



OpenShift Container Platform 4.15

Operator

在 OpenShift Container Platform 中使用 Operator

OpenShift Container Platform 4.15 Operator

在 OpenShift Container Platform 中使用 Operator

法律通告

Copyright © 2024 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux[®] is the registered trademark of Linus Torvalds in the United States and other countries.

Java[®] is a registered trademark of Oracle and/or its affiliates.

XFS[®] is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL[®] is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js[®] is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack[®] Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

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	4
1.1. 对于开发人员	4
1.2. 对于管理员	4
1.3. 后续步骤	5
第 2 章 了解 OPERATOR	6
2.1. 什么是 OPERATOR?	6
2.2. OPERATOR FRAMEWORK 打包格式	7
2.3. OPERATOR FRAMEWORK 常用术语表	25
2.4. OPERATOR LIFECYCLE MANAGER (OLM)	26
2.5. 了解 OPERATORHUB	64
2.6. 红帽提供的 OPERATOR 目录	66
2.7. 多租户集群中的 OPERATOR	68
2.8. CRD	70
第 3 章 用户任务	79
3.1. 从已安装的 OPERATOR 创建应用程序	79
3.2. 在命名空间中安装 OPERATOR	80
第 4 章 管理员任务	90
4.1. 在集群中添加 OPERATOR	90
4.2. 更新安装的 OPERATOR	107
4.3. 从集群中删除 OPERATOR	108
4.4. 配置 OPERATOR LIFECYCLE MANAGER 功能	111
4.5. 在 OPERATOR LIFECYCLE MANAGER 中配置代理支持	113
4.6. 查看 OPERATOR 状态	116
4.7. 管理 OPERATOR 条件	120
4.8. 允许非集群管理员安装 OPERATOR	121
4.9. 管理自定义目录	127
4.10. 在受限网络中使用 OPERATOR LIFECYCLE MANAGER	147
4.11. 目录源 POD 调度	151
4.12. 管理平台 OPERATOR (技术预览)	154
4.13. TROUBLESHOOTING OPERATOR 的问题	161
第 5 章 开发 OPERATOR	175
5.1. 关于 OPERATOR SDK	175
5.2. 安装 OPERATOR SDK CLI	176
5.3. 基于 GO 的 OPERATOR	178
5.4. 基于 ANSIBLE 的 OPERATOR	199
5.5. 基于 HELM 的 OPERATOR	230
5.6. 基于 JAVA 的 OPERATOR	259
5.7. 定义集群服务版本 (CSV)	278
5.8. 使用捆绑包镜像	306
5.9. 遵守 POD 安全准入	315
5.10. 令牌身份验证	320
5.11. 使用 SCORECARD 工具验证 OPERATOR	333
5.12. 验证 OPERATOR 捆绑包	341
5.13. 高可用性或单节点集群检测和支持	345
5.14. 使用 PROMETHEUS 配置内置监控	346
5.15. 配置领导选举机制	353
5.16. 为多平台支持配置 OPERATOR 项目	354
5.17. 基于 GO 的 OPERATOR 的对象修剪工具	361

5.18. 将软件包清单项目迁移到捆绑包格式	362
5.19. OPERATOR SDK CLI 参考	364
第 6 章 集群 OPERATOR 参考	372
6.1. CLUSTER BAREMETAL OPERATOR	372
6.2. 裸机事件中继	372
6.3. CLOUD CREDENTIAL OPERATOR	373
6.4. CLUSTER AUTHENTICATION OPERATOR	373
6.5. CLUSTER AUTOSCALER OPERATOR	374
6.6. CLUSTER CLOUD CONTROLLER MANAGER OPERATOR	374
6.7. CLUSTER CAPI OPERATOR	375
6.8. CLUSTER CONFIG OPERATOR	375
6.9. CLUSTER CSI SNAPSHOT CONTROLLER OPERATOR	376
6.10. CLUSTER IMAGE REGISTRY OPERATOR	376
6.11. CLUSTER MACHINE APPROVER OPERATOR	376
6.12. CLUSTER MONITORING OPERATOR	377
6.13. CLUSTER NETWORK OPERATOR	377
6.14. CLUSTER SAMPLES OPERATOR	377
6.15. CLUSTER STORAGE OPERATOR	378
6.16. CLUSTER VERSION OPERATOR	379
6.17. CONSOLE OPERATOR	379
6.18. CONTROL PLANE MACHINE SET OPERATOR	380
6.19. DNS OPERATOR	380
6.20. ETCD 集群 OPERATOR	380
6.21. INGRESS OPERATOR	381
6.22. INSIGHTS OPERATOR	382
6.23. KUBERNETES API SERVER OPERATOR	382
6.24. KUBERNETES CONTROLLER MANAGER OPERATOR	383
6.25. KUBERNETES SCHEDULER OPERATOR	383
6.26. KUBERNETES STORAGE VERSION MIGRATOR OPERATOR	384
6.27. MACHINE API OPERATOR	384
6.28. MACHINE CONFIG OPERATOR	384
6.29. MARKETPLACE OPERATOR	385
6.30. NODE TUNING OPERATOR	385
6.31. OPENSIFT API SERVER OPERATOR	386
6.32. OPENSIFT CONTROLLER MANAGER OPERATOR	386
6.33. OPERATOR LIFECYCLE MANAGER OPERATORS	387
6.34. OPENSIFT SERVICE CA OPERATOR	389
6.35. VSPHERE 问题检测器 (VSPHERE PROBLEM DETECTOR) OPERATOR	389
第 7 章 OLM 1.0 (技术预览)	391
7.1. 关于 OPERATOR LIFECYCLE MANAGER 1.0 (技术预览)	391
7.2. 组件和架构	392
7.3. 从 OLM 1.0 中的目录安装 OPERATOR (技术预览)	402
7.4. 在 OLM 1.0 中管理普通捆绑包 (技术预览)	430

