖项解析方案示例

在以下示例中，*provider（供应商）*是指"拥有" CRD 或 API 服务的 Operator。

**示例：弃用从属 API**
A 和 B 是 API（CRD）：

- A 的供应商依赖 B。
- B 的供应商有一个订阅。
- B 更新供应商提供 C，但弃用 B。

结果：

- B 不再有供应商。
- A 不再工作。

这是 OLM 通过升级策略阻止的一个案例。

**示例：版本死锁**
A 和 B 均为 API：

- A 的供应商需要 B。
- B 的供应商需要 A。
- A 更新的供应商到（提供 A2，需要 B2）并弃用 A。
- B 更新的供应商到（提供 B2,需要 A2）并弃用 B。

如果 OLM 试图在更新 A 的同时不更新 B，或更新 B 的同时不更新 A，则无法升级到新版 Operator，即使可找到新的兼容集也无法更新。

这是 OLM 通过升级策略阻止的另一案例。

## 2.4.5. operator 组

本指南概述了 OpenShift Container Platform 中 Operator Lifecycle Manager（OLM）的 Operator 组使用情况。

### 2.4.5.1. 关于 Operator 组

由 **OperatorGroup** 资源定义的 *Operator 组*，为 OLM 安装的 Operator 提供多租户配置。Operator 组选择目标命名空间，在其中为其成员 Operator 生成所需的 RBAC 访问权限。

这一组目标命名空间通过存储在 CSV 的 **olm.targetNamespaces** 注解中的以逗号分隔的字符串来提供。该注解应用于成员 Operator 的 CSV 实例，并注入它们的部署中。

### 2.4.5.2. Operator 组成员

满足以下任一条件，Operator 即可被视为 Operator 组的 *member*：

- Operator 的 CSV 与 Operator 组位于同一命名空间中。
- Operator CSV 中的安装模式支持 Operator 组的目标命名空间集。

CSV 中的安装模式由 **InstallModeType** 字段和 **Supported** 的布尔值字段组成。CSV 的 spec 可以包含一组由四个不同 **InstallModeTypes** 组成的安装模式：

**表 2.4. 安装模式和支持的 Operator 组**

| InstallModeType | 描述 |
|---|---|
| **OwnNamespace** | Operator 可以是选择其自有命名空间的 Operator 组的成员。 |
| **SingleNamespace** | Operator 可以是选择一个命名空间的 Operator 组的成员。 |
| **MultiNamespace** | Operator 可以是选择多个命名空间的 Operator 组的成员。 |
| **AllNamespaces** | Operator 可以是选择所有命名空间的 Operator 组的成员（目标命名空间集为空字符串 **""**）。 |

**注意**

如果 CSV 的 spec 省略 **InstallModeType** 条目，则该类型将被视为不受支持，除非可通过隐式支持的现有条目推断出支持。

#### 2.4.5.3. 目标命名空间选择

您可以使用 **spec.targetNamespaces** 参数为 Operator 组显式命名目标命名空间：

```
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: my-group
  namespace: my-namespace
spec:
  targetNamespaces:
  - my-namespace
```

您还可以使用带有 **spec.selector** 参数的标签选择器指定命名空间：

```
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: my-group
  namespace: my-namespace
spec:
  selector:
    cool.io/prod: "true"
```

**重要**

不建议通过 **spec.targetNamespaces** 列出多个命名空间，或通过 **spec.selector** 使用标签选择器，因为在以后的版本中可能会删除对 Operator 组中多个目标命名空间的支持。

如果 **spec.targetNamespaces** 和 **spec.selector** 均已定义，则会忽略 **spec.selector**。另外，您可以省略 **spec.selector** 和 **spec.targetNamespaces** 来指定一个 *全局* Operator 组，该组选择所有命名空间：

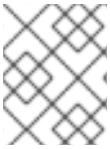
```
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: my-group
  namespace: my-namespace
```

Opeator 组的 **status.namespaces** 参数中会显示所选命名空间的解析集合。全局 OperatorGroup 的 **status.namespace** 包含空字符串 (**""**)，而该字符串会向正在使用的 Operator 发出信号，要求其监视所有命名空间。

#### 2.4.5.4. operator 组 CSV 注解

Operator 组的成员 CSV 具有以下注解：

| 注解 | 描述 |
| --- | --- |
| **olm.operatorGroup=<group_name>** | 包含 Operator 组的名称。 |
| **olm.operatorNamespace=<group_namespace>** | 包含 Operator 组的命名空间。 |
| **olm.targetNamespaces=<target_namespaces>** | 包含以逗号分隔的字符串，列出 Operator 组的目标命名空间选择。 |



**注意**

除 **olm.targetNamespaces** 以外的所有注解均包含在复制的 CSV 中。在复制的 CSV 上省略 **olm.targetNamespaces** 注解可防止租户之间目标命名空间出现重复。

#### 2.4.5.5. 所提供的 API 注解

*group/version/kind (GVK)* 是 Kubernetes API 的唯一标识符。 **olm.providedAPIs** 注解中会显示有关 Operator 组提供哪些 GVK 的信息。该注解值为一个字符串，由用逗号分隔的 **<kind>.<version>.<group>** 组成。其中包括由 Operator 组的所有活跃成员 CSV 提供的 CRD 和 APIService 的 GVK。

查看以下 **OperatorGroup** 示例，该 OperatorGroup 带有提供 **PackageManifest** 资源的单个活跃成员 CSV：

```
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  annotations:
    olm.providedAPIs: PackageManifest.v1alpha1.packages.apps.redhat.com
  name: olm-operators
  namespace: local
  ...
spec:
  selector: {}
  serviceAccount:
```

```
    metadata:
      creationTimestamp: null
  targetNamespaces:
  - local
status:
  lastUpdated: 2019-02-19T16:18:28Z
  namespaces:
  - local
```

#### 2.4.5.6. 基于角色的访问控制

创建 Operator 组时，会生成三个集群角色。每个 ClusterRole 均包含一个聚会规则，后者带有一个选择器以匹配标签，如下所示：

| 集群角色 | 要匹配的标签 |
| --- | --- |
| **<operatorgroup_name>-admin** | **olm.opgroup.permissions/aggregate-to-admin: <operatorgroup_name>** |
| **<operatorgroup_name>-edit** | **olm.opgroup.permissions/aggregate-to-edit: <operatorgroup_name>** |
| **<operatorgroup_name>-view** | **olm.opgroup.permissions/aggregate-to-view: <operatorgroup_name>** |

当 CSV 成为 Operator 组的活跃成员时，只要该 CSV 正在使用 **AllNamespaces** 安装模式来监视所有命名空间，且没有因 **InterOperatorGroupOwnerConflict** 原因处于故障状态，便会生成以下 RBAC 资源。

- 来自 CRD 的每个 API 资源的集群角色
- 来自 API 服务的每个 API 资源的集群角色
- 其他角色和角色绑定

**表 2.5. 来自 CRD 的为每个 API 资源生成的集群角色**

| 集群角色 | 设置 |
| --- | --- |
| **<kind>.<group>-<version>-admin** | **<kind>** 上的操作动词：<br>• *<br>聚合标签：<br>• **rbac.authorization.k8s.io/aggregate-to-admin: true**<br>• **olm.opgroup.permissions/aggregate-to-admin: <operatorgroup_name>** |

| 集群角色 | 设置 |
| --- | --- |
| **<kind>.<group>-<version>-edit** | **<kind>** 上的操作动词：<br>• **create**<br>• **update**<br>• **patch**<br>• **delete**<br>聚合标签：<br>• **rbac.authorization.k8s.io/aggregate-to-edit: true**<br>• **olm.opgroup.permissions/aggregate-to-edit: <operatorgroup_name>** |
| **<kind>.<group>-<version>-view** | **<kind>** 上的操作动词：<br>• **get**<br>• **list**<br>• **watch**<br>聚合标签：<br>• **rbac.authorization.k8s.io/aggregate-to-view: true**<br>• **olm.opgroup.permissions/aggregate-to-view: <operatorgroup_name>** |
| **<kind>.<group>-<version>-view-crdview** | **apiextensions.k8s.io customresourcedefinitions <crd-name>** 上的操作动词：<br>• **get**<br>聚合标签：<br>• **rbac.authorization.k8s.io/aggregate-to-view: true**<br>• **olm.opgroup.permissions/aggregate-to-view: <operatorgroup_name>** |

**表 2.6. 来自 API 服务的为每个 API 资源生成的集群角色**

| 集群角色 | 设置 |
| --- | --- |

| 集群角色 | 设置 |
| --- | --- |
| **<kind>.<group>-<version>-admin** | **<kind>** 上的操作动词：<br>• *****<br>聚合标签：<br>• **rbac.authorization.k8s.io/aggregate-to-admin: true**<br>• **olm.opgroup.permissions/aggregate-to-admin: <operatorgroup_name>** |
| **<kind>.<group>-<version>-edit** | **<kind>** 上的操作动词：<br>• **create**<br>• **update**<br>• **patch**<br>• **delete**<br>聚合标签：<br>• **rbac.authorization.k8s.io/aggregate-to-edit: true**<br>• **olm.opgroup.permissions/aggregate-to-edit: <operatorgroup_name>** |
| **<kind>.<group>-<version>-view** | **<kind>** 上的操作动词：<br>• **get**<br>• **list**<br>• **watch**<br>聚合标签：<br>• **rbac.authorization.k8s.io/aggregate-to-view: true**<br>• **olm.opgroup.permissions/aggregate-to-view: <operatorgroup_name>** |

**其他角色和角色绑定**

- 如果 CSV 定义了一个目标命名空间，其中包括 *****，则会针对 CSV 权限字段中定义的每个 **permissions** 生成集群角色和对应集群角色绑定。所有生成的资源均会标上 **olm.owner: <csv_name>** 和 **olm.owner.namespace: <csv_namespace>** 标签。

- 如果 CSV 没有定义一个包含 * 的目标命名空间，则 Operator 命名空间中的所有角色和角色绑定

- 如果 CSV 没有定义一个包含 * 的目标命名空间，则 Operator 命名空间中的所有角色和角色绑定都使用 **olm.owner: <csv_name>** 和 **olm.owner.namespace: <csv_namespace>** 标签复制到目标命名空间中。

#### 2.4.5.7. 复制的 CSV

OLM 会在 Operator 组的每个目标命名空间中创建 Operator 组的所有活跃成员 CSV 的副本。复制 CSV 的目的在于告诉目标命名空间的用户，特定 Operator 已配置为监视在此创建的资源。

复制的 CSV 会**复制**状态原因，并会更新以匹配其源 CSV 的状态。在集群上创建复制的 CSV 之前，会从这些 CSV 中分离 **olm.targetNamespaces** 注解。省略目标命名空间选择可避免租户之间存在目标命名空间重复的现象。

当所复制的 CSV 的源 CSV 不存在或其源 CSV 所属的 Operator 组不再指向复制 CSV 的命名空间时，会删除复制的 CSV。

#### 2.4.5.8. 静态 Operator 组

如果 Operator 组的 **spec.staticProvidedAPIs** 字段被设置为 **true**，则 Operator 组为*静态*。因此，OLM 不会修改 Operator 组的 **olm.providedAPIs** 注解，这意味着可以提前设置它。如果一组命名空间没有活跃的成员 CSV 来为资源提供 API，而用户想使用 Operator 组来防止命名空间集中发生资源争用，则这一操作十分有用。

以下是一个 Operator 组示例，它使用 **something.cool.io/cluster-monitoring: "true"** 注解来保护所有命名空间中的 **Prometheus** 资源：

```
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: cluster-monitoring
  namespace: cluster-monitoring
  annotations:
    olm.providedAPIs:
Alertmanager.v1.monitoring.coreos.com,Prometheus.v1.monitoring.coreos.com,PrometheusRule.v1.mo
nitoring.coreos.com,ServiceMonitor.v1.monitoring.coreos.com
spec:
  staticProvidedAPIs: true
  selector:
    matchLabels:
      something.cool.io/cluster-monitoring: "true"
```

#### 2.4.5.9. operator 组交集

如果两个 Operator 组的目标命名空间集的交集不是空集，且根据 **olm.providedAPIs** 注解的定义，所提供的 API 集的交集也不是空集，则称这两个 OperatorGroup 的*提供的 API 有交集*。

一个潜在问题是，提供的 API 有交集的 Operator 组可能在命名空间交集中竞争相同资源。

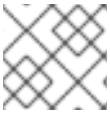
**注意**

在检查交集规则时，Operator 组的命名空间始终包含在其所选目标命名空间中。

**交集规则**

每次活跃成员 CSV 同步时，OLM 均会查询集群，以获取 CSV 组和其他所有 CSV 组之间提供的 API 交集。然后 OLM 会检查该交集是否为空集：

- 如果结果为 **true**，且 CSV 提供的 API 是 Operator 组提供的 API 的子集：
  - 继续转变。
- 如果结果为 **true**，且 CSV 提供的 API *不是* Operator 组提供的 API 的子集：
  - 如果 Operator 组是静态的：
    - 则清理属于 CSV 的所有部署。
    - 将 CSV 转变为故障状态，状态原因为：**CannotModifyStaticOperatorGroupProvidedAPIs**。
  - 如果 Operator 组*不是*静态的：
    - 将 Operator 组的 **olm.providedAPIs** 注解替换为其本身与 CSV 提供的 API 的并集。
- 如果结果为 **false**，且 CSV 提供的 API *不是* Operator 组提供的 API 的子集：
  - 则清理属于 CSV 的所有部署。
  - 将 CSV 转变为故障状态，状态原因为：**InterOperatorGroupOwnerConflict**。
- 如果结果为 **false**，且 CSV 提供的 API 是 Operator 组提供的 API 的子集：
  - 如果 Operator 组是静态的：
    - 则清理属于 CSV 的所有部署。
    - 将 CSV 转变为故障状态，状态原因为：**CannotModifyStaticOperatorGroupProvidedAPIs**。
  - 如果 Operator 组*不是*静态的：
    - 将 Operator 组的 **olm.providedAPIs** 注解替换为其本身与 CSV 提供的 API 的差集。



**注意**

Operator 组所造成的故障状态不是终端状态。

每次 Operator 组同步时都会执行以下操作：

- 来自活跃成员 CSV 的提供的 API 集是通过集群计算出来的。注意，复制的 CSV 会被忽略。
- 将集群集与 **olm.providedAPIs** 进行比较，如果 **olm.providedAPIs** 包含任何额外 API，则将删除这些 API。
- 在所有命名空间中提供相同 API 的所有 CSV 均会重新排序。这样可向交集组中的冲突 CSV 发送通知，表明可能已通过调整大小或删除冲突的 CSV 解决了冲突。

#### 2.4.5.10. 多租户 Operator 管理的限制

OpenShift Container Platform 对在集群中同时安装不同 Operator 版本提供有限支持。Operator 是 control plane 扩展。所有租户或命名空间共享同一集群的 control plane。因此，多租户环境中的租户也必须共享 Operator。

Operator Lifecycle Manager (OLM) 会在不同的命名空间中多次安装 Operator。其中一个限制是 Operator 的 API 版本必须相同。

Operator 的不同主要版本通常具有不兼容的自定义资源定义 (CRD)。这使得无法快速验证 OLM。

#### 2.4.5.10.1. 其它资源

- 允许非集群管理员安装 Operator

### 2.4.5.11. 对 Operator 组进行故障排除

**成员资格**

- 如果一个命名空间中存在多个 Operator 组，则在该命名空间中创建的所有 CSV 均会变成故障状态，故障原因为：**TooManyOperatorGroups**。处于故障状态的 CSV 会在命名空间中的 Operator 组达到 1 后转变为等待处理状态。
- 如果 CSV 的安装模式不支持其命名空间中 Operator 组的目标命名空间选择，CSV 会转变为故障状态，原因为 **UnsupportedOperatorGroup**。处于故障状态的 CSV 会在 Operator 组的目标命名空间选择变为受支持的配置后转变为待处理，或者 CSV 的安装模式被修改来支持目标命名空间选择。

## 2.4.6. Operator 条件

本指南概述了 Operator Lifecycle Manager（OLM）如何使用 Operator 条件。

### 2.4.6.1. 关于 Operator 条件

作为管理 Operator 生命周期的角色的一部分，Operator Lifecycle Manager（OLM）从定义 Operator 的 Kubernetes 资源状态中推断 Operator 状态。虽然此方法提供了一定程度的保证来确定 Operator 处于给定状态，但在有些情况下，Operator 可能需要直接向 OLM 提供信息，而这些信息不能被推断出来。这些信息可以被 OLM 使用来更好地管理 Operator 的生命周期。

OLM 提供了一个名为 **OperatorCondition** 的自定义资源定义（CRD），它允许 Operator 与 OLM 相互通信条件信息。当在一个 **OperatorCondition** 资源的 **Status.Conditions** 数组中存在时，则代表存在一组会影响 OLM 管理 Operator 的支持条件。

### 2.4.6.2. 支持的条件

Operator Lifecycle Manager（OLM）支持以下 Operator 条件。

#### 2.4.6.2.1. Upgradeable（可升级）条件

**Upgradeable** Operator 条件可防止现有集群服务版本（CSV）被 CSV 的新版本替换。这一条件在以下情况下很有用：

- Operator 即将启动关键进程，不应在进程完成前升级。
- Operator 正在执行一个自定义资源（CR）迁移，这个迁移必须在 Operator 准备进行升级前完成。

**Upgradeable Operator 条件**

```
apiVersion: operators.coreos.com/v1
kind: OperatorCondition
```

```
metadata:
  name: my-operator
  namespace: operators
status:
  conditions:
  - type: Upgradeable 1
    status: "False" 2
    reason: "migration"
    message: "The Operator is performing a migration."
    lastTransitionTime: "2020-08-24T23:15:55Z"
```

1 条件的名称。

2 **False** 值表示 Operator 未准备好升级。OLM 可防止替换 Operator 现有 CSV 的 CSV 离开 **Pending** 状态。

#### 2.4.6.3. 其他资源

- 管理 Operator 条件
- 启用 Operator 条件

### 2.4.7. Operator Lifecycle Manager 指标数据

#### 2.4.7.1. 公开的指标

Operator Lifecycle Manager（OLM）会公开某些 OLM 特定资源，供基于 Prometheus 的 OpenShift Container Platform 集群监控堆栈使用。

**表 2.7. OLM 公开的指标**

| 名称 | 描述 |
| --- | --- |
| **catalog_source_count** | 目录源数量。 |
| **csv_abnormal** | 在协调集群服务版本（CSV）时，每当 CSV 版本处于 **Succeeded** 以外的任何状态时（如没有安装它时）就会存在。包括 **name**、**namespace**、**phase**、**reason** 和 **version** 标签。当存在此指标数据时会创建一个 Prometheus 警报。 |
| **csv_count** | 成功注册的 CSV 数量。 |
| **csv_succeeded** | 在协调 CSV 时，代表 CSV 版本处于 **Succeeded** 状态（值为 **1**）或没有处于这个状态（值为 **0**）。包含 **name**、**namespace** 和 **version** 标签。 |
| **csv_upgrade_count** | CSV 升级的 Monotonic 计数。 |
| **install_plan_count** | 安装计划的数量。 |

| 名称 | 描述 |
| --- | --- |
| **subscription_count** | 订阅数。 |
| **subscription_sync_total** | 订阅同步的单调计数。包括 **channel**、**installed** CSV 和订阅 **name** 标签。 |

### 2.4.8. Operator Lifecycle Manager 中的 Webhook 管理

Webhook 允许 Operator 作者在资源被保存到对象存储并由 Operator 控制器处理之前，拦截、修改、接受或拒绝资源。当 webhook 与 Operator 一同提供时，Operator Lifecycle Manager（OLM）可以管理这些 webhook 的生命周期。

如需有关 Operator 开发人员如何为其 Operator 定义 webhook，以及 OLM 上运行时的注意事项的详细信息，请参阅定义集群服务版本（CSV）。

#### 2.4.8.1. 其他资源

- Webhook 准入插件类型
- Kubernetes 文档：
  - 验证准入 webhook
  - 变异准入 webhook
  - webhook 转换

## 2.5. 了解 OPERATORHUB

### 2.5.1. 关于 OperatorHub

*OperatorHub* 是集群管理员用来发现和安装 Operator 的 OpenShift Container Platform 中的 Web 控制台界面。只需单击一次，即可从其非集群源拉取 Operator，并将其安装和订阅至集群中，为工程团队使用 Operator Lifecycle Manager (OLM) 在部署环境中自助管理产品做好准备。

集群管理员可从划分为以下类别的目录进行选择：

| 类别 | 描述 |
| --- | --- |
| 红帽 Operator | 已由红帽打包并提供的红帽产品。受红帽支持。 |
| 经认证的 Operator | 来自主要独立软件供应商 (ISV) 的产品。红帽与 ISV 合作打包并提供。受 ISV 支持。 |
| Red Hat Marketplace | 可通过 Red Hat Marketplace 购买认证的软件。 |

| 类别 | 描述 |
| --- | --- |
| 社区 Operator | operator-framework/community-operators GitHub 存储库中由相关代表维护的可选可见软件。无官方支持。 |
| 自定义 Operator | 您自行添加至集群的 Operator。如果您尚未添加任何自定义 Operator，则您的 OperatorHub 上 Web 控制台中便不会出现**自定义**类别。 |

OperatorHub 上的操作员被打包在 OLM 上运行。这包括一个称为集群服务版本（CSV）的 YAML 文件，其中包含安装和安全运行 Operator 所需的所有 CRD、RBAC 规则、Deployment 和容器镜像。它还包含用户可见的信息，如功能描述和支持的 Kubernetes 版本。

Operator SDK 可以用来协助开发人员打包 Operators 以用于 OLM 和 OperatorHub。如果您有一个需要方便客户访问的商业应用程序，请使用红帽合作伙伴连接门户（connect.redhat.com）提供的认证工作流来包括这个应用程序。

### 2.5.2. OperatorHub 架构

OperatorHub UI 组件默认由 **openshift-marketplace** 命名空间中 OpenShift Container Platform 上的 Marketplace Operator 驱动。

#### 2.5.2.1. OperatorHub 自定义资源

Marketplace Operator 管理名为 **cluster** 的 **OperatorHub** 自定义资源（CR），用于管理 OperatorHub 提供的默认 **CatalogSource** 对象。您可以修改此资源以启用或禁用默认目录，这在受限网络环境中配置 OpenShift Container Platform 时非常有用。

**OperatorHub 自定义资源示例**

```
apiVersion: config.openshift.io/v1
kind: OperatorHub
metadata:
  name: cluster
spec:
  disableAllDefaultSources: true (1)
  sources: [ (2)
    {
      name: "community-operators",
      disabled: false
    }
  ]
```

1. **disableAllDefaultSources** 是一个覆盖，用于控制在 OpenShift Container Platform 安装期间默认配置的所有默认目录的可用性。
2. 通过更改每个源的 **disabled** 参数值来分别禁用默认目录。

### 2.5.3. 其他资源

- 目录源

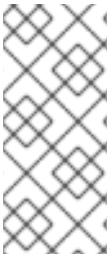
- 关于 Operator SDK
- 定义集群服务版本（CSV）
- OLM 中的 Operator 安装和升级工作流
- Red Hat Partner Connect
- Red Hat Marketplace

## 2.6. 红帽提供的 OPERATOR 目录

### 2.6.1. 关于 Operator 目录

Operator 目录是 Operator Lifecycle Manager（OLM）可以查询的元数据存储库，以在集群中发现和安装 Operator 及其依赖项。OLM 始终从目录的最新版本安装 Operator。从 OpenShift Container Platform 4.6 开始，红帽提供的目录会使用 *索引镜像*进行发布。

基于 Operator Bundle Format 的索引镜像是目录的容器化快照。这是一个不可变的工件，包含指向一组 Operator 清单内容的指针数据库。目录可以引用索引镜像来获取集群中 OLM 的内容。



**注意**

从 OpenShift Container Platform 4.6 开始，索引镜像由红帽提供来替代 App Registry 目录镜像。App Registry 目录镜像使用已弃用的 Package Manifest Format，它在以前的 OpenShift Container Platform 4 版本中分布。虽然红帽不会为 OpenShift Container Platform 4.6 及之后的版本发布 App Registry 目录镜像，但基于 Package Manifest Format 的自定义目录镜像仍被支持。

随着目录的更新，Operator 的最新版本会发生变化，旧版本可能会被删除或修改。另外，当 OLM 在受限网络环境中的 OpenShift Container Platform 集群上运行时，它无法直接从互联网访问目录来拉取最新内容。

作为集群管理员，您可以根据红帽提供的目录或从头创建自己的自定义索引镜像，该镜像可用于提供集群中的目录内容。创建和更新您自己的索引镜像提供了一种方法来自定义集群上可用的一组 Operator，同时避免了上面提到的受限网络环境中的问题。

**重要**

在创建自定义目录镜像时，**oc adm catalog build** 命令需要 OpenShift Container Platform 4 的早期版本，一些版本已弃用。从 OpenShift Container Platform 4.6 开始，红帽提供的索引镜像可用后，目录构建器应该开始切换为使用 **opm index** 命令来管理索引镜像，**oc adm catalog build** 命令会在以后的发行版本中被删除。

**其它资源**

- 管理自定义目录
- 在受限网络中使用 Operator Lifecycle Manager

### 2.6.2. 关于红帽提供的 Operator 目录

以下 Operator 目录由红帽发布：

| 目录 | 索引镜像 | 描述 |
| --- | --- | --- |
| **redhat-operators** | **registry.redhat.io/redhat/redhat-operator-index:v4.7** | 已由红帽打包并提供的红帽产品。受红帽支持。 |
| **certified-operators** | **registry.redhat.io/redhat/certified-operator-index:v4.7** | 来自主要独立软件供应商 (ISV) 的产品。红帽与 ISV 合作打包并提供。受 ISV 支持。 |
| **redhat-marketplace** | **registry.redhat.io/redhat/redhat-marketplace-index:v4.7** | 可通过 Red Hat Marketplace 购买认证的软件。 |
| **community-operators** | **registry.redhat.io/redhat/community-operator-index:v4.7** | 软件由 operator-framework/community-operators GitHub 仓库中相关方来维护。无官方支持。 |

## 2.7. CRD

### 2.7.1. 使用自定义资源定义来扩展 Kubernetes API

Operator 使用 Kubernetes 扩展机制（自定义资源定义（CRD）），以便使由 Operator 管理的自定义类似于内置的原生 Kubernetes 对象。本指南介绍了集群管理员如何通过创建和管理 CRD 来扩展其 OpenShift Container Platform 集群。

#### 2.7.1.1. 自定义资源定义

在 Kubernetes API 中，*resource（资源）*是存储某一类 API 对象集的端点。例如，内置 **Pod** 资源包含一组 **Pod** 对象。

*自定义资源定义*（CRD）对象在集群中定义一个新的、唯一的对象类型，称为 *kind*，并允许 Kubernetes API 服务器处理其整个生命周期。

*自定义资源* (CR) 对象由集群管理员通过集群中已添加的 CRD 创建，并支持所有集群用户在项目中增加新的资源类型。

当集群管理员增加新 CRD 至集群中时，Kubernetes API 服务器的回应方式是新建一个可由整个集群或单个项目（命名空间）访问的 RESTful 资源路径，并开始服务于指定的 CR。

集群管理员如果要向其他用户授予 CRD 访问权限，可使用集群角色聚合来向用户授予 **admin**、**edit** 或 **view** 默认集群角色访问权限。集群角色聚合支持将自定义策略规则插入到这些集群角色中。此行为将新资源整合到集群的 RBAC 策略中，就像内置资源一样。

Operator 会通过将 CRD 与任何所需 RBAC 策略和其他软件特定逻辑打包到一起来利用 CRD。集群管理员也可以手动将 CRD 添加到 Operator 生命周期之外的集群中，供所有用户使用。

**注意**

虽然只有集群管理员可创建 CRD，但具有 CRD 读写权限的开发人员也可通过现有 CRD 来创建 CR。

#### 2.7.1.2. 创建自定义资源定义

要创建自定义资源 (CR) 对象，集群管理员首先必须创建一个自定义资源定义 (CRD)。

**先决条件**

- 以 **cluster-admin** 用户身份访问 OpenShift Container Platform 集群。

**流程**

要创建 CRD：

1. 先创建一个包含以下字段类型的 YAML 文件：

**CRD 的 YAML 文件示例**

```
apiVersion: apiextensions.k8s.io/v1 1
kind: CustomResourceDefinition
metadata:
  name: crontabs.stable.example.com 2
spec:
  group: stable.example.com 3
  versions:
    name: v1 4
  scope: Namespaced 5
  names:
    plural: crontabs 6
    singular: crontab 7
    kind: CronTab 8
    shortNames:
    - ct 9
```

1. 使用 **apiextensions.k8s.io/v1** API。
2. 为定义指定名称。这必须采用 **<plural-name>.<group>** 格式，并使用来自 **group** 和 **plural** 字段的值。
3. 为 API 指定组名。API 组是一个逻辑上相关的对象集。例如，**Job** 或 **ScheduledJob** 等所有批处理对象，均可添加至批处理 API 组（如 **batch.api.example.com**）中。最好使用您机构的完全限定域名（FQDN）。
4. 指定 URL 中要用的版本名称。每个 API 组都可能存在于多个版本中，例如 **v1alpha**、**v1beta**、**v1**。
5. 指定自定义对象可用于某一个项目 (**Namespaced**) 还是集群中的所有项目 (**Cluster**)。
6. 指定 URL 中要用的复数名称。**plural** 字段与 API URL 网址中的资源相同。
7. 指定将在 CLI 上用作别名并用于显示的单数名称。

8 指定可创建的对象类型。类型可以采用 CamelCase。

9 指定与 CLI 中的资源相匹配的较短字符串。

**注意**

默认情况下，CRD 的覆盖范围为整个集群，适用于所有项目。

2. 创建 CRD 对象：

```
$ oc create -f <file_name>.yaml
```

在以下位置新建一个 RESTful API 端点：

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

例如，以下端点便是通过示例文件创建的：

```
/apis/stable.example.com/v1/namespaces/*/crontabs/...
```

现在，您即可使用该端点 URL 来创建和管理 CR。对象类型基于您创建的 CRD 对象的 **spec.kind** 字段。

#### 2.7.1.3. 为自定义资源定义创建集群角色

集群管理员可向现有集群范围的自定义资源定义 (CRD) 授予权限。如果使用 **admin**、**edit** 和 **view** 默认集群角色，请利用集群角色聚合来制定规则。

**重要**

您必须为每个角色明确分配权限。权限更多的角色不会继承权限较少角色的规则。如果要为某个角色分配规则，还必须将该操作动词分配给具有更多权限的角色。例如，如果要向 view 角色授予 **get crontabs** 的权限，也必须向 **edit** 和 **admin** 角色授予该权限。**admin** 或 **edit** 角色通常会分配给通过项目模板创建项目的用户。

**先决条件**

- 创建 CRD。

**流程**

1. 为 CRD 创建集群角色定义文件。集群角色定义是一个 YAML 文件，其中包含适用于各个集群角色的规则。OpenShift Container Platform 控制器会将您指定的规则添加至默认集群角色中。

**集群角色定义的 YAML 文件示例**

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1 1
metadata:
  name: aggregate-cron-tabs-admin-edit 2
  labels:
    rbac.authorization.k8s.io/aggregate-to-admin: "true" 3
```

```
    rbac.authorization.k8s.io/aggregate-to-edit: "true" 4
rules:
- apiGroups: ["stable.example.com"] 5
  resources: ["crontabs"] 6
  verbs: ["get", "list", "watch", "create", "update", "patch", "delete", "deletecollection"] 7
---
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: aggregate-cron-tabs-view 8
  labels:
    # Add these permissions to the "view" default role.
    rbac.authorization.k8s.io/aggregate-to-view: "true" 9
    rbac.authorization.k8s.io/aggregate-to-cluster-reader: "true" 10
rules:
- apiGroups: ["stable.example.com"] 11
  resources: ["crontabs"] 12
  verbs: ["get", "list", "watch"] 13
```

1. 使用 **rbac.authorization.k8s.io/v1** API。
2. 8 为定义指定名称。
3. 指定该标签向 admin 默认角色授予权限。
4. 指定该标签向 edit 默认角色授予权限。
5. 11 指定 CRD 的组名。
6. 12 指定适用于这些规则的 CRD 的复数名称。
7. 13 指定代表角色所获得的权限的操作动词。例如，对 **admin** 和 **edit** 角色应用读写权限，对 **view** 角色应用只读权限。
9. 指定该标签向 **view** 默认角色授予权限。
10. 指定该标签向 **cluster-reader** 默认角色授予权限。

2. 创建集群角色：

```
$ oc create -f <file_name>.yaml
```

#### 2.7.1.4. 通过文件创建自定义资源

将自定义资源定义 (CRD) 添加至集群后，可使用 CLI 按照自定义资源 (CR) 规范通过文件创建 CR。

**先决条件**

- 集群管理员已将 CRD 添加至集群中。

**流程**

1. 为 CR 创建 YAML 文件。在下面的定义示例中，**cronSpec** 和 **image** 自定义字段在 **Kind: CronTab** 的 CR 中设定。**Kind** 来自 CRD 对象的 **spec.kind** 字段：

   **CR 的 YAML 文件示例**

```
apiVersion: "stable.example.com/v1" 1
kind: CronTab 2
metadata:
  name: my-new-cron-object 3
  finalizers: 4
  - finalizer.stable.example.com
spec: 5
  cronSpec: "* * * * /5"
  image: my-awesome-cron-image
```

   1 指定 CRD 中的组名称和 API 版本（名称/版本）。

   2 指定 CRD 中的类型。

   3 指定对象的名称。

   4 指定对象的结束程序（如有）。结束程序可让控制器实现在删除对象之前必须完成的条件。

   5 指定特定于对象类型的条件。

2. 创建完文件后，再创建对象：

```
$ oc create -f <file_name>.yaml
```

#### 2.7.1.5. 检查自定义资源

您可使用 CLI 检查集群中存在的自定义资源 (CR) 对象。

**先决条件**

- 您有权访问的命名空间中已存在 CR 对象。

**流程**

1. 要获取特定类型的 CR 的信息，请运行：

```
$ oc get <kind>
```

   例如：

```
$ oc get crontab
```

   **输出示例**

```
NAME                 KIND
my-new-cron-object   CronTab.v1.stable.example.com
```

资源名称不区分大小写，您既可使用 CRD 中定义的单数或复数形式，也可使用简称。例如：

```
$ oc get crontabs
```

```
$ oc get crontab
```

```
$ oc get ct
```

2. 还可查看 CR 的原始 YAML 数据：

```
$ oc get <kind> -o yaml
```

例如：

```
$ oc get ct -o yaml
```

输**出示例**

```
apiVersion: v1
items:
- apiVersion: stable.example.com/v1
  kind: CronTab
  metadata:
    clusterName: ""
    creationTimestamp: 2017-05-31T12:56:35Z
    deletionGracePeriodSeconds: null
    deletionTimestamp: null
    name: my-new-cron-object
    namespace: default
    resourceVersion: "285"
    selfLink: /apis/stable.example.com/v1/namespaces/default/crontabs/my-new-cron-object
    uid: 9423255b-4600-11e7-af6a-28d2447dc82b
  spec:
    cronSpec: '* * * * /5' 1
    image: my-awesome-cron-image 2
```

1 2 显示用于创建对象的 YAML 的自定义数据。

### 2.7.2. 管理自定义资源定义中的资源

本指南向开发人员介绍了如何管理来自自定义资源定义 (CRD) 的自定义资源 (CR)。

#### 2.7.2.1. 自定义资源定义

在 Kubernetes API 中，*resource（资源）*是存储某一类 API 对象集的端点。例如，内置 **Pod** 资源包含一组 **Pod** 对象。

*自定义资源定义*（CRD）对象在集群中定义一个新的、唯一的对象类型，称为 *kind*，并允许 Kubernetes API 服务器处理其整个生命周期。

*自定义资源* (CR) 对象由集群管理员通过集群中已添加的 CRD 创建，并支持所有集群用户在项目中增加新的资源类型。

Operator 会通过将 CRD 与任何所需 RBAC 策略和其他软件特定逻辑打包到一起来利用 CRD。集群管理员也可以手动将 CRD 添加到 Operator 生命周期之外的集群中，供所有用户使用。

**注意**

虽然只有集群管理员可创建 CRD，但具有 CRD 读写权限的开发人员也可通过现有 CRD 来创建 CR。

#### 2.7.2.2. 通过文件创建自定义资源

将自定义资源定义 (CRD) 添加至集群后，可使用 CLI 按照自定义资源 (CR) 规范通过文件创建 CR。

**先决条件**

- 集群管理员已将 CRD 添加至集群中。

**流程**

1. 为 CR 创建 YAML 文件。在下面的定义示例中，**cronSpec** 和 **image** 自定义字段在 **Kind: CronTab** 的 CR 中设定。**Kind** 来自 CRD 对象的 **spec.kind** 字段：

   **CR 的 YAML 文件示例**

   ```
   apiVersion: "stable.example.com/v1" 1
   kind: CronTab 2
   metadata:
     name: my-new-cron-object 3
     finalizers: 4
     - finalizer.stable.example.com
   spec: 5
     cronSpec: "* * * * /5"
     image: my-awesome-cron-image
   ```

   1 指定 CRD 中的组名称和 API 版本（名称/版本）。

   2 指定 CRD 中的类型。

   3 指定对象的名称。

   4 指定对象的结束程序（如有）。结束程序可让控制器实现在删除对象之前必须完成的条件。

   5 指定特定于对象类型的条件。

2. 创建完文件后，再创建对象：

   ```
   $ oc create -f <file_name>.yaml
   ```

#### 2.7.2.3. 检查自定义资源

您可使用 CLI 检查集群中存在的自定义资源 (CR) 对象。

**先决条件**

- 您有权访问的命名空间中已存在 CR 对象。

**流程**

1. 要获取特定类型的 CR 的信息，请运行：

```
$ oc get <kind>
```

例如：

```
$ oc get crontab
```

输**出示例**

```
NAME                 KIND
my-new-cron-object   CronTab.v1.stable.example.com
```

资源名称不区分大小写，您既可使用 CRD 中定义的单数或复数形式，也可使用简称。例如：

```
$ oc get crontabs
```

```
$ oc get crontab
```

```
$ oc get ct
```

2. 还可查看 CR 的原始 YAML 数据：

```
$ oc get <kind> -o yaml
```

例如：

```
$ oc get ct -o yaml
```

输**出示例**

```
apiVersion: v1
items:
- apiVersion: stable.example.com/v1
  kind: CronTab
  metadata:
    clusterName: ""
    creationTimestamp: 2017-05-31T12:56:35Z
    deletionGracePeriodSeconds: null
    deletionTimestamp: null
    name: my-new-cron-object
    namespace: default
    resourceVersion: "285"
    selfLink: /apis/stable.example.com/v1/namespaces/default/crontabs/my-new-cron-object
    uid: 9423255b-4600-11e7-af6a-28d2447dc82b
```

```
spec:
  cronSpec: '* * * * /5' 1
  image: my-awesome-cron-image 2
```

1 2 显示用于创建对象的 YAML 的自定义数据。

# 第 3 章 用户任务

## 3.1. 从已安装的 OPERATOR 创建应用程序

本指南向开发人员介绍了如何使用 OpenShift Container Platform Web 控制台从已安装的 Operator 创建应用程序。

### 3.1.1. 使用 Operator 创建 etcd 集群

本流程介绍了如何通过由 Operator Lifecycle Manager (OLM) 管理的 etcd Operator 来新建一个 etcd 集群。

**先决条件**

- 访问 OpenShift Container Platform 4.7 集群
- 管理员已在集群范围内安装了 etcd Operator。

**流程**

1. 针对此流程在 OpenShift Container Platform Web 控制台中新建一个项目。这个示例使用名为 **my-etcd** 的项目。
2. 导航至 **Operators → Installed Operators** 页面。由集群管理员安装到集群且可供使用的 Operator 将以集群服务版本（CSV）列表形式显示在此处。CSV 用于启动和管理由 Operator 提供的软件。

   **提示**

   使用以下命令从 CLI 获得该列表：

   ```
   $ oc get csv
   ```

3. 在 **Installed Operators** 页面中，点 etcd Operator 查看更多详情和可用操作。
   正如 **Provided API** 下所示，该 Operator 提供了三类新资源，包括一种用于 **etcd Cluster** 的资源（**EtcdCluster** 资源）。这些对象的工作方式与内置的原生 Kubernetes 对象（如 **Deployment** 或 **ReplicaSet**）相似，但包含特定于管理 etcd 的逻辑。
4. 新建 etcd 集群：
   a. 在 **etcd Cluster** API 框中，点 **Create instance**。
   b. 在下一页上，您可对 **EtcdCluster** 对象的最小起始模板进行任何修改，比如集群大小。现在，点击 **Create** 即可完成。点击后即可触发 Operator 启动 pod、服务和新 etcd 集群的其他组件。
5. 点 **example** etcd 集群，然后点 **Resources** 选项卡，您可以看到项目现在包含很多由 Operator 自动创建和配置的资源。
   验证已创建了支持您从项目中的其他 pod 访问数据库的 Kubernetes 服务。
6. 给定项目中具有 **edit** 角色的所有用户均可创建、管理和删除应用程序实例（本例中为 etcd 集群），这些实例由已在项目中创建的 Operator 以自助方式管理，就像云服务一样。如果要赋予其他用户这一权利，项目管理员可使用以下命令添加角色：

```
$ oc policy add-role-to-user edit <user> -n <target_project>
```

现在您有了一个 etcd 集群，当 pod 运行不畅，或在集群中的节点之间迁移时，该集群将对故障做出反应并重新平衡数据。最重要的是，具有适当访问权限的集群管理员或开发人员现在可轻松将该数据库用于其应用程序。

## 3.2. 在命名空间中安装 OPERATOR

如果集群管理员将 Operator 安装权限委托给您的帐户，您可以以自助服务的方式将 Operator 安装并订阅到命名空间中。

### 3.2.1. 先决条件

- 集群管理员必须在 OpenShift Container Platform 用户帐户中添加某些权限，以便允许将自助服务 Operator 安装到命名空间。详情请参阅 允许非集群管理员安装 Operator。

### 3.2.2. 使用 OperatorHub 安装 operator

OperatorHub 是一个发现 Operator 的用户界面，它与 Operator Lifecycle Manager（OLM）一起工作，后者在集群中安装和管理 Operator。

作为具有适当权限的用户，您可以使用 OpenShift Container Platform Web 控制台或 CLI 安装来自 OperatorHub 的 Operator。

安装过程中，您必须为 Operator 确定以下初始设置：

**安装模式**

选择要在其中安装 Operator 的特定命名空间。

**更新频道**

如果某个 Operator 可通过多个频道获得，则可任选您想要订阅的频道。例如，要通过 **stable** 频道部署（如果可用），则从列表中选择这个选项。

**批准策略**

您可以选择自动或者手动更新。
如果选择自动更新某个已安装的 Operator，则当所选频道中有该 Operator 的新版本时，Operator Lifecycle Manager（OLM）将自动升级 Operator 的运行实例，而无需人为干预。

如果选择手动更新，则当有新版 Operator 可用时，OLM 会创建更新请求。作为集群管理员，您必须手动批准该更新请求，才可将 Operator 更新至新版本。

- 了解 OperatorHub

### 3.2.3. 使用 Web 控制台从 OperatorHub 安装

您可以使用 OpenShift Container Platform Web 控制台从 OperatorHub 安装并订阅 Operator。

**先决条件**

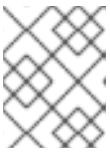
- 使用具有 Operator 安装权限的账户访问 OpenShift Container Platform 集群。

**流程**

1. 在 Web 控制台中导航至 **Operators → OperatorHub** 页面。

2. 找到您需要的 Operator（滚动页面会在 **Filter by keyword** 框中输入查找关键字）。例如，输入 **advanced** 来查找 Advanced Cluster Management for Kubernetes Operator。
   您还可以根据**基础架构功能**过滤选项。例如，如果您希望 Operator 在断开连接的环境中工作，请选择 **Disconnected**。

3. 选择要显示更多信息的 Operator。



**注意**

选择 Community Operator 会警告红帽没有认证社区 Operator ; 您必须确认该警告方可继续。

4. 阅读 Operator 信息并单击 **Install**。

5. 在 **Install Operator** 页面中：

   a. 选择要在其中安装 Operator 的特定单一命名空间。该 Operator 仅限在该单一命名空间中监视和使用。

   b. 选择一个**更新频道**（如有多个可用）。

   c. 如前面所述，选择**自动**或**手动**批准策略。

6. 点击 **Install** 使 Operator 可供 OpenShift Container Platform 集群上的所选命名空间使用。

   a. 如果选择了**手动**批准策略，订阅的升级状态将保持在 **Upgrading** 状态，直至您审核并批准安装计划。
   在 **Install Plan** 页面批准后，订阅的升级状态将变为 **Up to date**。

   b. 如果选择了 **Automatic** 批准策略，升级状态会在不用人工参与的情况下变为 **Up to date**。

7. 在订阅的升级状态成为 **Up to date** 后，选择 **Operators → Installed Operators** 来验证已安装 Operator 的 ClusterServiceVersion（CSV）是否最终出现了。**状态**最终会在相关命名空间中变为 **InstallSucceeded**。

**注意**

对于 **All namespaces...** 安装模式，状态在 **openshift-operators** 命名空间中解析为 **InstallSucceeded**，但如果检查其他命名空间，则状态为 **Copied**。

如果没有：

   a. 检查 **openshift-operators** 项目（如果选择了 **A specific namespace...** 安装模式）中的 openshift-operators 项目中的 pod 的日志，这会在 **Workloads → Pods** 页面中报告问题以便进一步排除故障。

### 3.2.4. 使用 CLI 从 OperatorHub 安装

您可以使用 CLI 从 OperatorHub 安装 Operator，而不必使用 OpenShift Container Platform Web 控制台。使用 **oc** 命令来创建或更新一个**订阅**对象。

**先决条件**

- 使用具有 Operator 安装权限的账户访问 OpenShift Container Platform 集群。
- 在您的本地系统安装 **oc** 命令。

**流程**

1. 查看 OperatorHub 中集群可用的 Operator 列表：

```
$ oc get packagemanifests -n openshift-marketplace
```

输**出示例**

```
NAME                              CATALOG               AGE
3scale-operator                   Red Hat Operators     91m
advanced-cluster-management       Red Hat Operators     91m
amq7-cert-manager                 Red Hat Operators     91m
...
couchbase-enterprise-certified    Certified Operators   91m
crunchy-postgres-operator         Certified Operators   91m
mongodb-enterprise                Certified Operators   91m
...
etcd                              Community Operators   91m
jaeger                            Community Operators   91m
kubefed                           Community Operators   91m
...
```

记录下所需 Operator 的目录。

2. 检查所需 Operator，以验证其支持的安装模式和可用频道：

```
$ oc describe packagemanifests <operator_name> -n openshift-marketplace
```

3. 一个 Operator 组（由 **OperatorGroup** 对象定义），在其中选择目标命名空间，在其中为与 Operator 组相同的命名空间中的所有 Operator 生成所需的 RBAC 访问权限。
订阅 Operator 的命名空间必须具有与 Operator 的安装模式相匹配的 Operator 组，可采用 **AllNamespaces** 模式，也可采用 **SingleNamespace** 模式。如果您要使用 **AllNamespaces** 安装 Operator，则 **openshift-operators** 命名空间已有适当的 Operator 组。

   如果要安装的 Operator 采用 **SingleNamespace** 模式，而您没有适当的 Operator 组，则必须创建一个。

**注意**

在选择 **SingleNamespace** 模式时，该流程的 Web 控制台版本会在后台自动为您处理 **OperatorGroup** 和 **Subscription** 对象的创建。

   a. 创建 **OperatorGroup** 对象 YAML 文件，如 **operatorgroup.yaml**：

**OperatorGroup** 对**象示例**

```
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
```

```
    name: <operatorgroup_name>
    namespace: <namespace>
  spec:
    targetNamespaces:
    - <namespace>
```

b. 创建 **OperatorGroup** 对象：

```
$ oc apply -f operatorgroup.yaml
```

4. 创建一个 **Subscription** 对象 YAML 文件，以便为 Operator 订阅一个命名空间，如 **sub.yaml**：

**Subscription 对象示例**

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: <subscription_name>
  namespace: openshift-operators (1)
spec:
  channel: <channel_name> (2)
  name: <operator_name> (3)
  source: redhat-operators (4)
  sourceNamespace: openshift-marketplace (5)
  config:
    env: (6)
    - name: ARGS
      value: "-v=10"
    envFrom: (7)
    - secretRef:
        name: license-secret
    volumes: (8)
    - name: <volume_name>
      configMap:
        name: <configmap_name>
    volumeMounts: (9)
    - mountPath: <directory_name>
      name: <volume_name>
    tolerations: (10)
    - operator: "Exists"
    resources: (11)
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
    nodeSelector: (12)
      foo: bar
```