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 Go-based Operators, Ansible-based Operators, Java-based Operators, 和 Helm-based Operators](#)。
- [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](#)。

其他资源

- [Operator 开发人员的机器删除生命周期 hook 示例](#)

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](#)。
- [升级已安装的 Operator](#)。
- [删除已安装的 Operator](#)。

- [配置代理支持](#)。
- [在受限网络中使用 Operator Lifecycle Manager](#)。

要了解红帽提供的所有集群 Operator，请参阅 [Cluster Operator 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。

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（扩展至 OpenShift Container Platform）包含构建复杂分布式系统（可在本地和云提供商之间工作）需要的所有原语，包括 secret 处理、负载均衡、服务发现、自动扩展。

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)。它默认部署在 OpenShift Container Platform 4.15 中。

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。

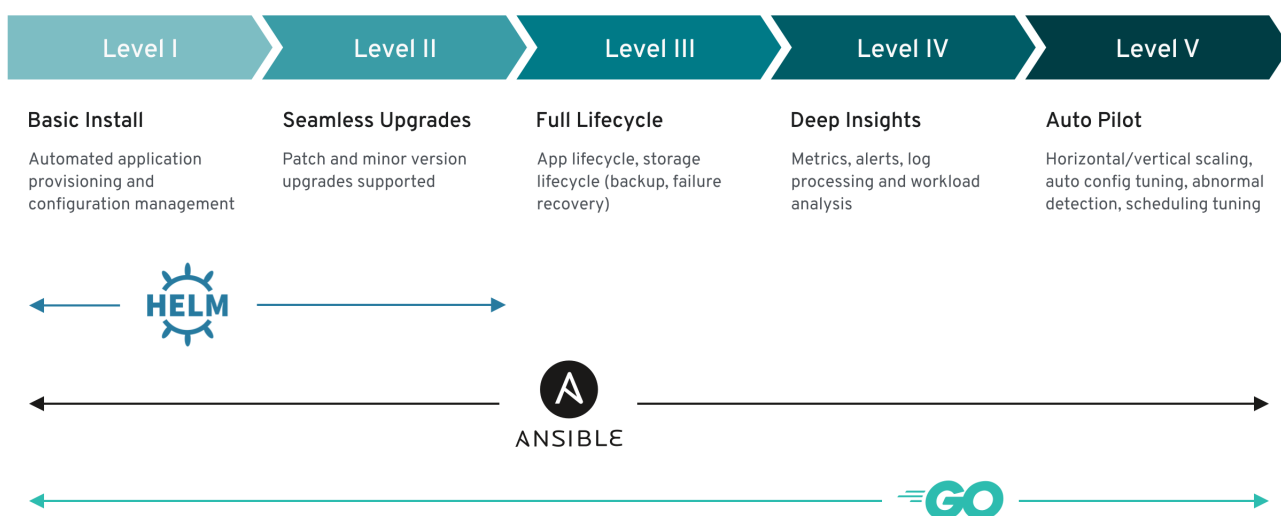
这些工具可组合使用，因此您可自由选择对您有用的工具。

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 的第二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. 捆绑包格式