(1) 对于 **AllNamespaces** 安装模式的使用，指定 **openshift-operators** 命名空间。否则，为 **SingleNamespace** 安装模式使用指定相关单一命名空间。

2. 要订阅的频道的名称。
3. 要订阅的 Operator 的名称。
4. 提供 Operator 的目录源的名称。
5. 目录源的命名空间。将 **openshift-marketplace** 用于默认的 OperatorHub 目录源。
6. **env** 参数定义必须存在于由 OLM 创建的 pod 中所有容器中的环境变量列表。
7. **envFrom** 参数定义要在容器中填充环境变量的源列表。
8. **volumes** 参数定义 OLM 创建的 pod 上必须存在的卷列表。
9. **volumeMounts** 参数定义由 OLM 创建的 pod 中必须存在的 VolumeMounts 列表。如果 **volumeMount** 引用不存在的 **卷**，OLM 无法部署 Operator。
10. **tolerations** 参数为 OLM 创建的 pod 定义 Tolerations 列表。
11. **resources** 参数为 OLM 创建的 pod 中所有容器定义资源限制。
12. **nodeSelector** 参数为 OLM 创建的 pod 定义 **NodeSelector**。

5. 创建 **Subscription** 对象：

```
$ oc apply -f sub.yaml
```

此时，OLM 已了解所选的 Operator。Operator 的集群服务版本（CSV）应出现在目标命名空间中，由 Operator 提供的 API 应可用于创建。

**其它资源**

- operator 组
- 频道名称

### 3.2.5. 安装 Operator 的特定版本

您可以通过在 **Subscription** 对象中设置集群服务版本（CSV）来安装 Operator 的特定版本。

**先决条件**

- 使用具有 Operator 安装权限的账户访问 OpenShift Container Platform 集群
- 已安装 OpenShift CLI（**oc**）

**流程**

1. 通过设置 **startingCSV** 字段，创建一个 **Subscription** 对象 YAML 文件，向带有特定版本的 Operator 订阅一个命名空间。将 **installPlanApproval** 字段设置为 **Manual**，以便在目录中存在更新的版本时防止 Operator 自动升级。
例如，可以使用以下 **sub.yaml** 文件安装 Red Hat Quay Operator，专门用于版本 3.4.0：

带**有特定起始 Operator 版本的**订阅

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: quay-operator
  namespace: quay
spec:
  channel: quay-v3.4
  installPlanApproval: Manual (1)
  name: quay-operator
  source: redhat-operators
  sourceNamespace: openshift-marketplace
  startingCSV: quay-operator.v3.4.0 (2)
```

1. 如果您指定的版本会被目录中的更新版本取代，则将批准策略设置为 **Manual**。此计划阻止自动升级到更新的版本，且需要在启动 CSV 可以完成安装前手动批准。
2. 设置 Operator CSV 的特定版本。

2. 创建 **Subscription** 对象：

```
$ oc apply -f sub.yaml
```

3. 手动批准待处理的安装计划以完成 Operator 安装。

**其它资源**

- 手动批准待处理的 Operator 升级

# 第 4 章 管理员任务

## 4.1. 在集群中添加 OPERATOR

集群管理员可以通过 OperatorHub 将 Operator 订阅到命名空间来将 Operator 安装到 OpenShift Container Platform 集群。

### 4.1.1. 使用 OperatorHub 安装 operator

OperatorHub 是一个发现 Operator 的用户界面，它与 Operator Lifecycle Manager（OLM）一起工作，后者在集群中安装和管理 Operator。

作为具有适当权限的用户，您可以使用 OpenShift Container Platform Web 控制台或 CLI 安装来自 OperatorHub 的 Operator。

安装过程中，您必须为 Operator 确定以下初始设置：

**安装模式**

选择要在其中安装 Operator 的特定命名空间。

**更新频道**

如果某个 Operator 可通过多个频道获得，则可任选您想要订阅的频道。例如，要通过 **stable** 频道部署（如果可用），则从列表中选择这个选项。

**批准策略**

您可以选择自动或者手动更新。
如果选择自动更新某个已安装的 Operator，则当所选频道中有该 Operator 的新版本时，Operator Lifecycle Manager（OLM）将自动升级 Operator 的运行实例，而无需人为干预。

如果选择手动更新，则当有新版 Operator 可用时，OLM 会创建更新请求。作为集群管理员，您必须手动批准该更新请求，才可将 Operator 更新至新版本。

- 了解 OperatorHub

### 4.1.2. 使用 Web 控制台从 OperatorHub 安装

您可以使用 OpenShift Container Platform Web 控制台从 OperatorHub 安装并订阅 Operator。

**先决条件**

- 使用具有 **cluster-admin** 权限的账户访问 OpenShift Container Platform 集群。
- 使用具有 Operator 安装权限的账户访问 OpenShift Container Platform 集群。

**流程**

1. 在 Web 控制台中导航至 **Operators → OperatorHub** 页面。
2. 找到您需要的 Operator（滚动页面会在 **Filter by keyword** 框中输入查找关键字）。例如，输入 **advanced** 来查找 Advanced Cluster Management for Kubernetes Operator。
   您还可以根据**基础架构功能**过滤选项。例如，如果您希望 Operator 在断开连接的环境中工作，请选择 **Disconnected**。
3. 选择要显示更多信息的 Operator。

**注意**

选择 Community Operator 会警告红帽没有认证社区 Operator ; 您必须确认该警告方可继续。

4. 阅读 Operator 信息并单击 **Install**。

5. 在 **Install Operator** 页面中：

   a. 任选以下一项：

      - **All namespaces on the cluster (default)** 选择该项会将 Operator 安装至默认 **openshift-operators** 命名空间，以便供集群中的所有命名空间监视和使用。该选项并非始终可用。

      - **A specific namespace on the cluster**，该项支持您选择单一特定命名空间来安装 Operator。该 Operator 仅限在该单一命名空间中监视和使用。

   b. 选择要在其中安装 Operator 的特定单一命名空间。该 Operator 仅限在该单一命名空间中监视和使用。

   c. 选择一个**更新频道**（如有多个可用）。

   d. 如前面所述，选择**自动**或**手动**批准策略。

6. 点击 **Install** 使 Operator 可供 OpenShift Container Platform 集群上的所选命名空间使用。

   a. 如果选择了**手动**批准策略，订阅的升级状态将保持在 **Upgrading** 状态，直至您审核并批准安装计划。
   在 **Install Plan** 页面批准后，订阅的升级状态将变为 **Up to date**。

   b. 如果选择了 **Automatic** 批准策略，升级状态会在不用人工参与的情况下变为 **Up to date**。

7. 在订阅的升级状态成为 **Up to date** 后，选择 **Operators → Installed Operators** 来验证已安装 Operator 的 ClusterServiceVersion（CSV）是否最终出现了。**状态**最终会在相关命名空间中变为 **InstallSucceeded**。

**注意**

对于 **All namespaces...** 安装模式，状态在 **openshift-operators** 命名空间中解析为 **InstallSucceeded**，但如果检查其他命名空间，则状态为 **Copied**。

如果没有：

   a. 检查 **openshift-operators** 项目（如果选择了 **A specific namespace...** 安装模式）中的 openshift-operators 项目中的 pod 的日志，这会在 **Workloads → Pods** 页面中报告问题以便进一步排除故障。

### 4.1.3. 使用 CLI 从 OperatorHub 安装

您可以使用 CLI 从 OperatorHub 安装 Operator，而不必使用 OpenShift Container Platform Web 控制台。使用 **oc** 命令来创建或更新一个**订阅**对象。

**先决条件**

- 使用具有 Operator 安装权限的账户访问 OpenShift Container Platform 集群。

- 在您的本地系统安装 **oc** 命令。

**流程**

1. 查看 OperatorHub 中集群可用的 Operator 列表：

```
$ oc get packagemanifests -n openshift-marketplace
```

输**出示例**

```
NAME                              CATALOG               AGE
3scale-operator                   Red Hat Operators     91m
advanced-cluster-management       Red Hat Operators     91m
amq7-cert-manager                 Red Hat Operators     91m
...
couchbase-enterprise-certified    Certified Operators   91m
crunchy-postgres-operator         Certified Operators   91m
mongodb-enterprise                Certified Operators   91m
...
etcd                              Community Operators   91m
jaeger                            Community Operators   91m
kubefed                           Community Operators   91m
...
```

记录下所需 Operator 的目录。

2. 检查所需 Operator，以验证其支持的安装模式和可用频道：

```
$ oc describe packagemanifests <operator_name> -n openshift-marketplace
```

3. 一个 Operator 组（由 **OperatorGroup** 对象定义），在其中选择目标命名空间，在其中为与 Operator 组相同的命名空间中的所有 Operator 生成所需的 RBAC 访问权限。
订阅 Operator 的命名空间必须具有与 Operator 的安装模式相匹配的 Operator 组，可采用 **AllNamespaces** 模式，也可采用 **SingleNamespace** 模式。如果您要使用 **AllNamespaces** 安装 Operator，则 **openshift-operators** 命名空间已有适当的 Operator 组。

   如果要安装的 Operator 采用 **SingleNamespace** 模式，而您没有适当的 Operator 组，则必须创建一个。

**注意**

在选择 **SingleNamespace** 模式时，该流程的 Web 控制台版本会在后台自动为您处理 **OperatorGroup** 和 **Subscription** 对象的创建。

   a. 创建 **OperatorGroup** 对象 YAML 文件，如 **operatorgroup.yaml**：

**OperatorGroup** 对**象示例**

```
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: <operatorgroup_name>
  namespace: <namespace>
```

```
spec:
  targetNamespaces:
  - <namespace>
```

b. 创建 **OperatorGroup** 对象：

```
$ oc apply -f operatorgroup.yaml
```

4. 创建一个 **Subscription** 对象 YAML 文件，以便为 Operator 订阅一个命名空间，如 **sub.yaml**：

**Subscription 对象示例**

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: <subscription_name>
  namespace: openshift-operators (1)
spec:
  channel: <channel_name> (2)
  name: <operator_name> (3)
  source: redhat-operators (4)
  sourceNamespace: openshift-marketplace (5)
  config:
    env: (6)
    - name: ARGS
      value: "-v=10"
    envFrom: (7)
    - secretRef:
        name: license-secret
    volumes: (8)
    - name: <volume_name>
      configMap:
        name: <configmap_name>
    volumeMounts: (9)
    - mountPath: <directory_name>
      name: <volume_name>
    tolerations: (10)
    - operator: "Exists"
    resources: (11)
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
    nodeSelector: (12)
      foo: bar
```

(1) 对于 **AllNamespaces** 安装模式的使用，指定 **openshift-operators** 命名空间。否则，为 **SingleNamespace** 安装模式使用指定相关单一命名空间。

(2) 要订阅的频道的名称。

3. 要订阅的 Operator 的名称。
4. 提供 Operator 的目录源的名称。
5. 目录源的命名空间。将 **openshift-marketplace** 用于默认的 OperatorHub 目录源。
6. **env** 参数定义必须存在于由 OLM 创建的 pod 中所有容器中的环境变量列表。
7. **envFrom** 参数定义要在容器中填充环境变量的源列表。
8. **volumes** 参数定义 OLM 创建的 pod 上必须存在的卷列表。
9. **volumeMounts** 参数定义由 OLM 创建的 pod 中必须存在的 VolumeMounts 列表。如果 **volumeMount** 引用不存在的 **卷**，OLM 无法部署 Operator。
10. **tolerations** 参数为 OLM 创建的 pod 定义 Tolerations 列表。
11. **resources** 参数为 OLM 创建的 pod 中所有容器定义资源限制。
12. **nodeSelector** 参数为 OLM 创建的 pod 定义 **NodeSelector**。

5. 创建 **Subscription** 对象：

```
$ oc apply -f sub.yaml
```

此时，OLM 已了解所选的 Operator。Operator 的集群服务版本（CSV）应出现在目标命名空间中，由 Operator 提供的 API 应可用于创建。

**其他资源**

- 关于 Operator 组

### 4.1.4. 安装 Operator 的特定版本

您可以通过在 **Subscription** 对象中设置集群服务版本（CSV）来安装 Operator 的特定版本。

**先决条件**

- 使用具有 Operator 安装权限的账户访问 OpenShift Container Platform 集群
- 已安装 OpenShift CLI（**oc**）

**流程**

1. 通过设置 **startingCSV** 字段，创建一个 **Subscription** 对象 YAML 文件，向带有特定版本的 Operator 订阅一个命名空间。将 **installPlanApproval** 字段设置为 **Manual**，以便在目录中存在更新的版本时防止 Operator 自动升级。
例如，可以使用以下 **sub.yaml** 文件安装 Red Hat Quay Operator，专门用于版本 3.4.0：

带**有特定起始 Operator 版本的**订阅

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
```

```
  name: quay-operator
  namespace: quay
spec:
  channel: quay-v3.4
  installPlanApproval: Manual (1)
  name: quay-operator
  source: redhat-operators
  sourceNamespace: openshift-marketplace
  startingCSV: quay-operator.v3.4.0 (2)
```

1. 如果您指定的版本会被目录中的更新版本取代，则将批准策略设置为 **Manual**。此计划阻止自动升级到更新的版本，且需要在启动 CSV 可以完成安装前手动批准。
2. 设置 Operator CSV 的特定版本。

2. 创建 **Subscription** 对象：

```
$ oc apply -f sub.yaml
```

3. 手动批准待处理的安装计划以完成 Operator 安装。

**其它资源**

- 手动批准待处理的 Operator 升级

### 4.1.5. Operator 工作负载的 Pod 放置

默认情况下，Operator Lifecycle Manager（OLM）在安装 Operator 或部署 Operand 工作负载时，会将 pod 放置到任意 worker 节点上。作为管理员，您可以使用节点选择器、污点和容限组合使用项目来控制将 Operator 和 Operands 放置到特定节点。

控制 Operator 和 Operand 工作负载的 pod 放置有以下先决条件：

1. 根据您的要求，确定 pod 的目标节点或一组节点。如果可用，请注意现有标签，如 **node-role.kubernetes.io/app**，用于标识节点。否则，使用机器集或直接编辑节点来添加标签，如 **myoperator**。您将在以后的步骤中使用此标签作为项目上的节点选择器。

2. 如果要确保只有具有特定标签的 pod 才能在节点上运行，同时将不相关的工作负载加载到其他节点，通过使用机器集或直接编辑节点为节点添加污点。使用一个效果来确保与污点不匹配的新 pod 不能调度到节点上。例如，**myoperator:NoSchedule** 污点确保与污点不匹配的新 pod 不能调度到该节点上，但节点上现有的 pod 可以保留。

3. 创建使用默认节点选择器配置的项目，如果您添加了污点，则创建一个匹配的容限。

此时，您创建的项目可在以下情况下用于将 pod 定向到指定节点：

**对于 Operator pod**

管理员可以在项目中创建 **Subscription** 对象。因此，Operator pod 放置在指定的节点上。

**对于 Operand pod**

通过使用已安装的 Operator，用户可以在项目中创建一个应用程序，这样可将 Operator 拥有的自定义资源（CR）放置到项目中。因此，Operand pod 放置到指定节点上，除非 Operator 在其他命名空间中部署集群范围对象或资源，在这种情况下，不会应用这个自定义的 pod 放置。

**其它资源**

- 手动向节点或机器集添加污点和容限
- 创建项目范围节点选择器
- 使用节点选择器和容限创建项目

## 4.2. 升级安装的 OPERATOR

作为集群管理员，您可以升级以前使用 OpenShift Container Platform 集群上的 Operator Lifecycle Manager（OLM）安装的 Operator。

### 4.2.1. 更改 Operator 的更新频道

已安装的 Operator 的订阅指定一个更新频道，用于跟踪和接收 Operator 的更新。要升级 Operator 以开始跟踪并从更新频道接受更新，您可以更改订阅中的更新频道。

订阅中更新频道的名称可能会因 Operator 而异，但应遵守给定 Operator 中的常规约定。例如，频道名称可能会遵循 Operator 提供的应用程序的次发行版本更新流（**1.2**、**1.3**）或发行的频率（**stable**、**fast**）。

**注意**

安装的 Operator 无法变为比当前频道旧的频道。

如果订阅中的批准策略被设置为 **Automatic**，则升级过程会在所选频道中提供新的 Operator 版本时立即启动。如果批准策略设为 **Manual**，则必须手动批准待处理的升级。

**先决条件**

- 之前使用 Operator Lifecycle Manager（OLM）安装的 Operator。

**流程**

1. 在 OpenShift Container Platform Web 控制台的 **Administrator** 视角中，进入 **Operators → Installed Operators**。
2. 点击您要更改更新频道的 Operator 名称。
3. 点 **Subscription** 标签页。
4. 点 **Channel** 中的更新频道的名称。
5. 点要更改的更新频道，然后点 **Save**。
6. 对于带有 **自动批准策略** 的订阅，升级会自动开始。返回 **Operators → Installed Operators** 页面来监控升级的过程。完成后，状态会变为 **Succeeded** 和 **Up to date**。
   对于采用**手动批准策略**的订阅，您可以使用 **Subscription** 选项卡手动批准升级。

### 4.2.2. 手动批准待处理的 Operator 升级

如果已安装的 Operator 的订阅被设置为 **Manual**，则当其当前更新频道中发布新更新时，在开始安装前必须手动批准更新。

**先决条件**

- 之前使用 Operator Lifecycle Manager（OLM）安装的 Operator。

**流程**

1. 在 OpenShift Container Platform Web 控制台的 **Administrator** 视角中，进入 **Operators → Installed Operators**。
2. 处于待定升级的 Operator 会显示 **Upgrade available** 状态。点您要升级的 Operator 的名称。
3. 点 **Subscription** 标签页。任何需要批准的升级都会在 **Upgrade Status** 旁边显示。例如：它可能会显示 **1 requires approval**。
4. 点 **1 requires approval**，然后点 **Preview Install Plan**。
5. 查看列出可用于升级的资源。在满意后，点 **Approve**。
6. 返回 **Operators → Installed Operators** 页面来监控升级的过程。完成后，状态会变为 **Succeeded** 和 **Up to date**。

## 4.3. 从集群中删除 OPERATOR

下面介绍如何删除以前使用 OpenShift Container Platform 集群上的 Operator Lifecycle Manager（OLM）安装的 Operator。

### 4.3.1. 使用 Web 控制台从集群中删除 Operator

集群管理员可以使用 Web 控制台从所选命名空间中删除已安装的 Operator。

**先决条件**

- 使用具有 **cluster-admin** 权限的账户访问 OpenShift Container Platform 集群 Web 控制台。

**流程**

1. 进入 **Operators → Installed Operators** 页面，在 **Filter by name** 字段滚动鼠标或键入关键词，以查找您想要的 Operator。然后点击。
2. 在 **Operator Details** 页面右侧，从 **Actions** 列表中选择 **Uninstall Operator**。此时会显示 **Uninstall Operator?** 对话框，提醒您：

   **删除 Operator 不会移除任何自定义资源定义或受管资源。如果 Operator 在集群中部署了应用程序，或者配置了非集群资源，则这些应用程序将继续运行，需要手动清理。**

   此操作将删除 Operator 以及 Operator 部署和 pod（若有）。任何 Operands 和由 Operator 管理的资源（包括 CRD 和 CR）都不会被删除。Web 控制台为一些 Operator 启用仪表板和导航项。要在卸载 Operator 后删除这些，您可能需要手动删除 Operator CRD。
3. 选择 **Uninstall**。此 Operator 将停止运行，并且不再接收更新。

### 4.3.2. 使用 CLI 从集群中删除 Operator

集群管理员可以使用 CLI 从所选命名空间中删除已安装的 Operator。

**先决条件**

- 使用具有 **cluster-admin** 权限的账户访问 OpenShift Container Platform 集群。
- 已在工作站上安装 **oc** 命令。

**流程**

1. 通过 **currentCSV** 字段检查已订阅 Operator 的当前版本（如 **jaeger**）：

```
$ oc get subscription jaeger -n openshift-operators -o yaml | grep currentCSV
```

输**出示例**

```
  currentCSV: jaeger-operator.v1.8.2
```

2. 删除订阅（如 **jaeger**）：

```
$ oc delete subscription jaeger -n openshift-operators
```

输**出示例**

```
subscription.operators.coreos.com "jaeger" deleted
```

3. 使用上一步中的 **currentCSV** 值来删除目标命名空间中相应 Operator 的 CSV：

```
$ oc delete clusterserviceversion jaeger-operator.v1.8.2 -n openshift-operators
```

输**出示例**

```
clusterserviceversion.operators.coreos.com "jaeger-operator.v1.8.2" deleted
```

### 4.3.3. 刷新失败的订阅

在 Operator Lifecycle Manager（OLM）中，如果您订阅的是引用网络中无法访问的镜像的 Operator，您可以在 **openshift-marketplace** 命名空间中找到带有以下错误的作业：

输**出示例**

```
ImagePullBackOff for
Back-off pulling image "example.com/openshift4/ose-elasticsearch-operator-
bundle@sha256:6d2587129c846ec28d384540322b40b05833e7e00b25cca584e004af9a1d292e"
```

输**出示例**

```
rpc error: code = Unknown desc = error pinging docker registry example.com: Get
"https://example.com/v2/": dial tcp: lookup example.com on 10.0.0.1:53: no such host
```

因此，订阅会处于这个失败状态，Operator 无法安装或升级。

您可以通过删除订阅、集群服务版本（CSV）及其他相关对象来刷新失败的订阅。重新创建订阅后，OLM 会重新安装 Operator 的正确版本。

**先决条件**

- 您有一个失败的订阅，无法拉取不能访问的捆绑包镜像。
- 已确认可以访问正确的捆绑包镜像。

**流程**

1. 从安装 Operator 的命名空间中获取 **Subscription** 和 **ClusterServiceVersion** 对象的名称：

```
$ oc get sub,csv -n <namespace>
```

输**出示例**

```
NAME                                         PACKAGE                SOURCE                CHANNEL
subscription.operators.coreos.com/elasticsearch-operator   elasticsearch-operator   redhat-operators   5.0

NAME                                                  DISPLAY                            VERSION    REPLACES   PHASE
clusterserviceversion.operators.coreos.com/elasticsearch-operator.5.0.0-65   OpenShift Elasticsearch Operator   5.0.0-65              Succeeded
```

2. 删除订阅：

```
$ oc delete subscription <subscription_name> -n <namespace>
```

3. 删除集群服务版本：

```
$ oc delete csv <csv_name> -n <namespace>
```

4. 在 **openshift-marketplace** 命名空间中获取所有失败的作业的名称和相关配置映射：

```
$ oc get job,configmap -n openshift-marketplace
```

输**出示例**

```
NAME                                                                COMPLETIONS   DURATION   AGE
job.batch/1de9443b6324e629ddf31fed0a853a121275806170e34c926d69e53a7fcbccb   1/1           26s        9m30s

NAME                                                                DATA   AGE
configmap/1de9443b6324e629ddf31fed0a853a121275806170e34c926d69e53a7fcbccb   3      9m30s
```

5. 删除作业：

```
$ oc delete job <job_name> -n openshift-marketplace
```

这样可确保尝试拉取无法访问的镜像的 Pod 不会被重新创建。

6. 删除配置映射：

```
$ oc delete configmap <configmap_name> -n openshift-marketplace
```

7. 在 Web 控制台中使用 OperatorHub 重新安装 Operator。

**验证**

- 检查是否已成功重新安装 Operator:

```
$ oc get sub,csv,installplan -n <namespace>
```

## 4.4. 在 OPERATOR LIFECYCLE MANAGER 中配置代理支持

如果在 OpenShift Container Platform 集群中配置了全局代理，Operator Lifecycle Manager（OLM）会自动配置使用集群范围代理管理的 Operator。但是，您也可以配置已安装的 Operator 来覆盖全局代理服务器或注入自定义 CA 证书。

**其他资源**

- 配置集群范围代理
- 配置自定义 PKI （自定义 CA 证书）

### 4.4.1. 覆盖 Operator 的代理设置

如果配置了集群范围的出口代理，使用 Operator Lifecycle Manager（OLM）运行的 Operator 会继承其部署上的集群范围代理设置。集群管理员还可以通过配置 Operator 的订阅来覆盖这些代理设置。

**重要**

操作员必须为任何受管 Operands 处理 pod 中的代理设置环境变量。

**先决条件**

- 使用具有 **cluster-admin** 权限的账户访问 OpenShift Container Platform 集群。

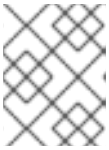
**流程**

1. 在 Web 控制台中导航至 **Operators → OperatorHub** 页面。
2. 选择 Operator 并点 **Install**。
3. 在 **Install Operator** 页面中，修改 **Subscription** 对象，使其在 **spec** 部分中包含一个或多个以下环境变量：
   - **HTTP_PROXY**
   - **HTTPS_PROXY**
   - **NO_PROXY**

   例如：

带有代理设置的Subscription 对象覆盖

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: etcd-config-test
  namespace: openshift-operators
spec:
  config:
    env:
    - name: HTTP_PROXY
      value: test_http
    - name: HTTPS_PROXY
      value: test_https
    - name: NO_PROXY
      value: test
  channel: clusterwide-alpha
  installPlanApproval: Automatic
  name: etcd
  source: community-operators
  sourceNamespace: openshift-marketplace
  startingCSV: etcdoperator.v0.9.4-clusterwide
```



**注意**

这些环境变量也可以使用空值取消设置，以删除所有之前设定的集群范围或自定义代理设置。

OLM 将这些环境变量作为一个单元处理; 如果至少设置了一个环境变量，则所有 3 个变量都将被视为覆盖，并且集群范围的默认值不会用于订阅的 Operator 部署。

4. 点击 **Install** 使 Operator 可供所选命名空间使用。

5. 当 Operator 的 CSV 出现在相关命名空间中后，您可以验证部署中是否设置了自定义代理环境变量。例如，使用 CLI：

```
$ oc get deployment -n openshift-operators \
    etcd-operator -o yaml \
    | grep -i "PROXY" -A 2
```

输出示例

```
        - name: HTTP_PROXY
          value: test_http
        - name: HTTPS_PROXY
          value: test_https
        - name: NO_PROXY
          value: test
        image: quay.io/coreos/etcd-
operator@sha256:66a37fd61a06a43969854ee6d3e21088a98b93838e284a6086b13917f96b0
d9c
...
```

### 4.4.2. 注入自定义 CA 证书

当集群管理员使用配置映射向集群添加自定义 CA 证书时，Cluster Network Operator 会将用户提供的证书和系统 CA 证书合并为一个捆绑包（bundle）。您可以将这个合并捆绑包注入 Operator Lifecycle Manager (OLM) 上运行的 Operator 中，如果您有一个中间人（man-in-the-middle）HTTPS 代理，这将会有用。

**先决条件**

- 使用具有 **cluster-admin** 权限的账户访问 OpenShift Container Platform 集群。
- 使用配置映射添加自定义 CA 证书至集群。
- 在 OLM 上安装并运行所需的 Operator。

**流程**

1. 在存在 Operator 订阅的命名空间中创建一个空配置映射，并包括以下标签：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: trusted-ca 1
  labels:
    config.openshift.io/inject-trusted-cabundle: "true" 2
```

1 配置映射的名称。

2 请求 Cluster Network Operator 注入合并的捆绑包。

创建此配置映射后，它会立即使用合并捆绑包的证书内容填充。

2. 更新您的 **Subscription** 对象，使其包含 **spec.config** 部分，该部分可将 **trusted-ca** 配置映射作为卷挂载到需要自定义 CA 的 pod 中的每个容器：

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: my-operator
spec:
  package: etcd
  channel: alpha
  config: 1
    selector:
      matchLabels:
        <labels_for_pods> 2
    volumes: 3
    - name: trusted-ca
      configMap:
        name: trusted-ca
        items:
          - key: ca-bundle.crt 4
            path: tls-ca-bundle.pem 5
    volumeMounts: 6
```

```
- name: trusted-ca
  mountPath: /etc/pki/ca-trust/extracted/pem
  readOnly: true
```

1. 如果不存在，请添加 **config** 部分。
2. 指定标签以匹配 Operator 拥有的 pod。
3. 创建一个 **trusted-ca** 卷。
4. **ca-bundle.crt** 需要作为配置映射键。
5. **tls-ca-bundle.pem** 需要作为配置映射路径。
6. 创建一个 **trusted-ca** 卷挂载。

## 4.5. 查看 OPERATOR 状态

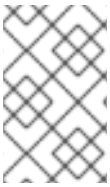
了解 Operator Lifecycle Manager (OLM) 中的系统状态，对于决定和调试已安装 Operator 的问题来说非常重要。OLM 可让您了解订阅和相关目录源的状态以及执行的操作。这样有助于用户更好地理解 Operator 的运行状况。

### 4.5.1. operator 订阅状况类型

订阅可报告以下状况类型：

**表 4.1. 订阅状况类型**

| 状况 | 描述 |
| --- | --- |
| **CatalogSourcesUnhealthy** | 用于解析的一个或多个目录源不健康。 |
| **InstallPlanMissing** | 缺少订阅的安装计划。 |
| **InstallPlanPending** | 订阅的安装计划正在安装中。 |
| **InstallPlanFailed** | 订阅的安装计划失败。 |



**注意**

默认 OpenShift Container Platform 集群 Operator 由 Cluster Version Operator（CVO）管理，它们没有 **Subscription** 对象。应用程序 Operator 由 Operator Lifecycle Manager（OLM）管理，它们具有 **Subscription** 对象。

**其它资源**

- 刷新失败的订阅

### 4.5.2. 使用 CLI 查看 Operator 订阅状态

您可以使用 CLI 查看 Operator 订阅状态。

**先决条件**

- 您可以使用具有 **cluster-admin** 角色的用户访问集群。
- 已安装 OpenShift CLI(**oc**)。

**流程**

1. 列出 Operator 订阅：

```
$ oc get subs -n <operator_namespace>
```

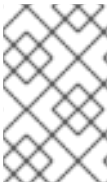
2. 使用 **oc describe** 命令检查 **Subscription** 资源：

```
$ oc describe sub <subscription_name> -n <operator_namespace>
```

3. 在命令输出中，找到 Operator 订阅状况类型的 **Conditions** 部分。在以下示例中，**CatalogSourcesUnhealthy** 条件类型具有 **false** 状态，因为所有可用目录源都健康：

**输出示例**

```
Conditions:
  Last Transition Time:  2019-07-29T13:42:57Z
  Message:               all available catalogsources are healthy
  Reason:                AllCatalogSourcesHealthy
  Status:                False
  Type:                  CatalogSourcesUnhealthy
```



**注意**

默认 OpenShift Container Platform 集群 Operator 由 Cluster Version Operator（CVO）管理，它们没有 **Subscription** 对象。应用程序 Operator 由 Operator Lifecycle Manager（OLM）管理，它们具有 **Subscription** 对象。

### 4.5.3. 使用 CLI 查看 Operator 目录源状态

您可以使用 CLI 查看 Operator 目录源的状态。

**先决条件**

- 您可以使用具有 **cluster-admin** 角色的用户访问集群。
- 已安装 OpenShift CLI(**oc**)。

**流程**

1. 列出命名空间中的目录源。例如，您可以检查 **openshift-marketplace** 命名空间，该命名空间用于集群范围的目录源：

```
$ oc get catalogsources -n openshift-marketplace
```

**输出示例**

```
NAME                  DISPLAY               TYPE   PUBLISHER   AGE
certified-operators   Certified Operators   grpc   Red Hat     55m
community-operators   Community Operators   grpc   Red Hat     55m
example-catalog       Example Catalog       grpc   Example Org 2m25s
redhat-marketplace    Red Hat Marketplace   grpc   Red Hat     55m
redhat-operators      Red Hat Operators     grpc   Red Hat     55m
```

2. 使用 **oc describe** 命令获取有关目录源的详情和状态：

```
$ oc describe catalogsource example-catalog -n openshift-marketplace
```

输**出示例**

```
Name:         example-catalog
Namespace:    openshift-marketplace
...
Status:
  Connection State:
    Address:              example-catalog.openshift-marketplace.svc:50051
    Last Connect:         2021-09-09T17:07:35Z
    Last Observed State:  TRANSIENT_FAILURE
  Registry Service:
    Created At:         2021-09-09T17:05:45Z
    Port:               50051
    Protocol:           grpc
    Service Name:       example-catalog
    Service Namespace:  openshift-marketplace
```

在上例的输出中，最后观察到的状态是 **TRANSIENT_FAILURE**。此状态表示目录源建立连接时出现问题。

3. 列出创建目录源的命名空间中的 pod：

```
$ oc get pods -n openshift-marketplace
```

输**出示例**

```
NAME                                  READY   STATUS             RESTARTS   AGE
certified-operators-cv9nn             1/1     Running            0          36m
community-operators-6v8lp             1/1     Running            0          36m
marketplace-operator-86bfc75f9b-jkgbc 1/1     Running            0          42m
example-catalog-bwt8z                 0/1     ImagePullBackOff   0          3m55s
redhat-marketplace-57p8c              1/1     Running            0          36m
redhat-operators-smxx8                1/1     Running            0          36m
```

在命名空间中创建目录源时，会在该命名空间中为目录源创建一个 pod。在前面的示例中，**example-catalog-bwt8z** pod 的状态是 **ImagePullBackOff**。此状态表示拉取目录源的索引镜像存在问题。

4. 使用 **oc describe** 命令检查 pod 以获取更多详细信息：

```
$ oc describe pod example-catalog-bwt8z -n openshift-marketplace
```

输**出示例**

```
Name:         example-catalog-bwt8z
Namespace:    openshift-marketplace
Priority:     0
Node:         ci-ln-jyryyg2-f76d1-ggdbq-worker-b-vsxjd/10.0.128.2
...
Events:
  Type     Reason          Age                From               Message
  ----     ------          ----               ----               -------
  Normal   Scheduled       48s                default-scheduler  Successfully assigned openshift-marketplace/example-catalog-bwt8z to ci-ln-jyryyf2-f76d1-fgdbq-worker-b-vsxjd
  Normal   AddedInterface  47s                multus             Add eth0 [10.131.0.40/23] from openshift-sdn
  Normal   BackOff         20s (x2 over 46s)  kubelet            Back-off pulling image "quay.io/example-org/example-catalog:v1"
  Warning  Failed          20s (x2 over 46s)  kubelet            Error: ImagePullBackOff
  Normal   Pulling         8s (x3 over 47s)   kubelet            Pulling image "quay.io/example-org/example-catalog:v1"
  Warning  Failed          8s (x3 over 47s)   kubelet            Failed to pull image "quay.io/example-org/example-catalog:v1": rpc error: code = Unknown desc = reading manifest v1 in quay.io/example-org/example-catalog: unauthorized: access to the requested resource is not authorized
  Warning  Failed          8s (x3 over 47s)   kubelet            Error: ErrImagePull
```

在前面的示例输出中，错误消息表示目录源的索引镜像因为授权问题而无法成功拉取。例如，索引镜像可能存储在需要登录凭证的 registry 中。

**其它资源**

- Operator Lifecycle Manager 概念和资源 → Catalog 源
- gRPC 文档：连接状态
- 从私有 registry 访问 Operator 的镜像

## 4.6. 管理 OPERATOR 条件

作为集群管理员，您可以使用 Operator Lifecycle Manager（OLM）来管理 Operator 状况。

### 4.6.1. 覆盖 Operator 条件

作为集群管理员，您可能想要忽略由 Operator 报告的、支持的 Operator 条件。**Spec.Overrides** 阵列中的 Operator 条件会覆盖 **Status.Conditions** 阵列中的条件，以便集群管理员可以处理 Operator 向 Operator Lifecycle Manager（OLM）报告了不正确状态的情况。

例如，一个 Operator 的已知版本，它始终会告知它是不可升级的。在这种情况下，尽管报告是不可升级的，您仍然希望升级 Operator。这可以通过在 **OperatorCondition** 资源中的 **Spec.Overrides** 阵列中添加 **type** 和 **status** 来覆盖 Operator 条件来实现。

**先决条件**

- 具有 **OperatorCondition** 资源的 Operator，使用 OLM 安装。

**流程**

1. 编辑 Operator 的 **OperatorCondition** 资源：

```
$ oc edit operatorcondition <name>
```

2. 在对象中添加 **Spec.Overrides** 数组：

**Operator 条件覆盖示例**

```
apiVersion: operators.coreos.com/v1
kind: OperatorCondition
metadata:
  name: my-operator
  namespace: operators
spec:
  overrides:
  - type: Upgradeable (1)
    status: "True"
    reason: "upgradeIsSafe"
    message: "This is a known issue with the Operator where it always reports that it cannot be upgraded."
status:
  conditions:
  - type: Upgradeable
    status: "False"
    reason: "migration"
    message: "The operator is performing a migration."
    lastTransitionTime: "2020-08-24T23:15:55Z"
```

(1) 允许集群管理员将升级就绪状态更改为 **True**。

### 4.6.2. 更新 Operator 以使用 Operator 条件

Operator Lifecycle Manager（OLM）会自动为每个它所协调的 **ClusterServiceVersion** 资源创建一个 **OperatorCondition** 资源。CSV 中的所有服务帐户都会被授予 RBAC，以便与 Operator 拥有的 **OperatorCondition** 交互。

Operator 作者可开发其自己的 Operator 来使用 **operator-lib** 库，以便在由 OLM 部署 Operator 后，它可以设置自己的条件。如需有关编写逻辑以将 Operator 条件设置为 Operator 作者的更多信息，请参阅 Operator SDK 文档。

#### 4.6.2.1. 设置默认值

为了保持向后兼容，OLM 认为在没有 **OperatorCondition** 时代表不使用条件。因此，要使用 Operator 条件的 Operator，在将 pod 的就绪探测设置为 **true** 前应设置默认条件。这为 Operator 提供了一个宽限期，用于将条件更新为正确的状态。

### 4.6.3. 其他资源

- Operator 条件

## 4.7. 允许非集群管理员安装 OPERATOR

Operator 可能需要广泛权限才可运行，且不同版本需要的权限也可能不同。Operator Lifecycle Manager (OLM) 需要 **cluster-admin** 权限才可运行。默认情况下，Operator 作者可在集群服务版本（CSV）中指定任意权限集，OLM 之后会将其授予 Operator。

集群管理员应采取措施，确保 Operator 无法获得全集群权限，且用户不可使用 OLM 来提升权限。要实现这一目的的一种方法要求集群管理员在将 Operator 添加至集群之前先对其进行审核。集群管理员还可获得一些工具来决定和限制在使用服务账户安装或升级 Operator 期间允许的操作。

通过将 *Operator 组*与获得一组权限的服务账户相关联，集群管理员便可对 Operator 设定策略，以确保它们仅在预先决定的边界范围内按照 RBAC 规则运行。Operator 无法执行任何这些规则未明确允许的操作。

非集群管理员可自行安装 Operator 但范围有限，这一规定意味着更多 Operator Framework 工具可安全提供给更多用户，为通过 Operator 构建应用程序提供更丰富的体验。

### 4.7.1. 了解 Operator 安装策略

通过使用 OLM，集群管理员可选择为一个 OperatorGroup 指定一个服务账户，以便部署与这个 Operator 组相关联的所有 Operator，并按照服务账户获得的权限运行。

**APIService** 和 **CustomResourceDefinition** 资源都由 OLM 使用 **cluster-admin** 角色来创建。不应向与 Operator 组相关联的服务账户授予写入这些资源的权限。

如果指定的服务账户对于正在安装或升级的 Operator 没有足够权限，则会在相应资源状态中添加实用背景信息，以便集群管理员轻松排除故障并解决问题。

与该 Operator 组相关联的所有 Operator 现已被限制在指定服务账户获得的权限范围内。如果 Operator 请求了超出服务账户范围的权限，安装将会失败，并将显示相应错误。

#### 4.7.1.1. 安装场景

在确定是否可在集群上安装或升级 Operator 时，Operator Lifecycle Manager（OLM）会考虑以下情况：

- 集群管理员新建了一个 Operator 组并指定了服务账户。已安装与该 Operator 组关联的所有 Operator，并根据相应服务账户获得的权限运行。
- 集群管理员新建了一个 Operator 组，且不指定任何服务帐户。OpenShift Container Platform 保持向后兼容性，因此会保留默认行为，并允许安装和升级 Operator。
- 对于未指定服务账户的现有 Operator 组，会保留默认行为,并允许安装和升级 Operator。
- 集群管理员更新了现有 Operator 组并指定了服务帐户。OLM 支持现有 Operator 继续根据当前权限运行。现有 Operator 升级后，它会重新安装并根据相应服务账户获得的权限运行，与新 Operator 一样。
- 由 Operator 组指定的服务帐户通过添加或删除权限来更改，或者现有服务账户被换为新服务帐户。现有 Operator 升级后，它会重新安装并根据更新后的服务账户获得的权限运行，与新 Operator 一样。
- 集群管理员从 Operator 组中删除服务账户。默认行为保留，并允许安装和升级 Operator。

#### 4.7.1.2. 安装工作流

当 Operator 组与服务账户绑定，并且安装或升级了 Operator 时，Operator Lifecycle Manager（OLM）会使用以下工作流：

1. OLM 会提取给定订阅对象。
2. OLM 获取与该订阅相关联的 Operator 组。
3. OLM 确定 Operator 组是否指定了服务帐户。
4. OLM 在服务账户范围内创建一个客户端，并使用该范围内客户端来安装 Operator。这样可确保 Operator 请求的任何权限始终限制在 Operator 组中服务账户的权限范围内。
5. OLM 新建一个服务账户，在 CSV 中指定其权限集，并将其分配至 Operator。Operator 将根据所分配的服务账户运行。

### 4.7.2. 限定 Operator 安装范围

要为 Operator Lifecycle Manager（OLM）上的 Operator 安装和升级提供范围规则，请将服务帐户与 Operator 组关联。

集群管理员可借鉴本例，将一组 Operator 限制到指定命名空间中。

**流程**

1. 新建命名空间：

```
$ cat <<EOF | oc create -f -
apiVersion: v1
kind: Namespace
metadata:
  name: scoped
EOF
```

2. 分配 Operator 的权限范围。这涉及创建新服务帐户、相关角色和角色绑定。

```
$ cat <<EOF | oc create -f -
apiVersion: v1
kind: ServiceAccount
metadata:
  name: scoped
  namespace: scoped
EOF
```

为简便起见，以下示例授予服务账户在指定命名空间进行任何操作的权限。在生产环境中，应创建更为精细的权限集：

```
$ cat <<EOF | oc create -f -
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: scoped
  namespace: scoped
rules:
- apiGroups: ["*"]
  resources: ["*"]
  verbs: ["*"]
---
apiVersion: rbac.authorization.k8s.io/v1
```

```
kind: RoleBinding
metadata:
  name: scoped-bindings
  namespace: scoped
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: scoped
subjects:
- kind: ServiceAccount
  name: scoped
  namespace: scoped
EOF
```

3. 在指定的命名空间中创建 **OperatorGroup** 对象。该 Operator 组以指定的命名空间为目标，以确保其租期仅限于该命名空间。
另外，Operator 组允许用户指定服务帐户。指定上一步中创建的服务帐户：

```
$ cat <<EOF | oc create -f -
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: scoped
  namespace: scoped
spec:
  serviceAccountName: scoped
  targetNamespaces:
  - scoped
EOF
```

在指定命名空间中安装的任何 Operator 均会关联至此 Operator 组，因此也会关联到指定的服务账户。

4. 在指定命名空间中创建 **Subscription** 对象以安装 Operator:

```
$ cat <<EOF | oc create -f -
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: etcd
  namespace: scoped
spec:
  channel: singlenamespace-alpha
  name: etcd
  source: <catalog_source_name> 1
  sourceNamespace: <catalog_source_namespace> 2
EOF
```

1 指定已存在于指定命名空间中或位于全局目录命名空间中的目录源。

2 指定创建目录源的命名空间。

与该 Operator 组相关联的所有 Operator 都仅限于为指定服务账户授予的权限。如果 Operator 请求的权限超出服务账户范围，安装会失败并显示相关错误。

#### 4.7.2.1. 细粒度权限

Operator Lifecycle Manager（OLM）使用 Operator 组中指定的服务账户来创建或更新与正在安装的 Operator 相关的以下资源：

- **ClusterServiceVersion**
- **Subscription**
- **Secret**
- **ServiceAccount**
- **Service**
- **ClusterRole** 和 **ClusterRoleBinding**
- **Role** 和 **RoleBinding**

要将 Operator 限制到指定命名空间，集群管理员可以首先向服务账户授予以下权限：

**注意**

以下角色只是一个通用示例，具体 Operator 可能需要额外规则。

```
kind: Role
rules:
- apiGroups: ["operators.coreos.com"]
  resources: ["subscriptions", "clusterserviceversions"]
  verbs: ["get", "create", "update", "patch"]
- apiGroups: [""]
  resources: ["services", "serviceaccounts"]
  verbs: ["get", "create", "update", "patch"]
- apiGroups: ["rbac.authorization.k8s.io"]
  resources: ["roles", "rolebindings"]
  verbs: ["get", "create", "update", "patch"]
- apiGroups: ["apps"] 1
  resources: ["deployments"]
  verbs: ["list", "watch", "get", "create", "update", "patch", "delete"]
- apiGroups: [""] 2
  resources: ["pods"]
  verbs: ["list", "watch", "get", "create", "update", "patch", "delete"]
```

1 2 增加创建其他资源的权限，如此处显示的部署和 pod。

另外，如果任何 Operator 指定了 pull secret，还必须增加以下权限：

```
kind: ClusterRole 1
rules:
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["get"]
---
kind: Role
```

```
rules:
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["create", "update", "patch"]
```

1 需要从 OLM 命名空间中获取 secret。

### 4.7.3. 故障排除权限失败

如果因为缺少权限而导致 Operator 安装失败，请按照以下流程找出错误。

**流程**

1. 查看 **Subscription** 对象。其状态中有一个指向 **InstallPlan** 对象的对象引用 **installPlanRef**，该对象试图为 Operator 创建需要的 **[Cluster]Role[Binding]**：

```
apiVersion: operators.coreos.com/v1
kind: Subscription
metadata:
  name: etcd
  namespace: scoped
status:
  installPlanRef:
    apiVersion: operators.coreos.com/v1
    kind: InstallPlan
    name: install-4plp8
    namespace: scoped
    resourceVersion: "117359"
    uid: 2c1df80e-afea-11e9-bce3-5254009c9c23
```

2. 检查 **InstallPlan** 对象的状态中的任何错误：

```
apiVersion: operators.coreos.com/v1
kind: InstallPlan
status:
  conditions:
  - lastTransitionTime: "2019-07-26T21:13:10Z"
    lastUpdateTime: "2019-07-26T21:13:10Z"
    message: 'error creating clusterrole etcdoperator.v0.9.4-clusterwide-dsfx4:
clusterroles.rbac.authorization.k8s.io
      is forbidden: User "system:serviceaccount:scoped:scoped" cannot create resource
      "clusterroles" in API group "rbac.authorization.k8s.io" at the cluster scope'
    reason: InstallComponentFailed
    status: "False"
    type: Installed
  phase: Failed
```

错误信息中会显示：

- 创建失败的资源类型，包括资源的 API 组。本例中为 **rbac.authorization.k8s.io** 组中的 **clusterroles**。
- 资源名称。

- 错误类型：**is forbidden** 表明相应用户没有足够权限来执行这一操作。
- 试图创建或更新资源的用户名称。本例中指的是 Operator 组中指定的服务账户。
- 操作范围：**集群范围内**或范围外。
  用户可在服务账户中增加所缺权限，然后迭代操作。

**注意**

Operator Lifecycle Manager（OLM）目前未提供第一次尝试的完整错误列表。

## 4.8. 管理自定义目录

本指南介绍了如何在 OpenShift Container Platform 的 Operator Lifecycle Manager (OLM) 中处理以 Bundle Format 或传统的 Bundle Format 打包的自定义目录。

**其它资源**

- 红帽提供的 Operator 目录

### 4.8.1. 使用捆绑格式（Bundle Format）的自定义目录

#### 4.8.1.1. 先决条件

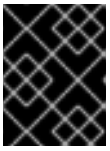
- 安装 **opm** CLI。

#### 4.8.1.2. 创建索引镜像

您可以使用 **opm** CLI 创建索引镜像。

**先决条件**

- **opm** 版本 1.12.3+
- **podman** 版本 1.9.3+
- 构建并推送到支持 Docker v2-2 的 registry 的捆绑包镜像



**重要**

OpenShift Container Platform 集群的内部 registry 不能用作目标 registry，因为它不支持没有标签的推送（在镜像过程中需要这个功能）。

**流程**

1. 启动一个新的索引：

```
$ opm index add \
    --bundles <registry>/<namespace>/<bundle_image_name>:<tag> \ 1
    --tag <registry>/<namespace>/<index_image_name>:<tag> \ 2
    [--binary-image <registry_base_image>] 3
```