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。

捆绑包格式布局示例

```

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**
- **ConsoleCLIDownload**
- **ConsoleLink**

- **ConsoleQuickStart**
- **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

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。支持以下 Operator 依赖项：

olm.package

这个类型表示特定 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

使用这个类型，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 的使用量。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。

olm.constraint

这个类型在任意 Operator 属性上声明通用限制。

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 CLI

opm CLI 工具由 Operator Framework 提供,用于 Operator 捆绑格式。您可以通过此工具从与软件存储库类似的 Operator 捆绑包列表中创建和维护 Operator 目录。其结果是一个容器镜像,它可以存储在容器的 registry 中,然后安装到集群中。

目录包含一个指向 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) 可以引用由 **CatalogSource** 对象定义的目录源中的镜像,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. 基于文件的目录

*基于文件的目录*是 Operator Lifecycle Manager (OLM) 中目录格式的最新迭代。它是基于纯文本 (JSON 或 YAML) 和早期 SQLite 数据库格式的声明式配置演变,并且完全向后兼容。此格式的目标是启用 Operator 目录编辑、可组合性和可扩展性。

编辑

使用基于文件的目录,与目录内容交互的用户可以对格式进行直接更改,并验证其更改是否有效。由于这种格式是纯文本 JSON 或 YAML,因此目录维护人员可以通过手动或广泛支持的 JSON 或 YAML 工具(如 **jq** CLI)轻松操作目录元数据。

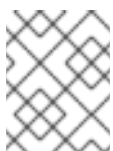
此可编辑功能启用以下功能和用户定义的扩展:

- 将现有捆绑包提升到新频道
- 更改软件包的默认频道
- 用于添加、更新和删除升级边缘的自定义算法

Composability

基于文件的目录存储在任意目录层次结构中,从而启用目录组成。例如,考虑两个单独的基于文件的目录目录:**catalogA** 和 **catalogB**。目录维护人员可以通过生成新目录 **catalogC** 并将 **catalogA** 和 **catalogB** 复制到其中来创建新的组合目录。

这种可组合性支持分散的目录。格式允许 Operator 作者维护特定于 Operator 的目录,它允许维护人员轻松构建由单个 Operator 目录组成的目录。基于文件的目录可以通过组合多个其他目录、提取一个目录的子集或两者的组合来组成。



注意

不允许软件包中重复软件包和重复捆绑包。如果找到任何重复项,**opm validate** 命令将返回错误。

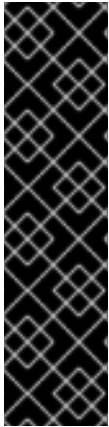
因为 Operator 作者最熟悉其 Operator、其依赖项及其升级兼容性,所以他们可以维护自己的 Operator 目录并直接控制其内容。对于基于文件的目录,Operator 作者负责在目录中构建和维护其软件包的任务。但是,复合目录维护者仅拥有在其目录中管理软件包并将目录发布到用户的任务。

可扩展性

基于文件的目录规格是目录的一个低级别表示。虽然目录维护器可以直接以低级形式维护,但目录维护人员可以在其自己的自定义工具上构建有趣的扩展,以供其自身的自定义工具用于实现任意数量的变异。

例如,工具可以将一个高级 API (如(**mode=semver**)) 转换为升级边缘基于文件的低级别目录格式。或目录维护人员可能需要通过添加新属性到符合特定标准的捆绑包来自定义所有捆绑包元数据。

虽然这种可扩展性允许在低级别 API 上开发额外的官方工具，用于将来的 OpenShift Container Platform 版本，但目录维护人员也具有此功能。



重要

从 OpenShift Container Platform 4.11 开始，默认的红帽提供的 Operator 目录以基于文件的目录格式发布。通过以过时的 SQLite 数据库格式发布的 4.10，用于 OpenShift Container Platform 4.6 的默认红帽提供的 Operator 目录。

与 SQLite 数据库格式相关的 **opm** 子命令、标志和功能已被弃用，并将在以后的版本中删除。功能仍被支持，且必须用于使用已弃用的 SQLite 数据库格式的目录。

许多 **opm** 子命令和标志都用于 SQLite 数据库格式，如 **opm index prune**，它们无法使用基于文件的目录格式。有关使用基于文件的目录的更多信息，请参阅[管理自定义目录](#)以及[使用 oc-mirror 插件为断开连接的安装 mirror 镜像](#)。

2.2.2.1. 目录结构

基于文件的目录可从基于目录的文件系统进行存储和加载。**opm** CLI 通过遍历根目录并递归到子目录来加载目录。CLI 尝试加载它找到的每个文件，如果发生错误，则会失败。

可以使用 **.indexignore** 文件忽略非目录文件，这些文件对模式和优先级与 **.gitignore** 文件具有相同的规则。

示例 .indexignore 文件

```
# Ignore everything except non-object .json and .yaml files
**/*
!*.json
!*.yaml
**/objects/*.json
**/objects/*.yaml
```

目录维护人员具有选择所需的布局的灵活性，但建议将每个软件包基于文件的目录 Blob 存储在单独的子目录中。每个单独的文件可以是 JSON 或 YAML；目录中的每一文件并不需要使用相同的格式。

基本推荐结构

```
catalog
├── packageA
│   └── index.yaml
├── packageB
│   ├── .indexignore
│   ├── index.yaml
│   └── objects
│       └── packageB.v0.1.0.clusterserviceversion.yaml
└── packageC
    ├── index.json
    └── deprecations.yaml
```

此推荐结构具有目录层次结构中的每个子目录都是自包含目录的属性，它使得目录组成、发现和导航简单文件系统操作。通过将目录复制到父目录的根目录，目录也可以包含在父目录中。

2.2.2.2. 模式

基于文件的目录使用基于 [CUE 语言规范](#) 的格式，该格式可使用任意模式进行扩展。以下 **_Meta** CUE 模式定义了所有基于文件的目录 Blob 必须遵循的格式：

_Meta 架构