1. 要添加到索引中的捆绑包镜像以逗号分隔的列表。
2. 希望索引镜像具有的镜像标签。
3. 可选：用于为目录提供服务的备选 registry 基础镜像。

2. 将索引镜像推送到 registry。

   a. 如果需要，与目标 registry 进行身份验证：

   ```
   $ podman login <registry>
   ```

   b. 推送索引镜像：

   ```
   $ podman push <registry>/<namespace>/test-catalog:latest
   ```

#### 4.8.1.3. 从索引镜像创建目录

您可以从索引镜像创建 Operator 目录，并将其应用到 OpenShift Container Platform 集群，供 Operator Lifecycle Manager（OLM）使用。

**先决条件**

- 构建并推送到 registry 的索引镜像。

**流程**

1. 创建一个 **CatalogSource** 对象来引用索引镜像。

   a. 根据您的规格修改以下内容，并将它保存为 **catalogSource.yaml** 文件：

   ```
   apiVersion: operators.coreos.com/v1alpha1
   kind: CatalogSource
   metadata:
     name: my-operator-catalog
     namespace: openshift-marketplace 1
   spec:
     sourceType: grpc
     image: <registry>:<port>/<namespace>/redhat-operator-index:v4.7 2
     displayName: My Operator Catalog
     publisher: <publisher_name> 3
     updateStrategy:
       registryPoll: 4
         interval: 30m
   ```

   1. 如果您希望目录源对所有命名空间中的用户全局可用，请指定 **openshift-marketplace** 命名空间。否则，您可以指定一个不同的命名空间来对目录进行作用域并只对该命名空间可用。
   2. 指定索引镜像。
   3. 指定发布目录的名称或机构名称。

4 目录源可以自动检查新版本以保持最新。

b. 使用该文件创建 **CatalogSource** 对象：

```
$ oc apply -f catalogSource.yaml
```

2. 确定成功创建以下资源。

a. 检查 pod:

```
$ oc get pods -n openshift-marketplace
```

输出示例

```
NAME                                READY   STATUS    RESTARTS  AGE
my-operator-catalog-6njx6           1/1     Running   0         28s
marketplace-operator-d9f549946-96sgr  1/1     Running   0         26h
```

b. 检查目录源：

```
$ oc get catalogsource -n openshift-marketplace
```

输出示例

```
NAME              DISPLAY             TYPE PUBLISHER  AGE
my-operator-catalog  My Operator Catalog  grpc          5s
```

c. 检查软件包清单：

```
$ oc get packagemanifest -n openshift-marketplace
```

输出示例

```
NAME                  CATALOG             AGE
jaeger-product          My Operator Catalog  93s
```

现在，您可以在 OpenShift Container Platform Web 控制台中通过 **OperatorHub** 安装 Operator。

**其他资源**

- 如果您的索引镜像托管在私有 registry 上且需要进行身份验证，请参阅 从私有 registry 访问 Operator 的镜像。

#### 4.8.1.4. 更新索引镜像

在将 OperatorHub 配置为使用引用自定义索引镜像的目录源后，集群管理员可通过将捆绑包镜像添加到索引镜像来保持其集群上的可用 Operator 最新状态。

您可以使用 **opm index add** 命令来更新存在的索引镜像。

**先决条件**

- **opm** 版本 1.12.3+
- **podman** 版本 1.9.3+
- 构建并推送到 registry 的索引镜像。
- 引用索引镜像的现有目录源。

**流程**

1. 通过添加捆绑包镜像来更新现有索引：

```
$ opm index add \
   --bundles <registry>/<namespace>/<new_bundle_image>@sha256:<digest> \ 1
   --from-index <registry>/<namespace>/<existing_index_image>:<existing_tag> \ 2
   --tag <registry>/<namespace>/<existing_index_image>:<updated_tag> \ 3
   --pull-tool podman 4
```

1. **--bundles** 标志指定要添加到索引中的、以逗号分隔的额外捆绑包镜像列表。
2. **--from-index** 标志指定之前推送的索引。
3. **--tag** 标志指定要应用到更新的索引镜像的镜像标签。
4. **--pull-tool** 标志指定用于拉取容器镜像的工具。

其中：

**<registry>**
指定 registry 的主机名，如 **quay.io** 或 **mirror.example.com**。

**<namespace>**
指定 registry 的命名空间，如 **ocs-dev** 或 **abc**。

**<new_bundle_image>**
指定要添加到 registry 的新捆绑包镜像，如 **ocs-operator**。

**<digest>**
指定捆绑包镜像的 SHA 镜像 ID 或摘要，如 **c7f11097a628f092d8bad148406aa0e0951094a03445fd4bc0775431ef683a41**。

**<existing_index_image>**
指定之前推送的镜像，如 **abc-redhat-operator-index**。

**<existing_tag>**
指定之前推送的镜像标签，如 **4.7**。

**<updated_tag>**
指定要应用到更新的索引镜像的镜像标签，如 **4.7.1**。

**示例命令**

```
$ opm index add \
   --bundles quay.io/ocs-dev/ocs-
operator@sha256:c7f11097a628f092d8bad148406aa0e0951094a03445fd4bc0775431ef683a
41 \
```

```
--from-index mirror.example.com/abc/abc-redhat-operator-index:4.7 \
--tag mirror.example.com/abc/abc-redhat-operator-index:4.7.1 \
--pull-tool podman
```

2. 推送更新的索引镜像：

```
$ podman push <registry>/<namespace>/<existing_index_image>:<updated_tag>
```

3. Operator Lifecycle Manager（OLM）会在常规时间段内自动轮询目录源中引用的索引镜像，验证是否已成功添加新软件包：

```
$ oc get packagemanifests -n openshift-marketplace
```

#### 4.8.1.5. 修剪索引镜像

基于 Operator Bundle Format 的索引镜像是 Operator 目录的容器化快照。您可以修剪除指定的软件包列表以外的所有索引，创建只包含您想要的 Operator 的源索引副本。

**先决条件**

- **podman** 版本 1.9.3+
- **grpcurl**（第三方命令行工具）
- **opm** 版本 1.18.0+
- 访问支持 Docker v2-2 的 registry

**重要**

OpenShift Container Platform 集群的内部 registry 不能用作目标 registry，因为它不支持没有标签的推送（在镜像过程中需要这个功能）。

**流程**

1. 通过目标 registry 进行身份验证：

```
$ podman login <target_registry>
```

2. 确定您要包括在您的修剪索引中的软件包列表。

   a. 运行您要修剪容器中的源索引镜像。例如：

```
$ podman run -p50051:50051 \
    -it registry.redhat.io/redhat/redhat-operator-index:v4.7
```

   输出示例

```
Trying to pull registry.redhat.io/redhat/redhat-operator-index:v4.7...
Getting image source signatures
Copying blob ae8a0c23f5b1 done
...
INFO[0000] serving registry                 database=/database/index.db port=50051
```

b. 在一个单独的终端会话中，使用 **grpcurl** 命令获取由索引提供的软件包列表：

```
$ grpcurl -plaintext localhost:50051 api.Registry/ListPackages > packages.out
```

c. 检查 **package.out** 文件，确定要保留在此列表中的哪个软件包名称。例如：

软**件包列表片断示例**

```
...
{
  "name": "advanced-cluster-management"
}
...
{
  "name": "jaeger-product"
}
...
{
{
  "name": "quay-operator"
}
...
```

d. 在您执行 **podman run** 命令的终端会话中，按 **Ctrl** 和 **C** 停止容器进程。

3. 运行以下命令来修剪指定软件包以外的所有源索引：

```
$ opm index prune \
   -f registry.redhat.io/redhat/redhat-operator-index:v4.7 \ 1
   -p advanced-cluster-management,jaeger-product,quay-operator \ 2
   [-i registry.redhat.io/openshift4/ose-operator-registry:v4.7] \ 3
   -t <target_registry>:<port>/<namespace>/redhat-operator-index:v4.7 4
```

1 到修剪的索引。

2 要保留的软件包用逗号隔开。

3 只适用于 IBM Power Systems 和 IBM Z 镜像： Operator Registry 基础镜像和与目标 OpenShift Container Platform 集群主版本和次版本匹配的标签。

4 用于正在构建新索引镜像的自定义标签。

4. 运行以下命令将新索引镜像推送到目标 registry:

```
$ podman push <target_registry>:<port>/<namespace>/redhat-operator-index:v4.7
```

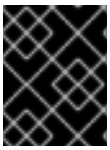
其中 **<namespace>** 是 registry 上的任何现有命名空间。

## 4.8.2. 使用 Package Manifest Format 的自定义目录

### 4.8.2.1. 构**建**软**件包清**单**格式目**录镜**像**

集群管理员可以根据 Operator Lifecycle Manager（OLM）使用的 Package Manifest Format 来构建自定

义 Operator 目录镜像。目录镜像可推送到支持 Docker v2-2 的容器镜像 registry。对于受限网络中的集群，此 registry 可以是集群有网络访问权限的 registry，如在受限网络集群安装过程中创建的镜像 registry。



**重要**

OpenShift Container Platform 集群的内部 registry 不能用作目标 registry，因为它不支持没有标签的推送（在镜像过程中需要这个功能）。

在这一示例中，流程假定在使用镜像 registry 时可访问您的网络以及互联网。

**注意**

只有 **oc** 客户端的 Linux 版本可用于此流程，因为 Windows 和 macOS 版本不提供 **oc adm catalog build** 命令。

**先决条件**

- 没有网络访问限制的工作站
- **oc** 版本 4.3.5+ Linux 客户端
- **podman** 版本 1.9.3+
- 访问支持 Docker v2-2 的镜像（mirror）registry
- 如果您正在使用私有 registry，请将 **REG_CREDS** 环境变量设置为到 registry 凭证的文件路径。例如，对于 **podman** CLI:

```
$ REG_CREDS=${XDG_RUNTIME_DIR}/containers/auth.json
```

- 如果您正在使用 quay.io 帐户可访问的私有命名空间，您必须设置 Quay 身份验证令牌。使用您的 quay.io 凭证对登录 API 发出请求，从而设置用于 **--auth-token** 标志的 **AUTH_TOKEN** 环境变量：

```
$ AUTH_TOKEN=$(curl -sH "Content-Type: application/json" \
    -XPOST https://quay.io/cnr/api/v1/users/login -d '
    {
        "user": {
            "username": """<quay_username>""",
            "password": """<quay_password>"""
        }
    }' | jq -r '.token')
```

**流程**

1. 在没有网络访问限制的工作站中，与目标镜像（mirror） registry 进行身份验证：

```
$ podman login <registry_host_name>
```

2. 使用 **registry.redhat.io** 进行身份验证，以便在构建期间拉取基础镜像：

```
$ podman login registry.redhat.io
```

3. 根据 Quay.io 中的 **redhat-operators** 目录构建目录镜像，进行标记并将其推送到您的镜像 registry:

```
$ oc adm catalog build \
    --appregistry-org redhat-operators \ 1
    --from=registry.redhat.io/openshift4/ose-operator-registry:v4.7 \ 2
    --filter-by-os="linux/amd64" \ 3
    --to=<registry_host_name>:<port>/olm/redhat-operators:v1 \ 4
    [-a ${REG_CREDS}] \ 5
    [--insecure] \ 6
    [--auth-token "${AUTH_TOKEN}"] 7
```

1. 从 App Registry 实例中拉取的机构（命名空间）。
2. 使用与目标 OpenShift Container Platform 集群主版本和次版本匹配的标签将 **--from** 设置为 Operator Registry 基础镜像。
3. 将 **--filter-by-os** 设置为用于基本镜像的操作系统和架构，该镜像必须与目标 OpenShift Container Platform 集群匹配。有效值是 **linux/amd64**、**linux/ppc64le** 和 **linux/s390x**。
4. 为您的目录镜像命名并包含标签，例如: **v1**。
5. 可选：如果需要，指定 registry 凭证文件的位置。
6. 可选：如果您不想为目标 registry 配置信任，请添加 **--insecure** 标志。
7. 可选：如果使用其他不公开的应用程序 registry 目录，则需要指定 Quay 身份验证令牌。

**输出示例**

```
INFO[0013] loading Bundles
dir=/var/folders/st/9cskxqs53ll3wdn434vw4cd80000gn/T/300666084/manifests-829192605
...
Pushed sha256:f73d42950021f9240389f99ddc5b0c7f1b533c054ba344654ff1edaf6bf827e3
to example_registry:5000/olm/redhat-operators:v1
```

有时红帽提供的无效清单是意外引入的目录；当发生这种情况时，您可能会看到一些错误：

**出错的输出示例**

```
...
INFO[0014] directory
dir=/var/folders/st/9cskxqs53ll3wdn434vw4cd80000gn/T/300666084/manifests-829192605
file=4.2 load=package
W1114 19:42:37.876180   34665 builder.go:141] error building database: error loading
package into db: fuse-camel-k-operator.v7.5.0 specifies replacement that couldn't be found
Uploading ... 244.9kB/s
```

这些错误通常不是致命的，如果所提及 Operator 软件包不包含您计划安装的 Operator 或其依赖项，则可以忽略它们。

**其它资源**

- 为断开连接的安装 mirror 镜像

#### 4.8.2.2. 对 Package Manifest Format 目录镜像进行镜像(mirror)

集群管理员可以根据 Package Manifest Format 将自定义 Operator 目录镜像(mirror)到 registry 中，并使用目录源将内容加载到其集群中。本例中的流程使用之前构建并推送到受支持的 registry 的自定义 **redhat-operators** 目录镜像。

**先决条件**

- 没有网络访问限制的工作站
- 基于推送到受支持的 registry 的 Package Manifest Format 的自定义 Operator 目录镜像
- **oc** 版本 4.3.5+
- **podman** 版本 1.9.3+
- 访问支持 Docker v2-2 的镜像（mirror）registry

**重要**

OpenShift Container Platform 集群的内部 registry 不能用作目标 registry，因为它不支持没有标签的推送（在镜像过程中需要这个功能）。

- 如果您正在使用私有 registry，请将 **REG_CREDS** 环境变量设置为到 registry 凭证的文件路径。例如，对于 **podman** CLI:

```
$ REG_CREDS=${XDG_RUNTIME_DIR}/containers/auth.json
```

**流程**

1. **oc adm catalog mirror** 命令提取自定义 Operator catalog 镜像的内容，以生成镜像（mirror）所需的清单：您可以选择：
   - 允许该命令的默认行为在生成清单后自动将所有镜像内容镜像到您的镜像 registry 中，或者
   - 添加 **--manifests-only** 标志来只生成镜像所需的清单，但并不实际将镜像内容镜像到 registry。这对检查哪些要镜像（mirror）非常有用。如果您只需要部分内容的话，可以对映射列表做出任何修改。然后，您可以使用该文件与 **oc image mirror** 命令一起，在以后的步骤中镜像修改的镜像列表。

   在没有网络访问限制的工作站中，运行以下命令：

```
$ oc adm catalog mirror \
    <registry_host_name>:<port>/olm/redhat-operators:v1 \ 1
    <registry_host_name>:<port> \ 2
    [-a ${REG_CREDS}] \ 3
    [--insecure] \ 4
    [--index-filter-by-os='<platform>/<arch>'] \ 5
    [--manifests-only] 6
```

1 指定 Operator 目录镜像。

2 指定目标 registry 的完全限定域名(FQDN)。

3 可选：如果需要，指定 registry 凭证文件的位置。

4 可选：如果您不想为目标 registry 配置信任，请添加 **--insecure** 标志。

5 可选：在有多个变体可用时，指定目录镜像的平台和架构。镜像被传递为 **'<platform>/<arch>[/<variant>]'**。这不适用于目录镜像引用的镜像。有效值是 **linux/amd64**、**linux/ppc64le** 和 **linux/s390x**。

6 可选：只生成镜像所需的清单，但并不实际将镜像内容镜像到 registry。

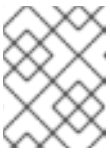
输**出示例**

```
using database path mapping: /:/tmp/190214037
wrote database to /tmp/190214037
using database at: /tmp/190214037/bundles.db 1
...
```

1 命令生成的临时数据库。

运行完该命令后，会在当前目录中创建 **manifests-<index_image_name>-<random_number>/** 目录并生成以下文件：

- **catalogSource.yaml** 文件是 **CatalogSource** 对象的基本定义，它预先填充目录镜像标签及其他相关元数据。此文件可原样使用，或进行相应修改来在集群中添加目录源。
- 用来定义 **ImageContentSourcePolicy** 对象的 **imageContentSourcePolicy.yaml**，它可以将节点配置为在 Operator 清单中存储的镜像（image）引用和镜像 (mirror) 的 registry 间进行转换。



**注意**

如果您的集群使用 **ImageContentSourcePolicy** 对象来配置存储库镜像，则只能将全局 pull secret 用于镜像 registry。您不能在项目中添加 pull secret。

- **mapping.txt** 文件，在其中包含所有源镜像，并将它们映射到目标 registry。此文件与 **oc image mirror** 命令兼容，可用于进一步自定义镜像（mirror）配置。

2. 如果您在上一步中使用 **--manifests-only** 标志，并只想镜像部分内容：

   a. 将 **mapping.txt** 文件中的镜像列表改为您的规格。如果您不确定要镜像的镜像子集的名称和版本，请使用以下步骤查找：

      i. 对 **oc adm catalog mirror** 命令生成的临时数据库运行 **sqlite3** 工具，以检索与一般搜索查询匹配的镜像列表。输出以后如何编辑 **mapping.txt** 文件的帮助信息。
      例如，要检索与字符串 **clusterlogging.4.3** 类似的镜像列表：

```
$ echo "select * from related_image \
    where operatorbundle_name like 'clusterlogging.4.3%';" \
    | sqlite3 -line /tmp/190214037/bundles.db 1
```

1 请参阅 **oc adm catalog mirror** 命令的输出结果来查找数据库文件的路径。

输出示例

```
image = registry.redhat.io/openshift-logging/kibana6-
rhel8@sha256:aa4a8b2a00836d0e28aa6497ad90a3c116f135f382d8211e3c55f34fb36
dfe61
operatorbundle_name = clusterlogging.4.3.33-202008111029.p0

image = registry.redhat.io/openshift4/ose-oauth-
proxy@sha256:6b4db07f6e6c962fc96473d86c44532c93b146bbefe311d0c348117bf75
9c506
operatorbundle_name = clusterlogging.4.3.33-202008111029.p0
...
```

ii. 使用上一步的结果来编辑 **mapping.txt** 文件，使其只包含您要镜像的镜像子集。例如，您可以使用前面示例输出中的 **image** 值来找出您的 **mapping.txt** 文件中存在以下匹配行：

**mapping.txt 中的匹配镜像映射**

```
registry.redhat.io/openshift-logging/kibana6-
rhel8@sha256:aa4a8b2a00836d0e28aa6497ad90a3c116f135f382d8211e3c55f34fb36
dfe61=<registry_host_name>:<port>/kibana6-rhel8:a767c8f0
registry.redhat.io/openshift4/ose-oauth-
proxy@sha256:6b4db07f6e6c962fc96473d86c44532c93b146bbefe311d0c348117bf75
9c506=<registry_host_name>:<port>/openshift4-ose-oauth-proxy:3754ea2b
```

在这个示例中，如果您只想镜像这些 image，可以在 **mapping.txt** 文件中删除所有其他条目，仅保留上述两行。

b. 在您的没有网络访问限制的工作站中，使用您修改的 **mapping.txt** 文件，使用 **oc image mirror** 命令将镜像镜像到 registry:

```
$ oc image mirror \
    [-a ${REG_CREDS}] \
    --filter-by-os='.*' \
    -f ./manifests-redhat-operators-<random_number>/mapping.txt
```

**警告**

如果未设置 **--filter-by-os** 标志或设置为 **.*** 以外的任何值，该命令会过滤不同的架构，用来更改清单列表摘要，也称为 *多架构镜像*。不正确的摘要会导致在断开连接的集群中部署这些镜像和 Operator 失败。

3. 创建 **ImageContentSourcePolicy** 对象：

```
$ oc create -f ./manifests-redhat-operators-
<random_number>/imageContentSourcePolicy.yaml
```

现在，您可以创建一个 **CatalogSource** 对象来引用您的镜像内容。

**其他资源**

- Operator 的架构和操作系统支持
- 如果您的目录镜像托管在私有 registry 上且需要进行身份验证，请参阅 从私有 registry 访问 Operator 的镜像。

#### 4.8.2.3. 更新软件包清单格式目录镜像

集群管理员将 OperatorHub 配置为使用自定义 Operator 目录镜像后，管理员可通过红帽提供的 App Registry 目录获得更新，使其 OpenShift Container Platform 集群与最新的 Operator 保持一致。这可以通过构建和推送新的 Operator 目录镜像，然后将 **CatalogSource** 对象中现有的 **spec.image** 参数替换为新镜像摘要。

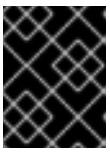
在本例中，该流程假设已配置了自定义 **redhat-operators** 目录镜像，以便与 OperatorHub 搭配使用。

**注意**

只有 **oc** 客户端的 Linux 版本可用于此流程，因为 Windows 和 macOS 版本不提供 **oc adm catalog build** 命令。

**先决条件**

- 没有网络访问限制的工作站
- **oc** 版本 4.3.5+ Linux 客户端
- **podman** 版本 1.9.3+
- 访问支持 Docker v2-2 的镜像（mirror）registry



**重要**

OpenShift Container Platform 集群的内部 registry 不能用作目标 registry，因为它不支持没有标签的推送（在镜像过程中需要这个功能）。

- OperatorHub 配置为使用自定义目录镜像
- 如果您正在使用私有 registry，请将 **REG_CREDS** 环境变量设置为到 registry 凭证的文件路径。例如，对于 **podman** CLI:

```
$ REG_CREDS=${XDG_RUNTIME_DIR}/containers/auth.json
```

- 如果您正在使用 quay.io 帐户可访问的私有命名空间，您必须设置 Quay 身份验证令牌。使用您的 quay.io 凭证对登录 API 发出请求，从而设置用于 **--auth-token** 标志的 **AUTH_TOKEN** 环境变量：

```
$ AUTH_TOKEN=$(curl -sH "Content-Type: application/json" \
    -XPOST https://quay.io/cnr/api/v1/users/login -d '
    {
        "user": {
            "username": "<quay_username>",
            "password": "<quay_password>"
        }
    }' | jq -r '.token')
```

- 

**流程**

1. 在没有网络访问限制的工作站中，与目标镜像（mirror） registry 进行身份验证：

```
$ podman login <registry_host_name>
```

2. 使用 **registry.redhat.io** 进行身份验证，以便在构建期间拉取基础镜像：

```
$ podman login registry.redhat.io
```

3. 根据 Quay.io 中的 **redhat-operators** 目录构建新目录镜像，标记并将其推送到您的镜像 registry:

```
$ oc adm catalog build \
  --appregistry-org redhat-operators \ 1
  --from=registry.redhat.io/openshift4/ose-operator-registry:v4.7 \ 2
  --filter-by-os="linux/amd64" \ 3
  --to=<registry_host_name>:<port>/olm/redhat-operators:v2 \ 4
  [-a ${REG_CREDS}] \ 5
  [--insecure] \ 6
  [--auth-token "${AUTH_TOKEN}"] 7
```

1. 从 App Registry 实例中拉取的机构（命名空间）。
2. 使用与目标 OpenShift Container Platform 集群主版本和次版本匹配的标签将 **--from** 设置为 Operator Registry 基础镜像。
3. 将 **--filter-by-os** 设置为用于基本镜像的操作系统和架构，该镜像必须与目标 OpenShift Container Platform 集群匹配。有效值是 **linux/amd64**、**linux/ppc64le** 和 **linux/s390x**。
4. 为您的目录镜像命名并包含一个标签，例如: **v2**，因为它是更新的目录。
5. 可选：如果需要，指定 registry 凭证文件的位置。
6. 可选：如果您不想为目标 registry 配置信任，请添加 **--insecure** 标志。
7. 可选：如果使用其他不公开的应用程序 registry 目录，则需要指定 Quay 身份验证令牌。

输**出示例**

```
INFO[0013] loading Bundles
dir=/var/folders/st/9cskxqs53ll3wdn434vw4cd80000gn/T/300666084/manifests-829192605
...
Pushed sha256:f73d42950021f9240389f99ddc5b0c7f1b533c054ba344654ff1edaf6bf827e3
to example_registry:5000/olm/redhat-operators:v2
```

4. 将目录镜像 (mirror) 到目标 registry。以下 **oc adm catalog mirror** 命令提取自定义 Operator catalog 镜像的内容，以生成镜像 (mirror) 和把镜像（image）镜像 (mirror)到 registry 所需的清单：

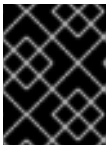
```
$ oc adm catalog mirror \
  <registry_host_name>:<port>/olm/redhat-operators:v2 \ 1
```

```
<registry_host_name>:<port> \ 2
[-a ${REG_CREDS}] \ 3
[--insecure] \ 4
[--index-filter-by-os='<platform>/<arch>'] 5
```

1. 指定新 Operator 目录镜像。
2. 指定目标 registry 的完全限定域名(FQDN)。
3. 可选：如果需要，指定 registry 凭证文件的位置。
4. 可选：如果您不想为目标 registry 配置信任，请添加 **--insecure** 标志。
5. 可选：在有多个变体可用时，指定目录镜像的平台和架构。镜像被传递为 **'<platform>/<arch>[/<variant>]'**。这不适用于目录镜像引用的镜像。有效值是 **linux/amd64**、**linux/ppc64le** 和 **linux/s390x**。

5. 应用新生成的清单：

```
$ oc replace -f ./manifests-redhat-operators-<random_number>
```



**重要**

您可能不需要应用 **imageContentSourcePolicy.yaml** 清单。对文件进行一个 **diff** 操作来决定是否需要改变。

6. 更新 **CatalogSource** 对象以引用您的目录镜像。
   a. 如果您有此 **CatalogSource** 对象的原始 **catalogsource.yaml** 文件：
      i. 编辑 **catalogsource.yaml** 文件，在 **spec.image** 字段中引用新目录镜像 ：

```
apiVersion: operators.coreos.com/v1alpha1
kind: CatalogSource
metadata:
  name: my-operator-catalog
  namespace: openshift-marketplace
spec:
  sourceType: grpc
  image: <registry_host_name>:<port>/olm/redhat-operators:v2 1
  displayName: My Operator Catalog
  publisher: grpc
```

1. 指定新 Operator 目录镜像。

      ii. 使用更新的文件替换 **CatalogSource** 对象：

```
$ oc replace -f catalogsource.yaml
```

   b. 或者，使用以下命令编辑目录源，在 **spec.image** 参数中引用您的新目录镜像 ：

```
$ oc edit catalogsource <catalog_source_name> -n openshift-marketplace
```

现在，OpenShift Container Platform 集群上的 **OperatorHub** 页中应可以提供更新的 Operator。

**其他资源**

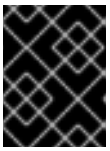
- Operator 的架构和操作系统支持

#### 4.8.2.4. 测试软件包清单格式目录镜像

要验证 Operator 目录镜像内容，您可以将其作为容器运行并查询其 gRPC API。要进一步测试镜像，您可以通过引用目录源中的镜像来在 Operator Lifecycle Manager（OLM）中解析订阅。本例中的流程使用之前构建并推送到受支持的 registry 的自定义 **redhat-operators** 目录镜像。

**先决条件**

- 推送到受支持的 registry 的自定义 Package Manifest Format 目录镜像
- **podman** 版本 1.9.3+
- **oc** 版本 4.3.5+
- 访问支持 Docker v2-2 的镜像（mirror）registry



**重要**

OpenShift Container Platform 集群的内部 registry 不能用作目标 registry，因为它不支持没有标签的推送（在镜像过程中需要这个功能）。

- **grpcurl**

**流程**

1. 拉取 Operator 目录镜像：

```
$ podman pull <registry_host_name>:<port>/olm/redhat-operators:v1
```

2. 运行镜像：

```
$ podman run -p 50051:50051 \
    -it <registry_host_name>:<port>/olm/redhat-operators:v1
```

3. 使用 **grpcurl** 查询正在运行的镜像中的可用软件包：

```
$ grpcurl -plaintext localhost:50051 api.Registry/ListPackages
```

输出示例

```
{
  "name": "3scale-operator"
}
{
  "name": "amq-broker"
}
```

```
{
  "name": "amq-online"
}
```

4. 在频道中获取最新的 Operator 捆绑包：

```
$  grpcurl -plaintext -d '{"pkgName":"kiali-ossm","channelName":"stable"}' localhost:50051
api.Registry/GetBundleForChannel
```

输**出示例**

```
{
  "csvName": "kiali-operator.v1.0.7",
  "packageName": "kiali-ossm",
  "channelName": "stable",
...
```

5. 获取镜像摘要：

```
$ podman inspect \
    --format='{{index .RepoDigests 0}}' \
    <registry_host_name>:<port>/olm/redhat-operators:v1
```

输**出示例**

```
example_registry:5000/olm/redhat-
operators@sha256:f73d42950021f9240389f99ddc5b0c7f1b533c054ba344654ff1edaf6bf827e3
```

6. 假设命名空间 **my-ns** 中存在一个支持您的 Operator 及其依赖项的 Operator 组，使用镜像摘要创建一个 **CatalogSource** 对象。例如：

```
apiVersion: operators.coreos.com/v1alpha1
kind: CatalogSource
metadata:
  name: custom-redhat-operators
  namespace: my-ns
spec:
  sourceType: grpc
  image: example_registry:5000/olm/redhat-
operators@sha256:f73d42950021f9240389f99ddc5b0c7f1b533c054ba344654ff1edaf6bf827e3

  displayName: Red Hat Operators
```

7. 创建一个可从您的目录镜像解析最新可用 **servicemeshoperator** 及其依赖项的订阅：

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: servicemeshoperator
  namespace: my-ns
spec:
  source: custom-redhat-operators
```

```
sourceNamespace: my-ns
name: servicemeshoperator
channel: "1.0"
```

### 4.8.3. 从私有 registry 访问 Operator 的镜像

如果与 Operator Lifecycle Manager（OLM）管理的 Operator 相关的某些镜像托管在需要经过身份验证的容器镜像 registry（也称为私有 registry）中时，在默认情况下，OLM 和 OperatorHub 将无法拉取镜像。要启用访问权限，可以创建一个包含 registry 验证凭证的 pull secret。通过引用一个或多个目录源的 pull secret，OLM 可以处理将 secret 放置到 Operator 和目录命名空间中以允许进行安装。

Operator 或其 Operands 所需的其他镜像可能会需要访问私有 registry。对于这种情况，OLM 不处理将 secret 放置到目标租户命名空间中，但身份验证凭证可以添加到全局范围集群 pull secret 中，或单独的命名空间服务帐户中，以启用所需的访问权限。

在决定由 OLM 管理的 Operator 是否有适当的拉取访问权限时，应该考虑以下类型的镜像：

**索引或目录镜像**

**CatalogSource** 对象可以引用索引镜像或目录镜像，这些镜像是作为托管在镜像 registry 中的容器镜像的目录源打包的。索引镜像使用 Bundle Format 并引用捆绑包镜像，而目录镜像使用 Package Manifest Format。如果索引或目录镜像托管在私有 registry 中，可以使用 secret 来启用拉取访问。

**捆绑包镜像**

Operator 捆绑包镜像是元数据和清单，并被打包为容器镜像，代表 Operator 的一个特定版本。如果目录源中引用的任何捆绑包镜像托管在一个或多个私有 registry 中，可以使用一个 secret 来启用拉取（pull）访问。

**Operator 和 Operand 镜像**

如果从目录源安装的 Operator 使用私有镜像，对于 Operator 镜像本身或它监视的 Operand 镜像之一，Operator 将无法安装，因为部署无法访问所需的 registry 身份验证。在目录源中引用 secret 不会使 OLM 将 secret 放置到安装 Operands 的目标租户命名空间中。
相反，身份验证详情被添加到 **openshift-config** 命名空间中的全局集群 pull secret 中，该 secret 提供对集群上所有命名空间的访问。另外，如果不允许访问整个集群，则可将 pull secret 添加到目标租户命名空间的 **default** 服务帐户中。

**先决条件**

- 至少以下之一被托管在私有 registry 中：
  - 一个索引镜像或目录镜像。
  - 一个 Operator 捆绑包镜像。
  - 一个 Operator 或 Operand 镜像。

**流程**

1. 为每个必需的私有 registry 创建 secret。
   a. 登录到私有 registry 以创建或更新 registry 凭证文件：

```
$ podman login <registry>:<port>
```

**注意**

registry 凭证的文件路径可能会根据用于登录到 registry 的容器工具的不同而有所不同。对于 **podman** CLI，默认位置为 **${XDG_RUNTIME_DIR}/containers/auth.json**。对于 **docker** CLI，默认位置为 **/root/.docker/config.json**。

b. 建议您为每个 secret 只包含一个 registry 的凭证，并在单独的 secret 中为多个 registry 管理凭证。后续步骤中的 **CatalogSource** 可包括多个 secret，OpenShift Container Platform 会将这些 secret 合并为一个单独的虚拟凭证文件，以便在镜像拉取过程中使用。
默认情况下，registry 凭证文件可以存储多个 registry 的详情。确认您的文件的当前内容。例如：

为两**个 registry 存**储**凭**证**的文件**

```
{
        "auths": {
                "registry.redhat.io": {
                        "auth": "FrNHNydQXdzclNqdg=="
                },
                "quay.io": {
                        "auth": "Xd2lhdsbnRib21iMQ=="
                }
        }
}
```

由于此文件用于后续步骤中创建 secret，请确保为每个文件只存储一个 registry 的详情。这可使用以下方法之一完成：

- 使用 **podman logout <registry>** 命令为额外 registry 删除凭证，直到您只保留一个 registry。
- 编辑 registry 凭证文件，将 registry 详情分开以存储在多个文件中。例如：

  为**一个 registry 存**储**凭**证**的文件**

  ```
  {
          "auths": {
                  "registry.redhat.io": {
                          "auth": "FrNHNydQXdzclNqdg=="
                  }
          }
  }
  ```

  为另**一个 registry 存**储**凭**证**的文件**

  ```
  {
          "auths": {
                  "quay.io": {
                          "auth": "Xd2lhdsbnRib21iMQ=="
                  }
          }
  }
  ```

c. 在 **openshift-marketplace** 命名空间中创建 secret，其中包含私有 registry 的身份验证凭证：

```
$ oc create secret generic <secret_name> \
    -n openshift-marketplace \
    --from-file=.dockerconfigjson=<path/to/registry/credentials> \
    --type=kubernetes.io/dockerconfigjson
```

重复此步骤，为任何其他需要的私有 registry 创建额外的 secret，更新 **--from-file** 标志以指定另一个 registry 凭证文件路径。

2. 创建或更新现有的 **CatalogSource** 对象以引用一个或多个 secret：

```
apiVersion: operators.coreos.com/v1alpha1
kind: CatalogSource
metadata:
  name: my-operator-catalog
  namespace: openshift-marketplace
spec:
  sourceType: grpc
  secrets: 1
  - "<secret_name_1>"
  - "<secret_name_2>"
  image: <registry>:<port>/<namespace>/<image>:<tag>
  displayName: My Operator Catalog
  publisher: <publisher_name>
  updateStrategy:
    registryPoll:
      interval: 30m
```

1 添加 **spec.secrets** 部分并指定任何所需 secret。

3. 如果订阅的 Operator 引用的任何 Operator 或 Operand 镜像需要访问私有 registry，您可以提供对集群中的所有命名空间或单独的目标租户命名空间的访问。

- 要访问集群中的所有命名空间，请将身份验证详情添加到 **openshift-config** 命名空间中的全局集群 pull secret 中。

**警告**

集群资源必须调整为新的全局 pull secret，这样可暂时限制集群的可用性。

a. 从全局 pull secret 中提取 **.dockerconfigjson** 文件：

```
$ oc extract secret/pull-secret -n openshift-config --confirm
```

b. 使用所需私有 registry 或 registry 的身份验证凭证更新 **.dockerconfigjson** 文件，并将它保存为新文件：

```
$ cat .dockerconfigjson | \
    jq --compact-output '.auths["<registry>:<port>/<namespace>/"] |= . + {"auth":"
<token>"}' \ 1
    > new_dockerconfigjson
```

1 将 **<registry>:<port> /<namespace>** 替换为私有 registry 的详情，将 **<token>** 替换为您的身份验证凭证。

c. 使用新文件更新全局 pull secret：

```
$ oc set data secret/pull-secret -n openshift-config \
    --from-file=.dockerconfigjson=new_dockerconfigjson
```

- 要更新单个命名空间，请向 Operator 的服务帐户添加一个 pull secret，以便在目标租户命名空间中访问该 Operator。

a. 在租户命名空间中为 **openshift-marketplace** 重新创建 secret：

```
$ oc create secret generic <secret_name> \
    -n <tenant_namespace> \
    --from-file=.dockerconfigjson=<path/to/registry/credentials> \
    --type=kubernetes.io/dockerconfigjson
```

b. 通过搜索租户命名空间来验证 Operator 的服务帐户名称：

```
$ oc get sa -n <tenant_namespace> 1
```

1 如果 Operator 安装在单独的命名空间中，请搜索该命名空间。如果 Operator 已为所有命名空间安装，请搜索 **openshift-operators** 命名空间。

**输出示例**

```
NAME          SECRETS   AGE
builder       2         6m1s
default       2         6m1s
deployer      2         6m1s
etcd-operator 2         5m18s 1
```

1 已安装的 etcd Operator 的服务帐户。

c. 将 secret 链接到 Operator 的服务帐户：

```
$ oc secrets link <operator_sa> \
    -n <tenant_namespace> \
     <secret_name> \
    --for=pull
```

**其他资源**

- 如需了解更多与 secret 相关的信息，包括用于 registry 凭证的 secret 类型的信息，请参阅什么是 secret?
- 如需了解有关更改此 secret 的影响的更多详细信息，请参阅更新全局集群 pull secret。
- 如需了解更多有关将 pull secret 链接到每个命名空间的服务帐户的详情，请参阅允许 Pod 引用其他安全 registry 中的镜像。

### 4.8.4. 禁用默认的 OperatorHub 源

在 OpenShift Container Platform 安装过程中，默认为 OperatorHub 配置由红帽和社区项目提供的源内容的 operator 目录。作为集群管理员，您可以禁用默认目录集。

**流程**

- 通过在 **OperatorHub** 对象中添加 **disableAllDefaultSources: true** 来禁用默认目录的源：

```
$ oc patch OperatorHub cluster --type json \
    -p '[{"op": "add", "path": "/spec/disableAllDefaultSources", "value": true}]'
```

**提示**

或者，您可以使用 Web 控制台管理目录源。在 **Administration → Cluster Settings → Global Configuration → OperatorHub** 页面中，点 **Sources** 选项卡，其中可创建、删除、禁用和启用单独的源。

### 4.8.5. 删除自定义目录

作为集群管理员，您可以删除之前添加到集群中的自定义 Operator 目录，方法是删除相关的目录源。

**流程**

1. 在 Web 控制台的 **Administrator** 视角中，导航至 **Administration → Cluster Settings**。
2. 点 **Global Configuration** 选项卡，然后点 **OperatorHub**。
3. 点 **Sources** 选项卡。
4. 选择您要删除的目录的 **Options** 菜单  ，然后点 **Delete CatalogSource**。

## 4.9. 在受限网络中使用 OPERATOR LIFECYCLE MANAGER

对于在受限网络中安装的 OpenShift Container Platform 集群（也称为 *断开连接的集群*），Operator Lifecycle Manager（OLM）默认无法访问托管在远程 registry 上的红帽提供的 OperatorHub 源，因为这些远程源需要足够的互联网连接。

但是，作为集群管理员，如果您有一个有完全互联网访问的工作站，则仍可以让集群在受限网络中使用 OLM。工作站需要完全访问互联网来拉取远程 OperatorHub 内容，用于准备远程源的本地镜像，并将内容推送到镜像 registry。

镜像 registry 可以位于堡垒主机上，它需要连接到您的工作站和断开连接的集群，或者一个完全断开连接的或 *airgapped* 主机，这需要可移动介质物理将镜像内容移到断开连接的环境中。

本指南描述了在受限网络中启用 OLM 所需的流程：

- 为 OLM 禁用默认远程 OperatorHub 源。
- 使用有完全互联网访问的工作站来创建并推送 OperatorHub 内容的本地镜像到镜像 registry。
- 将 OLM 配置为从镜像 registry 上的本地源而不是默认的远程源安装和管理 Operator。

在受限网络中启用 OLM 后，您可以继续使用不受限制的工作站在发布新版 Operator 时保持本地 OperatorHub 源的更新。

**重要**

虽然 OLM 可以从本地源管理 Operator，但给定 Operator 在受限网络中能否成功运行仍取决于 Operator 本身。Operator 必须：

- 在 **ClusterServiceVersion** (CSV) 对象的 **relatedImages**参数中列出所有相关的镜像，或 Operator 执行时可能需要的其他容器镜像。
- 通过摘要 (SHA) 而不是标签来引用所有指定的镜像。

如需了解在断开连接的模式模中可以运行的 Operator 列表，请参阅以下红帽知识库文章：

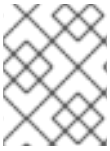
https://access.redhat.com/articles/4740011

**其它资源**

- 红帽提供的 Operator 目录
- 为受限网络环境启用 Operator

### 4.9.1. 先决条件

- 以具有 **cluster-admin** 权限的用户身份登录 OpenShift Container Platform 集群。
- 如果要修剪默认目录，且只有选择地镜像部分 Operator，请安装 **opm** CLI。



**注意**

如果您在 IBM Z 上的受限网络中使用 OLM，则必须至少为放置 registry 的目录分配 12 GB 的存储空间。

### 4.9.2. 禁用默认的 OperatorHub 源

在 OpenShift Container Platform 安装过程中，默认为 OperatorHub 配置由红帽和社区项目提供的源内容的 operator 目录。在受限网络环境中，必须以集群管理员身份禁用默认目录。然后，您可以将 OperatorHub 配置为使用本地目录源。

**流程**

- 通过在 **OperatorHub** 对象中添加 **disableAllDefaultSources: true 来** 禁用默认目录的源：

```
$ oc patch OperatorHub cluster --type json \
    -p '[{"op": "add", "path": "/spec/disableAllDefaultSources", "value": true}]'
```

**提示**

或者，您可以使用 Web 控制台管理目录源。在 **Administration → Cluster Settings → Global Configuration → OperatorHub** 页面中，点 **Sources** 选项卡，其中可创建、删除、禁用和启用单独的源。

### 4.9.3. 修剪索引镜像

基于 Operator Bundle Format 的索引镜像是 Operator 目录的容器化快照。您可以修剪除指定的软件包列表以外的所有索引，创建只包含您想要的 Operator 的源索引副本。

当将 Operator Lifecycle Manager（OLM）配置为在受限网络 OpenShift Container Platform 集群上使用镜像内容时，如果您只想从默认目录中镜像一部分 Operator，请使用此修剪方法。

对于此过程中的步骤，目标 registry 是一个存在的镜像 registry，您的具有无限网络访问权限的工作站可以访问该 registry。本例还显示修剪默认 **redhat-operators** 目录的索引镜像，但所有索引镜像的过程都是一样的。

**先决条件**

- 没有网络访问限制的工作站
- **podman** 版本 1.9.3+
- **grpcurl**（第三方命令行工具）
- **opm** 版本 1.18.0+
- 访问支持 Docker v2-2 的 registry

**重要**

OpenShift Container Platform 集群的内部 registry 不能用作目标 registry，因为它不支持没有标签的推送（在镜像过程中需要这个功能）。

**流程**

1. 通过 **registry.redhat.io** 进行身份验证：

```
$ podman login registry.redhat.io
```

2. 通过目标 registry 进行身份验证：

```
$ podman login <target_registry>
```

3. 确定您要包括在您的修剪索引中的软件包列表。

   a. 运行您要修剪容器中的源索引镜像。例如：

```
$ podman run -p50051:50051 \
    -it registry.redhat.io/redhat/redhat-operator-index:v4.7
```

   输**出示例**

```
Trying to pull registry.redhat.io/redhat/redhat-operator-index:v4.7...
Getting image source signatures
Copying blob ae8a0c23f5b1 done
...
INFO[0000] serving registry                 database=/database/index.db port=50051
```

b. 在一个单独的终端会话中，使用 **grpcurl** 命令获取由索引提供的软件包列表：

```
$ grpcurl -plaintext localhost:50051 api.Registry/ListPackages > packages.out
```

c. 检查 **package.out** 文件，确定要保留在此列表中的哪个软件包名称。例如：

**软件包列表片断示例**

```
...
{
  "name": "advanced-cluster-management"
}
...
{
  "name": "jaeger-product"
}
...
{
{
  "name": "quay-operator"
}
...
```

d. 在您执行 **podman run** 命令的终端会话中，按 **Ctrl** 和 **C** 停止容器进程。

4. 运行以下命令来修剪指定软件包以外的所有源索引：

```
$ opm index prune \
   -f registry.redhat.io/redhat/redhat-operator-index:v4.7 \ 1
   -p advanced-cluster-management,jaeger-product,quay-operator \ 2
   [-i registry.redhat.io/openshift4/ose-operator-registry:v4.7] \ 3
   -t <target_registry>:<port>/<namespace>/redhat-operator-index:v4.7 4
```

1 到修剪的索引。

2 要保留的软件包用逗号隔开。

3 只适用于 IBM Power Systems 和 IBM Z 镜像： Operator Registry 基础镜像和与目标 OpenShift Container Platform 集群主版本和次版本匹配的标签。

4 用于正在构建新索引镜像的自定义标签。

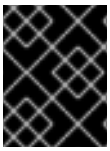
5. 运行以下命令将新索引镜像推送到目标 registry:

```
$ podman push <target_registry>:<port>/<namespace>/redhat-operator-index:v4.7
```

其中 **<namespace>** 是 registry 上的任何现有命名空间。例如，您可以创建一个 **olm-mirror** 命名空间来将所有镜像的内容推送到。

### 4.9.4. 对 Operator 目录进行镜像(mirror)

您可以使用 **oc adm catalog mirror** 命令将红帽提供的目录或自定义目录的 Operator 内容镜像到容器镜像 registry 中。目标 registry 必须支持 Docker v2-2。对于受限网络中的集群，此 registry 可以是集群有网络访问权限的 registry，如在受限网络集群安装过程中创建的镜像 registry。



**重要**

OpenShift Container Platform 集群的内部 registry 不能用作目标 registry，因为它不支持没有标签的推送（在镜像过程中需要这个功能）。

**oc adm catalog mirror** 命令还会自动将在镜像过程中指定的索引镜像（无论是红帽提供的索引镜像还是您自己的自定义构建索引镜像）镜像到目标 registry。然后，您可以使用镜像的索引镜像创建一个目录源，允许 Operator Lifecycle Manager（OLM）将镜像目录加载到 OpenShift Container Platform 集群。

**先决条件**

- 没有网络访问限制的工作站
- **podman** 1.9.3 或更高版本。
- 访问支持 Docker v2-2 的镜像 registry。
- 决定镜像 registry 上用于存储已镜像 Operator 内容的镜像 registry 内容。例如，您可以创建一个 **olm-mirror** 命名空间。
- 如果您的镜像 registry 无法访问互联网，请将可移动介质连接到您的没有网络访问限制的工作站。
- 如果您正在使用私有 registry，包括 **registry.redhat.io**，请将 **REG_CREDS** 环境变量设置为 registry 凭证的文件路径，以便在后续步骤中使用。例如，对于 **podman** CLI:

```
$ REG_CREDS=${XDG_RUNTIME_DIR}/containers/auth.json
```

**流程**

1. 如果要镜像红帽提供的目录，请在具有无网络访问限制的工作站中运行以下命令，以便与 **registry.redhat.io** 进行身份验证：

```
$ podman login registry.redhat.io
```

2. **oc adm catalog mirror** 命令提取索引镜像的内容，以生成镜像所需的清单。命令的默认行为会生成清单，然后会自动将索引镜像以及索引镜像本身中的所有镜像内容镜像（mirror）到您的镜像 registry。另外，如果您的镜像 registry 位于完全断开连接的主机上，或者断开连接的或 *airgapped* 主机上，您可以首先将内容镜像到可移动介质，将介质移到断开连接的环境中，然后将内容从介质镜像到 registry。

   - **选项 A: 如果您的镜像 registry 与您的没有网络访问限制的工作站位于同一个网络中**，请在您的工作站中执行以下操作：

     a. 如果您的镜像 registry 需要身份验证，请运行以下命令登录到 registry:

     ■

```
$ podman login <mirror_registry>
```

b. 运行以下命令镜像内容：

```
$ oc adm catalog mirror \
    <index_image> \ 1
    <mirror_registry>:<port>/<namespace> \ 2
    [-a ${REG_CREDS}] \ 3
    [--insecure] \ 4
    [--index-filter-by-os='<platform>/<arch>'] \ 5
    [--manifests-only] 6
```

1. 指定您要镜像的目录的索引镜像。例如，这可能是之前创建的已修剪索引镜像，也可以是默认目录的源索引镜像之一，如 **registry.redhat.io/redhat/redhat-operator-index:v4.7**。

2. 指定要将 Operator 内容镜像到的目标 registry 和命名空间的完全限定域名 (FQDN)，其中 **<namespace>** 是 registry 上的任何现有命名空间。例如，您可以创建一个 **olm-mirror** 命名空间来将所有镜像的内容推送到。

3. 可选：如果需要，指定 registry 凭证文件的位置。**registry.redhat.io** 需要 **{REG_CREDS}**。

4. 可选：如果您不想为目标 registry 配置信任，请添加 **--insecure** 标志。

5. 可选：在有多个变体可用时，指定索引镜像的平台和架构。镜像被传递为 **'<platform>/<arch>[/<variant>]'**。这不适用于索引引用的镜像。有效值是 **linux/amd64**、**linux/ppc64le** 和 **linux/s390x**。

6. 可选：只生成镜像所需的清单，但并不实际将镜像内容镜像到 registry。这个选项对检查哪些将被镜像（mirror）非常有用，如果您只需要一小部分软件包，可以对映射列表进行修改。然后，您可以使用带有 **oc image mirror** 命令的 **mapping.txt** 文件来在以后的步骤中镜像修改的镜像列表。此标志仅用于从目录中对内容进行高级选择性镜像；**opm index prune** 命令适用于大多数目录管理用例，如果之前用来修剪索引镜像。

**输出示例**

```
src image has index label for database path: /database/index.db
using database path mapping: /database/index.db:/tmp/153048078
wrote database to /tmp/153048078 1
...
wrote mirroring manifests to manifests-redhat-operator-index-1614211642 2
```

1. 命令生成的临时 **index.db** 数据库的目录。

2. 务必记录生成的 manifests 目录名称。该目录名将在以后的步骤中使用。

**注意**

Red Hat Quay 不支持嵌套存储库。因此，运行 **oc adm catalog mirror** 命令会失败，并显示 **401** 未授权错误。作为临时解决方案，您可以在运行 **oc adm catalog mirror** 命令时使用 **--max-components=2** 选项来禁用嵌套存储库的创建。有关这个临时解决方案的更多信息，请参阅使用带有 Quay registry 知识库文章的 catalog mirror 命令时出现 Unauthorized 错误。

- **选项 B：如果您的镜像 registry 位于断开连接的主机上**，请执行以下操作。

  a. 在您的工作站中运行以下命令，且没有网络访问权限将内容镜像到本地文件中：

```
$ oc adm catalog mirror \
    <index_image> \ 1
    file:///local/index \ 2
    [-a ${REG_CREDS}] \
    [--insecure]
```

  1 指定您要镜像的目录的索引镜像。例如，这可能是之前创建的已修剪索引镜像，也可以是默认目录的源索引镜像之一，如 **registry.redhat.io/redhat/redhat-operator-index:v4.7**。

  2 将内容镜像到您当前目录中的本地文件。

  **输出示例**

```
...
info: Mirroring completed in 5.93s (5.915MB/s)
wrote mirroring manifests to manifests-my-index-1614985528 1

To upload local images to a registry, run:

 oc adm catalog mirror file://local/index/myrepo/my-index:v1
REGISTRY/REPOSITORY 2
```

  1 务必记录生成的 manifests 目录名称。该目录名将在以后的步骤中使用。

  2 记录基于您提供的索引镜像展开的 **file://** 路径。该路径在以后的步骤中会使用。

  b. 将当前目录中生成的 **v2/** 目录复制到可移动介质中。

  c. 物理删除该介质并将其附加到断开连接的环境中可访问镜像 registry 的主机。

  d. 如果您的镜像 registry 需要身份验证，请在断开连接的环境中的主机上运行以下命令以登录到 registry：

```
$ podman login <mirror_registry>
```

  e. 从包含 **v2/** 目录的父目录运行以下命令，将镜像从本地文件上传到镜像 registry：

```
$ oc adm catalog mirror \
```

```
file://local/index/<repo>/<index_image>:<tag> \ 1
<mirror_registry>:<port>/<namespace> \ 2
[-a ${REG_CREDS}] \
[--insecure]
```

1 指定上一命令输出中的 **file://** 路径。

2 指定要将 Operator 内容镜像到的目标 registry 和命名空间的完全限定域名 (FQDN)，其中 **<namespace>** 是 registry 上的任何现有命名空间。例如，您可以创建一个 **olm-mirror** 命名空间来将所有镜像的内容推送到。

**注意**

Red Hat Quay 不支持嵌套存储库。因此，运行 **oc adm catalog mirror** 命令会失败，并显示 **401** 未授权错误。作为临时解决方案，您可以在运行 **oc adm catalog mirror** 命令时使用 **--max-components=2** 选项来禁用嵌套存储库的创建。有关这个临时解决方案的更多信息，请参阅使用带有 Quay registry 知识库文章的 catalog mirror 命令时出现 Unauthorized 错误。

f. 再次运行 **oc adm catalog mirror** 命令。使用新镜像的索引镜像作为源，以及上一步中与目标相同的镜像 registry 命名空间：

```
$ oc adm catalog mirror \
    <mirror_registry>:<port>/<index_image> \
    <mirror_registry>:<port>/<namespace> \
    --manifests-only \ 1
    [-a ${REG_CREDS}] \
    [--insecure]
```

1 此步骤需要 **--manifests-only** 标志，以便该命令不会再次复制所有镜像的内容。

**重要**

这一步是必需的，因为上一步中生成的 **imageContentSourcePolicy.yaml** 文件中的镜像映射必须从本地路径更新为有效的镜像位置。如果不这样做，会在稍后的步骤中创建 **ImageContentSourcePolicy** 对象时会导致错误。

3. 将内容镜像到 registry 后，检查当前目录中生成的清单目录。

**注意**

清单目录名会在以后的步骤中使用。

如果您在上一步中将内容镜像到同一网络上的 registry，则目录名称采用以下格式：

```
manifests-<index_image_name>-<random_number>
```

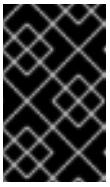
如果您在上一步中将内容镜像到断开连接的主机上的 registry，则目录名称采用以下格式：

-

```
manifests-index/<namespace>/<index_image_name>-<random_number>
```

manifests 目录包含以下文件，其中的一些文件可能需要进一步修改：

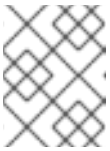
- **catalogSource.yaml** 文件是 **CatalogSource** 对象的基本定义，它预先填充索引镜像标签及其他相关元数据。此文件可原样使用，或进行相应修改来在集群中添加目录源。



**重要**

如果将内容镜像到本地文件，您必须修改 catalogSource **.yaml** 文件，从 **metadata.name** 字段中删除任何反斜杠(/)字符。否则，当您试图创建对象时，会失败并显示 "invalid resource name" 错误。

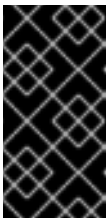
- 用来定义 **ImageContentSourcePolicy** 对象的 **imageContentSourcePolicy.yaml**，它可以将节点配置为在 Operator 清单中存储的镜像（image）引用和镜像 (mirror) 的 registry 间进行转换。



**注意**

如果您的集群使用 **ImageContentSourcePolicy** 对象来配置存储库镜像，则只能将全局 pull secret 用于镜像 registry。您不能在项目中添加 pull secret。

- **mapping.txt** 文件，在其中包含所有源镜像，并将它们映射到目标 registry。此文件与 **oc image mirror** 命令兼容，可用于进一步自定义镜像（mirror）配置。



**重要**

如果您在镜像过程中使用 **--manifests-only** 标志，并希望进一步调整要镜像的软件包子集，请参阅"镜像软件包清单格式目录镜像"中有关修改 **mapping.txt** 文件的步骤，并使用 **oc image mirror** 命令使用该文件。在进行了这些操作后，您可以继续这个过程。

4. 在可访问断开连接的集群的主机上，运行以下命令来指定 manifests 目录中的 **imageContentSourcePolicy.yaml** 文件，创建 **ImageContentSourcePolicy** (ICSP) 对象：

```
$ oc create -f <path/to/manifests/dir>/imageContentSourcePolicy.yaml
```

其中 **<path/to/manifests/dir>** 是镜像内容的 manifests 目录的路径。

**注意**

应用 ICSP 会导致集群中的所有 worker 节点重启。在继续操作前，您必须等待此重启过程完成每个 worker 节点的循环。

现在，您可以创建一个 **CatalogSource** 来引用您的镜像索引镜像和 Operator 内容。

**其它资源**

- 为断开连接的安装 mirror 镜像
- Operator 的架构和操作系统支持
- 对 Package Manifest Format 目录镜像进行镜像(mirror)

### 4.9.5. 从索引镜像创建目录

您可以从索引镜像创建 Operator 目录，并将其应用到 OpenShift Container Platform 集群，供 Operator Lifecycle Manager（OLM）使用。

**先决条件**

- 构建并推送到 registry 的索引镜像。

**流程**

1. 创建一个 **CatalogSource** 对象来引用索引镜像。如果使用 **oc adm catalog mirror** 命令将目录镜像到目标 registry，您可以使用生成的 **catalogSource.yaml** 文件作为起点。

   a. 根据您的规格修改以下内容，并将它保存为 **catalogSource.yaml** 文件：

```
apiVersion: operators.coreos.com/v1alpha1
kind: CatalogSource
metadata:
  name: my-operator-catalog (1)
  namespace: openshift-marketplace (2)
spec:
  sourceType: grpc
  image: <registry>:<port>/<namespace>/redhat-operator-index:v4.7 (3)
  displayName: My Operator Catalog
  publisher: <publisher_name> (4)
  updateStrategy:
    registryPoll: (5)
      interval: 30m
```

   (1) 如果您在上传到 registry 前将内容镜像到本地文件，请从 **metadata.name** 字段中删除任何反斜杠(/)字符，以避免在创建对象时出现 "invalid resource name" 错误。

   (2) 如果您希望目录源对所有命名空间中的用户全局可用，请指定 **openshift-marketplace** 命名空间。否则，您可以指定一个不同的命名空间来对目录进行作用域并只对该命名空间可用。

   (3) 指定索引镜像。

   (4) 指定发布目录的名称或机构名称。

   (5) 目录源可以自动检查新版本以保持最新。

   b. 使用该文件创建 **CatalogSource** 对象：

```
$ oc apply -f catalogSource.yaml
```

2. 确定成功创建以下资源。

   a. 检查 pod:

```
$ oc get pods -n openshift-marketplace
```

   **输出示例**

```
NAME                                READY   STATUS    RESTARTS  AGE
my-operator-catalog-6njx6           1/1     Running   0         28s
marketplace-operator-d9f549946-96sgr  1/1     Running   0         26h
```

b. 检查目录源：

```
$ oc get catalogsource -n openshift-marketplace
```

**输出示例**

```
NAME                  DISPLAY               TYPE PUBLISHER  AGE
my-operator-catalog   My Operator Catalog   grpc           5s
```

c. 检查软件包清单：

```
$ oc get packagemanifest -n openshift-marketplace
```

**输出示例**

```
NAME                  CATALOG               AGE
jaeger-product                My Operator Catalog   93s
```

现在，您可以在 OpenShift Container Platform Web 控制台中通过 **OperatorHub** 安装 Operator。

**其他资源**

- 如果您的索引镜像托管在私有 registry 上且需要进行身份验证，请参阅 从私有 registry 访问 Operator 的镜像。

### 4.9.6. 更新索引镜像

在将 OperatorHub 配置为使用引用自定义索引镜像的目录源后，集群管理员可通过将捆绑包镜像添加到索引镜像来保持其集群上的可用 Operator 最新状态。

您可以使用 **opm index add** 命令来更新存在的索引镜像。对于受限网络，还必须将更新的内容重新镜像到集群。

**先决条件**

- **opm** 版本 1.12.3+
- **podman** 版本 1.9.3+
- 构建并推送到 registry 的索引镜像。
- 引用索引镜像的现有目录源。

**流程**

1. 通过添加捆绑包镜像来更新现有索引：

```
$ opm index add \
```

```
--bundles <registry>/<namespace>/<new_bundle_image>@sha256:<digest> \ 1
--from-index <registry>/<namespace>/<existing_index_image>:<existing_tag> \ 2
--tag <registry>/<namespace>/<existing_index_image>:<updated_tag> \ 3
--pull-tool podman 4
```

1. **--bundles** 标志指定要添加到索引中的、以逗号分隔的额外捆绑包镜像列表。
2. **--from-index** 标志指定之前推送的索引。
3. **--tag** 标志指定要应用到更新的索引镜像的镜像标签。
4. **--pull-tool** 标志指定用于拉取容器镜像的工具。

其中：

**<registry>**

指定 registry 的主机名，如 **quay.io** 或 **mirror.example.com**。

**<namespace>**

指定 registry 的命名空间，如 **ocs-dev** 或 **abc**。

**<new_bundle_image>**

指定要添加到 registry 的新捆绑包镜像，如 **ocs-operator**。

**<digest>**

指定捆绑包镜像的 SHA 镜像 ID 或摘要，如 **c7f11097a628f092d8bad148406aa0e0951094a03445fd4bc0775431ef683a41**。

**<existing_index_image>**

指定之前推送的镜像，如 **abc-redhat-operator-index**。

**<existing_tag>**

指定之前推送的镜像标签，如 **4.7**。

**<updated_tag>**

指定要应用到更新的索引镜像的镜像标签，如 **4.7.1**。

**示例命令**

```
$ opm index add \
   --bundles quay.io/ocs-dev/ocs-
operator@sha256:c7f11097a628f092d8bad148406aa0e0951094a03445fd4bc0775431ef683a
41 \
   --from-index mirror.example.com/abc/abc-redhat-operator-index:4.7 \
   --tag mirror.example.com/abc/abc-redhat-operator-index:4.7.1 \
   --pull-tool podman
```

2. 推送更新的索引镜像：

```
$ podman push <registry>/<namespace>/<existing_index_image>:<updated_tag>
```

3. 按照 *镜像 Operator 目录*流程中的步骤来镜像更新的内容。但是，当进入创建 **ImageContentSourcePolicy**（ICSP）对象的步骤时，请使用 **oc replace** 命令而不是 **oc create** 命令。例如：

```
$ oc replace -f ./manifests-redhat-operator-index-
<random_number>/imageContentSourcePolicy.yaml
```

这一更改是必需的，因为对象已存在且必须更新。

**注意**

通常，**oc apply** 命令可用于更新之前使用 **oc apply** 创建的现有对象。但是，由于有关 ICSP 对象中的 **metadata.annotations** 字段大小的已知问题，**oc replace** 命令当前必须用于此步骤。

4. Operator Lifecycle Manager（OLM）会在常规时间段内自动轮询目录源中引用的索引镜像，验证是否已成功添加新软件包：

```
$ oc get packagemanifests -n openshift-marketplace
```

**其它资源**

- 对 Operator 目录进行镜像(mirror)

# 第 5 章 开发 OPERATOR

## 5.1. 关于 OPERATOR SDK

Operator Framework 是一个开源工具包，用于以有效、自动化且可扩展的方式管理 Kubernetes 原生应用程序，即 *Operator*。Operator 利用 Kubernetes 的可扩展性来展现云服务的自动化优势，如置备、扩展以及备份和恢复，同时能够在 Kubernetes 可运行的任何地方运行。

Operator 有助于简化对 Kubernetes 上的复杂、有状态的应用程序的管理。然而，现在编写 Operator 并不容易，会面临一些挑战，如使用低级别 API、编写样板文件以及缺乏模块化功能（这会导致重复工作）。

Operator SDK 是 Operator Framework 的一个组件，它提供了一个命令行界面（CLI）工具，供 Operator 开发人员用来构建、测试和部署 Operator。

**为什么使用 Operator SDK?**

Operator SDK 简化了这一构建 Kubernetes 原生应用程序的过程，它需要深入掌握特定于应用程序的操作知识。Operator SDK 不仅降低了这一障碍，而且有助于减少许多常见管理功能（如 metering 或监控）所需的样板代码量。

Operator SDK 是一个框架，它使用 controller-runtime 库来简化 Operator 的编写，并具有以下特色：

- 高级 API 和抽象，用于更直观地编写操作逻辑
- 支架和代码生成工具，用于快速引导新项目
- 与 Operator Lifecycle Manager（OLM）集成，简化了集群上的打包、安装和运行的 Operator
- 扩展项，覆盖常见的 Operator 用例
- 指标（metrics）在基于 Go 的 Operator 中自动设置，用于部署 Prometheus Operator 的集群

具有集群管理员访问权限的 operator 作者（如 OpenShift Container Platform）可以使用 Operator SDK CLI 根据 Go、Ansible 或 Helm 开发自己的 Operator。Kubebuilder 作为基于 Go 的 Operator 的构建解决方案嵌入到 Operator SDK 中，这意味着现有的 Kubebuilder 项目可以象 Operator SDK 一样使用并继续工作。

**注意**

OpenShift Container Platform 4.7 支持 Operator SDK v1.3.0 或更高版本。

### 5.1.1. 什么是 Operator?

有关基本 Operator 概念和术语的概述，请参阅了解 Operator。

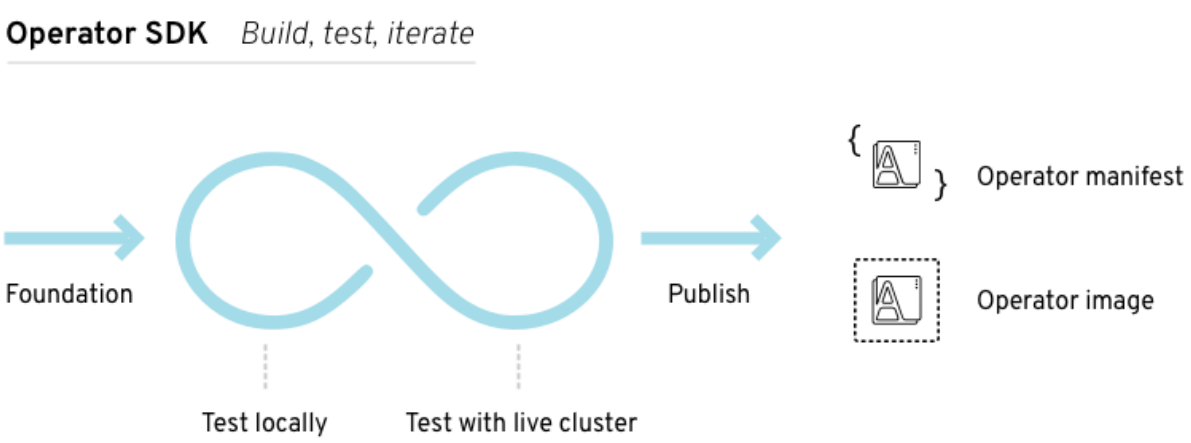
### 5.1.2. 开发工作流

Operator SDK 提供以下工作流来开发新的 Operator：

1. 使用 Operator SDK 命令行界面（CLI）创建 Operator 项目。
2. 通过添加自定义资源定义 (CRD) 来定义新的资源 API。
3. 使用 Operator SDK API 指定要监视的资源。

4. 在指定的处理程序中定义 Operator 协调逻辑，并使用 Operator SDK API 与资源交互。
5. 使用 Operator SDK CLI 来构建和生成 Operator 部署清单。

**图 5.1. Operator SDK 工作流**



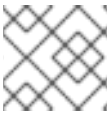

在高级别上，使用 Operator SDK 的 Operator 会在 Operator 作者定义的处理程序中处理与被监视资源相关的事件，并采取措施协调应用程序的状态。

### 5.1.3. 其他资源

- 认证的 Operator 构建指南

## 5.2. 安装 OPERATOR SDK CLI

Operator SDK 提供了一个命令行界面（CLI）工具，Operator 开发人员可使用它来构建、测试和部署 Operator。您可以在工作站上安装 Operator SDK CLI，以便准备开始编写自己的 Operator。



**注意**

OpenShift Container Platform 4.7 支持 Operator SDK v1.3.0。

### 5.2.1. 安装 Operator SDK CLI

您可以在 Linux 上安装 OpenShift SDK CLI 工具。

**先决条件**

- Go v1.13+
- **docker** v17.03+、**podman** v1.9.3+ 或 **buildah** v1.7+

**流程**

1. 导航到 OpenShift 镜像站点。
2. 在 **4.7.23** 目录中下载 Linux 的 tarball 的最新版本。
3. 解包存档：

```
$ tar xvf operator-sdk-v1.3.0-ocp-linux-x86_64.tar.gz
```

4. 使文件可执行：

```
$ chmod +x operator-sdk
```

5. 将提取的 **operator-sdk** 二进制文件移到 **PATH** 中的一个目录中。

**提示**

检查 **PATH**：

```
$ echo $PATH
```

```
$ sudo mv ./operator-sdk /usr/local/bin/operator-sdk
```

**验证**

- 安装 Operator SDK CLI 后，验证它是否可用：

```
$ operator-sdk version
```

**输出示例**

```
operator-sdk version: "v1.3.0-ocp", ...
```

## 5.3. 基于 GO 的 OPERATOR

### 5.3.1. 基于 Go 的 Operator 开始使用 Operator SDK

如需演示使用 Operator SDK 提供的工具和库来设置和运行基于 Go 的 Operator 的基本知识，Operator 开发人员可以为 Memcached 构建 Go-based Operator 示例，一个分布式键值存储，并将它部署到集群中。

#### 5.3.1.1. 先决条件

- 已安装 operator SDK CLI
- 已安装 OpenShift CLI（**oc**）v4.7+
- 使用具有 **cluster-admin** 权限的 **oc** 登录到 OpenShift Container Platform 4.7 集群
- 要允许集群拉取镜像，推送镜像的存储库必须设置为公共的存储库，或必须配置一个镜像 pull secret。

#### 5.3.1.2. 创建并部署基于 Go 的 Operator

您可以使用 Operator SDK 为 Memcached 构建和部署简单的 Go-based Operator。

**流程**

1. **创建一个项目。**

   a. 创建您的项目目录：

   ```
   $ mkdir memcached-operator
   ```

   b. 切换到项目所在的目录：

   ```
   $ cd memcached-operator
   ```

   c. 运行 **operator-sdk init** 命令以初始化项目：

   ```
   $ operator-sdk init \
     --domain=example.com \
     --repo=github.com/example-inc/memcached-operator
   ```

   命令默认使用 Go 插件。

   d. 要启用基于 Go 的 Operator 在 OpenShift Container Platform 上运行，编辑 **config/manager/manager.yaml** 文件并替换以下行：

   ```
   runAsUser: 65532
   ```

   使用：

   ```
   runAsNonRoot: true
   ```

   **注意**

   此步骤只是针对 Go 的 Operator 的一个临时解决方案。如需更多信息，请参阅 BZ#1914406。

2. **创建 API。**
   创建简单的 Memcached API：

   ```
   $ operator-sdk create api \
     --resource=true \
     --controller=true \
     --group cache \
     --version v1 \
     --kind Memcached
   ```

3. **构建并推送 Operator 镜像。**
   使用默认的 **Makefile** 目标来构建和推送 Operator。使用镜像的 pull spec 设置 **IMG**，该 spec 使用您可推送到的 registry：

   ```
   $ make docker-build docker-push IMG=<registry>/<user>/<image_name>:<tag>
   ```

4. **运行 Operator。**

   a. 安装 CRD：

```
$ make install
```

b. 将项目部署到集群中。将 **IMG** 设置为您推送的镜像：

```
$ make deploy IMG=<registry>/<user>/<image_name>:<tag>
```

5. **创建示例自定义资源（CR）。**

a. 创建一个示例 CR：

```
$ oc apply -f config/samples/cache_v1_memcached.yaml \
    -n memcached-operator-system
```

b. 查看 CR 协调 Operator：

```
$ oc logs deployment.apps/memcached-operator-controller-manager \
    -c manager \
    -n memcached-operator-system
```

6. **清理。**
运行以下命令清理在此流程中创建的资源：

```
$ make undeploy
```

#### 5.3.1.3. 后续步骤

- 如需更深入地了解如何构建基于 Go 的 Operator，请参阅基于 Go 的 Operator SDK 指南 。

### 5.3.2. 基于 Go 的 Operator 的 operator SDK 指南

Operator SDK 中的 Go 编程语言支持可以利用 Operator SDK 中的 Go 编程语言支持，为 Memcached 构建基于 Go 的 Operator 示例、分布式键值存储并管理其生命周期。

通过以下两个 Operator Framework 核心组件来完成此过程：

**Operator SDK**

**operator-sdk** CLI 工具和 **controller-runtime** 库 API

**Operator Lifecycle Manager (OLM)**

集群中 Operator 的安装、升级和基于角色的访问控制（RBAC）

**注意**

本教程的内容比基于 Go 的 Operator 开始使用 Operator SDK 更详细。

#### 5.3.2.1. 先决条件

- 已安装 operator SDK CLI
- 已安装 OpenShift CLI（**oc**）v4.7+
- 使用具有 **cluster-admin** 权限的 **oc** 登录到 OpenShift Container Platform 4.7 集群

- 要允许集群拉取镜像，推送镜像的存储库必须设置为公共的存储库，或必须配置一个镜像 pull secret。

#### 5.3.2.2. 创建一个项目

使用 Operator SDK CLI 创建名为 **memcached-operator 的** 项目。

**流程**

1. 为项目创建一个目录：

```
$ mkdir -p $HOME/projects/memcached-operator
```

2. 进入该目录：

```
$ cd $HOME/projects/memcached-operator
```

3. 激活对 Go 模块的支持：

```
$ export GO111MODULE=on
```

4. 运行 **operator-sdk init** 命令以初始化项目：

```
$ operator-sdk init \
    --domain=example.com \
    --repo=github.com/example-inc/memcached-operator
```

**注意**

**operator-sdk init** 命令默认使用 Go 插件。

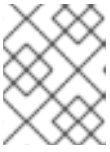
**operator-sdk init** 命令生成一个 **go.mod** 文件，用于 Go 模块。在创建 **$GOPATH/src/** 项目时需要 **--repo** 标志，因为生成的文件需要有效的模块路径。

5. 要启用基于 Go 的 Operator 在 OpenShift Container Platform 上运行，编辑 **config/manager/manager.yaml** 文件并替换以下行：

```
runAsUser: 65532
```

使用：

```
runAsNonRoot: true
```



**注意**

此步骤只是针对 Go 的 Operator 的一个临时解决方案。如需更多信息，请参阅 BZ#1914406。

##### 5.3.2.2.1. PROJECT 文件

**operator-sdk init** 命令生成的文件中是一个 Kubebuilder **PROJECT** 文件。从项目 root 运行的后续 **operator-sdk** 命令以及 **help** 输出会读取该文件，并注意到项目的类型为 Go。例如：

```
domain: example.com
layout: go.kubebuilder.io/v3
projectName: memcached-operator
repo: github.com/example-inc/memcached-operator
version: 3-alpha
plugins:
  manifests.sdk.operatorframework.io/v2: {}
  scorecard.sdk.operatorframework.io/v2: {}
```

#### 5.3.2.2.2. 关于 Manager

Operator 的主要程序是 **main.go** 文件，它初始化并运行 Manager。Manager 会自动注册所有自定义资源（CR）API 定义的方案，并设置和运行控制器和 webhook。

Manager 可以限制所有控制器监视资源的命名空间：

```
mgr, err := ctrl.NewManager(cfg, manager.Options{Namespace: namespace})
```

默认情况下， Manager 会监视 Operator 的运行命名空间。要监视所有命名空间，您可以将 **namespace** 选项留空：

```
mgr, err := ctrl.NewManager(cfg, manager.Options{Namespace: ""})
```

您还可以使用 **MultiNamespacedCacheBuilder** 功能监控特定命名空间集合：

```
var namespaces []string (1)
mgr, err := ctrl.NewManager(cfg, manager.Options{ (2)
  NewCache: cache.MultiNamespacedCacheBuilder(namespaces),
})
```

1. 命名空间列表。
2. 创建 **Cmd** 指令以提供共享依赖关系和启动组件。

#### 5.3.2.2.3. 关于多组 API

在创建 API 和控制器前，请考虑您的 Operator 是否需要多个 API 组。本教程涵盖了单个组 API 的默认情况，但要更改项目布局来支持多组 API，您可以运行以下命令：

```
$ operator-sdk edit --multigroup=true
```

这个命令更新了 **PROJECT** 文件，该文件应该类似以下示例：

```
domain: example.com
layout: go.kubebuilder.io/v3
multigroup: true
...
```

对于多组项目，API Go 类型文件会在 **apis/<group> /<version>** / 目录中创建，控制器在 **controllers/<group>** / 目录中创建。然后会相应地更新 Dockerfile。

**其他资源**

- 有关迁移到多组项目的详情，请参阅 Kubebuilder 文档。

#### 5.3.2.3. 创建 API 和控制器

使用 Operator SDK CLI 创建自定义资源定义（CRD）API 和控制器。

**流程**

1. 运行以下命令创建带有组 **cache**、版本 **v1** 和种类 **Memcached** 的 API：

```
$ operator-sdk create api \
    --group=cache \
    --version=v1 \
    --kind=Memcached
```

2. 提示时，输入 **y** 来创建资源和控制器：

```
Create Resource [y/n]
y
Create Controller [y/n]
y
```

**输出示例**

```
Writing scaffold for you to edit...
api/v1/memcached_types.go
controllers/memcached_controller.go
...
```

此过程会在 **api/v1/memcached_types.go** 和 **controllers/memcached_controller.go** 中生成 **Memcached** 资源 API。

##### 5.3.2.3.1. 定义 API

定义 **Memcached** 自定义资源（CR）的 API。

**流程**

1. 修改 **api/v1/memcached_types.go** 中的 Go 类型定义，使其具有以下 **spec** 和 **status**：

```
// MemcachedSpec defines the desired state of Memcached
type MemcachedSpec struct {
 // +kubebuilder:validation:Minimum=0
 // Size is the size of the memcached deployment
 Size int32 `json:"size"`
}

// MemcachedStatus defines the observed state of Memcached
type MemcachedStatus struct {
 // Nodes are the names of the memcached pods
 Nodes []string `json:"nodes"`
}
```

–

2. 为资源类型更新生成的代码：

```
$ make generate
```

**提示**

在修改了 ***_types.go** 文件后，您必须运行 **make generate** 命令来更新该资源类型生成的代码。

以上 Makefile 目标调用 **controller-gen** 程序来更新 **api/v1/zz_generated.deepcopy.go** 文件。这样可确保您的 API Go 类型定义实现了 **runtime.Object** 接口，所有 Kind 类型都必须实现。

#### 5.3.2.3.2. 生成 CRD 清单

在使用 **spec** 和 **status** 字段和自定义资源定义（CRD）验证标记定义后，您可以生成 CRD 清单。

**流程**

- 运行以下命令以生成和更新 CRD 清单：

```
$ make manifests
```

此 Makefile 目标调用 **controller-gen** 实用程序在 **config/crd/bases/cache.example.com_memcacheds.yaml** 文件中生成 CRD 清单。

##### 5.3.2.3.2.1. 关于 OpenAPI 验证

当生成清单时，openAPIV3 模式会添加到 **spec.validation** 块中的 CRD 清单中。此验证块允许 Kubernetes 在 Memcached 自定义资源（CR）创建或更新时验证其中的属性。

标记或注解可用于为您的 API 配置验证。这些标记始终具有 **+kubebuilder:validation** 前缀。

**其他资源**

- 如需有关在 API 代码中使用标记的更多详细信息，请参阅以下 Kubebuilder 文档：
  - CRD 生成
  - Markers（标记）
  - OpenAPIv3 验证标记列表
- 有关 CRD 中的 OpenAPIv3 验证模式的详情，请参阅 Kubernetes 文档。

### 5.3.2.4. 实现控制器

在创建新 API 和控制器后，您可以实现控制器逻辑。

**流程**

- 在本例中，将生成的控制器文件 **controllers/memcached_controller.go** 替换为以下示例实现：

**例 5.1. memcached_controller.go示例**

```
/*
Copyright 2020.

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
*/

package controllers

import (
 appsv1 "k8s.io/api/apps/v1"
 corev1 "k8s.io/api/core/v1"
 "k8s.io/apimachinery/pkg/api/errors"
 metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
 "k8s.io/apimachinery/pkg/types"
 "reflect"

 "context"

 "github.com/go-logr/logr"
 "k8s.io/apimachinery/pkg/runtime"
 ctrl "sigs.k8s.io/controller-runtime"
 "sigs.k8s.io/controller-runtime/pkg/client"

 cachev1alpha1 "github.com/example/memcached-operator/api/v1alpha1"
)

// MemcachedReconciler reconciles a Memcached object
type MemcachedReconciler struct {
 client.Client
 Log    logr.Logger
 Scheme *runtime.Scheme
}

//
+kubebuilder:rbac:groups=cache.example.com,resources=memcacheds,verbs=get;list;watch
;create;update;patch;delete
//
+kubebuilder:rbac:groups=cache.example.com,resources=memcacheds/status,verbs=get;up
date;patch
//
+kubebuilder:rbac:groups=cache.example.com,resources=memcacheds/finalizers,verbs=upd
ate
//
+kubebuilder:rbac:groups=apps,resources=deployments,verbs=get;list;watch;create;update;
patch;delete
// +kubebuilder:rbac:groups=core,resources=pods,verbs=get;list;
```

```
// Reconcile is part of the main kubernetes reconciliation loop which aims to
// move the current state of the cluster closer to the desired state.
// TODO(user): Modify the Reconcile function to compare the state specified by
// the Memcached object against the actual cluster state, and then
// perform operations to make the cluster state reflect the state specified by
// the user.
//
// For more details, check Reconcile and its Result here:
// - https://pkg.go.dev/sigs.k8s.io/controller-runtime@v0.7.0/pkg/reconcile
func (r *MemcachedReconciler) Reconcile(ctx context.Context, req ctrl.Request)
(ctrl.Result, error) {
 log := r.Log.WithValues("memcached", req.NamespacedName)

 // Fetch the Memcached instance
 memcached := &cachev1alpha1.Memcached{}
 err := r.Get(ctx, req.NamespacedName, memcached)
 if err != nil {
  if errors.IsNotFound(err) {
   // Request object not found, could have been deleted after reconcile request.
   // Owned objects are automatically garbage collected. For additional cleanup logic use
finalizers.
   // Return and don't requeue
   log.Info("Memcached resource not found. Ignoring since object must be deleted")
   return ctrl.Result{}, nil
  }
  // Error reading the object - requeue the request.
  log.Error(err, "Failed to get Memcached")
  return ctrl.Result{}, err
 }

 // Check if the deployment already exists, if not create a new one
 found := &appsv1.Deployment{}
 err = r.Get(ctx, types.NamespacedName{Name: memcached.Name, Namespace:
memcached.Namespace}, found)
 if err != nil && errors.IsNotFound(err) {
  // Define a new deployment
  dep := r.deploymentForMemcached(memcached)
  log.Info("Creating a new Deployment", "Deployment.Namespace", dep.Namespace,
"Deployment.Name", dep.Name)
  err = r.Create(ctx, dep)
  if err != nil {
   log.Error(err, "Failed to create new Deployment", "Deployment.Namespace",
dep.Namespace, "Deployment.Name", dep.Name)
   return ctrl.Result{}, err
  }
  // Deployment created successfully - return and requeue
  return ctrl.Result{Requeue: true}, nil
 } else if err != nil {
  log.Error(err, "Failed to get Deployment")
  return ctrl.Result{}, err
 }

 // Ensure the deployment size is the same as the spec
 size := memcached.Spec.Size
 if *found.Spec.Replicas != size {
```

```
 found.Spec.Replicas = &size
 err = r.Update(ctx, found)
 if err != nil {
  log.Error(err, "Failed to update Deployment", "Deployment.Namespace",
found.Namespace, "Deployment.Name", found.Name)
  return ctrl.Result{}, err
 }
 // Spec updated - return and requeue
 return ctrl.Result{Requeue: true}, nil
}

// Update the Memcached status with the pod names
// List the pods for this memcached's deployment
podList := &corev1.PodList{}
listOpts := []client.ListOption{
 client.InNamespace(memcached.Namespace),
 client.MatchingLabels(labelsForMemcached(memcached.Name)),
}
if err = r.List(ctx, podList, listOpts...); err != nil {
 log.Error(err, "Failed to list pods", "Memcached.Namespace", memcached.Namespace,
"Memcached.Name", memcached.Name)
 return ctrl.Result{}, err
}
podNames := getPodNames(podList.Items)

// Update status.Nodes if needed
if !reflect.DeepEqual(podNames, memcached.Status.Nodes) {
 memcached.Status.Nodes = podNames
 err := r.Status().Update(ctx, memcached)
 if err != nil {
  log.Error(err, "Failed to update Memcached status")
  return ctrl.Result{}, err
 }
}

return ctrl.Result{}, nil
}

// deploymentForMemcached returns a memcached Deployment object
func (r *MemcachedReconciler) deploymentForMemcached(m
*cachev1alpha1.Memcached) *appsv1.Deployment {
ls := labelsForMemcached(m.Name)
replicas := m.Spec.Size

dep := &appsv1.Deployment{
 ObjectMeta: metav1.ObjectMeta{
  Name:      m.Name,
  Namespace: m.Namespace,
 },
 Spec: appsv1.DeploymentSpec{
  Replicas: &replicas,
  Selector: &metav1.LabelSelector{
   MatchLabels: ls,
  },
  Template: corev1.PodTemplateSpec{
   ObjectMeta: metav1.ObjectMeta{
```

```
      Labels: ls,
     },
     Spec: corev1.PodSpec{
      Containers: []corev1.Container{{
       Image:   "memcached:1.4.36-alpine",
       Name:    "memcached",
       Command: []string{"memcached", "-m=64", "-o", "modern", "-v"},
       Ports: []corev1.ContainerPort{{
        ContainerPort: 11211,
        Name:          "memcached",
       }},
      }},
     },
    },
   },
  }
  // Set Memcached instance as the owner and controller
  ctrl.SetControllerReference(m, dep, r.Scheme)
  return dep
}

// labelsForMemcached returns the labels for selecting the resources
// belonging to the given memcached CR name.
func labelsForMemcached(name string) map[string]string {
  return map[string]string{"app": "memcached", "memcached_cr": name}
}

// getPodNames returns the pod names of the array of pods passed in
func getPodNames(pods []corev1.Pod) []string {
  var podNames []string
  for _, pod := range pods {
   podNames = append(podNames, pod.Name)
  }
  return podNames
}

// SetupWithManager sets up the controller with the Manager.
func (r *MemcachedReconciler) SetupWithManager(mgr ctrl.Manager) error {
  return ctrl.NewControllerManagedBy(mgr).
   For(&cachev1alpha1.Memcached{}).
   Owns(&appsv1.Deployment{}).
   Complete(r)
}
```

示例控制器为每个 **Memcached** 自定义资源（CR）运行以下协调逻辑：

- 如果尚无 Memcached 部署，创建一个。
- 确保部署大小与 **Memcached** CR spec 指定的大小相同。
- 使用 **memcached** Pod 的名称更新 **Memcached** CR 状态。

下面的小节解释了示例中的控制器如何监视资源以及如何触发协调循环。您可以跳过这些小节以，直接进入运行 Operator。

##### 5.3.2.4.1. 控制器监视的资源

**controllers/memcached_controller.go** 中的 **SetupWithManager()** 功能指定如何构建控制器来监视 CR 和其他控制器拥有和管理的资源。

```
import (
 ...
 appsv1 "k8s.io/api/apps/v1"
 ...
)

func (r *MemcachedReconciler) SetupWithManager(mgr ctrl.Manager) error {
 return ctrl.NewControllerManagedBy(mgr).
  For(&cachev1.Memcached{}).
  Owns(&appsv1.Deployment{}).
  Complete(r)
}
```

**NewControllerManagedBy()** 提供了一个控制器构建器，它允许各种控制器配置。

**for(&cachev1.Memcached{})** 将 **Memcached** 类型指定为要监视的主要资源。对于 **Memcached** 类型的每个 Add、Update 或 Delete 事件，协调循环都会为该 **Memcached** 对象发送一个协调 **Request** 参数，其中包括命名空间和名称键。

**owns(&appsv1.Deployment{})** 将 **Deployment** 类型指定为要监视的辅助资源。对于 **Deployment** 类型的每个 Add、Update 或 Delete 事件，事件处理程序会将每个事件映射到部署所有者的协调请求。在本例中，所有者是创建部署的 **Memcached** 对象。

##### 5.3.2.4.2. 控制器配置

您可以不同的配置来初始化控制器。例如：

- 使用 **MaxConcurrentReconciles** 选项设置控制器的并发协调的最大数量，其默认值为 **1**：

```
func (r *MemcachedReconciler) SetupWithManager(mgr ctrl.Manager) error {
    return ctrl.NewControllerManagedBy(mgr).
        For(&cachev1.Memcached{}).
        Owns(&appsv1.Deployment{}).
        WithOptions(controller.Options{
            MaxConcurrentReconciles: 2,
        }).
        Complete(r)
}
```

- 使用 predicates 过滤监视事件。
- 选择 EventHandler 类型来更改监视事件转换方式以协调协调循环的请求。对于比主和从属资源更复杂的 Operator 关系，您可以使用 **EnqueueRequestsFromMapFunc** 处理程序将监控事件转换为一组任意协调请求。

有关这些配置和其他配置的详情，请参阅上游 Builder 和 Controller GoDocs。

##### 5.3.2.4.3. 协调循环

每个控制器都有一个协调器对象，它带有实现了协调循环的 **Reconcile()** 方法。协调循环通过 **Request** 参数传递，该参数是从缓存中查找主资源对象 **Memcached** 的命名空间和名称键：

```
import (
 ctrl "sigs.k8s.io/controller-runtime"

 cachev1 "github.com/example-inc/memcached-operator/api/v1"
 ...
)

func (r *MemcachedReconciler) Reconcile(ctx context.Context, req ctrl.Request) (ctrl.Result, error) {
  // Lookup the Memcached instance for this reconcile request
  memcached := &cachev1.Memcached{}
  err := r.Get(ctx, req.NamespacedName, memcached)
  ...
}
```

根据返回值、结果和错误，请求可能会重新排序，协调循环可能会再次触发：

```
// Reconcile successful - don't requeue
return ctrl.Result{}, nil
// Reconcile failed due to error - requeue
return ctrl.Result{}, err
// Requeue for any reason other than an error
return ctrl.Result{Requeue: true}, nil
```

您可以将 **Result.RequeueAfter** 设置为在宽限期后重新排序请求：

```
import "time"

// Reconcile for any reason other than an error after 5 seconds
return ctrl.Result{RequeueAfter: time.Second*5}, nil
```

**注意**

您可以返回带有 **RequeueAfter** 设置的 **Result** 来定期协调一个 CR。

有关协调器、客户端并与资源事件交互的更多信息，请参阅 Controller Runtime Client API 文档。

#### 5.3.2.4.4. 权限和 RBAC 清单

控制器需要特定的 RBAC 权限与它管理的资源交互。它们通过 RBAC 标记来指定，如下所示：

```
//
+kubebuilder:rbac:groups=cache.example.com,resources=memcacheds,verbs=get;list;watch;create;upd
ate;patch;delete
//
+kubebuilder:rbac:groups=cache.example.com,resources=memcacheds/status,verbs=get;update;patch

// +kubebuilder:rbac:groups=cache.example.com,resources=memcacheds/finalizers,verbs=update
//
+kubebuilder:rbac:groups=apps,resources=deployments,verbs=get;list;watch;create;update;patch;delete

// +kubebuilder:rbac:groups=core,resources=pods,verbs=get;list;
```

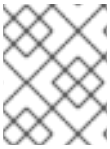
```
func (r *MemcachedReconciler) Reconcile(ctx context.Context, req ctrl.Request) (ctrl.Result, error) {
  ...
}
```

**config/rbac/role.yaml** 中的 **ClusterRole** 对象清单通过在每次运行 **manifests** 命令时使用 **controller-gen** 实用程序的以前的标记生成。

#### 5.3.2.5. 运行 Operator

您可以使用 Operator SDK CLI 构建和运行 Operator：

- 作为 Go 程序在集群外本地运行。
- 作为集群的部署运行。
- 捆绑 Operator，并使用 Operator Lifecycle Manager（OLM）在集群中部署。



**注意**

在将基于 Go 的 Operator 作为在 OpenShift Container Platform 上部署或作为使用 OLM 的捆绑包运行之前，请确保您的项目已更新为使用支持的镜像。

##### 5.3.2.5.1. 在集群外本地运行

您可以作为集群外的 Go 程序运行您的 Operator 项目。这可以加快部署和测试的速度，对于开发非常有用。

**流程**

- 运行以下命令，以在 **~/.kube/config** 文件中配置的集群中安装自定义资源定义（CRD），并在本地运行 Operator：

```
$ make install run
```

**输出示例**

```
...
2021-01-10T21:09:29.016-0700 INFO controller-runtime.metrics metrics server is starting to
listen {"addr": ":8080"}
2021-01-10T21:09:29.017-0700 INFO setup starting manager
2021-01-10T21:09:29.017-0700 INFO controller-runtime.manager starting metrics server
{"path": "/metrics"}
2021-01-10T21:09:29.018-0700 INFO controller-runtime.manager.controller.memcached
Starting EventSource {"reconciler group": "cache.example.com", "reconciler kind":
"Memcached", "source": "kind source: /, Kind="}
2021-01-10T21:09:29.218-0700 INFO controller-runtime.manager.controller.memcached
Starting Controller {"reconciler group": "cache.example.com", "reconciler kind":
"Memcached"}
2021-01-10T21:09:29.218-0700 INFO controller-runtime.manager.controller.memcached
Starting workers {"reconciler group": "cache.example.com", "reconciler kind": "Memcached",
"worker count": 1}
```

##### 5.3.2.5.2. 准备 Operator 以使用支持的镜像

在 OpenShift Container Platform 上运行基于 Go 的 Operator 之前，请将项目更新为使用支持的镜像。

**流程**

1. 更新项目根级别 Dockerfile 以使用支持的镜像。更改默认运行程序镜像引用：

```
FROM gcr.io/distroless/static:nonroot
```

改为：

```
FROM registry.access.redhat.com/ubi8/ubi-minimal:latest
```

2. 根据 Go 项目版本，您的 Dockerfile 可能包含 **USER 65532:65532** 或 **USER nonroot:nonroot** 指令。在这两种情况下，删除该行，因为受支持的 runner 镜像不需要该行。

3. 在 **config/default/manager_auth_proxy_patch.yaml** 文件中，修改 **image** 值：

```
gcr.io/kubebuilder/kube-rbac-proxy:<tag>
```

使用支持的镜像：

```
registry.redhat.io/openshift4/ose-kube-rbac-proxy:v4.7
```

#### 5.3.2.5.3. 作为集群的部署运行

您可以作为一个部署在集群中运行 Operator 项目。

**先决条件**

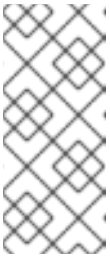
- 通过更新项目以使用支持的镜像，准备基于 Go 的 Operator 在 OpenShift Container Platform 上运行

**流程**

1. 运行以下 **make** 命令来构建和推送 Operator 镜像。在以下步骤中修改 **IMG** 参数来引用您可访问的库。您可以获取在存储库站点（如 Quay.io）存储容器的帐户。

   a. 构建镜像：

```
$ make docker-build IMG=<registry>/<user>/<image_name>:<tag>
```

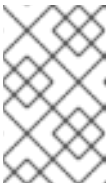


**注意**

由 SDK 为 Operator 生成的 Dockerfile 明确引用了 **go 构建** 的 **GOARCH=amd64**。对于非 AMD64 构架，这已被应用于 **GOARCH=$TARGETARCH**。Docker 会自动将 环境变量设置为 **-platform** 指定的值。使用 Buildah，**-build-arg** 需要用于目的。如需更多信息，请参阅 多个架构。

   b. 将镜像推送到存储库：

```
$ make docker-push IMG=<registry>/<user>/<image_name>:<tag>
```



**注意**

镜像的名称和标签，如 **IMG=<registry> /<user> /<image_name>:<tag>**，在两个命令中都可在您的 Makefile 中设置。修改 **IMG ?= controller:latest** 值来设置您的默认镜像名称。

2. 运行以下命令来部署 Operator：

```
$ make deploy IMG=<registry>/<user>/<image_name>:<tag>
```

默认情况下，这个命令会创建一个带有 Operator 项目名称的命名空间，格式为 **<project_name>-system**，用于部署。此命令还从 **config/rbac** 安装 RBAC 清单。

3. 验证 Operator 是否正在运行：