```
_Meta: {
  // schema is required and must be a non-empty string
  schema: string & !=""

  // package is optional, but if it's defined, it must be a non-empty string
  package?: string & !=""

  // properties is optional, but if it's defined, it must be a list of 0 or more properties
  properties?: [... #Property]
}

#Property: {
  // type is required
  type: string & !=""

  // value is required, and it must not be null
  value: !=null
}
```



注意

此规格中列出的 CUE 模式不可被视为详尽模式。**opm validate** 命令具有额外的验证，很难或不可能在 CUE 中简洁地表达。

Operator Lifecycle Manager (OLM) 目录目前使用三种模式 (**olm.package**、**olm.channel** 和 **olm.bundle**)，它们对应于 OLM 的现有软件包和捆绑包概念。

目录中的每个 Operator 软件包都需要一个 **olm.package** blob、至少一个 **olm.channel** blob 以及一个或多个 **olm.bundle** blob。



注意

所有 **olm.*** 模式都为 OLM 定义的模式保留。自定义模式必须使用唯一前缀，如您拥有的域。

2.2.2.2.1. olm.package schema

olm.package 模式为 Operator 定义软件包级别的元数据。这包括其名称、描述、默认频道和图标。

例 2.1. olm.package schema

```
#Package: {
  schema: "olm.package"

  // Package name
  name: string & !=""
}
```

```

// A description of the package
description?: string

// The package's default channel
defaultChannel: string & !=""

// An optional icon
icon?: {
  base64data: string
  mediatype: string
}
}

```

2.2.2.2.2. olm.channel schema

olm.channel 模式在软件包中定义频道、属于频道成员的捆绑包条目，以及这些捆绑包的升级边缘。

如果捆绑包条目代表多个 **olm.channel** blob 中的边缘，则每个频道只能显示一次。

它对条目的 **replaces** 值有效，以引用无法在此目录或其他目录中找到的另一捆绑包名称。但是，所有其他频道变量都必须为 true，比如频道没有多个磁头。

例 2.2. olm.channel schema

```

#Channel: {
  schema: "olm.channel"
  package: string & !=""
  name: string & !=""
  entries: [...#ChannelEntry]
}

#ChannelEntry: {
  // name is required. It is the name of an `olm.bundle` that
  // is present in the channel.
  name: string & !=""

  // replaces is optional. It is the name of bundle that is replaced
  // by this entry. It does not have to be present in the entry list.
  replaces?: string & !=""

  // skips is optional. It is a list of bundle names that are skipped by
  // this entry. The skipped bundles do not have to be present in the
  // entry list.
  skips?: [...string & !=""]

  // skipRange is optional. It is the semver range of bundle versions
  // that are skipped by this entry.
  skipRange?: string & !=""
}

```



警告

在使用 **skipRange** 字段时，跳过的 Operator 版本会从更新图中删除，因此不再可以被带有 **Subscription** 对象的 **spec.startingCSV** 属性的用户安装。

您可以使用 **skipRange** 和 **replaces** 字段以递增方式更新 Operator，同时保留以前安装的版本供用户使用。确保 **replaces** 字段指向相关的 Operator 版本前一个版本。

2.2.2.2.3. olm.bundle schema

例 2.3. olm.bundle schema

```
#Bundle: {
  schema: "olm.bundle"
  package: string & !=""
  name: string & !=""
  image: string & !=""
  properties: [...#Property]
  relatedImages?: [...#RelatedImage]
}

#Property: {
  // type is required
  type: string & !=""

  // value is required, and it must not be null
  value: !=null
}

#RelatedImage: {
  // image is the image reference
  image: string & !=""

  // name is an optional descriptive name for an image that
  // helps identify its purpose in the context of the bundle
  name?: string & !=""
}
```

2.2.2.2.4. olm.deprecations schema

可选的 **olm.deprecations** 模式定义了目录中软件包、捆绑包和频道的弃用信息。Operator 作者可使用此模式向从目录运行这些 Operator 的用户提供与 Operator 相关的信息，如支持状态和推荐的升级路径。

olm.deprecations schema 条目包含一个或多个 **reference** 类型，这表示弃用范围。安装 Operator 后，可以在相关的 **Subscription** 对象上查看任何指定的信息作为状态条件。

表 2.1. 弃用的 **reference** 类型

类型	影响范围	状态条件
olm.package	代表整个软件包	PackageDeprecated
olm.channel	代表一个频道	ChannelDeprecated
olm.bundle	表示一个捆绑包版本	BundleDeprecated

每个 **reference** 类型都有自己的要求，如下例所示。

例 2.4. 带有每个 **reference** 类型的 **olm.deprecations** 模式示例