```
$ oc get deployment -n <project_name>-system
```

输**出示例**

```
NAME                                    READY   UP-TO-DATE   AVAILABLE   AGE
<project_name>-controller-manager       1/1     1            1           8m
```

#### 5.3.2.5.4. 捆绑 Operator 并使用 Operator Lifecycle Manager 进行部署

Operator Lifecycle Manager（OLM）可帮助您在 Kubernetes 集群上安装、更新 Operator 及其相关服务的生命周期。OLM 在 OpenShift Container Platform 上默认安装，并作为 Kubernetes 扩展运行，以便您可以在没有任何额外工具的情况下将 Web 控制台和 OpenShift CLI（**oc**）用于所有 Operator 生命周期管理功能。

Operator Bundle Format 是 Operator SDK 和 OLM 的默认打包方法。您可以通过使用 Operator SDK 来通过 OLM 构建、推送、验证和运行捆绑包镜像，让 Operator 准备好进行 OLM。

**先决条件**

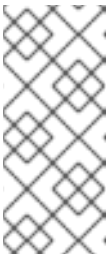
- 在开发工作站上安装 operator SDK CLI
- 已安装 OpenShift CLI（**oc**）v4.7+
- Operator Lifecycle Manager（OLM）安装在一个基于 Kubernetes 的集群上（如果使用 **apiextensions.k8s.io/v1** CRD，则为 v1.16.0 或更新版本，如 OpenShift Container Platform 4.7）
- 使用具有 **cluster-admin** 权限的账户使用 **oc** 登录到集群
- 使用 Operator SDK 初始化 operator 项目
- 如果 Operator 是基于 Go 的，则必须更新您的项目以使用支持的镜像在 OpenShift Container Platform 上运行

**流程**

1. 在 Operator 项目目录中运行以下 **make** 命令来构建和推送 Operator 镜像。在以下步骤中修改 **IMG** 参数来引用您可访问的库。您可以获取在存储库站点（如 Quay.io）存储容器的帐户。

a. 构建镜像：

```
$ make docker-build IMG=<registry>/<user>/<operator_image_name>:<tag>
```



**注意**

由 SDK 为 Operator 生成的 Dockerfile 明确引用了 **go 构建** 的 **GOARCH=amd64**。对于非 AMD64 构架，这已被应用于 **GOARCH=$TARGETARCH**。Docker 会自动将 环境变量设置为 **-platform** 指定的值。使用 Buildah，**-build-arg** 需要用于目的。如需更多信息，请参阅 多个架构。

b. 将镜像推送到存储库：

```
$ make docker-push IMG=<registry>/<user>/<operator_image_name>:<tag>
```

2. 运行 **make bundle** 命令创建 Operator 捆绑包清单，该命令调用多个命令，其中包括 Operator SDK **generate bundle** 和 **bundle validate** 子命令：

```
$ make bundle IMG=<registry>/<user>/<operator_image_name>:<tag>
```

Operator 的捆绑包清单描述了如何显示、创建和管理应用程序。**make bundle** 命令在 Operator 项目中创建以下文件和目录：

- 包含 **ClusterServiceVersion** 对象的捆绑包清单目录，名为 **bundle/manifests**
- 名为 **bundle/metadata** 的捆绑包元数据目录
- **config/crd** 目录中的所有自定义资源定义（CRD）
- 一个 Dockerfile **bundle.Dockerfile**

然后，使用 **operator-sdk bundle validate** 自动验证这些文件，以确保磁盘上的捆绑包的格式是正确的。

3. 运行以下命令来构建和推送捆绑包镜像。OLM 使用索引镜像来消耗 Operator 捆绑包，该镜像引用一个或多个捆绑包镜像。

a. 构建捆绑包镜像。使用您要推送镜像的 registry、用户命名空间和镜像标签的详情，设置 **BUNDLE_IMAGE**：

```
$ make bundle-build BUNDLE_IMG=<registry>/<user>/<bundle_image_name>:<tag>
```

b. 推送捆绑包镜像：

```
$ docker push <registry>/<user>/<bundle_image_name>:<tag>
```

4. 使用以下命令，检查集群中的 OLM 状态：

```
$ operator-sdk olm status \
   --olm-namespace=openshift-operator-lifecycle-manager
```

5. 使用 Operator SDK 中的 OLM 集成在集群中运行 Operator：

```
$ operator-sdk run bundle \
   [-n <namespace>] \ 1
   <registry>/<user>/<bundle_image_name>:<tag>
```

1 默认情况下，命令会在 ~/**.kube/config** 文件中当前活跃的项目中安装 Operator。您可以添加 **-n** 标志来为安装设置不同的命名空间范围。

这个命令执行以下操作：

- 使用注入的捆绑包镜像创建索引镜像。
- 创建指向新索引镜像的目录源，以便 OperatorHub 能够发现 Operator。
- 通过创建一个 Operator 组、订阅、安装计划以及所有其他必要的对象（包括 RBAC），将 Operator 部署到集群中。

#### 5.3.2.6. 创建自定义资源

安装 Operator 后，您可以通过创建一个由 Operator 在集群中提供的自定义资源（CR）来测试它。

**先决条件**

- 在集群中安装的 Memcached Operator 示例，它提供 **Memcached** CR）

**流程**

1. 切换到安装 Operator 的命名空间。例如，如果使用 **make deploy** 命令部署 Operator：

```
$ oc project memcached-operator-system
```

2. 编辑 **config/samples/cache_v1_memcached.yaml** 上的 **Memcached** CR 清单示例，使其包含以下规格：

```
apiVersion: cache.example.com/v1
kind: Memcached
metadata:
  name: memcached-sample
...
spec:
...
  size: 3
```

3. 创建 CR:

```
$ oc apply -f config/samples/cache_v1_memcached.yaml
```

4. 确保 **Memcached** Operator 为示例 CR 创建部署，其大小正确：

```
$ oc get deployments
```

输出示例

```
NAME                                    READY   UP-TO-DATE   AVAILABLE   AGE
memcached-operator-controller-manager   1/1     1            1           8m
memcached-sample                        3/3     3            3           1m
```

5. 检查 pod 和 CR 状态，以确认其状态是否使用 Memcached pod 名称更新。

   a. 检查 pod:

   ```
   $ oc get pods
   ```

   输**出示例**

   ```
   NAME                                READY     STATUS    RESTARTS   AGE
   memcached-sample-6fd7c98d8-7dqdr    1/1       Running   0          1m
   memcached-sample-6fd7c98d8-g5k7v    1/1       Running   0          1m
   memcached-sample-6fd7c98d8-m7vn7    1/1       Running   0          1m
   ```

   b. 检查 CR 状态：

   ```
   $ oc get memcached/memcached-sample -o yaml
   ```

   输**出示例**

   ```
   apiVersion: cache.example.com/v1
   kind: Memcached
   metadata:
   ...
     name: memcached-sample
   ...
   spec:
     size: 3
   status:
     nodes:
     - memcached-sample-6fd7c98d8-7dqdr
     - memcached-sample-6fd7c98d8-g5k7v
     - memcached-sample-6fd7c98d8-m7vn7
   ```

6. 更新部署大小。

   a. 更新 **config/samples/cache_v1_memcached.yaml** 文件，将 **Memcached** CR 中的 **spec.size** 字段从 **3** 改为 **5**：

   ```
   $ oc patch memcached memcached-sample \
       -p '{"spec":{"size": 5}}' \
       --type=merge
   ```

   b. 确认 Operator 已更改部署大小：

   ```
   $ oc get deployments
   ```

   输**出示例**

```
NAME                                    READY   UP-TO-DATE   AVAILABLE   AGE
memcached-operator-controller-manager   1/1     1            1           10m
memcached-sample                        5/5     5            5           3m
```

7. 清理本教程中创建的资源。

- 如果使用 **make deploy** 命令来测试 Operator，请运行以下命令：

```
$ make undeploy
```

- 如果使用 **operator-sdk run bundle** 命令来测试 Operator，请运行以下命令：

```
$ operator-sdk cleanup <project_name>
```

#### 5.3.2.7. 其他资源

- 请参阅 基于 Go 的 Operator 的项目布局，以了解 Operator SDK 创建的目录结构。

### 5.3.3. 基于 Go 的 Operator 的项目布局

**operator-sdk** CLI 可为每个 Operator 项目生成或 *scaffold* 多个 软件包和文件。

#### 5.3.3.1. 基于 Go 的项目布局

使用 **operator-sdk init** 命令生成的基于 Go 的 Operator 项目（默认类型）包含以下文件和目录：

| 文件或目录 | 用途 |
|---|---|
| **main.go** | Operator 的主要程序。这会实例化一个新管理器，它会在 **apis/** 目录中注册所有自定义资源定义（CRD），并启动 **controllers/** 目录中的所有控制器。 |
| **apis/** | 定义 CRD API 的目录树。您必须编辑 **apis/<version> /<kind>_types.go** 文件，为每个资源类型定义 API，并在控制器中导入这些软件包以监视这些资源类型。 |
| **controllers/** | 控制器的实现。编辑 **controller/<kind>_controller.go** 文件，以定义控制器处理指定种类资源类型的协调逻辑。 |
| **config/** | 用于在集群中部署控制器的 Kubernetes 清单，包括 CRD、RBAC 和证书。 |
| **Makefile** | 用于构建和部署控制器的目标。 |
| **Docker** | 容器引擎用来构建 Operator 的说明。 |
| **manifests/** | Kubernetes 清单用于注册 CRD、设置 RBAC 和将 Operator 部署为部署。 |

## 5.4. 基于 ANSIBLE 的 OPERATOR

### 5.4.1. 基于 Ansible 的 Operator 的 Operator SDK 入门

Operator SDK 包括生成 Operator 项目的选项，它利用现有 Ansible playbook 和模块将 Kubernetes 资源部署为统一应用程序，而无需编写任何 Go 代码。

如需演示使用 Operator SDK 提供的工具和库设置并运行基于 Ansible 的 Operator 的基本知识，Operator 开发人员可以为 Memcached、分布式键值存储构建基于 Ansible 的 Operator 示例，并将它部署到集群中。

#### 5.4.1.1. 先决条件

- 已安装 operator SDK CLI
- 已安装 OpenShift CLI（**oc**）v4.7+
- Ansible 版本 v2.9.0
- Ansible Runner 版本 v1.1.0+
- Ansible Runner HTTP Event Emitter plug-in 版本 v1.0.0+
- OpenShift Python client 版本 v0.11.2+
- 使用具有 **cluster-admin** 权限的 **oc** 登录到 OpenShift Container Platform 4.7 集群
- 要允许集群拉取镜像，推送镜像的存储库必须设置为公共的存储库，或必须配置一个镜像 pull secret。

#### 5.4.1.2. 创建并部署基于 Ansible 的 Operator

您可以使用 Operator SDK 为 Memcached 构建和部署简单的基于 Ansible 的 Operator。

**流程**

1. **创建一个项目。**
   a. 创建您的项目目录：

   ```
   $ mkdir memcached-operator
   ```

   b. 切换到项目所在的目录：

   ```
   $ cd memcached-operator
   ```

   c. 使用 **ansible** 插件运行 **operator-sdk init** 命令以初始化项目：

   ```
   $ operator-sdk init \
       --plugins=ansible \
       --domain=example.com
   ```

2. **创建 API。**
   创建简单的 Memcached API：

   ```
   $ operator-sdk create api \
       --group cache \
       --version v1 \
   ```

```
    --kind Memcached \
    --generate-role 1
```

1 为 API 生成 Ansible 角色。

3. **构建并推送 Operator 镜像。**
使用默认的 **Makefile** 目标来构建和推送 Operator。使用镜像的 pull spec 设置 **IMG**，该 spec 使用您可推送到的 registry：

```
$ make docker-build docker-push IMG=<registry>/<user>/<image_name>:<tag>
```

4. **运行 Operator。**

   a. 安装 CRD：

   ```
   $ make install
   ```

   b. 将项目部署到集群中。将 **IMG** 设置为您推送的镜像：

   ```
   $ make deploy IMG=<registry>/<user>/<image_name>:<tag>
   ```

5. **创建示例自定义资源（CR）。**

   a. 创建一个示例 CR：

   ```
   $ oc apply -f config/samples/cache_v1_memcached.yaml \
       -n memcached-operator-system
   ```

   b. 查看 CR 协调 Operator：

   ```
   $ oc logs deployment.apps/memcached-operator-controller-manager \
       -c manager \
       -n memcached-operator-system
   ```

   **输出示例**

   ```
   ...
   I0205 17:48:45.881666       7 leaderelection.go:253] successfully acquired lease
   memcached-operator-system/memcached-operator
   {"level":"info","ts":1612547325.8819902,"logger":"controller-
   runtime.manager.controller.memcached-controller","msg":"Starting
   EventSource","source":"kind source: cache.example.com/v1, Kind=Memcached"}
   {"level":"info","ts":1612547325.98242,"logger":"controller-
   runtime.manager.controller.memcached-controller","msg":"Starting Controller"}
   {"level":"info","ts":1612547325.9824686,"logger":"controller-
   runtime.manager.controller.memcached-controller","msg":"Starting workers","worker
   count":4}
   {"level":"info","ts":1612547348.8311093,"logger":"runner","msg":"Ansible-runner exited
   successfully","job":"4037200794235010051","name":"memcached-
   sample","namespace":"memcached-operator-system"}
   ```

6. **清理。**

运行以下命令清理在此流程中创建的资源：

```
$ make undeploy
```

#### 5.4.1.3. 后续步骤

- 如需更深入地了解如何构建基于 Ansible 的 Operator，请参阅基于 Ansible 的 Operator SDK 指南。

### 5.4.2. 基于 Ansible 的 Operator 的 operator SDK 指南

operator 开发人员可以利用 Operator SDK 中的 Ansible 支持来为 Memcached 构建基于 Ansible 的示例 Operator、分布式键值存储并管理其生命周期。本教程介绍了以下过程：

- 创建 Memcached 部署
- 确保部署大小与 **Memcached** 自定义资源（CR）spec 指定的大小相同
- 使用 status writer 带有 **memcached** Pod 的名称来更新 **Memcached** CR 状态

此过程可通过以下两个 Operator Framework 核心组件完成：

**Operator SDK**

**operator-sdk** CLI 工具和 **controller-runtime** 库 API

**Operator Lifecycle Manager (OLM)**

集群中 Operator 的安装、升级和基于角色的访问控制（RBAC）

**注意**

本教程的内容比基于 Ansible 的 Operator 开始使用 Operator SDK 内容更详细。

#### 5.4.2.1. 先决条件

- 已安装 operator SDK CLI
- 已安装 OpenShift CLI（**oc**）v4.7+
- Ansible 版本 v2.9.0
- Ansible Runner 版本 v1.1.0+
- Ansible Runner HTTP Event Emitter plug-in 版本 v1.0.0+
- OpenShift Python client 版本 v0.11.2+
- 使用具有 **cluster-admin** 权限的 **oc** 登录到 OpenShift Container Platform 4.7 集群
- 要允许集群拉取镜像，推送镜像的存储库必须设置为公共的存储库，或必须配置一个镜像 pull secret。

#### 5.4.2.2. 创建一个项目

使用 Operator SDK CLI 创建名为 **memcached-operator 的** 项目。

**流程**

1. 为项目创建一个目录：

```
$ mkdir -p $HOME/projects/memcached-operator
```

2. 进入该目录：

```
$ cd $HOME/projects/memcached-operator
```

3. 使用 **ansible** 插件运行 **operator-sdk init** 命令以初始化项目：

```
$ operator-sdk init \
    --plugins=ansible \
    --domain=example.com
```

#### 5.4.2.2.1. PROJECT 文件

**operator-sdk init** 命令生成的文件中是一个 Kubebuilder **PROJECT** 文件。从项目 root 运行的后续 **operator-sdk** 命令以及 **help** 输出可读取该文件，并注意到项目的类型是 Ansible。例如：

```
domain: example.com
layout: ansible.sdk.operatorframework.io/v1
projectName: memcached-operator
version: 3-alpha
```

### 5.4.2.3. 创建 API

使用 Operator SDK CLI 创建 Memcached API。

**流程**

- 运行以下命令创建带有组 **cache**、版本 **v1** 和种类 **Memcached** 的 API：

```
$ operator-sdk create api \
    --group cache \
    --version v1 \
    --kind Memcached \
    --generate-role 1
```

1 为 API 生成 Ansible 角色。

创建 API 后，Operator 项目会以以下结构更新：

**Memcached CRD**

包括一个 **Memcached** 资源示例

**Manager（管理者）**

使用以下方法将集群状态协调到所需状态的程序：

- 一个协调器，可以是 Ansible 角色或 playbook
- 一个 **watches.yaml** 文件，将 **Memcached** 资源连接到 **memcached** Ansible 角色

#### 5.4.2.4. 修改管理者

更新您的 Operator 项目，以提供协调逻辑，其格式为 Ansible 角色，它在每次创建、更新或删除 **Memcached** 资源时运行。

**流程**

1. 用下列结构更新 **roles/memcached/tasks/main.yml** 文件：

```
---
- name: start memcached
  community.kubernetes.k8s:
    definition:
      kind: Deployment
      apiVersion: apps/v1
      metadata:
        name: '{{ ansible_operator_meta.name }}-memcached'
        namespace: '{{ ansible_operator_meta.namespace }}'
      spec:
        replicas: "{{size}}"
        selector:
          matchLabels:
            app: memcached
        template:
          metadata:
            labels:
              app: memcached
          spec:
            containers:
            - name: memcached
              command:
              - memcached
              - -m=64
              - -o
              - modern
              - -v
              image: "docker.io/memcached:1.4.36-alpine"
              ports:
                - containerPort: 11211
```

这个 **memcached** 角色可确保存在 **memcached** 部署并设置部署大小。

2. 通过编辑 **roles/memcached/defaults/main.yml** 文件，为您的 Ansible 角色中使用的变量设置默认值：

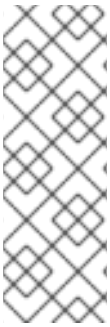
```
---
# defaults file for Memcached
size: 1
```

3. 使用以下结构更新 **config/samples/cache_v1_memcached.yaml** 文件中的 **Memcached** 示例资源：

```
apiVersion: cache.example.com/v1
kind: Memcached
metadata:
```

```
  name: memcached-sample
spec:
  size: 3
```

自定义资源（CR）spec 中的键值对作为额外变量传递给 Ansible。



**注意**

在运行 Ansible 前，Operator 会将 **spec** 字段中所有变量的名称转换为 snake case，即小写并附带下划线。例如，spec 中的 **serviceAccount** 在 Ansible 中会变成 **service_account**。

您可以通过在 **watches.yaml** 文件中将 **snakeCaseParameters** 选项设置为 **false** 来禁用大小写转换。建议您在 Ansible 中对变量执行一些类型验证，以确保应用程序收到所需输入。

#### 5.4.2.5. 运行 Operator

您可以使用 Operator SDK CLI 构建和运行 Operator：

- 作为 Go 程序在集群外本地运行。
- 作为集群的部署运行。
- 捆绑 Operator，并使用 Operator Lifecycle Manager（OLM）在集群中部署。

##### 5.4.2.5.1. 在集群外本地运行

您可以作为集群外的 Go 程序运行您的 Operator 项目。这可以加快部署和测试的速度，对于开发非常有用。

**流程**

- 运行以下命令，以在 **~/.kube/config** 文件中配置的集群中安装自定义资源定义（CRD），并在本地运行 Operator：

```
$ make install run
```

输出示例

```
...
{"level":"info","ts":1612589622.7888272,"logger":"ansible-controller","msg":"Watching
resource","Options.Group":"cache.example.com","Options.Version":"v1","Options.Kind":"Memc
ached"}
{"level":"info","ts":1612589622.7897573,"logger":"proxy","msg":"Starting to
serve","Address":"127.0.0.1:8888"}
{"level":"info","ts":1612589622.789971,"logger":"controller-runtime.manager","msg":"starting
metrics server","path":"/metrics"}
{"level":"info","ts":1612589622.7899997,"logger":"controller-
runtime.manager.controller.memcached-controller","msg":"Starting
EventSource","source":"kind source: cache.example.com/v1, Kind=Memcached"}
{"level":"info","ts":1612589622.8904517,"logger":"controller-
runtime.manager.controller.memcached-controller","msg":"Starting Controller"}
```

```
{"level":"info","ts":1612589622.8905244,"logger":"controller-
runtime.manager.controller.memcached-controller","msg":"Starting workers","worker
count":8}
```

#### 5.4.2.5.2. 准备 Operator 以使用支持的镜像

在 OpenShift Container Platform 上运行基于 Ansible 的 Operator 之前，请将项目更新为使用支持的镜像。

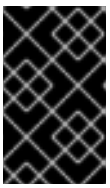
**流程**

1. 更新项目根级别 Dockerfile 以使用支持的镜像。更改默认构建器镜像引用：

```
FROM quay.io/operator-framework/ansible-operator:v1.3.0
```

改为：

```
FROM registry.redhat.io/openshift4/ose-ansible-operator:v4.7
```



**重要**

使用与 Operator SDK 版本匹配的构建器镜像版本。因项目布局或 *scaffolding* 而导致问题，特别是将较新的 Operator SDK 版本与下游 OpenShift Container Platform 构建器镜像混合时。

2. 在 **config/default/manager_auth_proxy_patch.yaml** 文件中，修改 **image** 值：

```
gcr.io/kubebuilder/kube-rbac-proxy:<tag>
```

使用支持的镜像：

```
registry.redhat.io/openshift4/ose-kube-rbac-proxy:v4.7
```

#### 5.4.2.5.3. 作为集群的部署运行

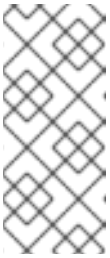
您可以作为一个部署在集群中运行 Operator 项目。

**流程**

1. 运行以下 **make** 命令来构建和推送 Operator 镜像。在以下步骤中修改 **IMG** 参数来引用您可访问的库。您可以获取在存储库站点（如 Quay.io）存储容器的帐户。

   a. 构建镜像：

```
$ make docker-build IMG=<registry>/<user>/<image_name>:<tag>
```

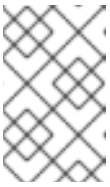


**注意**

由 SDK 为 Operator 生成的 Dockerfile 明确引用了 **go 构建** 的 **GOARCH=amd64**。对于非 AMD64 构架，这已被应用于 **GOARCH=$TARGETARCH**。Docker 会自动将 环境变量设置为 **-platform** 指定的值。使用 Buildah，**-build-arg** 需要用于目的。如需更多信息，请参阅 多个架构。

b. 将镜像推送到存储库：

```
$ make docker-push IMG=<registry>/<user>/<image_name>:<tag>
```



**注意**

镜像的名称和标签，如 **IMG=<registry> /<user> /<image_name>:<tag>**，在两个命令中都可在您的 Makefile 中设置。修改 **IMG ?= controller:latest** 值来设置您的默认镜像名称。

2. 运行以下命令来部署 Operator：

```
$ make deploy IMG=<registry>/<user>/<image_name>:<tag>
```

默认情况下，这个命令会创建一个带有 Operator 项目名称的命名空间，格式为 **<project_name>-system**，用于部署。此命令还从 **config/rbac** 安装 RBAC 清单。

3. 验证 Operator 是否正在运行：

```
$ oc get deployment -n <project_name>-system
```

输**出示例**

```
NAME                                 READY   UP-TO-DATE   AVAILABLE   AGE
<project_name>-controller-manager    1/1     1            1           8m
```

#### 5.4.2.5.4. 捆绑 Operator 并使用 Operator Lifecycle Manager 进行部署

Operator Lifecycle Manager（OLM）可帮助您在 Kubernetes 集群上安装、更新 Operator 及其相关服务的生命周期。OLM 在 OpenShift Container Platform 上默认安装，并作为 Kubernetes 扩展运行，以便您可以在没有任何额外工具的情况下将 Web 控制台和 OpenShift CLI（**oc**）用于所有 Operator 生命周期管理功能。

Operator Bundle Format 是 Operator SDK 和 OLM 的默认打包方法。您可以通过使用 Operator SDK 来通过 OLM 构建、推送、验证和运行捆绑包镜像，让 Operator 准备好进行 OLM。

**先决条件**

- 在开发工作站上安装 operator SDK CLI
- 已安装 OpenShift CLI（**oc**）v4.7+
- Operator Lifecycle Manager（OLM）安装在一个基于 Kubernetes 的集群上（如果使用 **apiextensions.k8s.io/v1** CRD，则为 v1.16.0 或更新版本，如 OpenShift Container Platform 4.7）

- 使用具有 **cluster-admin** 权限的账户使用 **oc** 登录到集群
- 使用 Operator SDK 初始化 operator 项目

**流程**

1. 在 Operator 项目目录中运行以下 **make** 命令来构建和推送 Operator 镜像。在以下步骤中修改 **IMG** 参数来引用您可访问的库。您可以获取在存储库站点（如 Quay.io）存储容器的帐户。

   a. 构建镜像：

   ```
   $ make docker-build IMG=<registry>/<user>/<operator_image_name>:<tag>
   ```

   **注意**

   由 SDK 为 Operator 生成的 Dockerfile 明确引用了 **go 构建** 的 **GOARCH=amd64**。对于非 AMD64 构架，这已被应用于 **GOARCH=$TARGETARCH**。Docker 会自动将 环境变量设置为 **-platform** 指定的值。使用 Buildah，**-build-arg** 需要用于目的。如需更多信息，请参阅 多个架构。

   b. 将镜像推送到存储库：

   ```
   $ make docker-push IMG=<registry>/<user>/<operator_image_name>:<tag>
   ```

2. 运行 **make bundle** 命令创建 Operator 捆绑包清单，该命令调用多个命令，其中包括 Operator SDK **generate bundle** 和 **bundle validate** 子命令：

   ```
   $ make bundle IMG=<registry>/<user>/<operator_image_name>:<tag>
   ```

   Operator 的捆绑包清单描述了如何显示、创建和管理应用程序。**make bundle** 命令在 Operator 项目中创建以下文件和目录：

   - 包含 **ClusterServiceVersion** 对象的捆绑包清单目录，名为 **bundle/manifests**
   - 名为 **bundle/metadata** 的捆绑包元数据目录
   - **config/crd** 目录中的所有自定义资源定义（CRD）
   - 一个 Dockerfile **bundle.Dockerfile**

   然后，使用 **operator-sdk bundle validate** 自动验证这些文件，以确保磁盘上的捆绑包的格式是正确的。

3. 运行以下命令来构建和推送捆绑包镜像。OLM 使用索引镜像来消耗 Operator 捆绑包，该镜像引用一个或多个捆绑包镜像。

   a. 构建捆绑包镜像。使用您要推送镜像的 registry、用户命名空间和镜像标签的详情，设置 **BUNDLE_IMAGE**：

   ```
   $ make bundle-build BUNDLE_IMG=<registry>/<user>/<bundle_image_name>:<tag>
   ```

   b. 推送捆绑包镜像：

```
$ docker push <registry>/<user>/<bundle_image_name>:<tag>
```

4. 使用以下命令，检查集群中的 OLM 状态：

```
$ operator-sdk olm status \
   --olm-namespace=openshift-operator-lifecycle-manager
```

5. 使用 Operator SDK 中的 OLM 集成在集群中运行 Operator：

```
$ operator-sdk run bundle \
   [-n <namespace>] \ 1
   <registry>/<user>/<bundle_image_name>:<tag>
```

1 默认情况下，命令会在 **~/.kube/config** 文件中当前活跃的项目中安装 Operator。您可以添加 **-n** 标志来为安装设置不同的命名空间范围。

这个命令执行以下操作：

- 使用注入的捆绑包镜像创建索引镜像。
- 创建指向新索引镜像的目录源，以便 OperatorHub 能够发现 Operator。
- 通过创建一个 Operator 组、订阅、安装计划以及所有其他必要的对象（包括 RBAC），将 Operator 部署到集群中。

#### 5.4.2.6. 创建自定义资源

安装 Operator 后，您可以通过创建一个由 Operator 在集群中提供的自定义资源（CR）来测试它。

**先决条件**

- 在集群中安装的 Memcached Operator 示例，它提供 **Memcached** CR）

**流程**

1. 切换到安装 Operator 的命名空间。例如，如果使用 **make deploy** 命令部署 Operator：

```
$ oc project memcached-operator-system
```

2. 编辑 **config/samples/cache_v1_memcached.yaml** 上的 **Memcached** CR 清单示例，使其包含以下规格：

```
apiVersion: cache.example.com/v1
kind: Memcached
metadata:
  name: memcached-sample
...
spec:
...
  size: 3
```

3. 创建 CR:

```
$ oc apply -f config/samples/cache_v1_memcached.yaml
```

4. 确保 **Memcached** Operator 为示例 CR 创建部署，其大小正确：

```
$ oc get deployments
```

输**出示例**

```
NAME                                    READY   UP-TO-DATE   AVAILABLE   AGE
memcached-operator-controller-manager   1/1     1            1           8m
memcached-sample                        3/3     3            3           1m
```

5. 检查 pod 和 CR 状态，以确认其状态是否使用 Memcached pod 名称更新。

   a. 检查 pod:

```
$ oc get pods
```

   输**出示例**

```
NAME                                READY     STATUS    RESTARTS   AGE
memcached-sample-6fd7c98d8-7dqdr    1/1       Running   0          1m
memcached-sample-6fd7c98d8-g5k7v    1/1       Running   0          1m
memcached-sample-6fd7c98d8-m7vn7    1/1       Running   0          1m
```

   b. 检查 CR 状态：

```
$ oc get memcached/memcached-sample -o yaml
```

   输**出示例**

```
apiVersion: cache.example.com/v1
kind: Memcached
metadata:
...
  name: memcached-sample
...
spec:
  size: 3
status:
  nodes:
  - memcached-sample-6fd7c98d8-7dqdr
  - memcached-sample-6fd7c98d8-g5k7v
  - memcached-sample-6fd7c98d8-m7vn7
```

6. 更新部署大小。

   a. 更新 **config/samples/cache_v1_memcached.yaml** 文件，将 **Memcached** CR 中的 **spec.size** 字段从 **3** 改为 **5**：

```
$ oc patch memcached memcached-sample \
    -p '{"spec":{"size": 5}}' \
    --type=merge
```

b. 确认 Operator 已更改部署大小：

```
$ oc get deployments
```

输出示例

```
NAME                                    READY   UP-TO-DATE   AVAILABLE   AGE
memcached-operator-controller-manager   1/1     1            1           10m
memcached-sample                        5/5     5            5           3m
```

7. 清理本教程中创建的资源。

- 如果使用 **make deploy** 命令来测试 Operator，请运行以下命令：

```
$ make undeploy
```

- 如果使用 **operator-sdk run bundle** 命令来测试 Operator，请运行以下命令：

```
$ operator-sdk cleanup <project_name>
```

#### 5.4.2.7. 其他资源

- 请参阅 基于 Ansible 的 Operator 的项目布局，以了解 Operator SDK 创建的目录结构。

### 5.4.3. 基于 Ansible 的 Operator 的项目布局

**operator-sdk** CLI 可为每个 Operator 项目生成或 *scaffold* 多个 软件包和文件。

#### 5.4.3.1. 基于 Ansible 的项目布局

使用 **operator-sdk init --plugins ansible** 命令生成的基于 Ansible 的 Operator 项目包含以下目录和文件：

| 文件或目录 | 用途 |
| --- | --- |
| **Docker** | 用于为 Operator 构建容器镜像的 Dockerfile。 |
| **Makefile** | 用于构建、发布、部署容器镜像的目标，其中包含 Operator 二进制文件，用于安装和卸载自定义资源定义（CRD）。 |
| **PROJECT** | 包含 Operator 元数据信息的 YAML 文件。 |
| **config/crd** | 基本 CRD 文件和 **kustomization.yaml** 文件的设置。 |
| **config/default** | 为部署收集所有 Operator 清单。被 **make deploy** 命令使用。 |
| **config/manager** | Controller Manager 部署。 |
| **config/prometheus** | 用于监控 Operator 的**ServiceMonitor** 资源。 |

| 文件或目录 | 用途 |
| --- | --- |
| **config/rbac** | 领导选举和身份验证代理的角色和角色绑定。 |
| **config/samples** | 为 CRD 创建的资源示例。 |
| **config/testing** | 用于测试的示例配置。 |
| **playbooks/** | 要运行的 playbook 的子目录。 |
| **roles/** | 要运行的角色树的子目录。 |
| **watches.yaml** | 要监视的资源的 Group/version/kind（GVK）和 Ansible 调用方法。使用 **create api** 命令添加新条目。 |
| **requirements.yml** | 包含要在构建期间安装的 Ansible 集合和角色依赖项的 YAML 文件。 |
| **molecule/** | 模拟您角色和 Operator 端到端测试的场景。 |

## 5.4.4. Operator SDK 中的 Ansible 支持

### 5.4.4.1. 自定义资源文件

Operator 会使用 Kubernetes 的扩展机制，即自定义资源定义 (CRD)，这样您的自定义资源 (CR) 的外观和行为均类似于内置的原生 Kubernetes 对象。

CR 文件格式是一个 Kubernetes 资源文件。该对象具有必填和选填字段：

**表 5.1. 自定义资源字段**

| 字段 | 描述 |
| --- | --- |
| **apiVersion** | 要创建 CR 的版本。 |
| **kind** | 要创建 CR 的类型。 |
| **metadata** | 要创建的 Kubernetes 特定元数据。 |
| **spec**（选填） | 传输至 Ansible 的变量键值列表。本字段默认为空。 |
| **status** | 总结对象的当前状态。对于基于 Ansible 的 Operator，**status** 子资源默认为 CRD 启用，由 **operator_sdk.util.k8s_status** Ansible 模块管理，其中包含 CR **status** 的 **condition** 信息。 |
| **annotations** | 要附于 CR 的 Kubernetes 特定注解。 |

以下 CR 注解列表会修改 Operator 的行为：

**表 5.2. 基于 Ansible 的 Operator 注解**

| 注解 | 描述 |
| --- | --- |
| **ansible.operator-sdk/reconcile-period** | 为 CR 指定协调间隔。该值将通过标准 Golang 软件包 **time** 来解析。具体来说，使用 **ParseDuration**，默认后缀 **s**，给出的数值以秒为单位。 |

### 基于 Ansible 的 Operator 注解示例

```
apiVersion: "test1.example.com/v1alpha1"
kind: "Test1"
metadata:
  name: "example"
annotations:
  ansible.operator-sdk/reconcile-period: "30s"
```

#### 5.4.4.2. watches.yaml 文件

*group/version/kind (GVK)* 是 Kubernetes API 的唯一标识符。 **watches.yaml** 文件包含从自定义资源（CR）中标识的自定义资源（CR）到 Ansible 角色或 playbook 的映射列表。Operator 期望这个映射文件位于 **/opt/ansible/watches.yaml** 的预定义位置。

**表 5.3. watches.yaml 文件映射**

| 字段 | 描述 |
| --- | --- |
| **group** | 要监视的 CR 组。 |
| **version** | 要监视的 CR 版本。 |
| **kind** | 要监视的 CR 类型 |
| **role**（默认） | 添加至容器中的 Ansible 角色的路径。例如：如果您的 **roles** 目录位于 **/opt/ansible/roles/** 中，角色名为 **busybox**，则该值应为 **/opt/ansible/roles/busybox**。该字段与 **playbook** 字段相互排斥。 |
| **playbook** | 添加至容器中的 Ansible playbook 的路径。期望这个 playbook 作为一种调用角色的方法。该字段与 **role** 字段相互排斥。 |
| **reconcilePeriod**（选填） | 给定 CR 的协调间隔，角色或 playbook 运行的频率。 |
| **manageStatus**（选填） | 如果设置为 **true**（默认），则 CR 的状态通常由 Operator 来管理。如果设置为 **false**，则 CR 的状态则会由指定角色或 playbook 在别处管理，或在单独控制器中管理。 |

### watches.yaml 文件示例

```
- version: v1alpha1 (1)
```

```
  group: test1.example.com
  kind: Test1
  role: /opt/ansible/roles/Test1

- version: v1alpha1 (2)
  group: test2.example.com
  kind: Test2
  playbook: /opt/ansible/playbook.yml

- version: v1alpha1 (3)
  group: test3.example.com
  kind: Test3
  playbook: /opt/ansible/test3.yml
  reconcilePeriod: 0
  manageStatus: false
```

1. 将 **Test1** 映射到 **test1** 角色的简单示例。
2. 将 **Test2** 映射到 playbook 的简单示例。
3. **Test3** kind 更复杂的示例。在 playbook 中禁止对 CR 状态重新排队和管理。

#### 5.4.4.2.1. 高级选项

高级功能可通过添加至每个 GVK 的 **watches.yaml** 文件中来启用。它们可放在 **group**、**version**、**kind** 和 **playbook** 或 **role** 字段下方。

可使用 CR 上的注解覆盖每个资源的一些功能。可覆盖的选项会指定以下注解。

**表 5.4. 高级的 watches.yaml 文件选项**

| 功能 | YAML 密钥 | 描述 | 覆盖注解 | 默认值 |
|---|---|---|---|---|
| 协调周期 | **reconcilePeriod** | 特定 CR 的协调运行间隔时间。 | **ansible.operator-sdk/reconcile-period** | **1m** |
| 管理状态 | **manageStatus** | 允许 Operator 管理每个 CR **status** 部分中的 **conditions** 部分。 | | **true** |
| 监视依赖资源 | **watchDependentResources** | 支持 Operator 动态监视由 Ansible 创建的资源。 | | **true** |
| 监控集群范围内的资源 | **watchClusterScopedResources** | 支持 Operator 监视由 Ansible 创建的集群范围的资源。 | | **false** |

| 功能 | YAML 密钥 | 描述 | 覆盖注解 | 默认值 |
| --- | --- | --- | --- | --- |
| 最大运行程序工件 | **maxRunnerArtifacts** | 管理 Ansible Runner 在 Operator 容器中为每个单独资源保存的构件目录的数量。 | **ansible.operator-sdk/max-runner-artifacts** | **20** |

带有高级选项的 watches.yml 文件示例

```
- version: v1alpha1
  group: app.example.com
  kind: AppService
  playbook: /opt/ansible/playbook.yml
  maxRunnerArtifacts: 30
  reconcilePeriod: 5s
  manageStatus: False
  watchDependentResources: False
```

#### 5.4.4.3. 发送至 Ansible 的额外变量

额外变量可发送至 Ansible，然后由 Operator 管理。自定义资源 (CR) 的 **spec** 部分作为额外变量按照键值对传递。等同于传递给 **ansible-playbook** 命令的额外变量。

Operator 还会在 **meta** 字段下传递额外变量，用于 CR 的名称和 CR 的命名空间。

对于以下 CR 示例：

```
apiVersion: "app.example.com/v1alpha1"
kind: "Database"
metadata:
  name: "example"
spec:
  message: "Hello world 2"
  newParameter: "newParam"
```

作为额外变量传递至 Ansible 的结构为：

```
{ "meta": {
        "name": "<cr_name>",
        "namespace": "<cr_namespace>",
  },
  "message": "Hello world 2",
  "new_parameter": "newParam",
  "_app_example_com_database": {
     <full_crd>
   },
}
```

**message** 和 **newParameter** 字段在顶层被设置为额外变量，**meta** 则为 Operator 中定义的 CR 提供相关元数据。**meta** 字段可使用 Ansible 中的点符号来访问，如：

```
---
- debug:
    msg: "name: {{ ansible_operator_meta.name }}, {{ ansible_operator_meta.namespace }}"
```

#### 5.4.4.4. Ansible Runner 目录

Ansible Runner 会将与 Ansible 运行相关的信息保存至容器中。具体位于：**/tmp/ansible-operator/runner/<group>/<version>/<kind>/<namespace>/<name>**。

**其他资源**

- 要了解有关 **runner** 目录的更多信息，请参阅 Ansible Runner 文档。

### 5.4.5. Kubernetes Collection for Ansible

要使用 Ansible 管理 Kubernetes 上的应用程序生命周期，您可以使用 Kubernetes Collection for Ansible。此 Ansible 模块集合允许开发人员利用通过 YAML 编写的现有 Kubernetes 资源文件，或用原生 Ansible 表达生命周期管理。

将 Ansible 与现有 Kubernetes 资源文件相结合的一个最大好处在于可使用 Jinja 模板，这样您只需借助 Ansible 中的几个变量即可轻松自定义资源。

本节详细介绍了 Kubernetes 集合的使用方法。开始之前，在本地工作站上安装集合，并使用 playbook 进行测试，然后再移至 Operator 内使用它。

#### 5.4.5.1. 为 Ansible 安装 Kubernetes 集合

您可以在本地工作站上安装 Kubernetes Collection for Ansible。

**流程**

1. 安装 Ansible 2.9+：

```
$ sudo dnf install ansible
```

2. 安装 OpenShift python 客户端 软件包：

```
$ pip3 install openshift
```

3. 使用以下方法之一安装 Kubernetes Collection：
   - 您可以直接从 Ansible Galaxy 安装集合：

```
$ ansible-galaxy collection install community.kubernetes
```

   - 如果您已初始化了 Operator，则可能在项目顶层都有一个 **requirements.yml** 文件。此文件指定必须安装的 Ansible 依赖项，才能让 Operator 正常工作。在默认情况下，这个文件会安装 **community.kubernetes** 集合以及 **operator_sdk.util** 集合，它为特定 Operator 的单元提供模块和插件。
     安装来自 **requirements.yml** 文件的依赖模块：

```
$ ansible-galaxy collection install -r requirements.yml
```

#### 5.4.5.2. 本地测试 Kubernetes Collection

operator 开发人员可以从其本地机器运行 Ansible 代码，而不是每次运行和重建 Operator。

**先决条件**

- 初始化基于 Ansible 的 Operator 项目，并使用 Operator SDK 创建具有生成 Ansible 角色的 API
- 安装 Kubernetes Collection for Ansible

**流程**

1. 在基于 Ansible 的 Operator 项目目录中，使用您想要的 Ansible 逻辑来修改 **roles/<kind> /tasks/main.yml** 文件。在创建 API 时，当使用 **--generate-role** 标志时，会创建 **roles/<kind>** / 目录。**<kind>** 可替换与您为 API 指定的类型匹配。
   以下示例根据名为 **state** 的变量值创建并删除配置映射：

```
---
- name: set ConfigMap example-config to {{ state }}
  community.kubernetes.k8s:
    api_version: v1
    kind: ConfigMap
    name: example-config
    namespace: default (1)
    state: "{{ state }}"
  ignore_errors: true (2)
```

(1) 如果您希望在一个与 **default** 不同的命名空间中创建配置映射，请更改此值。

(2) 设置 **ignore_errors: true** 可确保删除不存在的配置映射不会失败。

2. 修改 **roles/<kind> /defaults/main.yml** 文件，将默认 **state** 设置为 **present**：

```
---
state: present
```

3. 通过在项目目录的顶层创建一个 **playbook.yml** 文件来创建一个 Ansible playbook，其中包含您的 **<kind>** 角色：

```
---
- hosts: localhost
  roles:
    - <kind>
```

4. 运行 playbook：

```
$ ansible-playbook playbook.yml
```

输**出示例**

```
[WARNING]: provided hosts list is empty, only localhost is available. Note that the implicit
localhost does not match 'all'

PLAY [localhost] ***************************************************************************

TASK [Gathering Facts]
*********************************************************************************
ok: [localhost]

TASK [memcached : set ConfigMap example-config to present]
*********************************************************************************
changed: [localhost]

PLAY RECAP *********************************************************************************
localhost                  : ok=2    changed=1    unreachable=0    failed=0    skipped=0
rescued=0    ignored=0
```

5. 验证配置映射是否已创建：

```
$ oc get configmaps
```

输**出示例**

```
NAME             DATA   AGE
example-config    0     2m1s
```

6. 重新运行 playbook，设置 **state** 为 **absent**：

```
$ ansible-playbook playbook.yml --extra-vars state=absent
```

输**出示例**

```
[WARNING]: provided hosts list is empty, only localhost is available. Note that the implicit
localhost does not match 'all'

PLAY [localhost] ***************************************************************************

TASK [Gathering Facts]
*********************************************************************************
ok: [localhost]

TASK [memcached : set ConfigMap example-config to absent]
*********************************************************************************
changed: [localhost]

PLAY RECAP *********************************************************************************
localhost                  : ok=2    changed=1    unreachable=0    failed=0    skipped=0
rescued=0    ignored=0
```

7. 验证配置映射是否已删除：

```
$ oc get configmaps
```

#### 5.4.5.3. 后续步骤

- 如需了解当自定义资源（CR）更改时在 Operator 内触发自定义 Ansible 逻辑的详情，请参阅在 Operator 中使用 Ansible。

### 5.4.6. 在 Operator 中使用 Ansible

熟悉在本地使用 Kubernetes Collection for Ansible 后，当自定义资源（CR）发生变化时，您可以在 Operator 内部触发相同的 Ansible 逻辑。本示例将 Ansible 角色映射到 Operator 所监视的特定 Kubernetes 资源。该映射在 **watches.yaml** 文件中完成。

#### 5.4.6.1. 自定义资源文件

Operator 会使用 Kubernetes 的扩展机制，即自定义资源定义 (CRD)，这样您的自定义资源 (CR) 的外观和行为均类似于内置的原生 Kubernetes 对象。

CR 文件格式是一个 Kubernetes 资源文件。该对象具有必填和选填字段：

**表 5.5. 自定义资源字段**

| 字段 | 描述 |
| --- | --- |
| **apiVersion** | 要创建 CR 的版本。 |
| **kind** | 要创建 CR 的类型。 |
| **metadata** | 要创建的 Kubernetes 特定元数据。 |
| **spec**（选填） | 传输至 Ansible 的变量键值列表。本字段默认为空。 |
| **status** | 总结对象的当前状态。对于基于 Ansible 的 Operator，**status** 子资源默认为 CRD 启用，由 **operator_sdk.util.k8s_status** Ansible 模块管理，其中包含 CR **status** 的 **condition** 信息。 |
| **annotations** | 要附于 CR 的 Kubernetes 特定注解。 |

以下 CR 注解列表会修改 Operator 的行为：

**表 5.6. 基于 Ansible 的 Operator 注解**

| 注解 | 描述 |
| --- | --- |
| **ansible.operator-sdk/reconcile-period** | 为 CR 指定协调间隔。该值将通过标准 Golang 软件包 **time** 来解析。具体来说，使用 **ParseDuration**，默认后缀 **s**，给出的数值以秒为单位。 |

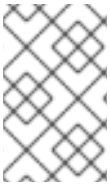
#### 基于 Ansible 的 Operator 注解示例

```
apiVersion: "test1.example.com/v1alpha1"
kind: "Test1"
metadata:
```

```
  name: "example"
annotations:
  ansible.operator-sdk/reconcile-period: "30s"
```

#### 5.4.6.2. 本地测试基于 Ansible 的 Operator

您可以使用 Operator 项目的顶层目录中的 **make run** 命令，测试本地运行的基于 Ansible 的 Operator 内部的逻辑。**make run** Makefile 目标在本地运行 **ansible-operator** 二进制文件，从 **watches.yaml** 文件中读取并使用 **~/.kube/config** 文件与 Kubernetes 集群通信，就像 **k8s** 模块一样。



**注意**

您可以通过设置环境变量 **ANSIBLE_ROLES_PATH** 或者使用 **ansible-roles-path** 标记来自定义角色路径。如果在 **ANSIBLE_ROLES_PATH** 值中没有找到该角色，Operator 会在 **{{current directory}}/roles** 中查找它。

**先决条件**

- Ansible Runner 版本 v1.1.0+
- Ansible Runner HTTP Event Emitter plug-in 版本 v1.0.0+
- 执行前面的步骤在本地测试 Kubernetes Collection

**流程**

1. 为自定义资源（CR）安装自定义资源定义（CRD）和正确的基于角色的访问控制（RBAC）定义：

```
$ make install
```

输出示例

```
/usr/bin/kustomize build config/crd | kubectl apply -f -
customresourcedefinition.apiextensions.k8s.io/memcacheds.cache.example.com created
```

2. 运行 **make run** 命令：

```
$ make run
```

输出示例

```
/home/user/memcached-operator/bin/ansible-operator run
{"level":"info","ts":1612739145.2871568,"logger":"cmd","msg":"Version","Go
Version":"go1.15.5","GOOS":"linux","GOARCH":"amd64","ansible-
operator":"v1.3.0","commit":"1abf57985b43bf6a59dcd18147b3c574fa57d3f6"}
...
{"level":"info","ts":1612739148.347306,"logger":"controller-runtime.metrics","msg":"metrics
server is starting to listen","addr":":8080"}
{"level":"info","ts":1612739148.3488882,"logger":"watches","msg":"Environment variable not
set; using default
value","envVar":"ANSIBLE_VERBOSITY_MEMCACHED_CACHE_EXAMPLE_COM","default":
2}
```

```
{"level":"info","ts":1612739148.3490262,"logger":"cmd","msg":"Environment variable not set; using default
value","Namespace":"","envVar":"ANSIBLE_DEBUG_LOGS","ANSIBLE_DEBUG_LOGS":false}
{"level":"info","ts":1612739148.3490646,"logger":"ansible-controller","msg":"Watching
resource","Options.Group":"cache.example.com","Options.Version":"v1","Options.Kind":"Memcached"}
{"level":"info","ts":1612739148.350217,"logger":"proxy","msg":"Starting to
serve","Address":"127.0.0.1:8888"}
{"level":"info","ts":1612739148.3506632,"logger":"controller-runtime.manager","msg":"starting
metrics server","path":"/metrics"}
{"level":"info","ts":1612739148.350784,"logger":"controller-
runtime.manager.controller.memcached-controller","msg":"Starting
EventSource","source":"kind source: cache.example.com/v1, Kind=Memcached"}
{"level":"info","ts":1612739148.5511978,"logger":"controller-
runtime.manager.controller.memcached-controller","msg":"Starting Controller"}
{"level":"info","ts":1612739148.5512562,"logger":"controller-
runtime.manager.controller.memcached-controller","msg":"Starting workers","worker
count":8}
```

现在，Operator 会监控 CR 的事件，创建 CR 将触发您的 Ansible 角色运行。

**注意**

考虑 **config/samples/<gvk>.yaml** CR 清单示例：

```
apiVersion: <group>.example.com/v1alpha1
kind: <kind>
metadata:
  name: "<kind>-sample"
```

因为未设置 **spec** 字段，所以调用 Ansible 时无额外变量。其他部分将涵盖从 CR 传递给 Ansible 的额外变量。为 Operator 设置适当的默认值是很重要的。

3. 创建 CR 实例，并将默认变量 **state** 设置为 **present**：

```
$ oc apply -f config/samples/<gvk>.yaml
```

4. 检查 **example-config** 配置映射是否已创建：

```
$ oc get configmaps
```

输出示例

```
NAME              STATUS   AGE
example-config      Active   3s
```

5. 修改 **config/samples/<gvk>.yaml** 文件，将 **state** 字段设置为 **absent**。例如：

```
apiVersion: cache.example.com/v1
kind: Memcached
metadata:
```

```
  name: memcached-sample
spec:
  state: absent
```

6. 应用更改：

```
$ oc apply -f config/samples/<gvk>.yaml
```

7. 确认配置映射已被删除：

```
$ oc get configmap
```

#### 5.4.6.3. 在集群上测试基于 Ansible 的 Operator

在 Operator 本地测试了自定义 Ansible 逻辑后，您可以在 OpenShift Container Platform 集群的 pod 内测试 Operator，该集群首选在生产环境中使用该逻辑。

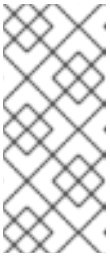
您可以作为一个部署在集群中运行 Operator 项目。

**流程**

1. 运行以下 **make** 命令来构建和推送 Operator 镜像。在以下步骤中修改 **IMG** 参数来引用您可访问的库。您可以获取在存储库站点（如 Quay.io）存储容器的帐户。

   a. 构建镜像：

```
$ make docker-build IMG=<registry>/<user>/<image_name>:<tag>
```



**注意**

由 SDK 为 Operator 生成的 Dockerfile 明确引用了 **go 构建** 的 **GOARCH=amd64**。对于非 AMD64 构架，这已被应用于 **GOARCH=$TARGETARCH**。Docker 会自动将 环境变量设置为 **-platform** 指定的值。使用 Buildah，**-build-arg** 需要用于目的。如需更多信息，请参阅 多个架构。

   b. 将镜像推送到存储库：

```
$ make docker-push IMG=<registry>/<user>/<image_name>:<tag>
```

**注意**

镜像的名称和标签，如 **IMG=<registry> /<user> /<image_name>:<tag>**，在两个命令中都可在您的 Makefile 中设置。修改 **IMG ?= controller:latest** 值来设置您的默认镜像名称。

2. 运行以下命令来部署 Operator：

```
$ make deploy IMG=<registry>/<user>/<image_name>:<tag>
```

默认情况下，这个命令会创建一个带有 Operator 项目名称的命名空间，格式为 **<project_name>-system**，用于部署。此命令还从 **config/rbac** 安装 RBAC 清单。

3. 验证 Operator 是否正在运行：

```
$ oc get deployment -n <project_name>-system
```

输出示例

```
NAME                                    READY   UP-TO-DATE   AVAILABLE   AGE
<project_name>-controller-manager       1/1     1            1           8m
```

#### 5.4.6.4. Ansible 日志

基于 Ansible 的 Operator 提供有关 Ansible 运行的日志，可用于调试 Ansible 任务。日志也可以包含有关 Operator 内部及其与 Kubernetes 交互的详细信息。

##### 5.4.6.4.1. 查看 Ansible 日志

**先决条件**

- 基于 Ansible 的 Operator 作为在集群中的部署方式运行

**流程**

- 要查看基于 Ansible 的 Operator 的日志，请运行以下命令：

```
$ oc logs deployment/<project_name>-controller-manager \
  -c manager \ 1
  -n <namespace> 2
```

1. 查看 **Manager** 容器的日志。
2. 如果您使用 **make deploy** 命令作为部署运行 Operator，使用 **<project_name>-system** 命名空间。

输出示例

```
{"level":"info","ts":1612732105.0579333,"logger":"cmd","msg":"Version","Go
Version":"go1.15.5","GOOS":"linux","GOARCH":"amd64","ansible-
operator":"v1.3.0","commit":"1abf57985b43bf6a59dcd18147b3c574fa57d3f6"}
{"level":"info","ts":1612732105.0587437,"logger":"cmd","msg":"WATCH_NAMESPACE
environment variable not set. Watching all namespaces.","Namespace":""}
I0207 21:08:26.110949       7 request.go:645] Throttling request took 1.035521578s, request:
GET:https://172.30.0.1:443/apis/flowcontrol.apiserver.k8s.io/v1alpha1?timeout=32s
{"level":"info","ts":1612732107.768025,"logger":"controller-runtime.metrics","msg":"metrics
server is starting to listen","addr":"127.0.0.1:8080"}
{"level":"info","ts":1612732107.768796,"logger":"watches","msg":"Environment variable not
set; using default
value","envVar":"ANSIBLE_VERBOSITY_MEMCACHED_CACHE_EXAMPLE_COM","default":
2}
{"level":"info","ts":1612732107.7688773,"logger":"cmd","msg":"Environment variable not set;
using default
value","Namespace":"","envVar":"ANSIBLE_DEBUG_LOGS","ANSIBLE_DEBUG_LOGS":fals
e}
```

```
{"level":"info","ts":1612732107.7688901,"logger":"ansible-controller","msg":"Watching
resource","Options.Group":"cache.example.com","Options.Version":"v1","Options.Kind":"Memc
ached"}
{"level":"info","ts":1612732107.770032,"logger":"proxy","msg":"Starting to
serve","Address":"127.0.0.1:8888"}
I0207 21:08:27.770185       7 leaderelection.go:243] attempting to acquire leader lease
memcached-operator-system/memcached-operator...
{"level":"info","ts":1612732107.770202,"logger":"controller-runtime.manager","msg":"starting
metrics server","path":"/metrics"}
I0207 21:08:27.784854       7 leaderelection.go:253] successfully acquired lease
memcached-operator-system/memcached-operator
{"level":"info","ts":1612732107.7850506,"logger":"controller-
runtime.manager.controller.memcached-controller","msg":"Starting
EventSource","source":"kind source: cache.example.com/v1, Kind=Memcached"}
{"level":"info","ts":1612732107.8853772,"logger":"controller-
runtime.manager.controller.memcached-controller","msg":"Starting Controller"}
{"level":"info","ts":1612732107.8854098,"logger":"controller-
runtime.manager.controller.memcached-controller","msg":"Starting workers","worker
count":4}
```

#### 5.4.6.4.2. 启用完整的 Ansible 结果会包括在日志中

您可以将环境变量 **ANSIBLE_DEBUG_LOGS** 设置为 **True**，以启用检查完整 Ansible 结果日志，这在调试时很有用。

**流程**

- 编辑 **config/manager/manager.yaml** 和 **config/default/manager_auth_proxy_patch.yaml** 文件，使其包含以下配置：

```
    containers:
    - name: manager
      env:
      - name: ANSIBLE_DEBUG_LOGS
        value: "True"
```

#### 5.4.6.4.3. 在日志中启用详细调试

在开发基于 Ansible 的 Operator 时，在日志中启用额外的调试可能会有所帮助。

**流程**

- 在自定义资源中添加 **ansible.sdk.operatorframework.io/verbosity** 注解，以启用您想要的详细程度。例如：

```
apiVersion: "cache.example.com/v1alpha1"
kind: "Memcached"
metadata:
  name: "example-memcached"
  annotations:
    "ansible.sdk.operatorframework.io/verbosity": "4"
spec:
  size: 4
```

### 5.4.7. 自定义资源状态管理

#### 5.4.7.1. 基于 Ansible 的 Operator 中的自定义资源状态

基于 Ansible 的 Operator 会自动将上一次 Ansible 运行的一般信息更新到自定义资源 (CR) **status** 子资源中。其中包括成功和失败任务的数量以及相关的错误消息，如下所示：

```
status:
  conditions:
  - ansibleResult:
      changed: 3
      completion: 2018-12-03T13:45:57.13329
      failures: 1
      ok: 6
      skipped: 0
    lastTransitionTime: 2018-12-03T13:45:57Z
    message: 'Status code was -1 and not [200]: Request failed: <urlopen error [Errno
      113] No route to host>'
    reason: Failed
    status: "True"
    type: Failure
  - lastTransitionTime: 2018-12-03T13:46:13Z
    message: Running reconciliation
    reason: Running
    status: "True"
    type: Running
```

基于 Ansible 的 Operator 还支持 Operator 作者通过 **k8s_status** Ansible 模块提供自定义状态值，该模块包含在 **operator_sdk.util** 集中。作者可以根据需要使用任意键值对从 Ansible 内部更新 **status**。

基于 Ansible 的 Operator 默认始终包含如上所示的通用 Ansible 运行输出。如果不希望您的应用程序使用 Ansible 输出来更新状态，您可以通过应用程序来手动跟踪状态。

#### 5.4.7.2. 手动跟踪自定义资源状态

您可以使用 **operator_sdk.util** 集合来修改基于 Ansible 的 Operator，以手动从应用程序跟踪自定义资源（CR）状态。

**先决条件**

- 使用 Operator SDK 创建基于 Ansible 的 Operator 项目

**流程**

1. 更新 **watches.yaml** 文件，把一个 **manageStatus** 项设置为 **false**：

```
- version: v1
  group: api.example.com
  kind: <kind>
  role: <role>
  manageStatus: false
```

2. 使用 **operator_sdk.util.k8s_status** Ansible 模块来更新子资源。例如，使用键 **test** 和值 **data** 更新，**operator_sdk.util** 可以按以下方式使用：

```
- operator_sdk.util.k8s_status:
    api_version: app.example.com/v1
    kind: <kind>
    name: "{{ ansible_operator_meta.name }}"
    namespace: "{{ ansible_operator_meta.namespace }}"
    status:
      test: data
```

3. 您可以为角色在 **meta/main.yml** 文件中声明集合，用于构建基于 Ansible 的 Operator：

```
collections:
  - operator_sdk.util
```

4. 在角色 meta 中声明集合后，您可以直接调用 **k8s_status** 模块：

```
k8s_status:
  ...
  status:
    key1: value1
```

## 5.5. 基于 HELM 的 OPERATOR

### 5.5.1. 开始使用基于 Helm 的 Operator 的 Operator SDK

Operator SDK 包括生成一个 Operator 项目的选项，它利用现有 Helm chart 将 Kubernetes 资源部署为统一应用程序，而无需编写任何 Go 代码。

如需演示使用 Operator SDK 提供的工具和库来设置并运行基于 Helm 的 Operator 的基本知识，Operator 开发人员可以为 Nginx 构建一个基于 Helm 的 Operator 示例，并将它部署到集群中。

#### 5.5.1.1. 先决条件

- 已安装 operator SDK CLI
- 已安装 OpenShift CLI（**oc**）v4.7+
- 使用具有 **cluster-admin** 权限的 **oc** 登录到 OpenShift Container Platform 4.7 集群
- 要允许集群拉取镜像，推送镜像的存储库必须设置为公共的存储库，或必须配置一个镜像 pull secret。

#### 5.5.1.2. 创建并部署基于 Helm 的 Operator

您可以使用 Operator SDK 为 Nginx 构建和部署简单基于 Helm 的 Operator。

**流程**

1. **创建一个项目。**
   a. 创建您的项目目录：

```
$ mkdir nginx-operator
```

b. 切换到项目所在的目录：

```
$ cd nginx-operator
```

c. 使用 **helm** 插件运行 **operator-sdk init** 命令以初始化项目：

```
$ operator-sdk init \
    --plugins=helm
```

2. **创建 API。**
创建简单的 Nginx API：

```
$ operator-sdk create api \
    --group demo \
    --version v1 \
    --kind Nginx
```

此 API 使用 **helm create** 命令的内置 Helm Chart 网卡。

3. **构建并推送 Operator 镜像。**
使用默认的 **Makefile** 目标来构建和推送 Operator。使用镜像的 pull spec 设置 **IMG**，该 spec 使用您可推送到的 registry：

```
$ make docker-build docker-push IMG=<registry>/<user>/<image_name>:<tag>
```

4. **运行 Operator。**

a. 安装 CRD：

```
$ make install
```

b. 将项目部署到集群中。将 **IMG** 设置为您推送的镜像：

```
$ make deploy IMG=<registry>/<user>/<image_name>:<tag>
```

5. **添加安全性上下文约束(SCC)。**
Nginx 服务帐户需要特权访问权限才能在 OpenShift Container Platform 中运行。将以下 SCC 添加到 **nginx-sample** pod 的服务帐户中：

```
$ oc adm policy add-scc-to-user \
    anyuid system:serviceaccount:nginx-operator-system:nginx-sample
```

6. **创建示例自定义资源（CR）。**

a. 创建一个示例 CR：

```
$ oc apply -f config/samples/demo_v1_nginx.yaml \
    -n nginx-operator-system
```

b. 查看 CR 协调 Operator：

```
$ oc logs deployment.apps/nginx-operator-controller-manager \
    -c manager \
    -n nginx-operator-system
```

7. **清理。**
   运行以下命令清理在此流程中创建的资源：

```
$ make undeploy
```

#### 5.5.1.3. 后续步骤

- 如需更深入地了解如何构建基于 Helm 的 Operator，请参阅基于 Helm 的 Operator SDK 指南 。

### 5.5.2. 基于 Helm 的 Operator 的 operator SDK 指南

Operator 开发人员可以利用 Operator SDK 中的 Helm 支持来为 Nginx 构建基于 Helm 的 Operator 示例，并管理其生命周期。本教程介绍了以下过程：

- 创建 Nginx 部署
- 确保部署大小与 **Nginx** 自定义资源（CR）spec 指定的大小相同
- 使用 status writer 带有 **nginx** Pod 的名称来更新 **Nginx** CR 状态

通过以下两个 Operator Framework 核心组件来完成此过程：

**Operator SDK**

**operator-sdk** CLI 工具和 **controller-runtime** 库 API

**Operator Lifecycle Manager (OLM)**

集群中 Operator 的安装、升级和基于角色的访问控制（RBAC）

**注意**

本教程的内容比基于 Helm 的 Operator 开始使用 Operator SDK 的内容更详细。

#### 5.5.2.1. 先决条件

- 已安装 operator SDK CLI
- 已安装 OpenShift CLI（**oc**）v4.7+
- 使用具有 **cluster-admin** 权限的 **oc** 登录到 OpenShift Container Platform 4.7 集群
- 要允许集群拉取镜像，推送镜像的存储库必须设置为公共的存储库，或必须配置一个镜像 pull secret。

#### 5.5.2.2. 创建一个项目

使用 Operator SDK CLI 创建名为 **nginx-operator** 的项目。

**流程**

1. 为项目创建一个目录：

```
$ mkdir -p $HOME/projects/nginx-operator
```

2. 进入该目录：

```
$ cd $HOME/projects/nginx-operator
```

3. 使用 **helm** 插件运行 **operator-sdk init** 命令以初始化项目：

```
$ operator-sdk init \
    --plugins=helm \
    --domain=example.com \
    --group=demo \
    --version=v1 \
    --kind=Nginx
```

**注意**

默认情况下，**helm** 插件使用样板 Helm Chart 初始化项目。您可以使用其他标记（如 **--helm-chart** 标志）使用现有 Helm chart 初始化项目。

**init** 命令创建 **nginx-operator** 项目，专门用于监视 API 版本为 **example.com/v1** 和 kind **Nginx** 的资源。

4. 对于基于 Helm 的项目，**init** 命令根据 chart 的默认清单部署的资源，在 **config/rbac/role.yaml** 文件中生成 RBAC 规则。验证此文件生成的规则是否满足 Operator 的权限要求。

##### 5.5.2.2.1. 现有 Helm chart

您可以使用以下标记，而不是使用样板 Helm Chart 创建项目，而是使用现有 chart（可以从本地文件系统或远程 Chart 仓库中）使用现有 chart:

- **--helm-chart**
- **--helm-chart-repo**
- **--helm-chart-version**

如果指定了 **--helm-chart** 标志，**--group**、**--version** 和 **--kind** 标志将变为可选。如果保留未设置，则使用以下默认值：

| 标记 | 值 |
|---|---|
| **--domain** | **my.domain** |
| **--group** | **charts** |
| **--version** | **v1** |
| **--kind** | 从指定的 chart 中分离 |

如果 **--helm-chart** 标志指定本地 chart 归档，如 **example-chart-1.2.0.tgz** 或目录，则 chart 被验证并解包或复制到项目中。否则，Operator SDK 会尝试从远程存储库中获取 chart。

如果没有通过 **--helm-chart-repo** 标志指定自定义存储库 URL，则支持以下 chart 引用格式：

| 格式 | 描述 |
| --- | --- |
| **<repo_name>/<chart_name>** | 从名为 **<repo_name>** 的 helm chart 中获取名为 **<chart_name>** 的 Helm chart，如 **$HELM_HOME/repositories/repositories.yaml** 文件指定。使用 **helm repo add** 命令来配置此文件。 |
| **<url>** | 通过指定的 URL 获取 Helm Chart 归档。 |

如果自定义仓库 URL 由 **--helm-chart-repo** 指定，则支持以下 chart 引用格式：

| 格式 | 描述 |
| --- | --- |
| **<chart_name>** | 在由 **--helm-chart-repo** URL 值指定的 Helm Chart 仓库中获取名为 **<chart_name>** 的 Helm Chart。 |

如果 **--helm-chart-version** 标志未设置，Operator SDK 会获取最新可用的 Helm Chart 版本。否则，它会获取指定的版本。当使用 **--helm-chart** 标记指定一个特定版本（例如一个本地路径或 URL）的 chart 时，**--helm-chart-version** 标志不会被使用。

如需更多详细信息和示例，请运行：

```
$ operator-sdk init --plugins helm --help
```

#### 5.5.2.2.2. PROJECT 文件

**operator-sdk init** 命令生成的文件中是一个 Kubebuilder **PROJECT** 文件。从项目 root 运行的后续 **operator-sdk** 命令以及 **help** 输出可读取该文件，并注意项目类型是 Helm。例如：

```
domain: example.com
layout: helm.sdk.operatorframework.io/v1
projectName: helm-operator
resources:
- group: demo
  kind: Nginx
  version: v1
version: 3-alpha
```

### 5.5.2.3. 了解 Operator 逻辑

在本例中，**nginx-operator** 项目会针对每个 **Nginx** 自定义资源 (CR) 执行以下协调逻辑：

- 如果尚无 Nginx 部署，请创建一个。
- 如果尚无 Nginx 服务，请创建一个。
- 如果被启用且不存在，请创建一个 Nginx ingress。

- 确保部署、服务和可选入口与 **Nginx** CR 指定的配置匹配，如副本数、镜像和服务类型。

默认情况下，**nginx-operator** 项目会监视 **Vginx** 资源事件，如 **watches.yaml** 文件中所示，并使用指定 Chart 执行 Helm 发行版本：

```
# Use the 'create api' subcommand to add watches to this file.
- group: demo
  version: v1
  kind: Nginx
  chart: helm-charts/nginx
# +kubebuilder:scaffold:watch
```

#### 5.5.2.3.1. Helm chart 示例

创建 Helm Operator 项目后，Operator SDK 会创建一个 Helm Chart 示例，其中包含一组模板，用于简单的 Nginx 发行版本。

本例中，针对部署、服务和 Ingress 资源提供了模板，另外还有 **NOTES.txt** 模板，Helm Chart 开发人员可利用该模板传达有关发型版本的实用信息。

如果您对 Helm chart 有一定的了解，请参阅 Helm 开发人员文档。

#### 5.5.2.3.2. 修改自定义资源规格

Helm 使用名为 values 的概念来自定义 Helm Chart 的默认配置，该 chart 在 **values.yaml** 文件中定义。

您可以通过在自定义资源（CR）spec 中设置所需的值来覆盖这些默认值。以副本数量为例。

**流程**

1. 在默认情况下，**helm-charts/nginx/values.yaml** 文件有一个设置为 **1** 的名为 **replicaCount** 的值。要在部署中有两个 Nginx 实例，您的 CR spec 必须包含 **replicaCount: 2**。
   编辑 **config/samples/demo_v1_nginx.yaml** 文件以设置 **replicaCount: 2**：

```
apiVersion: demo.example.com/v1
kind: Nginx
metadata:
  name: nginx-sample
...
spec:
...
  replicaCount: 2
```

2. 同样，服务端口默认设置为 **80**。要使用 **8080**，编辑 **config/samples/demo_v1_nginx.yaml** 文件来设置 **spec.port: 8080**，它会添加服务端口覆盖：

```
apiVersion: demo.example.com/v1
kind: Nginx
metadata:
  name: nginx-sample
spec:
  replicaCount: 2
  service:
    port: 8080
```

Helm Operator 应用整个 spec，将其视为 values 文件内容，与 **helm install -f ./overrides.yaml** 命令的工作方式类似。

#### 5.5.2.4. 运行 Operator

您可以使用 Operator SDK CLI 构建和运行 Operator：

- 作为 Go 程序在集群外本地运行。
- 作为集群的部署运行。
- 捆绑 Operator，并使用 Operator Lifecycle Manager（OLM）在集群中部署。

##### 5.5.2.4.1. 在集群外本地运行

您可以作为集群外的 Go 程序运行您的 Operator 项目。这可以加快部署和测试的速度，对于开发非常有用。

**流程**

- 运行以下命令，以在 **~/.kube/config** 文件中配置的集群中安装自定义资源定义（CRD），并在本地运行 Operator：

```
$ make install run
```

输出示例

```
...
{"level":"info","ts":1612652419.9289865,"logger":"controller-runtime.metrics","msg":"metrics
server is starting to listen","addr":":8080"}
{"level":"info","ts":1612652419.9296563,"logger":"helm.controller","msg":"Watching
resource","apiVersion":"demo.example.com/v1","kind":"Nginx","namespace":"","reconcilePeriod
":"1m0s"}
{"level":"info","ts":1612652419.929983,"logger":"controller-runtime.manager","msg":"starting
metrics server","path":"/metrics"}
{"level":"info","ts":1612652419.930015,"logger":"controller-runtime.manager.controller.nginx-
controller","msg":"Starting EventSource","source":"kind source: demo.example.com/v1,
Kind=Nginx"}
{"level":"info","ts":1612652420.2307851,"logger":"controller-runtime.manager.controller.nginx-
controller","msg":"Starting Controller"}
{"level":"info","ts":1612652420.2309358,"logger":"controller-runtime.manager.controller.nginx-
controller","msg":"Starting workers","worker count":8}
```

##### 5.5.2.4.2. 准备 Operator 以使用支持的镜像

在 OpenShift Container Platform 上运行基于 Helm 的 Operator 之前，请将项目更新为使用支持的镜像。

**流程**

1. 更新项目根级别 Dockerfile 以使用支持的镜像。更改默认构建器镜像引用：

```
FROM quay.io/operator-framework/helm-operator:v1.3.0
```

改为：

```
FROM registry.redhat.io/openshift4/ose-helm-operator:v4.7
```

**重要**

使用与 Operator SDK 版本匹配的构建器镜像版本。因项目布局或 *scaffolding* 而导致问题，特别是将较新的 Operator SDK 版本与下游 OpenShift Container Platform 构建器镜像混合时。

2. 在 **config/default/manager_auth_proxy_patch.yaml** 文件中，修改 **image** 值：

```
gcr.io/kubebuilder/kube-rbac-proxy:<tag>
```

使用支持的镜像：

```
registry.redhat.io/openshift4/ose-kube-rbac-proxy:v4.7
```

#### 5.5.2.4.3. 作为集群的部署运行

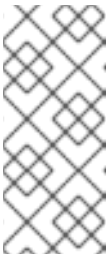
您可以作为一个部署在集群中运行 Operator 项目。

**流程**

1. 运行以下 **make** 命令来构建和推送 Operator 镜像。在以下步骤中修改 **IMG** 参数来引用您可访问的库。您可以获取在存储库站点（如 Quay.io）存储容器的帐户。

   a. 构建镜像：

   ```
   $ make docker-build IMG=<registry>/<user>/<image_name>:<tag>
   ```

   **注意**

   由 SDK 为 Operator 生成的 Dockerfile 明确引用了 **go 构建** 的 **GOARCH=amd64**。对于非 AMD64 构架，这已被应用于 **GOARCH=$TARGETARCH**。Docker 会自动将 环境变量设置为 **-platform** 指定的值。使用 Buildah，**-build-arg** 需要用于目的。如需更多信息，请参阅 多个架构。

   b. 将镜像推送到存储库：

   ```
   $ make docker-push IMG=<registry>/<user>/<image_name>:<tag>
   ```

   **注意**

   镜像的名称和标签，如 **IMG=<registry> /<user> /<image_name>:<tag>**，在两个命令中都可在您的 Makefile 中设置。修改 **IMG ?= controller:latest** 值来设置您的默认镜像名称。

2. 运行以下命令来部署 Operator：

```
$ make deploy IMG=<registry>/<user>/<image_name>:<tag>
```

默认情况下，这个命令会创建一个带有 Operator 项目名称的命名空间，格式为 **<project_name>-system**，用于部署。此命令还从 **config/rbac** 安装 RBAC 清单。

3. 验证 Operator 是否正在运行：

```
$ oc get deployment -n <project_name>-system
```

输**出示例**

```
NAME                                    READY   UP-TO-DATE   AVAILABLE   AGE
<project_name>-controller-manager       1/1     1            1           8m
```

#### 5.5.2.4.4. 捆绑 Operator 并使用 Operator Lifecycle Manager 进行部署

Operator Lifecycle Manager（OLM）可帮助您在 Kubernetes 集群上安装、更新 Operator 及其相关服务的生命周期。OLM 在 OpenShift Container Platform 上默认安装，并作为 Kubernetes 扩展运行，以便您可以在没有任何额外工具的情况下将 Web 控制台和 OpenShift CLI（**oc**）用于所有 Operator 生命周期管理功能。

Operator Bundle Format 是 Operator SDK 和 OLM 的默认打包方法。您可以通过使用 Operator SDK 来通过 OLM 构建、推送、验证和运行捆绑包镜像，让 Operator 准备好进行 OLM。

**先决条件**

- 在开发工作站上安装 operator SDK CLI
- 已安装 OpenShift CLI（**oc**）v4.7+
- Operator Lifecycle Manager（OLM）安装在一个基于 Kubernetes 的集群上（如果使用 **apiextensions.k8s.io/v1** CRD，则为 v1.16.0 或更新版本，如 OpenShift Container Platform 4.7）
- 使用具有 **cluster-admin** 权限的账户使用 **oc** 登录到集群
- 使用 Operator SDK 初始化 operator 项目

**流程**

1. 在 Operator 项目目录中运行以下 **make** 命令来构建和推送 Operator 镜像。在以下步骤中修改 **IMG** 参数来引用您可访问的库。您可以获取在存储库站点（如 Quay.io）存储容器的帐户。

   a. 构建镜像：

```
$ make docker-build IMG=<registry>/<user>/<operator_image_name>:<tag>
```

**注意**

由 SDK 为 Operator 生成的 Dockerfile 明确引用了 **go 构建** 的 **GOARCH=amd64**。对于非 AMD64 构架，这已被应用于 **GOARCH=$TARGETARCH**。Docker 会自动将 环境变量设置为 **-platform** 指定的值。使用 Buildah，**-build-arg** 需要用于目的。如需更多信息，请参阅 多个架构。

b. 将镜像推送到存储库：

```
$ make docker-push IMG=<registry>/<user>/<operator_image_name>:<tag>
```

2. 运行 **make bundle** 命令创建 Operator 捆绑包清单，该命令调用多个命令，其中包括 Operator SDK **generate bundle** 和 **bundle validate** 子命令：

```
$ make bundle IMG=<registry>/<user>/<operator_image_name>:<tag>
```

Operator 的捆绑包清单描述了如何显示、创建和管理应用程序。**make bundle** 命令在 Operator 项目中创建以下文件和目录：

- 包含 **ClusterServiceVersion** 对象的捆绑包清单目录，名为 **bundle/manifests**
- 名为 **bundle/metadata** 的捆绑包元数据目录
- **config/crd** 目录中的所有自定义资源定义（CRD）
- 一个 Dockerfile **bundle.Dockerfile**

然后，使用 **operator-sdk bundle validate** 自动验证这些文件，以确保磁盘上的捆绑包的格式是正确的。

3. 运行以下命令来构建和推送捆绑包镜像。OLM 使用索引镜像来消耗 Operator 捆绑包，该镜像引用一个或多个捆绑包镜像。

a. 构建捆绑包镜像。使用您要推送镜像的 registry、用户命名空间和镜像标签的详情，设置 **BUNDLE_IMAGE**：

```
$ make bundle-build BUNDLE_IMG=<registry>/<user>/<bundle_image_name>:<tag>
```

b. 推送捆绑包镜像：

```
$ docker push <registry>/<user>/<bundle_image_name>:<tag>
```

4. 使用以下命令，检查集群中的 OLM 状态：

```
$ operator-sdk olm status \
   --olm-namespace=openshift-operator-lifecycle-manager
```

5. 使用 Operator SDK 中的 OLM 集成在集群中运行 Operator：

```
$ operator-sdk run bundle \
   [-n <namespace>] \ 1
   <registry>/<user>/<bundle_image_name>:<tag>
```

1 默认情况下，命令会在 **~/.kube/config** 文件中当前活跃的项目中安装 Operator。您可以添加 **-n** 标志来为安装设置不同的命名空间范围。

这个命令执行以下操作：

- 使用注入的捆绑包镜像创建索引镜像。
- 创建指向新索引镜像的目录源，以便 OperatorHub 能够发现 Operator。

- 通过创建一个 Operator 组、订阅、安装计划以及所有其他必要的对象（包括 RBAC），将 Operator 部署到集群中。

#### 5.5.2.5. 创建自定义资源

安装 Operator 后，您可以通过创建一个由 Operator 在集群中提供的自定义资源（CR）来测试它。

**先决条件**

- Nginx Operator 示例，它提供了 **Nginx** CR，在集群中安装

**流程**

1. 切换到安装 Operator 的命名空间。例如，如果使用 **make deploy** 命令部署 Operator：

```
$ oc project nginx-operator-system
```

2. 编辑 **config/samples/demo_v1_nginx.yaml** 中的 **Nginx** CR 清单示例，使其包含以下规格：

```
apiVersion: demo.example.com/v1
kind: Nginx
metadata:
  name: nginx-sample
...
spec:
...
  replicaCount: 3
```

3. Nginx 服务帐户需要特权访问权限才能在 OpenShift Container Platform 中运行。将以下安全性上下文约束 (SCC) 添加到 **nginx-sample** pod 的服务帐户中：

```
$ oc adm policy add-scc-to-user \
    anyuid system:serviceaccount:nginx-operator-system:nginx-sample
```

4. 创建 CR:

```
$ oc apply -f config/samples/demo_v1_nginx.yaml
```

5. 确保 **Nginx** Operator 为示例 CR 创建部署，其大小正确：

```
$ oc get deployments
```

输出示例

```
NAME                                  READY   UP-TO-DATE   AVAILABLE   AGE
nginx-operator-controller-manager     1/1     1            1           8m
nginx-sample                          3/3     3            3           1m
```

6. 检查 pod 和 CR 状态，以确认其状态是否使用 Nginx pod 名称更新。

   a. 检查 pod:

```
$ oc get pods
```

-

输**出示例**

```
NAME                                READY   STATUS    RESTARTS   AGE
nginx-sample-6fd7c98d8-7dqdr        1/1     Running   0          1m
nginx-sample-6fd7c98d8-g5k7v        1/1     Running   0          1m
nginx-sample-6fd7c98d8-m7vn7        1/1     Running   0          1m
```

b. 检查 CR 状态：

```
$ oc get nginx/nginx-sample -o yaml
```

输**出示例**

```
apiVersion: demo.example.com/v1
kind: Nginx
metadata:
...
  name: nginx-sample
...
spec:
  replicaCount: 3
status:
  nodes:
  - nginx-sample-6fd7c98d8-7dqdr
  - nginx-sample-6fd7c98d8-g5k7v
  - nginx-sample-6fd7c98d8-m7vn7
```

7. 更新部署大小。

a. 更新 **config/samples/demo_v1_nginx.yaml** 文件，将 **Nginx** CR 中的 **spec.size** 字段从 **3** 改为 **5**：

```
$ oc patch nginx nginx-sample \
    -p '{"spec":{"replicaCount": 5}}' \
    --type=merge
```

b. 确认 Operator 已更改部署大小：

```
$ oc get deployments
```

输**出示例**

```
NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
nginx-operator-controller-manager   1/1     1            1           10m
nginx-sample                        5/5     5            5           3m
```

8. 清理本教程中创建的资源。

- 如果使用 **make deploy** 命令来测试 Operator，请运行以下命令：

```
$ make undeploy
```

- 如果使用 **operator-sdk run bundle** 命令来测试 Operator，请运行以下命令：

```
$ operator-sdk cleanup <project_name>
```

#### 5.5.2.6. 其他资源

- 请参阅 基于 Helm 的 Operator 的项目布局，以了解 Operator SDK 创建的目录结构。

### 5.5.3. 基于 Helm 的 Operator 的项目布局

**operator-sdk** CLI 可为每个 Operator 项目生成或 *scaffold* 多个 软件包和文件。

#### 5.5.3.1. 基于 Helm 的项目布局

使用 **operator-sdk init --plugins helm** 命令生成的基于 Helm 的 Operator 项目包含以下目录和文件：

| 文件/文件夹 | 用途 |
| --- | --- |
| **config** | kustomize 清单，用于在 Kubernetes 集群上部署 Operator。 |
| **helm-charts/** | Helm Chart 使用 **operator-sdk create api** 命令初始化。 |
| **Docker** | 用于使用 **make docker-build** 命令构建 Operator 镜像。 |
| **watches.yaml** | Group/version/kind（GVK）和 Helm Chart 的位置。 |
| **Makefile** | 用于管理项目的目标。 |
| **PROJECT** | 包含 Operator 元数据信息的 YAML 文件。 |

### 5.5.4. Operator SDK 中的 Helm 支持

#### 5.5.4.1. Helm chart

通过 Operator SDK 生成 Operator 项目的其中一种方案是利用现有 Helm Chart 来部署 Kubernetes 资源作为统一应用程序，而无需编写任何 Go 代码。这种基于 Helm 的 Operator 非常适合于推出时所需逻辑极少的无状态应用程序，因为更改应该应用于作为 Chart 一部分生成的 Kubernetes 对象。这听起来似乎很有局限性，但就 Kubernetes 社区构建的 Helm Chart 的增长而言，这足以满足它们的大量用例需要。

Operator 的主要功能是从代表应用程序实例的自定义对象中读取数据，并使其所需状态与正在运行的状态相匹配。对于基于 Helm 的 Operator，对象的 **spec** 字段是一个配置选项列表，通常在 Helm **values.yaml** 文件中描述。您可以不使用 Helm CLI（如 **helm install -f values.yaml**）来通过标志设置这些值，而是在自定义资源 (CR) 中表达这些值，因为 CR 作为原生 Kubernetes 对象能够实现应用的 RBAC 以及审核跟踪所带来的好处。

举一个名为 **Tomcat** 的简单 CR 示例：

```
apiVersion: apache.org/v1alpha1
kind: Tomcat
metadata:
```

```
  name: example-app
spec:
  replicaCount: 2
```

**replicaCount** 值（本例中为 **2**）会被传播到使用以下内容的 Chart 模板中：

```
{{ .Values.replicaCount }}
```

构建并部署完 Operator 后，您可通过新建一个 CR 实例来部署新的应用实例，或使用 **oc** 命令列出所有环境中运行的不同实例：

```
$ oc get Tomcats --all-namespaces
```

不要求使用 Helm CLI 或安装 Tiller；基于 Helm 的 Operator 会从 Helm 项目中导入代码。您要做的只是运行一个 Operator 实例，并使用自定义资源定义 (CRD) 注册 CR。因其遵循 RBAC，所以可以更容易防止生产环境改变。

## 5.6. 定义集群服务版本（CSV）

由 **ClusterServiceVersion** 对象定义的 *集群服务版本*（CSV）是一个利用 Operator 元数据创建的 YAML 清单，可辅助 Operator Lifecycle Manager（OLM）在集群中运行 Operator。它是 Operator 容器镜像附带的元数据，用于在用户界面填充徽标、描述和版本等信息。此外，CSV 还是运行 Operator 所需的技术信息来源，类似于其需要的 RBAC 规则及其管理或依赖的自定义资源 (CR)。

Operator SDK 包括 CSV 生成器，用于为当前 Operator 项目生成 CSV，使用 YAML 清单和 Operator 源文件中包含的信息自定义。

借助生成 CSV 的命令，Operator 作者便无需深入掌握为了让其 Operator 与 OLM 交互或向 Catalog Registry 发布元数据所需的 OLM 知识。此外，因为实现了新的 Kubernetes 和 OLM 功能，CSV spec 可能会随着时间的推移而有所变化，而 Operator SDK 可轻松扩展其更新系统，以应对 CSV 的未来新功能。

### 5.6.1. CSV 生成的工作方式

Operator 捆绑包清单，其中包括集群服务版本（CSV），描述如何使用 Operator Lifecycle Manager（OLM）显示、创建和管理应用程序。Operator SDK 中的 CSV 生成器（由 **generate bundle** 子命令调用）是将 Operator 发布到目录并使用 OLM 部署的第一个步骤。子命令需要特定的输入清单来构造 CSV 清单，在调用命令时会读取所有输入，以及 CSV 基础，以便预先生成或重新生成 CSV。

通常，**generate kustomize manifests** 子命令会首先运行，以生成由 **generate bundle** 子命令使用的输入 Kustomize 基础。但是，Operator SDK 提供 **make bundle** 命令，它自动执行一些任务，包括按顺序运行以下子命令：

1. **generate kustomize manifests**
2. **generate bundle**
3. **bundle validate**

**其他资源**

- 如需包含生成捆绑包和 CSV 的完整流程，请参阅捆绑 Operator 并使用 Operator Lifecycle Manager 部署。

#### 5.6.1.1. 生成的文件和资源

**make bundle** 命令在 Operator 项目中创建以下文件和目录：

- 名为 **bundle/manifests** 的捆绑包清单目录，其中包含 **ClusterServiceVersion**（CSV）对象
- 名为 **bundle/metadata** 的捆绑包元数据目录
- **config/crd** 目录中的所有自定义资源定义（CRD）
- 一个 Dockerfile **bundle.Dockerfile**

以下资源通常包含在 CSV 中：

**角色**
定义命名空间中的 Operator 权限。

**ClusterRole**
定义集群范围的 Operator 权限。

**Deployment**
定义如何在 pod 中运行 Operator 的 Operand。

**CustomResourceDefinition (CRD)**
定义 Operator 协调的自定义资源。

**自定义资源示例**
遵循特定 CRD 规格的资源示例。

#### 5.6.1.2. 版本管理

**generate bundle** 子命令的 **--version** 标志在首次创建以及升级现有捆绑包时，为您提供语义版本。

通过在 **Makefile** 中设置 **VERSION** 变量，当使用 **make bundle** 命令运行 **generate bundle** 子命令时使用该值自动调用 **--version** 标志。CSV 版本与 Operator 版本相同，在升级 Operator 版本时会生成新 CSV。

### 5.6.2. 手动定义的 CSV 字段

很多 CSV 字段无法使用生成的、不属于 Operator SDK 的特殊通用清单进行填充。这些字段大多由人工编写，是一些有关 Operator 和各种自定义资源定义 (CRD) 的元数据。

Operator 作者必须直接修改其集群服务版本(CSV)YAML 文件，将个性化数据添加到以下必填字段。当检测到任何必填字段中缺少数据时，Operator SDK 在生成 CSV 时发出警告。

下表详细介绍了需要手动定义的 CSV 字段，哪些是可选的。

**表 5.7. 必填**

| 字段 | 描述 |
| --- | --- |
| **metadata.name** | 该 CSV 的唯一名称。Operator 版本应包含在名称中，以保证唯一性，如 **app-operator.v0.1.1**。 |
| **metadata.capabilities** | 根据 Operator 成熟度模型划分的能力等级。选项包括 **Basic Install**、**Seamless Upgrades**、**Full Lifecycle**、**Deep Insights** 和 **Auto Pilot**。 |

| 字段 | 描述 |
| --- | --- |
| **spec.displayName** | 用于标识 Operator 的公共名称。 |
| **spec.description** | 有关 Operator 功能的简短描述。 |
| **spec.keywords** | 描述 Operator 的关键词。 |
| **spec.maintainers** | 维护 Operator 的个人或组织实体，含**名称**和**电子邮件地址**。 |
| **spec.provider** | Operator 的供应商（通常是机构）,含 **名称**。 |
| **spec.labels** | 供 Operator 内部使用的键值对。 |
| **spec.version** | Operator 的语义版本，如 **0.1.1**。 |
| **spec.customresourcedefinitions** | Operator 使用的任何 CRD。如果 **deploy/** 中存在任何 CRD YAML 文件，Operator SDK 将自动填充该字段。但 CRD 清单 spec 中没有的几个字段需要用户输入：<br>• **description**：CRD 描述。<br>• **resources**：CRD 利用的任何 Kubernetes 资源，如**Pod** 和 **StatefulSet** 对象。<br>• **specDescriptors**：用于 Operator 输入和输出的 UI 提示。 |

表 5.8. 选填

| 字段 | 描述 |
| --- | --- |
| **spec.replaces** | 被该 CSV 替换的 CSV 名称。 |
| **spec.links** | 与被管理的 Operator 或应用程序相关的 URL（如网站和文档），各自含**名称**和 **url**。 |
| **spec.selector** | Operator 可用于配对群集中资源的选择器。 |
| **spec.icon** | Operator 独有的 base64 编码图标，通过 **mediatype** 在 **base64data** 字段中设置。 |
| **spec.maturity** | 软件在这个版本中达到的成熟度。选项包括 **planning**、**pre-alpha**、**alpha**、**beta**、**stable**、**mature**、**inactive** 和 **deprecated**。 |

有关以上每个字段应包含哪些数据的更多详情，请参见 CSV spec。

**注意**

目前需要用户干预的几个 YAML 字段可能会从 Operator 代码中解析。

**其他资源**

- Operator 成熟度模型

#### 5.6.2.1. Operator 元数据注解

operator 开发人员可以在集群服务版本（CSV）的元数据中手动定义某些注解，以启用功能或在用户界面（UI）中突出功能，如 OperatorHub。

下表列出了可使用 **metadata.annotations** 字段手动定义的 Operator 元数据注解。

**表 5.9. 注解**

| 字段 | 描述 |
| --- | --- |
| **alm-examples** | 提供自定义资源定义（CRD）模板最小配置集。兼容的 UI 会预先填充此模板，供用户进一步自定义。 |
| **operatorframework.io/initialization-resource** | 指定安装 Operator 时必须创建的一个所需的自定义资源。必须包含带有完整 YAML 定义的模板。 |
| **operatorframework.io/suggested-namespace** | 设置部署 Operator 的建议命名空间。 |

| 字段 | 描述 |
| --- | --- |
| **operators.openshift.io/infrastructure-features** | Operator 支持的基础架构功能。在 web 控制台中通过 OperatorHub 发现 Operator 时，用户可以查看和过滤这些功能。有效的、区分大小写的值：<br>● **disconnected**：Operator 支持被镜像到断开连接的目录中，包括所有依赖项，且不需要访问互联网。Operator 列出了镜像所需的所有相关镜像。<br>● **cnf**：Operator 提供了一个 Cloud-native Network Function（CNF）Kubernetes 插件。<br>● **CNI**：Operator 提供了一个 Container Network Interface（CNI）Kubernetes 插件。<br>● **CSI**：Operator 提供了一个 Container Storage Interface（CSI）Kubernetes 插件。<br>● **FIPS**：Operator 接受底层平台的 FIPS 模式，并可用于引导到 FIPS 模式的节点。<br>  **重要**<br>只有在 **x86_64** 架构中的 OpenShift Container Platform 部署支持 FIPS 验证的/Modules in Process 加密库。<br>● **proxy-aware**：Operator 支持在代理后面的集群上运行。Operator 接受标准代理环境变量 **HTTP_PROXY** 和 **HTTPS_PROXY**，Operator Lifecycle Manager（OLM）在集群配置为使用代理时自动为 Operator 提供这些环境变量。传递给受管工作负载的 Operands 所需的环境变量。 |
| **operators.openshift.io/valid-subscription** | 用于列出使用 Operator 所需的任何特定订阅的空闲数组。例如，**'["3Scale Commercial License", "Red Hat Managed Integration"]'**。 |
| **operators.operatorframework.io/internal-objects** | 在 UI 中隐藏不用于用户操作的 CRD。 |

**使用案例示例**

**Operator 支持断开连接和代理**

```
operators.openshift.io/infrastructure-features: '["disconnected", "proxy-aware"]'
```

**Operator 需要 OpenShift Container Platform 许可证**

```
operators.openshift.io/valid-subscription: '["OpenShift Container Platform"]'
```

**Operator 需要 3scale 许可证**

```
operators.openshift.io/valid-subscription: '["3Scale Commercial License", "Red Hat Managed
Integration"]'
```

**Operator 支持断开连接和代理，且需要一个 OpenShift Container Platform 许可证**

```
operators.openshift.io/infrastructure-features: '["disconnected", "proxy-aware"]'
operators.openshift.io/valid-subscription: '["OpenShift Container Platform"]'
```

**其它资源**

- CRD 模板
- 初始化所需的自定义资源
- 设置建议的命名空间
- 为受限网络环境启用 Operator （断开连接模式）
- 隐藏内部对象
- 支持 FIPS 加密

### 5.6.3. 为受限网络环境启用 Operator

作为 Operator 作者，您的 Operator 必须满足额外要求才能在受限网络或断开连接的环境中正常运行。

**支持断开连接模式的 Operator 的要求**

- 在 Operator 的集群服务版本（CSV）中：
  - 列出 Operator 执行其功能可能需要的任何 *相关镜像*或其他容器镜像。
  - 通过摘要 (SHA) 而不是标签来引用所有指定的镜像。
- Operator 的所有依赖项还必须支持以断开连接的模式运行。
- 您的 Operator 不得要求任何非集群资源。

对于 CSV 要求，您可以以 Operator 作者的身份进行以下更改。

**先决条件**

- 包含 CSV 的 Operator 项目

**流程**

1. 为 Operator 在 CSV 的两个位置中使用 SHA 引用相关镜像：
   a. 更新 **spec.relatedImages**:

```
...
spec:
  relatedImages: (1)
    - name: etcd-operator (2)
      image: quay.io/etcd-
operator/operator@sha256:d134a9865524c29fcf75bbc4469013bc38d8a15cb5f41acfddb6
b9e492f556e4 (3)
    - name: etcd-image
      image: quay.io/etcd-
operator/etcd@sha256:13348c15263bd8838ec1d5fc4550ede9860fcbb0f843e48cbccec07
810eebb68
...
```

(1) 创建 **relatedImages** 部分并设置相关镜像列表。

(2) 指定镜像的唯一标识符。

(3) 通过摘要 (SHA) 而不是通过镜像标签指定每个镜像。

b. 当声明环境变量以注入 Operator 应使用的镜像时，更新部署的 **env** 部分：