```

schema: olm.deprecations
package: my-operator ❶
entries:
- reference:
  schema: olm.package ❷
  message: | ❸
  The 'my-operator' package is end of life. Please use the
  'my-operator-new' package for support.
- reference:
  schema: olm.channel
  name: alpha ❹
  message: |
  The 'alpha' channel is no longer supported. Please switch to the
  'stable' channel.
- reference:
  schema: olm.bundle
  name: my-operator.v1.68.0 ❺
  message: |
  my-operator.v1.68.0 is deprecated. Uninstall my-operator.v1.68.0 and
  install my-operator.v1.72.0 for support.

```

- ❶ 每个弃用模式都必须有一个 **package** 值，且该软件包引用必须在目录间唯一。不能有关联的 **name** 字段。
- ❷ **olm.package** 模式不得包含 **name** 字段，因为它由之前在 schema 中定义的 **package** 字段决定。
- ❸ 所有 **message** 字段（对于任何 **reference** 类型）都必须是一个非零长度，并以 opaque 文本 blob 表示。
- ❹ **olm.channel** schema 的 **name** 字段是必需的。
- ❺ **olm.bundle** schema 的 **name** 字段是必需的。



注意

弃用功能没有考虑重叠的弃用，例如，软件包与频道与捆绑包的比较。

Operator 作者可在与软件包的 `index.yaml` 文件相同的目录中将 `olm.deprecations` schema 条目保存为 `deprecations.yaml` 文件：

带有弃用的目录的目录结构示例

```
my-catalog
├── my-operator
│   ├── index.yaml
│   └── deprecations.yaml
```

其他资源

- [更新或过滤基于文件的目录镜像](#)

2.2.2.3. Properties

属性是可附加到基于文件的目录方案的任意元数据片段。`type` 字段是一个有效指定 `value` 字段语义和语法含义的字符串。该值可以是任意 JSON 或 YAML。

OLM 定义几个属性类型，再次使用保留的 `olm.*` 前缀。

2.2.2.3.1. olm.package 属性

`olm.package` 属性定义软件包名称和版本。这是捆绑包上的必要属性，必须正好有一个这些属性。`packageName` 字段必须与捆绑包的 first-class `package` 字段匹配，并且 `version` 字段必须是有效的语义版本。

例 2.5. olm.package 属性

```
#PropertyPackage: {
  type: "olm.package"
  value: {
    packageName: string & !=""
    version: string & !=""
  }
}
```

2.2.2.3.2. olm.gvk 属性

`olm.gvk` 属性定义此捆绑包提供的 Kubernetes API 的 group/version/kind (GVK)。OLM 使用此属性解析捆绑包，作为列出与所需 API 相同的 GVK 的其他捆绑包的依赖项。GVK 必须遵循 Kubernetes GVK 验证。

例 2.6. olm.gvk 属性

```
#PropertyGVK: {
  type: "olm.gvk"
  value: {
    group: string & !=""
    version: string & !=""
  }
}
```

```

kind: string & !=""
}
}

```

2.2.2.3.3. olm.package.required

olm.package.required 属性定义此捆绑包需要的另一软件包的软件包名称和版本范围。对于捆绑包列表的每个所需软件包属性，OLM 确保集群中为列出的软件包和所需版本范围安装了一个 Operator。**versionRange** 字段必须是有效的语义版本（模拟）范围。

例 2.7. olm.package.required 属性

```

#PropertyPackageRequired: {
  type: "olm.package.required"
  value: {
    packageName: string & !=""
    versionRange: string & !=""
  }
}

```

2.2.2.3.4. olm.gvk.required

olm.gvk.required 属性定义此捆绑包需要的 Kubernetes API 的 group/version/kind (GVK)。对于捆绑包列表的每个必需的 GVK 属性，OLM 确保集群中安装了提供它的 Operator。GVK 必须遵循 Kubernetes GVK 验证。

例 2.8. olm.gvk.required 属性

```

#PropertyGVKRequired: {
  type: "olm.gvk.required"
  value: {
    group: string & !=""
    version: string & !=""
    kind: string & !=""
  }
}

```

2.2.2.4. 目录示例

对于基于文件的目录，目录维护人员可以专注于 Operator 策展和兼容性。由于 Operator 作者已为其 Operator 创建了特定于 Operator 的目录，因此目录维护人员可以通过将每个 Operator 目录渲染到目录根目录的子目录来构建其目录。

构建基于文件的目录的方法有很多；以下步骤概述了一个简单的方法：

1. 为目录维护一个配置文件，其中包含目录中每个 Operator 的镜像引用：

目录配置文件示例

```

name: community-operators

```

```

repo: quay.io/community-operators/catalog
tag: latest
references:
- name: etcd-operator
  image: quay.io/etcd-
operator/index@sha256:5891b5b522d5df086d0ff0b110fbd9d21bb4fc7163af34d08286a2e846f
6be03
- name: prometheus-operator
  image: quay.io/prometheus-
operator/index@sha256:e258d248fda94c63753607f7c4494ee0fcbe92f1a76bfdac795c9d84101
eb317

```

2. 运行一个脚本，该脚本将解析配置文件并从其引用中创建新目录：

脚本示例