```
spec:
  install:
    spec:
      deployments:
      - name: etcd-operator-v3.1.1
        spec:
          replicas: 1
          selector:
            matchLabels:
              name: etcd-operator
          strategy:
            type: Recreate
          template:
            metadata:
              labels:
                name: etcd-operator
            spec:
              containers:
              - args:
                - /opt/etcd/bin/etcd_operator_run.sh
                env:
                - name: WATCH_NAMESPACE
                  valueFrom:
                    fieldRef:
                      fieldPath: metadata.annotations['olm.targetNamespaces']
                - name: ETCD_OPERATOR_DEFAULT_ETCD_IMAGE (1)
                  value: quay.io/etcd-
operator/etcd@sha256:13348c15263bd8838ec1d5fc4550ede9860fcbb0f843e48cbccec07
810eebb68 (2)
                - name: ETCD_LOG_LEVEL
                  value: INFO
                image: quay.io/etcd-
operator/operator@sha256:d134a9865524c29fcf75bbc4469013bc38d8a15cb5f41acfddb6
```

```
b9e492f556e4 3
          imagePullPolicy: IfNotPresent
          livenessProbe:
            httpGet:
              path: /healthy
              port: 8080
            initialDelaySeconds: 10
            periodSeconds: 30
          name: etcd-operator
          readinessProbe:
            httpGet:
              path: /ready
              port: 8080
            initialDelaySeconds: 10
            periodSeconds: 30
          resources: {}
        serviceAccountName: etcd-operator
  strategy: deployment
```

1. 使用环境变量注入 Operator 引用的镜像。
2. 通过摘要 (SHA) 而不是通过镜像标签指定每个镜像。
3. 另外，使用摘要(SHA)而不是镜像标签（tag）引用 Operator 容器镜像。

**注意**

配置探测时，**timeoutSeconds** 值必须小于 **periodSeconds** 值。**timeoutSeconds** 默认值为 **1**。**periodSeconds** 默认值为 **10**。

2. 添加 **disconnected** 注解，这表示 Operator 在断开连接的环境中工作：

```
metadata:
  annotations:
    operators.openshift.io/infrastructure-features: '["disconnected"]'
```

OperatorHub 中可根据此基础架构功能来过滤 Operator。

### 5.6.4. 为多个架构和操作系统启用您的 Operator

Operator Lifecycle Manager (OLM) 假设所有 Operator 都在 Linux 主机中运行。但是，作为 Operator 的开发者，如果 OpenShift Container Platform 集群中有 worker 节点，您可以指定您的 Operator 是否支持管理其他架构上的工作负载。

如果 Operator 支持 AMD64 和 Linux 以外的变体，您可以向 CSV 添加标签，从而提供 Operator 列出支持的变体。标注支持的架构和操作系统的标签定义如下：

```
labels:
    operatorframework.io/arch.<arch>: supported 1
    operatorframework.io/os.<os>: supported 2
```

1. 将 **<arch>** 设置为受支持的字符串。

2 将 **<os>** 设置为受支持的字符串。

**注意**

只有默认频道的频道头的标签才会在根据标签进行过滤时考虑软件包清单。例如，这表示有可能在非默认频道中为 Operator 提供额外的架构，但该架构在 **PackageManifest** API 中不可用。

如果 CSV 不包括 **os** 标签，它将被视为默认具有以下 Linux 支持标签：

```
labels:
    operatorframework.io/os.linux: supported
```

如果 CSV 不包括 **arch** 标签，它将被视为默认具有以下 AMD64 支持标签：

```
labels:
    operatorframework.io/arch.amd64: supported
```

如果 Operator 支持多个节点架构或操作系统，您也可以添加多个标签。

**先决条件**

- 包含 CSV 的 Operator 项目
- 要支持列出多个架构和操作系统，CSV 中引用的 Operator 镜像必须是清单列表镜像。
- 要使 Operator 在受限网络或断开连接的环境中正常工作，还必须使用摘要（SHA）而不是标签（tag）来指定引用的镜像。

**流程**

- 在 CSV 的 **metadata.labels** 中为每个 Operator 支持的架构和操作系统添加标签：

```
labels:
  operatorframework.io/arch.s390x: supported
  operatorframework.io/os.zos: supported
  operatorframework.io/os.linux: supported 1
  operatorframework.io/arch.amd64: supported 2
```

1 2 在添加新的构架或操作系统后，您必须明确包含默认的 **os.linux** 和 **arch.amd64** 变体。

**其他资源**

- 如需有关清单列表的更多信息，请参阅Image Manifest V 2, Schema 2 说明。

### 5.6.4.1. Operator 的架构和操作系统支持

在标记或过滤支持多个架构和操作系统的 Operator 时，OpenShift Container Platform 上的 Operator Lifecycle Manager (OLM) 支持以下字符串：

**表 5.10. OpenShift Container Platform 支持的架构**

| 架构 | 字符串 |
| --- | --- |
| AMD64 | **amd64** |
| 64-bit PowerPC little-endian | **ppc64le** |
| IBM Z | **s390x** |

**表 5.11. OpenShift Container Platform 支持的操作系统**

| 操作系统 | 字符串 |
| --- | --- |
| Linux | **linux** |
| z/OS | **zos** |

**注意**

OpenShift Container Platform 的不同版本和其他基于 Kubernetes 的发行版本可能支持不同的架构和操作系统集合。

### 5.6.5. 设置建议的命名空间

有些 Operator 必须部署到特定命名空间中，或使用特定命名空间中的辅助资源进行部署，才能正常工作。如果从订阅中解析，Operator Lifecycle Manager(OLM)会将 Operator 的命名空间的资源默认设置为订阅的命名空间。

作为 Operator 作者，您可以将所需的目标命名空间作为集群服务版本(CSV)的一部分来控制为 Operator 安装的资源的最终命名空间。使用 OperatorHub 将 Operator 添加到集群时，此操作可让 Web 控制台在安装过程中为集群管理员自动填充建议的命名空间。

**流程**

- 在 CSV 中，将 **operatorframework.io/suggested-namespace** 注解设置为建议的命名空间：

```
metadata:
  annotations:
    operatorframework.io/suggested-namespace: <namespace> 1
```

1 设置建议的命名空间。

### 5.6.6. 启用 Operator 条件

Operator Lifecycle Manager（OLM）为 Operator 提供一个频道来交流影响 Operator 在管理 Operator 的复杂状态。默认情况下，OLM 在安装 Operator 时会创建一个 **OperatorCondition** 自定义资源定义（CRD）。根据 **OperatorCondition** 自定义资源（CR）中设置的条件，OLM 的行为会相应更改。

要支持 Operator 条件，Operator 必须能够读取由 OLM 创建的 **OperatorCondition** CR，并具有完成以下任务的能力：

- 获取特定条件。
- 设置特定条件的状态。

这可以通过使用 **operator-lib** 库来实现。Operator 作者可在 Operator 中提供 **controller-runtime** 客户端，以便该程序库访问集群中 Operator 拥有的 **OperatorCondition** CR。

该程序库提供了一个通用的 **Conditions** 接口，它使以下方法在 **OperatorCondition** CR 中 **Get** 和 **Set** 一个 **conditionType**：

**Get**

要获得特定条件，程序库使用来自 **controller-runtime** 的 **client.Get** 函数，它需要在 **conditionAccessor** 中存在类型 **type.NamespacedName** 的 **ObjectKey**。

**Set**

要更新特定条件的状态，程序库使用来自 **controller-runtime** 的 **client.Update** 功能。如果 CRD 中不存在 **conditionType**，则会出现错误。

Operator 只允许修改 CR 的 **status** 子资源。operator 可以删除或更新 **status.conditions** 数组，使其包含条件。有关条件中字段的格式和描述的详情，请查看上游的条件 GoDocs。

**注意**

Operator SDK v1.3.0 支持 **operator-lib** v0.3.0。

**先决条件**

- 使用 Operator SDK 生成一个 Operator 项目。

**流程**

在 Operator 项目中启用 Operator 条件：

1. 在 Operator 项目的 **go.mod** 文件中，将 **operator-framework/operator-lib** 添加为所需的库：

```
module github.com/example-inc/memcached-operator

go 1.15

require (
  k8s.io/apimachinery v0.19.2
  k8s.io/client-go v0.19.2
  sigs.k8s.io/controller-runtime v0.7.0
  operator-framework/operator-lib v0.3.0
)
```

2. 在 Operator 逻辑中编写自己的构造器，会导致以下结果：
   - 接受 **controller-runtime** 客户端。
   - 接受 **conditionType**。
   - 返回一个 **Condition** 接口以更新或添加条件。

   由于 OLM 目前支持 **Upgradeable** 条件，因此可以创建一个接口，它具有访问 **Upgradeable** 条件的方法。例如：

```
import (
  ...
  apiv1 "github.com/operator-framework/api/pkg/operators/v1"
)

func NewUpgradeable(cl client.Client) (Condition, error) {
  return NewCondition(cl, "apiv1.OperatorUpgradeable")
}

cond, err := NewUpgradeable(cl);
```

在这个示例中，**NewUpgradeable** constructor 被进一步使用来创建类型为 **Condition** 的一个变量 **cond**。**cond** 变量依次使用 **Get** 和 **Set** 方法，可用于处理 OLM 的 **Upgradeable** 条件。

**其他资源**

- Operator 条件

### 5.6.7. 定义 webhook

Webhook 允许 Operator 作者在资源被保存到对象存储并由 Operator 控制器处理之前，拦截、修改、接受或拒绝资源。当 webhook 与 Operator 一同提供时，Operator Lifecycle Manager（OLM）可以管理这些 webhook 的生命周期。

Operator 的集群服务版本(CSV)资源可能包含 **webhookdefinitions** 部分，以定义以下 Webhook 类型：

- Admission webhook（validating and mutating）
- webhook 转换

**流程**

- 在 Operator 的 **spec** 部分添加 **webhookdefinitions** 部分，并使用 **ValidatingAdmissionWebhook**、**MutatingAdmissionWebhook** 或 **ConversionWebhook type** 包括任何 webhook 定义。以下示例包含所有三种类型的 Webhook:

  **包含 Webhook 的 CSV**

```
apiVersion: operators.coreos.com/v1alpha1
kind: ClusterServiceVersion
metadata:
  name: webhook-operator.v0.0.1
spec:
  customresourcedefinitions:
    owned:
    - kind: WebhookTest
      name: webhooktests.webhook.operators.coreos.io 1
      version: v1
  install:
    spec:
      deployments:
      - name: webhook-operator-webhook
        ...
        ...
        ...
```

```
  strategy: deployment
installModes:
- supported: false
  type: OwnNamespace
- supported: false
  type: SingleNamespace
- supported: false
  type: MultiNamespace
- supported: true
  type: AllNamespaces
webhookdefinitions:
- type: ValidatingAdmissionWebhook (2)
  admissionReviewVersions:
  - v1beta1
  - v1
  containerPort: 443
  targetPort: 4343
  deploymentName: webhook-operator-webhook
  failurePolicy: Fail
  generateName: vwebhooktest.kb.io
  rules:
  - apiGroups:
    - webhook.operators.coreos.io
    apiVersions:
    - v1
    operations:
    - CREATE
    - UPDATE
    resources:
    - webhooktests
  sideEffects: None
  webhookPath: /validate-webhook-operators-coreos-io-v1-webhooktest
- type: MutatingAdmissionWebhook (3)
  admissionReviewVersions:
  - v1beta1
  - v1
  containerPort: 443
  targetPort: 4343
  deploymentName: webhook-operator-webhook
  failurePolicy: Fail
  generateName: mwebhooktest.kb.io
  rules:
  - apiGroups:
    - webhook.operators.coreos.io
    apiVersions:
    - v1
    operations:
    - CREATE
    - UPDATE
    resources:
    - webhooktests
  sideEffects: None
  webhookPath: /mutate-webhook-operators-coreos-io-v1-webhooktest
- type: ConversionWebhook (4)
  admissionReviewVersions:
  - v1beta1
```

```
      - v1
      containerPort: 443
      targetPort: 4343
      deploymentName: webhook-operator-webhook
      generateName: cwebhooktest.kb.io
      sideEffects: None
      webhookPath: /convert
      conversionCRDs:
      - webhooktests.webhook.operators.coreos.io 5
...
```

1. 转换 Webhook 的目标 CRD 必须在这里存在。
2. 验证准入 Webhook。
3. 变异准入 Webhook。
4. 转换 Webhook。
5. 每个 CRD 的 **spec.PreserveUnknownFields** 属性必须设置为 **false** 或 **nil**。

**其他资源**

- Webhook 准入插件类型
- Kubernetes 文档：
  - 验证准入 webhook
  - 变异准入 webhook
  - webhook 转换

#### 5.6.7.1. 针对 OLM 的 Webhook 注意事项

使用 Operator Lifecycle Manager（OLM）部署带有 webhook 的 Operator 时，您必须定义以下内容：

- **type** 字段必须设置为 **ValidatingAdmissionWebhook**、**MutatingAdmissionWebhook** 或 **ConversionWebhook**，否则 CSV 会进入失败的阶段。
- CSV 必须包含一个部署，它的名称相当于 **webhookdefinition** 的 **deploymentName** 字段中提供的值。

创建 webhook 时，OLM 确保 webhook 仅在与 Operator 部署的 Operator 组相匹配的命名空间上操作。

**证书颁发机构限制**
将 OLM 配置为为每个部署提供一个单独的证书颁发机构（CA）。将 CA 生成并挂载到部署的逻辑最初由 API 服务生命周期逻辑使用。因此：

- TLS 证书文件挂载到部署的 **/apiserver.local.config/certificates/apiserver.crt**。
- TLS 密钥文件挂载到部署的 **/apiserver.local.config/certificates/apiserver.key**。

**Admission webhook 规则约束**
为防止 Operator 将集群配置为无法恢复的状态，OLM 如果准入 webhook 中定义的规则拦截了以下请求中的规则，则 OLM 会将 CSV 放置到失败的阶段：

- 请求目标所有组
- 请求以 **operators.coreos.com** 组为目标
- 请求目标为 **ValidatingWebhookConfigurations** 或 **MutatingWebhookConfigurations** 资源

**转换 Webhook 约束**
如果转换 Webhook 定义未遵循以下限制，OLM 会将 CSV 放置到失败的阶段：

- 带有转换 Webhook 的 CSV 只能支持 **AllNamespaces** 安装模式。
- 转换 Webhook 的目标 CRD 必须将其 **spec.preserveUnknownFields** 字段设置为 **false** 或 **nil**。
- CSV 中定义的转换 webhook 必须针对拥有的 CRD。
- 在整个集群中，给定 CRD 只能有一个转换 Webhook。

## 5.6.8. 了解您的自定义资源定义（CRD）

您的 Operator 可能会使用两类自定义资源定义 (CRD)：一类归 Operator *拥有*，另一类为 Operator 依赖的*必要* CRD。

### 5.6.8.1. 拥有的 CRD

Operator 拥有的自定义资源定义（CRD）是 CSV 最重要的部分。这类 CRD 会在您的 Operator 与所需 RBAC 规则、依赖项管理和其他 Kubernetes 概念之间建立联系。

Operator 通常会使用多个 CRD 将各个概念链接在一起，例如一个对象中的顶级数据库配置和另一对象中的副本集表示代表。这在 CSV 文件中应逐一列出。

**表 5.12. 拥有的 CRD 字段**

| 字段 | 描述 | 必需/可选 |
|---|---|---|
| **名称** | CRD 的全名。 | 必填 |
| **Version** | 该对象 API 的版本。 | 必填 |
| **Kind** | CRD 的机器可读名称。 | 必填 |
| **DisplayName** | CRD 名称的人类可读版本，如 **MongoDB Standalone**。 | 必填 |
| **描述** | 有关 Operator 如何使用该 CRD 的简短描述，或有关 CRD 所提供功能的描述。 | 必填 |
| **Group** | 该 CRD 所属的 API 组，如 **database.example.com**。 | 选填 |

| 字段 | 描述 | 必需/可选 |
| --- | --- | --- |
| **Resources** | 您的 CRD 可能拥有一类或多类 Kubernetes 对象。它们将在 **resources** 部分列出，用于告知用户他们可能需要排除故障的对象或如何连接至应用程序，如公开数据库的服务或 Ingress 规则。<br><br>建议仅列出对人重要的对象，而不必列出您编排的所有对象。例如，不要列出存储用户不会修改的内部状态的配置映射。 | 选填 |
| **SpecDescriptors**、**StatusDescriptors** 和 **ActionDescriptors** | 这些描述符是通过对终端用户来说最重要的 Operator 的某些输入或输出提示 UI 的一种方式。如果您的 CRD 包含用户必须提供的 Secret 或 ConfigMap 的名称，您可在此处指定。这些项目在兼容的 UI 中链接并突出显示。<br><br>共有以下三类描述符：<br>• **SpecDescriptors**：引用对象 **spec** 块中的字段。<br>• **StatusDescriptors**：引用对象 **status** 块中的字段。<br>• **ActionDescriptors**：引用对象上可执行的操作。<br><br>所有描述符都接受以下字段：<br>• **DisplayName**: **Spec**、**Status** 或 **Action** 的人类可读名称。<br>• **Description**:有关 **Spec**、**Status** 或 **Action** 以及 Operator 如何使用它的简短描述。<br>• **Path**：描述符描述的对象上字段的点分隔路径。<br>• **X-Descriptors**：用于决定该描述符拥有哪些"功能"以及要使用哪个 UI 组件。有关 OpenShift Container Platform 的标准 React UI X-Descriptors 列表的信息，请参见 **openshift/console** 项目。<br><br>有关描述符的更多一般信息，请参见 **openshift/console** 项目。 | 选填 |

以下示例描述了一个 **MongoDB Standalone** CRD，要求某些用户以 Secret 和配置映射的形式输入，并编排服务、有状态集、pod 和 配置映射：

### 拥有的 CRD 示例

```
- displayName: MongoDB Standalone
  group: mongodb.com
  kind: MongoDbStandalone
  name: mongodbstandalones.mongodb.com
  resources:
    - kind: Service
      name: ''
      version: v1
    - kind: StatefulSet
```

```
      name: ''
      version: v1beta2
    - kind: Pod
      name: ''
      version: v1
    - kind: ConfigMap
      name: ''
      version: v1
  specDescriptors:
    - description: Credentials for Ops Manager or Cloud Manager.
      displayName: Credentials
      path: credentials
      x-descriptors:
        - 'urn:alm:descriptor:com.tectonic.ui:selector:core:v1:Secret'
    - description: Project this deployment belongs to.
      displayName: Project
      path: project
      x-descriptors:
        - 'urn:alm:descriptor:com.tectonic.ui:selector:core:v1:ConfigMap'
    - description: MongoDB version to be installed.
      displayName: Version
      path: version
      x-descriptors:
        - 'urn:alm:descriptor:com.tectonic.ui:label'
  statusDescriptors:
    - description: The status of each of the pods for the MongoDB cluster.
      displayName: Pod Status
      path: pods
      x-descriptors:
        - 'urn:alm:descriptor:com.tectonic.ui:podStatuses'
  version: v1
  description: >-
    MongoDB Deployment consisting of only one host. No replication of
    data.
```

#### 5.6.8.2. 必需的 CRD

是否依赖其他必需 CRD 完全可以自由选择，它们存在的目的只是为了缩小单个 Operator 的范围，并提供一种将多个 Operator 组合到一起来解决端到端用例的办法。

例如，一个 Operator 可设置一个应用程序并（从 etcd Operator）安装一个 etcd 集群以用于分布式锁定，以及一个 Postgres 数据库（来自 Postgres Operator）以用于数据存储。

Operator Lifecycle Manager (OLM) 对照集群中可用的 CRD 和 Operator 进行检查，以满足这些要求。如果找到合适的版本，Operator 将在所需命名空间中启动，并为每个 Operator 创建一个服务账户，以创建、监视和修改所需的 Kubernetes 资源。

**表 5.13. 必需的 CRD 字段**

| 字段 | 描述 | 必需/可选 |
| --- | --- | --- |
| **名称** | 所需 CRD 的全称。 | 必填 |
| **Version** | 该对象 API 的版本。 | 必填 |

| 字段 | 描述 | 必需/可选 |
| --- | --- | --- |
| **Kind** | Kubernetes 对象类型。 | 必填 |
| **DisplayName** | CRD 的人类可读版本。 | 必填 |
| **描述** | 概述该组件如何适合您的更大架构。 | 必填 |

**必需的 CRD 示例**

```
required:
- name: etcdclusters.etcd.database.coreos.com
  version: v1beta2
  kind: EtcdCluster
  displayName: etcd Cluster
  description: Represents a cluster of etcd nodes.
```

### 5.6.8.3. CRD 升级

如果自定义资源定义（CRD）属于单一集群服务版本（CSV），OLM 会立即对其升级。如果某个 CRD 被多个 CSV 拥有，则当该 CRD 满足以下所有向后兼容条件时才会升级：

- 所有已存在于当前 CRD 中的服务版本都包括在新 CRD 中。
- 在根据新 CRD 的验证模式（schema）进行验证后，与 CRD 的服务版本关联的所有现有实例或自定义资源均有效。

#### 5.6.8.3.1. 添加新版 CRD

**流程**

将新版 CRD 添加到 Operator:

1. 在 CSV 的 **versions** 部分的 CRD 资源中添加新条目。
   例如，如果当前 CRD 有一个 **v1alpha1** 版本，而您想要添加新的 **v1beta1** 版本并 将其标记为新的存储版本，请为 **v1beta1** 添加新条目：

```
versions:
  - name: v1alpha1
    served: true
    storage: false
  - name: v1beta1 (1)
    served: true
    storage: true
```

(1) 新条目。

2. 如果 CSV 打算使用新版本，请确保更新您的 CSV **owned** 部分中的 CRD 引用版本：

```
customresourcedefinitions:
  owned:
```

```
- name: cluster.example.com
  version: v1beta1 1
  kind: cluster
  displayName: Cluster
```

1 更新 **version**。

3. 将更新的 CRD 和 CSV 推送至您的捆绑包中。

#### 5.6.8.3.2. 弃用或删除 CRD 版本

Operator Lifecycle Manager(OLM)不允许立即删除自定义资源定义(CRD)的服务版本。弃用的 CRD 版本应首先通过将 CRD 的 **served** 字段设置为 **false** 来禁用。随后在升级 CRD 时便可将非服务版本删除。

**流程**

要弃用和删除特定 CRD 版本：

1. 将弃用版本标记为非服务版本，表明该版本已不再使用且后续升级时可删除。例如：

```
versions:
  - name: v1alpha1
    served: false 1
    storage: true
```

1 设置为 **false**。

2. 如果要弃用的版本目前为 **storage** 版本，则将该 **storage** 版本切换至服务版本。例如：

```
versions:
  - name: v1alpha1
    served: false
    storage: false 1
  - name: v1beta1
    served: true
    storage: true 2
```

1 2 对应更新 **storage** 字段。

**注意**

要从 CRD 中删除曾是或现在是 **storage** 的特定版本，该版本必须从 CRD 状态下的 **storedVersion** 中删除。OLM 一旦检测到某个已存储版本在新 CRD 中不再存在，OLM 将尝试执行这一操作。

3. 使用以上更改来升级 CRD。

4. 在后续升级周期中，非服务版本可从 CRD 中完全删除。例如：

```
versions:
  - name: v1beta1
    served: true
```

```
    storage: true
```

5. 如果该版本已从 CRD 中删除，请确保相应更新您的 CSV **owned** 部分中的引用 CRD 版本。

#### 5.6.8.4. CRD 模板

Operator 用户必须了解哪个选项必填，而不是可选选项。您可为您的每个 CRD 提供模板，并以最小配置集作为名为 **alm-examples** 的注解。兼容 UI 会预先填充该模板，供用户进一步自定义。

该注解由一个 kind 列表组成，如 CRD 名称和对应的 Kubernetes 对象的 **metadata** 和 **spec**。

以下完整示例提供了 **EtcdCluster**、**EtcdBackup** 和 **EtcdRestore** 模板：

```
metadata:
  annotations:
    alm-examples: >-
      [{"apiVersion":"etcd.database.coreos.com/v1beta2","kind":"EtcdCluster","metadata":
{"name":"example","namespace":"default"},"spec":{"size":3,"version":"3.2.13"}},
{"apiVersion":"etcd.database.coreos.com/v1beta2","kind":"EtcdRestore","metadata":
{"name":"example-etcd-cluster"},"spec":{"etcdCluster":{"name":"example-etcd-
cluster"},"backupStorageType":"S3","s3":{"path":"<full-s3-path>","awsSecret":"<aws-secret>"}}},
{"apiVersion":"etcd.database.coreos.com/v1beta2","kind":"EtcdBackup","metadata":
{"name":"example-etcd-cluster-backup"},"spec":{"etcdEndpoints":["<etcd-cluster-
endpoints>"],"storageType":"S3","s3":{"path":"<full-s3-path>","awsSecret":"<aws-secret>"}}}]
```

#### 5.6.8.5. 隐藏内部对象

Operator 在内部使用自定义资源定义 (CRD) 来完成任务是常见的。这些对象并不是供用户操作的，且可能会让 Operator 用户混淆。例如，数据库 Operator 可能会有一个 **Replication** CRD，当用户创建带有 **replication:true** 的数据库对象时就会创建它。

作为 Operator 作者，您可以通过将 **operators.operatorframework.io/internal-objects** 注解添加到 Operator 的 ClusterServiceVersion（CSV）来隐藏用户界面中不用于用户操作的任何 CRD。

**流程**

1. 在将一个 CRD 标记为 internal 之前，请确保任何管理应用程序所需的调试信息或配置都会反映在 CR 的状态或 **spec** 块中（如果适用于您的 Operator）。

2. 向 Operator 的 CSV 添加 **operators.operatorframework.io/internal-objects** 注解，以指定要在用户界面中隐藏的任何内部对象：

   **内部对象注解**

```
apiVersion: operators.coreos.com/v1alpha1
kind: ClusterServiceVersion
metadata:
  name: my-operator-v1.2.3
  annotations:
    operators.operatorframework.io/internal-objects:
'["my.internal.crd1.io","my.internal.crd2.io"]' (1)
...
```

(1) 将任何内部 CRD 设置为字符串数组。

#### 5.6.8.6. 初始化所需的自定义资源

Operator 可能需要用户在 Operator 完全正常工作前实例化自定义资源。然而，用户很难确定需要什么或怎样定义资源。

作为 Operator 开发人员，您可以通过将 **operatorframework.io/initialization-resource** 注解添加到集群服务版本（CSV）来指定在安装 Operator 时必须创建的单个自定义资源。该注解必须有包含完整 YAML 定义模板，该定义是在安装过程中初始化资源所需的。

如果定义了此注解，在从 OpenShift Container Platform Web 控制台安装 Operator 后，会提示用户使用 CSV 中提供的模板创建资源。

**流程**

- 为 Operator 的 CSV 添加 **operatorframework.io/initialization-resource** 注解，以指定所需的自定义资源。例如，以下注解需要创建 **StorageCluster** 资源，并提供完整的 YAML 定义：

  **初始化资源注解**

```
apiVersion: operators.coreos.com/v1alpha1
kind: ClusterServiceVersion
metadata:
  name: my-operator-v1.2.3
  annotations:
    operatorframework.io/initialization-resource: |-
       {
          "apiVersion": "ocs.openshift.io/v1",
          "kind": "StorageCluster",
          "metadata": {
             "name": "example-storagecluster"
          },
          "spec": {
             "manageNodes": false,
             "monPVCTemplate": {
                "spec": {
                   "accessModes": [
                      "ReadWriteOnce"
                   ],
                   "resources": {
                      "requests": {
                         "storage": "10Gi"
                      }
                   },
                   "storageClassName": "gp2"
                }
             },
             "storageDeviceSets": [
                {
                   "count": 3,
                   "dataPVCTemplate": {
                      "spec": {
                         "accessModes": [
                            "ReadWriteOnce"
                         ],
                         "resources": {
                            "requests": {
```

```
                                "storage": "1Ti"
                            }
                        },
                        "storageClassName": "gp2",
                        "volumeMode": "Block"
                    }
                },
                "name": "example-deviceset",
                "placement": {},
                "portable": true,
                "resources": {}
            }
        ]
    }
}
...
```

### 5.6.9. 了解您的 API 服务

与 CRD 一样，您的 Operator 可使用两类 APIService：*拥有的*和*必需的*。

#### 5.6.9.1. 拥有的 API 服务

当 CSV 拥有 API 服务时，它将负责描述为其提供支持的扩展 **api-server** 的部署及其提供的组/version/kind（GVK）。

API 服务由它提供的 group/version 唯一标识，并可以多次列出，以表示期望提供的不同类型。

**表 5.14. 拥有的 API 服务字段**

| 字段 | 描述 | 必需/可选 |
|---|---|---|
| **Group** | API 服务提供的组，如 **database.example.com**。 | 必填 |
| **Version** | API 服务的版本，如 **v1alpha1**。 | 必填 |
| **Kind** | API 服务应提供的类型。 | 必填 |
| **名称** | 提供的 API 服务的复数名称。 | 必填 |
| **DeploymentName** | 由您的 CSV 定义的部署名称，对应您的 API 服务（对于拥有的 API 服务是必需的）。在 CSV 待定阶段，OLM Operator 会在您的 CSV **InstallStrategy** 中搜索具有匹配名称的 **Deployment** spec，如果未找到，则不会将 CSV 转换至安装就绪阶段。 | 必填 |
| **DisplayName** | API 服务名称的人类可读版本，如 **MongoDB Standalone**。 | 必填 |
| **描述** | 有关 Operator 如何使用此 API 服务的简短描述，或有关 API 服务提供的功能描述。 | 必填 |

| 字段 | 描述 | 必需/可选 |
| --- | --- | --- |
| **资源** | 您的 API 服务拥有一类或多类 Kubernetes 对象。它们将在 resources 部分列出，用于告知用户他们可能需要排除故障的对象或如何连接至应用程序，如公开数据库的服务或 Ingress 规则。<br><br>建议仅列出对人重要的对象，而不必列出您编排的所有对象。例如，不要列出存储用户不会修改的内部状态的配置映射。 | 选填 |
| **SpecDescriptors**、**StatusDescriptors** 和 **ActionDescriptors** | 与拥有的 CRD 基本相同。 | 选填 |

##### 5.6.9.1.1. API 服务资源创建

Operator Lifecycle Manager (OLM) 负责为每个唯一拥有的 API 服务创建或替换服务及 API 服务资源：

- Service pod 选择器从与 API 服务描述的 **DeploymentName** 字段匹配的 CSV 部署中复制。
- 每次安装都会生成一个新的 CA 密钥/证书对，并且将 base64 编码的 CA 捆绑包嵌入到对应的 API 服务资源中。

##### 5.6.9.1.2. API service serving 证书

每当安装拥有的 API 服务时，OLM 均会处理服务密钥/证书对的生成。服务证书有一个通用名称（CN），其中包含生成的 **Service** 资源的主机名，并由嵌入在对应 API 服务资源中的 CA 捆绑包的私钥签名。

该证书作为类型 **kubernetes.io/tls** secret 存储在部署命名空间中，名为 **apiservice-cert** 的卷会自动附加至 CSV 中与 API 服务描述的 **DeploymentName** 字段匹配的 volumes 部分中。

如果尚不存在，则具有匹配名称的卷挂载也会附加至该部署的所有容器中。这样用户便可使用预期名称来定义卷挂载，以适应任何自定义路径要求。所生成的卷挂载的默认路径为 **/apiserver.local.config/certificates**，具有相同路径的任何现有卷挂载都会被替换。

#### 5.6.9.2. 所需的 API 服务

OLM 可保证所有必需的 CSV 均有可用的 API 服务，且所有预期的 GVK 在试图安装前均可发现。这允许 CSV 依赖于由它拥有的 API 服务提供的特定类型。

**表 5.15. 所需的 API 服务字段**

| 字段 | 描述 | 必需/可选 |
| --- | --- | --- |
| **Group** | API 服务提供的组，如 **database.example.com**。 | 必填 |
| **Version** | API 服务的版本，如 **v1alpha1**。 | 必填 |

| 字段 | 描述 | 必需/可选 |
| --- | --- | --- |
| **Kind** | API 服务应提供的类型。 | 必填 |
| **DisplayName** | API 服务名称的人类可读版本，如 **MongoDB Standalone**。 | 必填 |
| **描述** | 有关 Operator 如何使用此 API 服务的简短描述，或有关 API 服务提供的功能描述。 | 必填 |

## 5.7. 使用捆绑包镜像

您可以使用 Operator SDK 在 Operator Lifecycle Manager（OLM）中以捆绑格式（Bundle Format）打包、部署和升级 Operator。

### 5.7.1. 捆绑 Operator 并使用 Operator Lifecycle Manager 进行部署

Operator Lifecycle Manager（OLM）可帮助您在 Kubernetes 集群上安装、更新 Operator 及其相关服务的生命周期。OLM 在 OpenShift Container Platform 上默认安装，并作为 Kubernetes 扩展运行，以便您可以在没有任何额外工具的情况下将 Web 控制台和 OpenShift CLI（**oc**）用于所有 Operator 生命周期管理功能。

Operator Bundle Format 是 Operator SDK 和 OLM 的默认打包方法。您可以通过使用 Operator SDK 来通过 OLM 构建、推送、验证和运行捆绑包镜像，让 Operator 准备好进行 OLM。

**先决条件**

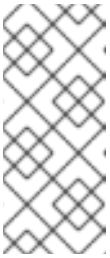
- 在开发工作站上安装 operator SDK CLI
- 已安装 OpenShift CLI（**oc**）v4.7+
- Operator Lifecycle Manager（OLM）安装在一个基于 Kubernetes 的集群上（如果使用 **apiextensions.k8s.io/v1** CRD，则为 v1.16.0 或更新版本，如 OpenShift Container Platform 4.7）
- 使用具有 **cluster-admin** 权限的账户使用 **oc** 登录到集群
- 使用 Operator SDK 初始化 operator 项目
- 如果 Operator 是基于 Go 的，则必须更新您的项目以使用支持的镜像在 OpenShift Container Platform 上运行

**流程**

1. 在 Operator 项目目录中运行以下 **make** 命令来构建和推送 Operator 镜像。在以下步骤中修改 **IMG** 参数来引用您可访问的库。您可以获取在存储库站点（如 Quay.io）存储容器的帐户。

   a. 构建镜像：

   ```
   $ make docker-build IMG=<registry>/<user>/<operator_image_name>:<tag>
   ```



**注意**

由 SDK 为 Operator 生成的 Dockerfile 明确引用了 **go 构建** 的 **GOARCH=amd64**。对于非 AMD64 构架，这已被应用于 **GOARCH=$TARGETARCH**。Docker 会自动将 环境变量设置为 **-platform** 指定的值。使用 Buildah，**-build-arg** 需要用于目的。如需更多信息，请参阅 多个架构。

b. 将镜像推送到存储库：

```
$ make docker-push IMG=<registry>/<user>/<operator_image_name>:<tag>
```

2. 运行 **make bundle** 命令创建 Operator 捆绑包清单，该命令调用多个命令，其中包括 Operator SDK **generate bundle** 和 **bundle validate** 子命令：

```
$ make bundle IMG=<registry>/<user>/<operator_image_name>:<tag>
```

Operator 的捆绑包清单描述了如何显示、创建和管理应用程序。**make bundle** 命令在 Operator 项目中创建以下文件和目录：

- 包含 **ClusterServiceVersion** 对象的捆绑包清单目录，名为 **bundle/manifests**
- 名为 **bundle/metadata** 的捆绑包元数据目录
- **config/crd** 目录中的所有自定义资源定义（CRD）
- 一个 Dockerfile **bundle.Dockerfile**

然后，使用 **operator-sdk bundle validate** 自动验证这些文件，以确保磁盘上的捆绑包的格式是正确的。

3. 运行以下命令来构建和推送捆绑包镜像。OLM 使用索引镜像来消耗 Operator 捆绑包，该镜像引用一个或多个捆绑包镜像。

a. 构建捆绑包镜像。使用您要推送镜像的 registry、用户命名空间和镜像标签的详情，设置 **BUNDLE_IMAGE**：

```
$ make bundle-build BUNDLE_IMG=<registry>/<user>/<bundle_image_name>:<tag>
```

b. 推送捆绑包镜像：

```
$ docker push <registry>/<user>/<bundle_image_name>:<tag>
```

4. 使用以下命令，检查集群中的 OLM 状态：

```
$ operator-sdk olm status \
   --olm-namespace=openshift-operator-lifecycle-manager
```

5. 使用 Operator SDK 中的 OLM 集成在集群中运行 Operator：

```
$ operator-sdk run bundle \
   [-n <namespace>] \(1)
   <registry>/<user>/<bundle_image_name>:<tag>
```

1 默认情况下，命令会在 **~/.kube/config** 文件中当前活跃的项目中安装 Operator。您可以添加 **-n** 标志来为安装设置不同的命名空间范围。

这个命令执行以下操作：

- 使用注入的捆绑包镜像创建索引镜像。
- 创建指向新索引镜像的目录源，以便 OperatorHub 能够发现 Operator。
- 通过创建一个 Operator 组、订阅、安装计划以及所有其他必要的对象（包括 RBAC），将 Operator 部署到集群中。

### 5.7.2. 在 Operator Lifecycle Manager 中测试 Operator 升级

您可以使用 Operator Lifecycle Manager（OLM）集成 Operator SDK 来快速测试 Operator 升级，而无需手动管理索引镜像和目录源。

**run bundle-upgrade** 子命令通过为以后的版本指定捆绑包镜像来自动触发已安装的 Operator 以升级到更新的版本。

**先决条件**

- 使用 **run bundle** 子命令安装 OLM 的 operator
- 代表已安装 Operator 的更新版本的捆绑包镜像

**流程**

1. 如果使用 **run bundle** 子命令的 OLM 尚未安装 Operator，请通过指定捆绑包镜像来安装 Operator 的早期版本。例如，对于 Memcached Operator：

```
$ operator-sdk run bundle <registry>/<user>/memcached-operator:v0.0.1
```

输**出示例**

```
INFO[0009] Successfully created registry pod: quay-io-demo-memcached-operator-v0-0-1
INFO[0009] Created CatalogSource: memcached-operator-catalog
INFO[0010] OperatorGroup "operator-sdk-og" created
INFO[0010] Created Subscription: memcached-operator-v0-0-1-sub
INFO[0013] Approved InstallPlan install-bqggr for the Subscription: memcached-operator-v0-0-1-sub
INFO[0013] Waiting for ClusterServiceVersion "my-project/memcached-operator.v0.0.1" to reach 'Succeeded' phase
INFO[0013]  Waiting for ClusterServiceVersion "my-project/memcached-operator.v0.0.1" to appear
INFO[0019]  Found ClusterServiceVersion "my-project/memcached-operator.v0.0.1" phase: Succeeded
```

2. 通过为后续的 Operator 版本指定捆绑包镜像来升级已安装的 Operator：

```
$ operator-sdk run bundle-upgrade <registry>/<user>/memcached-operator:v0.0.2
```

输**出示例**

```
INFO[0002] Found existing subscription with name memcached-operator-v0-0-1-sub and
namespace my-project
INFO[0002] Found existing catalog source with name memcached-operator-catalog and
namespace my-project
INFO[0009] Successfully created registry pod: quay-io-demo-memcached-operator-v0-0-2
INFO[0009] Updated catalog source memcached-operator-catalog with address and
annotations
INFO[0010] Deleted previous registry pod with name "quay-io-demo-memcached-operator-
v0-0-1"
INFO[0041] Approved InstallPlan install-gvcjh for the Subscription: memcached-operator-v0-
0-1-sub
INFO[0042] Waiting for ClusterServiceVersion "my-project/memcached-operator.v0.0.2" to
reach 'Succeeded' phase
INFO[0042]   Found ClusterServiceVersion "my-project/memcached-operator.v0.0.2" phase:
InstallReady
INFO[0043]   Found ClusterServiceVersion "my-project/memcached-operator.v0.0.2" phase:
Installing
INFO[0044]   Found ClusterServiceVersion "my-project/memcached-operator.v0.0.2" phase:
Succeeded
INFO[0044] Successfully upgraded to "memcached-operator.v0.0.2"
```

3. 清理已安装的 Operator：

```
$ operator-sdk cleanup memcached-operator
```

### 5.7.3. 其他资源

- 如需有关捆绑格式的更多详情，请参阅 Operator Framework 打包格式。
- 有关使用 **opm** 命令将捆绑包镜像添加到索引镜像的详情，请参阅管理自定义目录。
- 如需了解有关升级已安装的 Operator 的工作原理的详细信息，请参阅 Operator Lifecycle Manager 工作流。

## 5.8. 使用 SCORECARD 工具验证 OPERATOR

作为 Operator 作者，您可以使用 Operator SDK 中的 scorecard 工具来执行以下任务：

- 验证您的 Operator 项目没有语法错误，并正确打包
- 查看有关如何改进 Operator 的建议

### 5.8.1. 关于 scorecard 工具

虽然 Operator SDK **bundle validate** 子命令可为内容和结构验证本地捆绑包目录和远程捆绑包镜像，但您可以使用 **scorecard** 命令基于配置文件和测试镜像对 Operator 运行测试。这些测试在由 scorecard 配置并组成执行的测试镜像中实施。

Scorecard 假设它是在可以访问已配置的 Kubernetes 集群（如 OpenShift Container Platform）的情况下运行的。Scorecard 在 pod 中运行每个测试，从中聚合 pod 日志并将测试结果发送到控制台。Scorecard 内置了基本测试和 Operator Lifecycle Manager（OLM）测试，同时还提供了执行自定义测试定义的方法。

#### Scorecard 工作流

1. 创建任何相关的自定义资源（CR）和 Operator 所需的所有资源
2. 在 Operator 部署中创建代理容器，记录对 API 服务器的调用并运行测试
3. 检查 CR 中的参数

Scorecard 测试不会假定要测试的 Operator 状态。为 Operator 创建 Operator 和 CR 超出了 scorecard 本身的范围。但是，如果测试是为创建资源而设计的，则 scorecard 测试可以创建其所需的任何资源。

**scorecard 命令语法**

```
$ operator-sdk scorecard <bundle_dir_or_image> [flags]
```

Scorecard 需要一个位置参数，它是指向 Operator 捆绑包的磁盘路径或捆绑包镜像的名称。

如需有关标记的更多信息，请运行：

```
$ operator-sdk scorecard -h
```

### 5.8.2. Scorecard 配置

Scorecard 工具使用一个配置来供您配置内部插件以及几个全局配置选项。测试是由名为 **config.yaml** 的配置文件驱动的，该文件由 **make bundle** 命令生成，位于 **bundle/** 目录中：

```
./bundle
...
└── tests
    └── scorecard
        └── config.yaml
```

**Scorecard 配置文件示例**

```
kind: Configuration
apiversion: scorecard.operatorframework.io/v1alpha3
metadata:
  name: config
stages:
- parallel: true
  tests:
  - image: quay.io/operator-framework/scorecard-test:v1.3.0
    entrypoint:
    - scorecard-test
    - basic-check-spec
    labels:
      suite: basic
      test: basic-check-spec-test
  - image: quay.io/operator-framework/scorecard-test:v1.3.0
    entrypoint:
    - scorecard-test
    - olm-bundle-validation
    labels:
      suite: olm
      test: olm-bundle-validation-test
```

配置文件定义 scorecard 可执行的每个测试。Scorecard 配置文件的以下字段定义测试，如下所示：

| 配置字段 | 描述 |
| --- | --- |
| **image** | 测试实现测试的容器镜像名称 |
| **entrypoint** | 测试镜像中调用的命令和参数来执行测试 |
| **labels** | 选择要运行的测试的 scorecard 定义或自定义标签 |

### 5.8.3. 内置 scorecard 测试

Scorecard 附带预定义的测试，这些测试被放在套件中：基本测试套件和 Operator Lifecycle Manager（OLM）套件。

**表 5.16. 基本测试套件**

| 测试 | 描述 | 短名称 |
| --- | --- | --- |
| Spec Block Exists | 此测试会检查集群中创建的自定义资源（CR）以确保所有 CR 都有一个 **spec** 块。 | **basic-check-spec-test** |

**表 5.17. OLM 测试套件**

| 测试 | 描述 | 短名称 |
| --- | --- | --- |
| 捆绑包验证 | 此测试会验证传递给 scorecard 的捆绑包中的捆绑包清单。如果捆绑包内容包含错误，那么测试结果输出中将包括验证器日志以及验证库中的错误消息。 | **olm-bundle-validation-test** |
| Provided APIs Have Validation | 此测试会验证提供的 CR 的自定义资源定义（CRD）是否包含一个验证部分，并且 CR 中检测到的每个 **spec** 和 **status** 字段是否已验证。 | **olm-crds-have-validation-test** |
| Owned CRDs Have Resources Listed | 此测试确保通过 **cr-manifest** 选项提供的每个 CR 的 CRD 在 ClusterServiceVersion（CSV）的 **owned CRDs** 部分中有一个 **resources** 子部分。如果测试检测到未在 resources 部分中列出的已使用资源，它会在测试结束时将它们列在建议中。为这个测试通过初始代码生成后，用户需要填写 resources 部分。 | **olm-crds-have-resources-test** |
| Spec Fields With Descriptors | 此测试会验证 CRs **spec** 部分中的每一个字段是否都在 CSV 中列出对应的描述符。 | **olm-spec-descriptors-test** |
| Status Fields With Descriptors | 此测试会验证 CRs **status** 部分中的每一个字段是否都在 CSV 中列出对应的描述符。 | **olm-status-descriptors-test** |

### 5.8.4. 运行 scorecard 工具

Operator SDK 在运行 **init** 命令后生成一组默认 Kustomize 文件。生成的默认 **bundle/tests/scorecard/config.yaml** 文件可立即用于针对 Operator 运行 scorecard 工具，或者您可以根据测试规格修改该文件。

**先决条件**

- 使用 Operator SDK 生成的 operator 项目

**流程**

1. 为 Operator 生成或重新生成捆绑包清单和元数据：

```
$ make bundle
```

此命令自动将 scorecard 注解添加到捆绑包元数据中，由 **scorecard** 命令用来运行测试。

2. 针对 Operator 捆绑包的磁盘路径或捆绑包镜像的名称运行 scorecard:

```
$ operator-sdk scorecard <bundle_dir_or_image>
```

### 5.8.5. Scorecard 输出

**scorecard** 命令的 **--output** 标志指定 scorecard 结果输出格式： **text** 或 **json**。

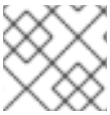
**例 5.2. JSON 输出片断示例**


```
{
  "apiVersion": "scorecard.operatorframework.io/v1alpha3",
  "kind": "TestList",
  "items": [
   {
    "kind": "Test",
    "apiVersion": "scorecard.operatorframework.io/v1alpha3",
    "spec": {
     "image": "quay.io/operator-framework/scorecard-test:v1.3.0",
     "entrypoint": [
      "scorecard-test",
      "olm-bundle-validation"
     ],
     "labels": {
      "suite": "olm",
      "test": "olm-bundle-validation-test"
     }
    },
    "status": {
     "results": [
      {
       "name": "olm-bundle-validation",
       "log": "time=\"2020-06-10T19:02:49Z\" level=debug msg=\"Found manifests directory\"
name=bundle-test\ntime=\"2020-06-10T19:02:49Z\" level=debug msg=\"Found metadata
directory\" name=bundle-test\ntime=\"2020-06-10T19:02:49Z\" level=debug msg=\"Getting
mediaType info from manifests directory\" name=bundle-test\ntime=\"2020-06-10T19:02:49Z\"
level=info msg=\"Found annotations file\" name=bundle-test\ntime=\"2020-06-10T19:02:49Z\"
level=info msg=\"Could not find optional dependencies file\" name=bundle-test\n",
```

```
            "state": "pass"
          }
        ]
      }
    }
  ]
}
```

**例 5.3. 文本输出片段示例**

```
--------------------------------------------------------------------------------
Image:      quay.io/operator-framework/scorecard-test:v1.3.0
Entrypoint: [scorecard-test olm-bundle-validation]
Labels:
 "suite":"olm"
 "test":"olm-bundle-validation-test"
Results:
 Name: olm-bundle-validation
 State: pass
 Log:
  time="2020-07-15T03:19:02Z" level=debug msg="Found manifests directory" name=bundle-test
  time="2020-07-15T03:19:02Z" level=debug msg="Found metadata directory" name=bundle-test
  time="2020-07-15T03:19:02Z" level=debug msg="Getting mediaType info from manifests
directory" name=bundle-test
  time="2020-07-15T03:19:02Z" level=info msg="Found annotations file" name=bundle-test
  time="2020-07-15T03:19:02Z" level=info msg="Could not find optional dependencies file"
name=bundle-test
```



**注意**

输出格式 spec 与 **Test** 类型布局匹配。

### 5.8.6. 选择测试

Scorecard 测试通过将 **--selector** CLI 标志设置为一组标签字符串来选择。如果没有提供选择器标志，则运行 scorecard 配置文件中的所有测试。

测试通过 scorecard 聚合并写入标准输出或 *stdout* 以序列方式运行。

**流程**

1. 要选择单个测试（如 **basic-check-spec-test**），使用 **--selector** 标志来指定测试：