```

name=$(yq eval '.name' catalog.yaml)
mkdir "$name"
yq eval '.name + "/" + .references[].name' catalog.yaml | xargs mkdir
for I in $(yq e '.name as $catalog | .references[] | .image + "|" + $catalog + "/" + .name +
"/index.yaml"' catalog.yaml); do
  image=$(echo $I | cut -d'|' -f1)
  file=$(echo $I | cut -d'|' -f2)
  opm render "$image" > "$file"
done
opm generate dockerfile "$name"
indexImage=$(yq eval '.repo + ":" + .tag' catalog.yaml)
docker build -t "$indexImage" -f "$name.Dockerfile" .
docker push "$indexImage"

```

2.2.2.5. 指南

在维护基于文件的目录时，请考虑以下准则。

2.2.2.5.1. 不可变捆绑包

Operator Lifecycle Manager (OLM) 的常规建议是捆绑包镜像及其元数据应视为不可变。

如果一个错误的捆绑包被推送到目录，您必须假设至少有一个用户已升级到该捆绑包。基于这种假设，您必须从损坏的捆绑包中发布另一个带有升级边缘的捆绑包，以确保安装了有问题的捆绑包的用户收到升级。如果目录中更新了该捆绑包的内容，OLM 将不会重新安装已安装的捆绑包。

然而，在某些情况下首选更改目录元数据：

- 频道升级：如果您已发布了捆绑包，且之后决定将其添加到另一个频道，您可以在另一个 **olm.channel** blob 中添加捆绑包条目。
- 新的升级边缘：如果您发布一个新的 **1.2.z** 捆绑包版本，如 **1.2.4**，但 **1.3.0** 已发布，您可以更新 **1.3.0** 的目录元数据以跳过 **1.2.4**。

2.2.2.5.2. 源控制

目录元数据应存储在源控制中，并被视为事实来源。目录镜像的更新应包括以下步骤：

1. 使用新的提交来更新源控制的目录目录。

2. 构建并推送目录镜像。使用一致的标记分类，如 `:latest` 或 `:<target_cluster_version>`，以便用户可以在目录可用时接收到更新。

2.2.2.6. CLI 用法

有关使用 **opm** CLI 创建基于文件的目录的说明，请参阅[管理自定义目录](#)。

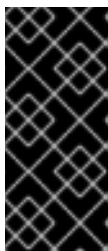
有关管理基于文件的目录的 **opm** CLI 命令的参考文档，请参阅 [CLI 工具](#)。

2.2.2.7. 自动化

建议 Operator 作者和目录维护人员使用 CI/CD 工作流自动化其目录维护。目录维护人员可通过构建 GitOps 自动化以完成以下任务来进一步改进：

- 检查是否允许拉取请求 (PR) 作者进行请求的更改，例如更新其软件包的镜像引用。
- 检查目录更新是否通过 **opm validate** 命令。
- 检查是否有更新的捆绑包或目录镜像引用，目录镜像在集群中成功运行，来自该软件包的 Operator 可以成功安装。
- 自动合并通过之前检查的 PR。
- 自动重新构建和重新发布目录镜像。

2.2.3. RukPak（技术预览）



重要

RukPak 只是一个技术预览功能。技术预览功能不受红帽产品服务等级协议 (SLA) 支持，且功能可能并不完整。红帽不推荐在生产环境中使用它们。这些技术预览功能可以使用户提早试用新的功能，并有机会在开发阶段提供反馈意见。

有关红帽技术预览功能支持范围的更多信息，请参阅[技术预览功能支持范围](#)。

OpenShift Container Platform 4.12 引入了 *平台 Operator* 类型作为技术预览功能。平台 Operator 机制依赖于 RukPak 组件，也包含在 OpenShift Container Platform 4.12 中，及其用于管理内容的资源。

OpenShift Container Platform 4.14 引入了 Operator Lifecycle Manager (OLM) 1.0 作为技术预览功能，它基于 RukPak 组件。

RukPak 是一个可插拔式解决方案，用于打包和分发云原生内容。它支持安装、更新和策略的高级策略。

RukPak 提供用于在 Kubernetes 集群上安装各种工件的内容生态系统。工件示例包括 Git 仓库、Helm chart 和 OLM 捆绑包。然后，RukPak 可以以安全的方式管理、扩展和升级这些工件，以启用强大的集群扩展。

在其核心上，RukPak 是一组 API 和控制器。API 打包为自定义资源定义 (CRD)，用于表达要在集群中安装的内容以及如何创建运行的内容。控制器会监视 API。

常见术语

捆绑包 (Bundle)

定义要部署到集群的内容的 Kubernetes 清单集合

捆绑包镜像

在其文件系统中包含捆绑包的容器镜像

捆绑包 Git 存储库

在目录中包含捆绑包的 Git 存储库

Provisioner

在 Kubernetes 集群上安装和管理内容的控制器

捆绑包部署

生成捆绑包部署的实例

其他资源

- [管理平台 Operator](#)
- [平台 Operator 的技术预览限制](#)
- [关于 Operator Lifecycle Manager 1.0（技术预览）](#)

2.2.3.1. 捆绑包（Bundle）

RukPak **Bundle** 对象代表可供集群中的其他使用者使用的内容。就像需要拉取和解包容器镜像的内容一样，pod 才能开始使用它们，使用 **Bundle** 对象来引用可能需要拉取和解包的内容。因此，捆绑包是镜像概念的规范化，可用于表示任何类型的内容。

捆绑包无法自行执行；它们需要置备程序来解包并在集群中提供其内容。它们可以解包到任何任意存储介质，如挂载到 provisioner pod 目录中的 **tar.gz** 文件。每个 **Bundle** 对象都有一个关联的 **spec.provisionerClassName** 字段，它指示 **Provisioner** 对象监视和解包该特定捆绑包类型。

配置为与普通置备程序一起使用的 Bundle 对象示例

```

apiVersion: core.rukpak.io/v1alpha1
kind: Bundle
metadata:
  name: my-bundle
spec:
  source:
    type: image
    image:
      ref: my-bundle@sha256:xyz123
  provisionerClassName: core-rukpak-io-plain

```



注意

捆绑包在创建后被视为不可变。

2.2.3.1.1. 捆绑包不可变

在 API 服务器接受 **Bundle** 对象后，捆绑包被视为 RukPak 系统的其余部分的不可变工件。这个行为强制捆绑包代表集群中要 **source** 的一些唯一静态内容。用户可以放心，特定捆绑包指向特定的清单集合，在没有创建新捆绑包的情况下无法更新。对于由嵌入式 **BundleTemplate** 对象创建的单机捆绑包和动态捆绑包，此属性为 **true**。

捆绑包的不可变性由核心 RukPak webhook 强制执行。此 webhook 监视 **Bundle** 对象事件，并对捆绑包

的任何更新，检查现有捆绑包的 **spec** 字段是否与建议的捆绑包中的 **spec** 字段相同。如果它们不相等，则 Webhook 将拒绝更新。在捆绑包的生命周期中，其他 **Bundle** 对象字段（如 **metadata** 或 **status**）会更新，它只是被视为不可变的 **spec** 字段。

应用 **Bundle** 对象，然后尝试更新其 **spec** 会失败。例如，以下示例会创建一个捆绑包：

```
$ oc apply -f -<<EOF
apiVersion: core.rukpak.io/v1alpha1
kind: Bundle
metadata:
  name: combo-tag-ref
spec:
  source:
    type: git
    git:
      ref:
        tag: v0.0.2
      repository: https://github.com/operator-framework/combo
  provisionerClassName: core-rukpak-io-plain
EOF
```

输出示例

```
bundle.core.rukpak.io/combo-tag-ref created
```

然后，修补捆绑包以指向较新的标签会返回错误：

```
$ oc patch bundle combo-tag-ref --type='merge' -p '{"spec":{"source":{"git":{"ref":{"tag":"v0.0.3"}}}}}'
```

输出示例

```
Error from server (bundle.spec is immutable): admission webhook "vbundles.core.rukpak.io" denied the request: bundle.spec is immutable
```

核心 RukPak 准入 Webhook 会拒绝补丁，因为捆绑包的 **spec** 是不可变的。推荐的修改捆绑包内容的方法是通过创建一个新的 **Bundle** 对象而不是原位更新。

进一步的不可变注意事项

虽然 **Bundle** 对象的 **spec** 字段不可变，但 **BundleDeployment** 对象仍有可能变为较新的捆绑包内容版本，而无需更改底层 **spec** 字段。这个非预期的行为可能会在以下情况中出现：

1. 用户在 **Bundle** 对象的 **spec.source** 字段中设置镜像标签、Git 分支或 Git 标签。
2. 镜像标签移到新的摘要、用户将更改推送到 Git 分支，或者用户删除并在不同的提交上重新推送 Git 标签。
3. 用户执行一些操作，以便重新创建捆绑包解包 pod，如删除解包 pod。

如果发生这种情况，步骤 2 中的新内容会因为第 3 步而解包。捆绑包部署会检测更改，并探测到新版本的内容。

这与 pod 的行为类似，其中一个 pod 的容器镜像使用标签，标签将移到不同的摘要中，然后在将来将现有 pod 重新调度到其他节点上。此时，节点会将新镜像拉取到新摘要中，并在没有用户明确要求的情况下运行不同的镜像。

为了确定底层 **Bundle** spec 内容没有改变，请在创建捆绑包时使用基于摘要的镜像或 Git 提交引用。

2.2.3.1.2. 普通捆绑包规格

RukPak 中的普通捆绑包是给定目录中静态、任意的 Kubernetes YAML 清单的集合。

当前实施的普通捆绑包格式是 **plain+v0** 格式。捆绑包格式的名称 (**plain+v0**) 组合了捆绑包类型 (**plain**) 和当前模式版本 (**v0**)。



注意

plain+v0 捆绑包格式是 schema 版本 **v0**，这意味着它可能会在以后有所变化。

例如，下面显示了 **plain+v0** 捆绑包中的文件树。它必须具有一个 **manifests/** 目录，其中包含部署应用程序所需的 Kubernetes 资源。

plain+v0 捆绑包文件树示例