```
$ operator-sdk scorecard <bundle_dir_or_image> \
    -o text \
    --selector=test=basic-check-spec-test
```

2. 要选择一组测试（如 **olm** ），请指定所有 OLM 测试使用的标签：

```
$ operator-sdk scorecard <bundle_dir_or_image> \
    -o text \
    --selector=suite=olm
```

3. 要选择多个测试，按照以下语法使用 **selector** 标记指定测试名称：

```
$ operator-sdk scorecard <bundle_dir_or_image> \
    -o text \
    --selector='test in (basic-check-spec-test,olm-bundle-validation-test)'
```

### 5.8.7. 启用并行测试

作为 Operator 作者，您可以使用 scorecard 配置文件为测试定义独立阶段。阶段会根据配置文件中定义的顺序按照顺序运行。一个阶段（stage）包含测试列表以及一个可配置的 **parallel** 设置。

默认情况，或当阶段把 **parallel** 明确设置为 **false** 时，阶段中的测试会按配置文件中定义的顺序运行。每次只运行一个测试有助于保证两个测试间不会相互交互和冲突。

但是，如果测试被设计为完全隔离，则可以实现并行化。

**流程**

- 要并行运行一组隔离测试，在同一个阶段中包括它们，并把 **parallel** 设置为 **true**：

```
apiVersion: scorecard.operatorframework.io/v1alpha3
kind: Configuration
metadata:
  name: config
stages:
- parallel: true (1)
  tests:
  - entrypoint:
    - scorecard-test
    - basic-check-spec
    image: quay.io/operator-framework/scorecard-test:v1.3.0
    labels:
      suite: basic
      test: basic-check-spec-test
  - entrypoint:
    - scorecard-test
    - olm-bundle-validation
    image: quay.io/operator-framework/scorecard-test:v1.3.0
    labels:
      suite: olm
      test: olm-bundle-validation-test
```

(1) 启用并行测试

所有并行阶段中的测试都会同时执行，scorecard 会在进入下一阶段前等待所有测试完成。这使得测试可以更快地运行。

### 5.8.8. 自定义 scorecard 测试

scorecard 工具可按照以下强制约定运行自定义测试：

- 测试在容器镜像内实施
- 测试可以接受包含命令和参数的入口点
- 测试以 JSON 格式生成 **v1alpha3** scorecard 输出，在测试输出中没有无关的日志信息
- 测试可在 **/bundle** 的共享挂载点获取捆绑包内容
- 测试可以使用集群内客户端连接访问 Kubernetes API

如果测试镜像遵循上述指南，则可以使用其他编程语言编写自定义测试。

以下示例显示了在 Go 中写入的自定义测试镜像：

**例 5.4. 自定义 scorecard 测试示例**

```
// Copyright 2020 The Operator-SDK Authors
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//     http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

package main

import (
 "encoding/json"
 "fmt"
 "log"
 "os"

 scapiv1alpha3 "github.com/operator-framework/api/pkg/apis/scorecard/v1alpha3"
 apimanifests "github.com/operator-framework/api/pkg/manifests"
)

// This is the custom scorecard test example binary
// As with the Redhat scorecard test image, the bundle that is under
// test is expected to be mounted so that tests can inspect the
// bundle contents as part of their test implementations.
// The actual test is to be run is named and that name is passed
// as an argument to this binary.  This argument mechanism allows
// this binary to run various tests all from within a single
// test image.

const PodBundleRoot = "/bundle"

func main() {
```

```
 entrypoint := os.Args[1:]
 if len(entrypoint) == 0 {
  log.Fatal("Test name argument is required")
 }

 // Read the pod's untar'd bundle from a well-known path.
 cfg, err := apimanifests.GetBundleFromDir(PodBundleRoot)
 if err != nil {
  log.Fatal(err.Error())
 }

 var result scapiv1alpha3.TestStatus

 // Names of the custom tests which would be passed in the
 // `operator-sdk` command.
 switch entrypoint[0] {
 case CustomTest1Name:
  result = CustomTest1(cfg)
 case CustomTest2Name:
  result = CustomTest2(cfg)
 default:
  result = printValidTests()
 }

 // Convert scapiv1alpha3.TestResult to json.
 prettyJSON, err := json.MarshalIndent(result, "", "    ")
 if err != nil {
  log.Fatal("Failed to generate json", err)
 }
 fmt.Printf("%s\n", string(prettyJSON))

}

// printValidTests will print out full list of test names to give a hint to the end user on what the valid
tests are.
func printValidTests() scapiv1alpha3.TestStatus {
 result := scapiv1alpha3.TestResult{}
 result.State = scapiv1alpha3.FailState
 result.Errors = make([]string, 0)
 result.Suggestions = make([]string, 0)

 str := fmt.Sprintf("Valid tests for this image include: %s %s",
  CustomTest1Name,
  CustomTest2Name)
 result.Errors = append(result.Errors, str)
 return scapiv1alpha3.TestStatus{
  Results: []scapiv1alpha3.TestResult{result},
 }
}

const (
 CustomTest1Name = "customtest1"
 CustomTest2Name = "customtest2"
)

// Define any operator specific custom tests here.
```

```
// CustomTest1 and CustomTest2 are example test functions. Relevant operator specific
// test logic is to be implemented in similarly.

func CustomTest1(bundle *apimanifests.Bundle) scapiv1alpha3.TestStatus {
 r := scapiv1alpha3.TestResult{}
 r.Name = CustomTest1Name
 r.State = scapiv1alpha3.PassState
 r.Errors = make([]string, 0)
 r.Suggestions = make([]string, 0)
 almExamples := bundle.CSV.GetAnnotations()["alm-examples"]
 if almExamples == "" {
  fmt.Println("no alm-examples in the bundle CSV")
 }

 return wrapResult(r)
}

func CustomTest2(bundle *apimanifests.Bundle) scapiv1alpha3.TestStatus {
 r := scapiv1alpha3.TestResult{}
 r.Name = CustomTest2Name
 r.State = scapiv1alpha3.PassState
 r.Errors = make([]string, 0)
 r.Suggestions = make([]string, 0)
 almExamples := bundle.CSV.GetAnnotations()["alm-examples"]
 if almExamples == "" {
  fmt.Println("no alm-examples in the bundle CSV")
 }
 return wrapResult(r)
}

func wrapResult(r scapiv1alpha3.TestResult) scapiv1alpha3.TestStatus {
 return scapiv1alpha3.TestStatus{
  Results: []scapiv1alpha3.TestResult{r},
 }
}
```

## 5.9. 使用 PROMETHEUS 配置内置监控

本指南介绍了 Operator SDK 通过 Prometheus Operator 提供的内置监控支持，及其对 Operator 作者的详细用途。

### 5.9.1. Prometheus Operator 支持

Prometheus 是一个开源系统监视和警报工具包。Prometheus Operator 会创建、配置和管理在基于 Kubernetes 的集群（如 OpenShift Container Platform）中运行的 Prometheus 集群。

默认情况下，Operator SDK 中包括帮助函数，用于在任何生成的 Go-based Operator 中自动设置指标，以便在部署了 Prometheus Operator 的集群上使用。

### 5.9.2. 指标帮助函数

在使用 Operator SDK 生成的基于 Go 的 Operator 中，以下函数会公开有关运行中程序的一般指标：

```
func ExposeMetricsPort(ctx context.Context, port int32) (*v1.Service, error)
```

这些指标从 **controller-runtime** 库 API 继承而来。默认在 **0.0.0.0:8383/metrics** 上提供指标。

创建一个 **Service** 对象并公开指标端口，之后可通过 Prometheus 访问该端口。删除领导 Pod 的 **root** 所有者时，**Service** 对象便会被垃圾回收。

以下示例出现在使用 Operator SDK 生成的所有 Operator 的 **cmd/manager/main.go** 文件中：

```
import(
    "github.com/operator-framework/operator-sdk/pkg/metrics"
    "machine.openshift.io/controller-runtime/pkg/manager"
)

var (
    // Change the below variables to serve metrics on a different host or port.
    metricsHost       = "0.0.0.0" (1)
    metricsPort int32 = 8383 (2)
)
...
func main() {
    ...
    // Pass metrics address to controller-runtime manager
    mgr, err := manager.New(cfg, manager.Options{
        Namespace:          namespace,
        MetricsBindAddress: fmt.Sprintf("%s:%d", metricsHost, metricsPort),
    })

    ...
    // Create Service object to expose the metrics port.
    _, err = metrics.ExposeMetricsPort(ctx, metricsPort)
    if err != nil {
        // handle error
        log.Info(err.Error())
    }
    ...
}
```

(1) 在其上公开指标的主机。

(2) 在其上公开指标的端口。

#### 5.9.2.1. 修改指标端口

Operator 作者可修改在其上公开指标的端口。

**先决条件**

- 使用 Operator SDK 生成基于 Go 的 Operator
- 基于 Kubernetes 的集群已部署 Prometheus Operator

**流程**

- 在生成的 Operator 的 **cmd/manager/main.go** 文件中，在以下行中更改 **metricsPort** 的值：

```
var metricsPort int32 = 8383
```

### 5.9.3. 服务监控器

**ServiceMonitor** 是 Prometheus Operator 提供的自定义资源，用于发现 **Service** 对象中的 **Endpoints**，配置 Prometheus 以监控这些 pod。

在使用 Operator SDK 生成的基于 Go 的 Operator 中，**GenerateServiceMonitor()** 帮助函数可以获取 **Service** 对象并基于该对象生成 **ServiceMonitor** 对象。

#### 其他资源

- 如需有关 **ServiceMonitor** 自定义资源定义（CRD）的更多信息，请参阅 Prometheus Operator 文档。

#### 5.9.3.1. 创建服务监控器

Operator 作者可使用 **metrics.CreateServiceMonitor()** 帮助函数来添加已创建监控服务的服务目标发现，该函数接受新创建的 Service。

#### 先决条件

- 使用 Operator SDK 生成基于 Go 的 Operator
- 基于 Kubernetes 的集群已部署 Prometheus Operator

#### 流程

- 将 **metrics.CreateServiceMonitor()** 帮助函数添加至您的 Operator 代码中：

```
import(
    "k8s.io/api/core/v1"
    "github.com/operator-framework/operator-sdk/pkg/metrics"
    "machine.openshift.io/controller-runtime/pkg/client/config"
)
func main() {

    ...
    // Populate below with the Service(s) for which you want to create ServiceMonitors.
    services := []*v1.Service{}
    // Create one ServiceMonitor per application per namespace.
    // Change the below value to name of the Namespace you want the ServiceMonitor to be
created in.
    ns := "default"
    // restConfig is used for talking to the Kubernetes apiserver
    restConfig := config.GetConfig()

    // Pass the Service(s) to the helper function, which in turn returns the array of
ServiceMonitor objects.
    serviceMonitors, err := metrics.CreateServiceMonitors(restConfig, ns, services)
    if err != nil {
        // Handle errors here.
```

```
    }
    ...
}
```

## 5.10. 配置领导选举机制

在 Operator 的生命周期中，在任意给定时间可能有多个实例在运行，例如，推出 Operator 升级程序。这种情况下，需要使用领导选举机制来避免多个 Operator 实例争用。这样可确保只有一个领导实例处理协调，其他实例均不活跃，但却会做好准备，随时接管领导实例的的工作。

有两种不同的领导选举实现可供选择，每种机制都有各自的利弊权衡问题：

**leader-for-life**

领导 pod 只在删除垃圾回收时放弃领导权。这种实现可以避免两个实例错误地作为领导运行，一个也被称为"裂脑（split brain）"的状态。但这种方法可能会延迟选举新的领导。例如，当领导 pod 位于无响应或分区的节点上时， **pod-eviction-timeout** 会规定领导 pod 从节点上删除和停止所需要的时间，默认值为 **5m**。详情请参见 Leader-for-life Go 文档。

**Leader-with-lease**

领导 pod 定期更新领导租期，并在无法更新租期时放弃领导权。当现有领导 Pod 被隔离时，这种实现方式可更快速地过渡至新领导，但在某些情况下存在脑裂的可能性。详情请参见 Leader-with-lease Go 文档。

Operator SDK 默认启用 Leader-for-life 实现。请查阅相关的 Go 文档来了解这两种方法，以考虑对您的用例来说有意义的利弊得失。

### 5.10.1. Operator 领导选举示例

以下示例演示了如何为 Operator、Leader-for-life 和 Leader-with-lease 使用两个领导选举选项。

#### 5.10.1.1. leader-for-life 选举机制

实现 Leader-for-life 选举机制时，调用 **leader.Become()** 会在 Operator 重试时进行阻止，直至通过创建名为 **memcached-operator-lock** 的配置映射使其成为领导：

```
import (
  ...
  "github.com/operator-framework/operator-sdk/pkg/leader"
)

func main() {
  ...
  err = leader.Become(context.TODO(), "memcached-operator-lock")
  if err != nil {
    log.Error(err, "Failed to retry for leader lock")
    os.Exit(1)
  }
  ...
}
```

如果 Operator 不在集群内运行，则只会返回 **leader.Become()** 而无任何错误，以跳过该领导选举机制，因其无法检测 Operator 的名称。

#### 5.10.1.2. Leader-with-lease 选举机制

Leader-with-lease 实现可使用 Manager Options 来启用以作为领导选举机制：

```
import (
  ...
  "sigs.k8s.io/controller-runtime/pkg/manager"
)

func main() {
  ...
  opts := manager.Options{
    ...
    LeaderElection: true,
    LeaderElectionID: "memcached-operator-lock"
  }
  mgr, err := manager.New(cfg, opts)
  ...
}
```

当 Operator 没有在集群中运行时，Manager 会在启动时返回一个错误，因为它无法检测到 Operator 的命名空间，以便为领导选举机制创建配置映射。您可通过设置 Manager 的 **LeaderElectionNamespace** 选项来覆盖该命名空间。

## 5.11. OPERATOR SDK CLI 参考

Operator SDK 命令行界面（CLI）是一个开发组件，旨在更轻松地编写 Operator。

### operator SDK CLI 语法

```
$ operator-sdk <command> [<subcommand>] [<argument>] [<flags>]
```

具有集群管理员访问权限的 operator 作者（如 OpenShift Container Platform）可以使用 Operator SDK CLI 根据 Go、Ansible 或 Helm 开发自己的 Operator。Kubebuilder 作为基于 Go 的 Operator 的构建解决方案嵌入到 Operator SDK 中，这意味着现有的 Kubebuilder 项目可以象 Operator SDK 一样使用并继续工作。

### 5.11.1. bundle

**operator-sdk bundle** 命令管理 Operator 捆绑包元数据。

#### 5.11.1.1. validate

**bundle validate** 子命令会验证 Operator 捆绑包。

**表 5.18. bundle validate 标记**

| 标记 | 描述 |
|---|---|
| **-h，--help** | **bundle validate** 子命令的帮助输出。 |
| **--index-builder**（字符串） | 拉取和解包捆绑包镜像的工具。仅在验证捆绑包镜像时使用。可用选项是 **docker**（默认值）、**podman** 或 **none**。 |

| 标记 | 描述 |
| --- | --- |
| **--list-optional** | 列出所有可用的可选验证器。设置后，不会运行验证器。 |
| **--select-optional**（字符串） | 选择要运行的可选验证器的标签选择器。当使用 **--list-optional** 标志运行时，会列出可用的可选验证器。 |

### 5.11.2. cleanup

**operator-sdk cleanup** 命令会销毁并删除为通过 **run** 命令部署的 Operator 创建的资源。

表 5.19. **cleanup** 标记

| 标记 | 描述 |
| --- | --- |
| **-h**，**--help** | **run bundle** 子命令的帮助输出。 |
| **--kubeconfig** (string) | 用于 CLI 请求的 **kubeconfig** 文件的路径。 |
| **n**、**--namespace**（字符串） | 如果存在，代表在其中运行 CLI 请求的命名空间。 |
| **--timeout <duration>** | 失败前，等待命令完成的时间。默认值为 **2m0s**。 |

### 5.11.3. completion

**operator-sdk completion** 命令生成 shell completion，以便更迅速、更轻松地发出 CLI 命令。

表 5.20. **completion 子命令**

| 子命令 | 描述 |
| --- | --- |
| **bash** | 生成 bash completion。 |
| **zsh** | 生成 zsh completion。 |

表 5.21. **completion** 标记

| 标记 | 描述 |
| --- | --- |
| **-h, --help** | 使用方法帮助输出。 |

例如：

```
$ operator-sdk completion bash
```

输**出示例**

```
# bash completion for operator-sdk                    -*- shell-script -*-
...
# ex: ts=4 sw=4 et filetype=sh
```

### 5.11.4. create

**operator-sdk create** 命令用于创建或 *scaffold* Kubernetes API。

#### 5.11.4.1. api

**create api** 子命令构建 Kubernetes API。子命令必须在 **init** 命令初始化的项目中运行。

表 5.22. **create api** 标记

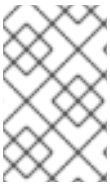
| 标记 | 描述 |
| --- | --- |
| **-h，--help** | **run bundle** 子命令的帮助输出。 |

### 5.11.5. generate

**operator-sdk generate** 命令调用特定的生成器来生成代码或清单。

#### 5.11.5.1. bundle

**generate bundle** 子命令为您的 Operator 项目生成一组捆绑包清单、元数据和 **bundle.Dockerfile** 文件。



**注意**

通常，您首先运行 **generate kustomize manifests** 子命令来生成由 **generate bundle** 子命令使用的输入 Kustomize 基础。但是，您可以使用初始项目中的 **make bundle** 命令按顺序自动运行这些命令。

表 5.23. **generate bundle** 标记

| 标记 | 描述 |
| --- | --- |
| **--channels**（字符串） | 捆绑包所属频道的以逗号分隔的列表。默认值为 **alpha**。 |
| **--crds-dir**（字符串） | **CustomResoureDefinition** 清单的根目录。 |
| **--default-channel**（字符串） | 捆绑包的默认频道。 |
| **--deploy-dir**（字符串） | Operator 清单的根目录，如部署和 RBAC。这个目录与传递给 **--input-dir** 标记的目录不同。 |
| **-h，--help** | **generate bundle** 的帮助信息 |

| 标记 | 描述 |
| --- | --- |
| **--input-dir**（字符串） | 从中读取现有捆绑包的目录。这个目录是捆绑包 **manifests** 目录的父目录，它与 **--deploy-dir** 目录不同。 |
| **--kustomize-dir**（字符串） | 包含 Kustomize 基础的目录以及用于捆绑包清单的 **kustomization.yaml** 文件。默认路径为 **config/manifests**。 |
| **--manifests** | 生成捆绑包清单。 |
| **--metadata** | 生成捆绑包元数据和 Dockerfile。 |
| **--output-dir**（字符串） | 将捆绑包写入的目录。 |
| **--overwrite** | 如果捆绑包元数据和 Dockerfile 存在，则覆盖它们。默认值为 **true**。 |
| **--package**（字符串） | 捆绑包的软件包名称。 |
| **-q**、**--quiet** | 在静默模式下运行。 |
| **--stdout** | 将捆绑包清单写入标准输出。 |
| **--version**（字符串） | 生成的捆绑包中的 Operator 语义版本。仅在创建新捆绑包或升级 Operator 时设置。 |

**其他资源**

- 如需了解包括使用 **make bundle** 命令来调用 **generate bundle** 子命令的完整流程，请参阅捆绑 Operator 和 Operator Lifecycle Manager 部署。

### 5.11.5.2. kustomize

**generate kustomize** 子命令包含为 Operator 生成 Kustomize 数据的子命令。

#### 5.11.5.2.1. 清单

**generate kustomize manifests** 子命令生成或重新生成 Kustomize 基础以及 **config/manifests** 目录中的 **kustomization.yaml** 文件，用于其他 Operator SDK 命令构建捆绑包清单。在默认情况下，这个命令会以互动方式询问 UI 元数据，即清单基础的重要组件，除非基础已存在或设置了 **--interactive=false** 标志。

**表 5.24. generate kustomize manifests 标记**

| 标记 | 描述 |
| --- | --- |
| **--apis-dir**（字符串） | API 类型定义的根目录。 |
| **-h**，**--help** | **generate kustomize manifests** 的帮助信息。 |

| 标记 | 描述 |
| --- | --- |
| **--input-dir**（字符串） | 包含现有 Kustomize 文件的目录。 |
| **--interactive** | 当设置为 **false** 时，如果没有 Kustomize 基础，则会出现交互式命令提示符来接受自定义元数据。 |
| **--output-dir**（字符串） | 写入 Kustomize 文件的目录。 |
| **--package**（字符串） | 软件包名称。 |
| **-q、--quiet** | 在静默模式下运行。 |

### 5.11.6. init

**operator-sdk init** 命令初始化 Operator 项目，并为给定插件生成或 *scaffolds* 默认项目目录布局。

这个命令会写入以下文件：

- boilerplate 许可证文件
- 带有域和库的**PROJECT** 文件
- 构建项目的**Makefile**
- **go.mod** 文件带有项目依赖项
- 用于自定义清单的 **kustomization.yaml** 文件
- 用于为管理器清单自定义镜像的补丁文件
- 启用 Prometheus 指标的补丁文件
- 运行的 **main.go** 文件

**表 5.25. init 标记**

| 标记 | 描述 |
| --- | --- |
| **--help, -h** | **init** 命令的帮助输出。 |
| **--plugins**（字符串） | 插件的名称和可选版本，用于初始化项目。可用插件包括 **ansible.sdk.operatorframework.io/v1**、**go.kubebuilder.io/v2**、**go.kubebuilder.io/v3** 和 **helm.sdk.operatorframework.io/v1**。 |
| **--project-version** | 项目版本。可用值为 **2** 和 **3-alpha**（默认值）。 |

### 5.11.7. run

**operator-sdk run** 命令提供可在各种环境中启动 Operator 的选项。

#### 5.11.7.1. bundle

**run bundle** 子命令使用 Operator Lifecycle Manager（OLM）以捆绑包格式部署 Operator。

**表 5.26. run bundle 标记**

| 标记 | 描述 |
| --- | --- |
| **--index-image**（字符串） | 在其中注入捆绑包的索引镜像。默认镜像为 **quay.io/operator-framework/upstream-opm-builder:latest**。 |
| **--install-mode <install_mode_value>** | 安装 Operator 的集群服务版本（CSV）支持的模式，如 **AllNamespaces** 或 **SingleNamespace**。 |
| **--timeout <duration>** | 安装超时。默认值为 **2m0s**。 |
| **--kubeconfig** (string) | 用于 CLI 请求的 **kubeconfig** 文件的路径。 |
| **n、--namespace**（字符串） | 如果存在，代表在其中运行 CLI 请求的命名空间。 |
| **-h，--help** | **run bundle** 子命令的帮助输出。 |

**其他资源**

- 如需有关可能安装模式的详细信息，请参阅 Operator 组成员资格。

#### 5.11.7.2. bundle-upgrade

**run bundle-upgrade** 子命令升级之前使用 Operator Lifecycle Manager（OLM）以捆绑包格式安装的 Operator。

**表 5.27. run bundle-upgrade 标记**

| 标记 | 描述 |
| --- | --- |
| **--timeout <duration>** | 升级超时。默认值为 **2m0s**。 |
| **--kubeconfig** (string) | 用于 CLI 请求的 **kubeconfig** 文件的路径。 |
| **n、--namespace**（字符串） | 如果存在，代表在其中运行 CLI 请求的命名空间。 |
| **-h，--help** | **run bundle** 子命令的帮助输出。 |

### 5.11.8. scorecard

**operator-sdk scorecard** 命令运行 scorecard 工具来验证 Operator 捆绑包并提供改进建议。该命令使用一个参数，可以是捆绑包镜像，也可以是包含清单和元数据的目录。如果参数包含镜像标签，则镜像必须远程存在。

表 5.28. **scorecard** 标记

| 标记 | 描述 |
| --- | --- |
| **-c, --config**（字符串） | scorecard 配置文件的路径。默认路径为 **bundle/tests/scorecard/config.yaml**。 |
| **-h，--help** | **scorecard** 命令的帮助输出。 |
| **--kubeconfig** (string) | **kubeconfig** 文件的路径。 |
| **-L、--list** | 列出哪些测试可以运行。 |
| **-n**、--namespace（字符串） | 运行测试镜像的命名空间。 |
| **-o, --output**（字符串） | 结果的输出格式。可用值为 **text** （默认值）和 **json**。 |
| **-l、--selector**（字符串） | 标识选择器以确定要运行哪个测试。 |
| **-s、--service-account**（字符串） | 用于测试的服务帐户。默认值为 **default**。 |
| **-x、--skip-cleanup** | 运行测试后禁用资源清理。 |
| **-w、--wait-time <duration>** | 等待测试完成的时间，如 **35s**。默认值为 **30s**。 |

**其他资源**

- 如需有关运行scorecard 工具的详细信息，请参阅使用 scorecard 工具验证 Operator。

# 第 6 章 集群 OPERATOR 参考

本参考指南将红帽提供的 *集群 Operator* 组成，该 Operator 作为 OpenShift Container Platform 的架构基础。默认情况下会安装集群 Operator，除非另有说明，并由 Cluster Version Operator(CVO)管理。如需有关 control plane 架构的更多信息，请参阅 OpenShift Container Platform 中的 Operator。

集群管理员可以通过 **Administration → Cluster Settings** 页面，在 OpenShift Container Platform Web 控制台中查看集群 Operator。

**注意**

Cluster Operator 不由 Operator Lifecycle Manager(OLM)和 OperatorHub 管理。OLM 和 OperatorHub 是 OpenShift Container Platform 中用于安装和运行可选附加组件 Operator 的 Operator Framework 的一部分。

## 6.1. CLOUD CREDENTIAL OPERATOR

### 用途

Cloud Credential Operator(CCO)将云供应商凭证作为 Kubernetes 自定义资源定义(CRD)进行管理。**CredentialsRequest** 自定义资源（CR）的 CCO 同步，允许 OpenShift Container Platform 组件使用集群运行所需的特定权限请求云供应商凭证。

通过在 **install-config.yaml** 文件中为 **credentialsMode** 参数设置不同的值，可将 CCO 配置为以几种不同模式操作。如果没有指定模式，或将 **credentialsMode** 参数被设置为空字符串（""）。

### project

openshift-cloud-credential-operator

### CRD

- **credentialsrequests.cloudcredential.openshift.io**
  - Scope: Namespaced
  - CR：**CredentialsRequest**
  - Validation: Yes

### Configuration objects

不需要配置。

### 其它资源

- CredentialsRequest 自定义资源
- 关于 Cloud Credential Operator

## 6.2. CLUSTER AUTHENTICATION OPERATOR

### 用途

Cluster Authentication Operator 在集群中安装并维护 **Authentication** 自定义资源，并可以通过以下方式查看：

```
$ oc get clusteroperator authentication -o yaml
```

**project**
cluster-authentication-operator

## 6.3. CLUSTER AUTOSCALER OPERATOR

**用途**
Cluster Autoscaler Operator 使用 **cluster-api** 供应商管理 OpenShift Cluster Autoscaler 的部署。

**project**
cluster-autoscaler-operator

**CRD**

- **ClusterAutoscaler**：这是一个单一的资源，用于控制集群自动扩展实例的配置。Operator 只响应受管命名空间中名为 **default** 的 **ClusterAutoscaler** 资源，即 **WATCH_NAMESPACE** 环境变量的值。
- **MachineAutoscaler**：此资源针对一个节点组，并管理注解来为那个组启用和配置自动扩展，**min** 和 **max** 的值。目前只能将 **MachineSet** 对象作为目标。

## 6.4. CLUSTER CONFIG OPERATOR

**用途**
Cluster Config Operator 执行与 **config.openshift.io** 相关的以下任务：

- 创建 CRD。
- 呈现初始自定义资源。
- 处理迁移。

**project**
cluster-config-operator

## 6.5. CLUSTER CSI SNAPSHOT CONTROLLER OPERATOR

**用途**
Cluster CSI Snapshot Controller Operator 安装和维护 CSI Snapshot Controller。CSI Snapshot Controller 负责监视 **VolumeSnapshot** CRD 对象，并管理卷快照的创建和删除生命周期。

**project**
cluster-csi-snapshot-controller-operator

## 6.6. CLUSTER IMAGE REGISTRY OPERATOR

**用途**
Cluster Image Registry Operator 管理 OpenShift Container Platform registry 的单个实例。它管理 registry 的所有配置，包括创建存储。

在初始启动时，Operator 会基于集群中检测到的配置创建默认的 **image-registry** 资源实例。这代表了根据云供应商要使用的云存储类型。

如果没有足够的信息来定义完整的 **image-registry** 资源，则会定义一个不完整的资源，Operator 将更新资源状态以提供缺失的内容。

Cluster Image Registry Operator在**openshift-image-registry**命名空间中运行，并管理该位置中的 registry 实例。registry的所有配置和工作负载资源都位于该命名空间中。

**project**
cluster-image-registry-operator

## 6.7. CLUSTER MACHINE APPROVER OPERATOR

**用途**
Cluster Machine Approver Operator 在集群安装后自动批准为新 worker 节点请求的 CSR。

**注意**

对于 control plane 节点，bootstrap 节点上的 **approve-csr** 服务会在集群引导阶段自动批准所有 CSR。

**project**
cluster-machine-approver-operator

## 6.8. CLUSTER MONITORING OPERATOR

**用途**
Cluster Monitoring Operator 管理并更新 OpenShift Container Platform 上部署的基于 Prometheus 的集群监控堆栈。

**project**
openshift-monitoring

**CRD**

- **alertmanagers.monitoring.coreos.com**
  - Scope: Namespaced
  - CR: **alertmanager**
  - Validation: Yes
- **prometheuses.monitoring.coreos.com**
  - Scope: Namespaced
  - CR: **prometheus**
  - Validation: Yes
- **prometheusrules.monitoring.coreos.com**
  - Scope: Namespaced
  - CR: **prometheusrule**
  - Validation: Yes
- **servicemonitors.monitoring.coreos.com**

- Scope: Namespaced
- CR: **servicemonitor**
- Validation: Yes

### Configuration objects

```
$ oc -n openshift-monitoring edit cm cluster-monitoring-config
```

## 6.9. CLUSTER NETWORK OPERATOR

**用途**
Cluster Network Operator 在 OpenShift Container Platform 集群上安装并升级网络组件。

## 6.10. OPENSHIFT CONTROLLER MANAGER OPERATOR

**用途**
OpenShift Controller Manager Operator 在集群中安装和维护 **OpenShiftControllerManager** 自定义资源，并可使用以下方法查看：

```
$ oc get clusteroperator openshift-controller-manager -o yaml
```

自定义资源（CRD） **openshiftcontrollermanagers.operator.openshift.io** 可以在具有以下内容的集群中查看：

```
$ oc get crd openshiftcontrollermanagers.operator.openshift.io -o yaml
```

**project**
cluster-openshift-controller-manager-operator

## 6.11. CLUSTER SAMPLES OPERATOR

**用途**
Cluster Samples Operator 管理存储在 **openshift** 命名空间中的示例镜像流和模板。

在初始启动时，Operator 会创建默认样本配置资源来启动镜像流和模板的创建。配置对象是一个集群范围内的对象，它带有一个键 **cluster** 和类型 **configs.samples**。

镜像流是基于 Red Hat Enterprise Linux CoreOS (RHCOS) 的 OpenShift Container Platform 镜像流，指向 **registry.redhat.io** 上的镜像。同样，模板也被归类为 OpenShift Container Platform 模板。

Cluster Samples Operator 部署包含在 **openshift-cluster-samples-operator** 命名空间中。在开始时，内部 registry 中的镜像流导入逻辑和 API 服务器使用安装 pull secret 与 **registry.redhat.io** 进行身份验证。如果管理员更改了用于示例镜像流的 registry，则管理员可在 **openshift** 命名空间中创建额外的 secret。如果创建，这些 secret 包含用于简化镜像导入所需的 **docker** 的 **config.json** 的内容。

Cluster Samples Operator 的镜像包含关联的 OpenShift Container Platform 发行版本的镜像流和模板定义。Cluster Samples Operator 创建示例后，它会添加一个注解，表示其兼容的 OpenShift Container Platform 版本。Operator 使用此注解来确保每个示例与兼容发行版本匹配。清单（inventory）以外的示例会与跳过的示例一样被忽略。

只要版本注解没有修改或删除，则允许对 Operator 管理的任何样本进行修改。但是，在升级中，当版本注解改变时，这些修改可能会被替换，因为样本会使用更新的版本进行更新。Jenkins 镜像是安装后镜像有效负载的一部分，并直接标记到镜像流中。

Samples Operator 配置资源包含一个终结器（finalizer），它会在删除时清除以下内容：

- Operator 管理的镜像流
- Operator 管理的模板。
- Operator 生成的配置资源。
- 集群状态资源。

删除样本资源后，Samples Operator 会使用默认配置重新创建资源。

**project**
cluster-samples-operator

## 6.12. CLUSTER STORAGE OPERATOR

**用途**
Cluster Storage Operator 设置 OpenShift Container Platform 集群范围内的存储默认设置。它确保了为 OpenShift Container Platform 集群存在一个默认存储类。

**project**
cluster-storage-operator

**Configuration**
不需要配置。

备注

- Cluster Storage Operator 支持 Amazon Web Services (AWS) 和 Red Hat OpenStack Platform (RHOSP)。
- 创建的存储类可以通过编辑其注解来实现非默认设置，但只要 Operator 运行，存储类就无法被删除。

## 6.13. CLUSTER VERSION OPERATOR

**用途**
集群 Operator 管理集群功能的特定区域。Cluster Version Operator (CVO)管理集群 Operator 的生命周期，其中许多默认安装在 OpenShift Container Platform 中。

CVO 还检查 OpenShift Update Service，以查看基于当前组件版本及图中信息的有效更新和更新路径。

**project**
cluster-version-operator

**其它资源**

- OpenShift Container Platform 中的 Operator

## 6.14. CONSOLE OPERATOR

**用途**
Console Operator 在集群中安装和维护 OpenShift Container Platform web 控制台。

**project**
console-operator

## 6.15. DNS OPERATOR

**用途**
DNS Operator 部署并管理 CoreDNS，以为 pod 提供名称解析服务。它在 OpenShift Container Platform 中启用了基于 DNS 的 Kubernetes 服务发现。

Operator 根据集群的配置创建可正常工作的默认部署。

- 默认集群域是 **cluster.local**。
- CoreDNS Corefile 和 Kubernetes 插件配置还不被支持。

DNS Operator 把 CoreDNS 做为一个 Kubernetes 守护进程集进行管理。它会使用一个带有静态 IP 的服务向外界公开这个功能。CoreDNS 在集群中的所有节点上运行。

**project**
cluster-dns-operator

## 6.16. ETCD 集群 OPERATOR

**用途**
etcd 集群 Operator 自动执行 etcd 集群扩展，启用 etcd 监控和指标，并简化灾难恢复流程。

**project**
cluster-etcd-operator

**CRD**

- **etcds.operator.openshift.io**
  - Scope: Cluster
  - CR: **etcd**
  - Validation: Yes

**Configuration objects**

```
$ oc edit etcd cluster
```

## 6.17. INGRESS OPERATOR

**用途**
Ingress Operator 配置并管理 OpenShift Container Platform 路由。

**project**
openshift-ingress-operator

### CRD

- **clusteringresses.ingress.openshift.io**
  - Scope: Namespaced
  - CR: **clusteringresses**
  - Validation: No

### Configuration objects

- Cluster config
  - 类型名：**clusteringresses.ingress.openshift.io**
  - 实例名称： **default**
  - 查看命令：

```
$ oc get clusteringresses.ingress.openshift.io -n openshift-ingress-operator default -o yaml
```

备**注**
Ingress Operator 在 **openshift-ingress** 项目中设置路由，并为路由创建部署：

```
$ oc get deployment -n openshift-ingress
```

Ingress Operator 使用来自 **network/cluster** 状态的 **clusterNetwork[].cidr** 来决定受管入口控制器（路由器）应该在其中操作的模式(IPv4、IPv6 或双堆栈）。例如，如果 **clusterNetwork** 只包含 v6 **cidr**，则 ingress 控制器以 IPv6 模式操作。

在以下示例中，Ingress Operator 管理的 ingress 控制器将以 IPv4 模式运行，因为只有一个集群网络存在，网络是 IPv4 **cidr**：

```
$ oc get network/cluster -o jsonpath='{.status.clusterNetwork[*]}'
```

输**出示例**

```
map[cidr:10.128.0.0/14 hostPrefix:23]
```

## 6.18. INSIGHTS OPERATOR

### 用途
Insights Operator 收集 OpenShift Container Platform 配置数据并将其发送到红帽。数据用于生成有关集群可能暴露的潜在问题的主动分析建议。这些建议通过 console.redhat.com 上的 Insights Advisor 与集群管理员通信。

### project
insights-operator

### Configuration
不需要配置。

备注
Insights Operator 是 OpenShift Container Platform Telemetry 的一个良好补充。

**其它资源**

- 关于远程健康监控，了解有关 Insights Operator 和 Telemetry 的详细信息

## 6.19. KUBERNETES API SERVER OPERATOR

**用途**
Kubernetes API Server Operator 管理并更新在 OpenShift Container Platform 上部署的 Kubernetes API 服务器。Operator 基于 OpenShift library-go 框架，并使用 Cluster Version Operator (CVO) 安装。

**project**
openshift-kube-apiserver-operator

**CRD**

- **kubeapiservers.operator.openshift.io**
  - Scope: Cluster
  - CR: **kubeapiserver**
  - Validation: Yes

**Configuration objects**

```
$ oc edit kubeapiserver
```

## 6.20. KUBERNETES CONTROLLER MANAGER OPERATOR

**用途**
Kubernetes Controller Manager Operator 管理并更新在 OpenShift Container Platform 上部署的 Kubernetes Controller Manager。Operator 基于 OpenShift **library-go** 框架，并通过 Cluster Version Operator (CVO) 安装。

它包含以下组件：

- Operator
- Bootstrap 清单解析器
- 基于静态 pod 的安装程序
- 配置观察

默认情况下，Operator 通过 **metrics** 服务公开 Prometheus 指标数据。

**project**
cluster-kube-controller-manager-operator

## 6.21. KUBERNETES SCHEDULER OPERATOR

**用途**

Kubernetes Scheduler Operator 管理并更新在 OpenShift Container Platform 上部署的 Kubernetes 调度程序。Operator 基于 OpenShift Container Platform **library-go** 框架，它与 Cluster Version Operator (CVO) 一起安装。

Kubernetes Scheduler Operator 包含以下组件：

- Operator
- Bootstrap 清单解析器
- 基于静态 pod 的安装程序
- 配置观察

默认情况下，Operator 通过 metrics 服务公开 Prometheus 指标数据。

### project
cluster-kube-scheduler-operator

### Configuration
Kubernetes 调度程序的配置是以下合并的结果：

- 默认配置
- 从 spec **schedulers.config.openshift.io** 获得的配置。

所有这些都是稀疏配置，无效的 JSON 片断会在结尾进行合并，以便形成有效的配置。

## 6.22. KUBERNETES STORAGE VERSION MIGRATOR OPERATOR

### 用途
Kubernetes Storage Version Migrator Operator 检测到默认存储版本的更改，在存储版本更改时为资源类型创建迁移请求，并处理迁移请求。

### project
cluster-kube-storage-version-migrator-operator

## 6.23. MACHINE API OPERATOR

### 用途
Machine API Operator 管理用于扩展 Kubernetes API 的特定目的自定义资源定义（CRD）、控制器和 RBAC 对象的生命周期。它声明集群中机器的所需状态。

### project
machine-api-operator

### CRD

- **MachineSet**
- **机器**
- **MachineHealthCheck**

## 6.24. MACHINE CONFIG OPERATOR

**用途**
Machine Congig Operator 管理并应用基本操作系统和容器运行时的配置和更新，包括内核和 kubelet 之间的所有配置和更新。

有四个组件：

- **machine-config-server**：为加入集群的新机器提供 Ignition 配置。
- **machine-config-controller**：协调机器升级到 **MachineConfig** 对象定义的配置。提供用来控制单独一组机器升级的选项。
- **machine-config-daemon**：在更新过程中应用新机器配置。验证并验证机器的状态到请求的机器配置。
- **machine-config**：提供安装、首次启动和更新一个机器的完整机器配置源。

**project**
openshift-machine-config-operator

## 6.25. MARKETPLACE OPERATOR

**用途**
Marketplace Operator 是一个将非集群 Operator 放置到集群中的机制。

**project**
operator-marketplace

## 6.26. NODE TUNING OPERATOR

**用途**
Node Tuning Operator 可以帮助您通过编排 Tuned 守护进程来管理节点级别的性能优化。大多数高性能应用程序都需要一定程度的内核级性能优化。Node Tuning Operator 为用户提供了一个统一的、节点一级的 sysctl 管理接口，并可以根据具体用户的需要灵活地添加自定义性能优化设置。

Operator 将为 OpenShift Container Platform 容器化 Tuned 守护进程作为一个 Kubernetes 守护进程集进行管理。它保证了自定义性能优化设置以可被守护进程支持的格式传递到在集群中运行的所有容器化的 Tuned 守护进程中。相应的守护进程会在集群的所有节点上运行，每个节点上运行一个。

在发生触发配置集更改的事件时，或通过接收和处理终止信号安全终止容器化 Tuned 守护进程时，容器化 Tuned 守护进程所应用的节点级设置将被回滚。

在版本 4.1 及更高版本中，OpenShift Container Platform 标准安装中包含了 Node Tuning Operator。

**project**
cluster-node-tuning-operator

## 6.27. OPENSHIFT API SERVER OPERATOR

**用途**
OpenShift API Server Operator 在集群中安装和维护 **openshift-apiserver** 。

**project**
openshift-apiserver-operator

**CRD**

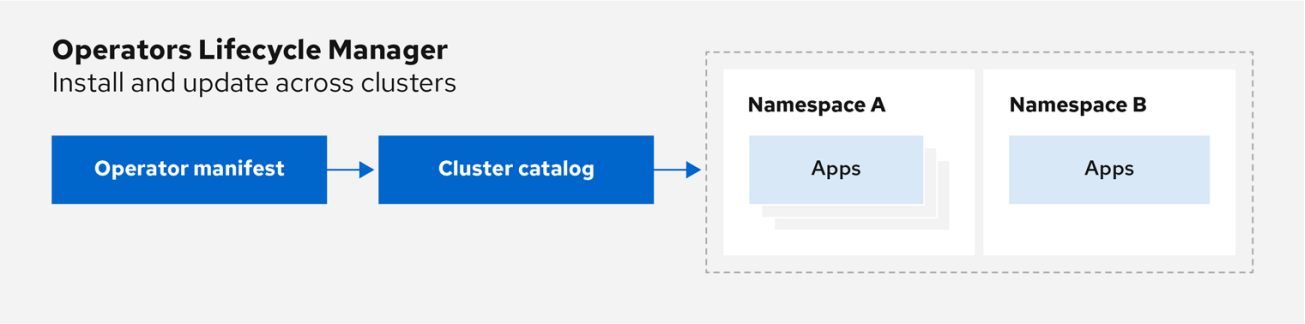
- **openshiftapiservers.operator.openshift.io**
  - Scope: Cluster
  - CR: **openshiftapiserver**
  - Validation: Yes

## 6.28. OPERATOR LIFECYCLE MANAGER OPERATORS

### 用途

*Operator Lifecycle Manager* （OLM）可帮助用户安装、更新和管理所有 Kubernetes 原生应用程序（Operator）以及在 OpenShift Container Platform 集群中运行的关联服务的生命周期。它是 Operator Framework 的一部分，后者是一个开源工具包，用于以有效、自动化且可扩展的方式管理 Operator。

**图 6.1. Operator Lifecycle Manager 工作流**




OLM 默认在 OpenShift Container Platform 4.7 中运行，辅助集群管理员对集群上运行的 Operator 进行安装、升级和授予访问权。OpenShift Container Platform Web 控制台提供一些管理界面，供集群管理员安装 Operator，以及为特定项目授权以便使用集群上的可用 Operator 目录。

开发人员通过自助服务体验，无需成为相关问题的专家也可自由置备和配置数据库、监控和大数据服务的实例，因为 Operator 已将相关知识融入其中。

### CRD

Operator Lifecycle Manager (OLM) 由两个 Operator 组成，分别为：OLM Operator 和 Catalog Operator。

每个 Operator 均负责管理 CRD，而 CRD 是 OLM 的框架基础：

**表 6.1. 由 OLM 和 Catalog Operator 管理的 CRD**

| 资源 | 短名称 | 所有者 | 描述 |
|---|---|---|---|
| **ClusterServiceVersion** (CSV) | **csv** | OLM | 应用程序元数据：名称、版本、图标、所需资源、安装等。 |
| **InstallPlan** | **ip** | Catalog | 为自动安装或升级 CSV 而需创建的资源的计算列表。 |

| 资源 | 短名称 | 所有者 | 描述 |
| --- | --- | --- | --- |
| **CatalogSource** | **catsrc** | Catalog | 定义应用程序的 CSV、CRD 和软件包存储库。 |
| **Subscription** | **sub** | Catalog | 用于通过跟踪软件包中的频道来保持 CSV 最新。 |
| **OperatorGroup** | **og** | OLM | 将部署在同一命名空间中的所有 Operator 配置为 **OperatorGroup** 对象，以便在一系列命名空间或集群范围内监视其自定义资源 ( CR)。 |

每个 Operator 还负责创建以下资源：

**表 6.2. 由 OLM 和 Catalog Operator 创建的资源**

| 资源 | 所有者 |
| --- | --- |
| **部署** | OLM |
| **ServiceAccounts** | |
| **(Cluster)Roles** | |
| **(Cluster)RoleBindings** | |
| **CustomResourceDefinitions** (CRD) | Catalog |
| **ClusterServiceVersions** | |

### OLM Operator

集群中存在 CSV 中指定需要的资源后，OLM Operator 将负责部署由 CSV 资源定义的应用程序。

OLM Operator 不负责创建所需资源；用户可选择使用 CLI 手动创建这些资源，也可选择使用 Catalog Operator 来创建这些资源。这种关注点分离的机制可以使得用户逐渐增加他们选择用于其应用程序的 OLM 框架量。

OLM Operator 使用以下工作流：

1. 观察命名空间中的集群服务版本（CSV），并检查是否满足要求。
2. 如果满足要求，请运行 CSV 的安装策略。

**注意**

CSV 必须是 Operator 组的活跃成员，才可运行该安装策略。

### Catalog Operator

Catalog Operator 负责解析和安装集群服务版本（CSV）以及它们指定的所需资源。另外还负责监视频道中的目录源中是否有软件包更新，并将其升级（可选择自动）至最新可用版本。

要跟踪频道中的软件包，您可以创建一个 **Subscription** 对象来配置所需的软件包、频道和 **CatalogSource** 对象，以便拉取更新。在找到更新后，便会代表用户将一个适当的 **InstallPlan** 对象写入命名空间。

Catalog Operator 使用以下工作流：

1. 连接到集群中的每个目录源。
2. 监视是否有用户创建的未解析安装计划，如果有：
   a. 查找与请求名称相匹配的 CSV，并将此 CSC 添加为已解析的资源。
   b. 对于每个受管或所需 CRD，将其添加为已解析的资源。
   c. 对于每个所需 CRD，找到管理相应 CRD 的 CSV。
3. 监视是否有已解析的安装计划并为其创建已发现的所有资源（用户批准或自动）。
4. 观察目录源和订阅并根据它们创建安装计划。

### Catalog Registry

Catalog Registry 存储 CSV 和 CRD 以便在集群中创建，并存储有关软件包和频道的元数据。

*package manifest* 是 Catalog Registry 中的一个条目，用于将软件包标识与 CSV 集相关联。在软件包中，频道指向特定 CSV。因为 CSV 明确引用了所替换的 CSV，软件包清单向 Catalog Operator 提供了将 CSV 更新至频道中最新版本所需的信息，逐步安装和替换每个中间版本。

### 其他资源

- 了解 Operator Lifecycle Manager (OLM)

## 6.29. OPENSHIFT SERVICE CA OPERATOR

### 用途

OpenShift Service CA Operator mint 并管理 Kubernetes 服务的服务证书。

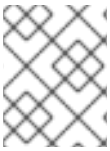
### project

openshift-service-ca-operator

## 6.30. VSPHERE 问题检测器（VSPHERE PROBLEM DETECTOR）OPERATOR

### 用途

vSphere 问题检测器 Operator 会检查在 vSphere 上部署的集群，以获取与存储相关的常见安装和错误配置问题。



**注意**

只有 Cluster Storage Operator 检测到集群部署在 vSphere 上时，Cluster Storage Operator 才会启动 vSphere 问题检测器 Operator。

### Configuration

不需要配置。

备**注**

- Operator 支持 vSphere 上的 OpenShift Container Platform 安装。
- Operator 使用 **vsphere-cloud-credentials** 与 vSphere 通信。
- Operator 会执行与存储相关的检查。

**其它**资**源**

- 使用 vSphere 问题检测器 Operator