```
$ tree manifests
manifests
├── namespace.yaml
├── service_account.yaml
├── cluster_role.yaml
├── cluster_role_binding.yaml
└── deployment.yaml
```

静态清单必须位于 **manifests/** 目录中，其中至少包含一个资源，以便捆绑包是置备程序可以解包的有效 **plain+v0** 捆绑包。**manifests/** 目录还必须是扁平的；所有清单都必须位于顶层，且无子目录。



重要

不要在不是静态清单的普通捆绑包的 **manifests/** 目录中包含任何内容。否则，从该捆绑包创建内容时会出现错误。任何不能使用 **oc apply** 命令应用的文件将导致错误。多对象 YAML 或 JSON 文件也有效。

2.2.3.1.3. registry 捆绑包规格

registry 捆绑包或 **registry+v1** 捆绑包包含一组以旧 Operator Lifecycle Manager (OLM) 捆绑包格式组织的静态 Kubernetes YAML 清单。

其他资源

- [旧的 OLM 捆绑包格式](#)

2.2.3.2. BundleDeployment



警告

BundleDeployment 对象通过安装和删除对象来更改 Kubernetes 集群的状态。务必要验证并信任正在安装和限制访问权限的内容，方法是使用 RBAC 到 **BundleDeployment** API 到需要这些权限的用户。

RukPak **BundleDeployment** API 指向 **Bundle** 对象，并表明它应当处于活动状态。这包括从活跃捆绑包的旧版本获取。**BundleDeployment** 对象可能还包括所需捆绑包的嵌入式 spec。

与 pod 生成容器镜像实例一样，捆绑包部署会生成捆绑包部署的版本。捆绑包部署可被视为 pod 概念的一般化。

捆绑包部署如何根据引用的捆绑包对集群进行更改，具体由配置为监视该捆绑包部署的置备程序定义。

配置为与普通置备程序一起工作的 BundleDeployment 对象示例

```
apiVersion: core.rukpak.io/v1alpha1
kind: BundleDeployment
metadata:
  name: my-bundle-deployment
spec:
  provisionerClassName: core-rukpak-io-plain
  template:
    metadata:
      labels:
        app: my-bundle
    spec:
      source:
        type: image
        image:
          ref: my-bundle@sha256:xyz123
        provisionerClassName: core-rukpak-io-plain
```

2.2.3.3. 关于置备程序

RukPak 由一系列控制器组成，称为 *provisioners*，后者在 Kubernetes 集群上安装和管理内容。RukPak 还提供两个主要 API：**Bundle** 和 **BundleDeployment**。这些组件协同工作，将内容整合到集群中，并安装它，并在集群中生成资源。

目前，两个置备程序被实施并捆绑到 RukPak：*plain provisioner* source 并解包 **plain+v0** 捆绑包；*registry provisioner* source Operator Lifecycle Manager (OLM) **registry+v1** 捆绑包。

每个置备程序都会被分配一个唯一 ID，它负责协调与该特定 ID 匹配的 **spec.provisionerClassName** 字段的 **Bundle** 和 **BundleDeployment** 对象。例如，普通置备程序可以解包给定 **plain+v0** 捆绑包到集群中，然后实例化它，使捆绑包的内容在集群中可用。

置备程序会在 **Bundle** 和 **BundleDeployment** 资源上明确观察到引用置备程序的资源。对于给定捆绑包，置备程序会将 **Bundle** 资源的内容解压缩到集群中。然后，如果一个指向该捆绑包的 **BundleDeployment** 资源，置备程序会安装捆绑包内容，并负责管理这些资源的生命周期。

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，包括 Operator Lifecycle Manager (OLM) 和 Operator SDK。

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* 代表 OLM 可查询的元数据存储，以发现和安装 Operator 及其依赖项。

2.3.1.4. Channel

*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。

2.3.1.5. 频道头

*频道头*是指特定频道中最新已知的更新。

2.3.1.6. 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)。

2.3.1.7. 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。

2.3.1.8. 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。

2.3.1.9. 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 而需创建的资源计算列表。

2.3.1.10. 多租户

OpenShift Container Platform 中的 *租户* 是为了一组部署的工作负载（通常由命名空间或项目表示）共享共同访问权限和特权的用户或组。您可以使用租户在不同的组或团队之间提供一定程度的隔离。

当集群由多个用户或组共享时，它被视为 *多租户* 集群。

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)

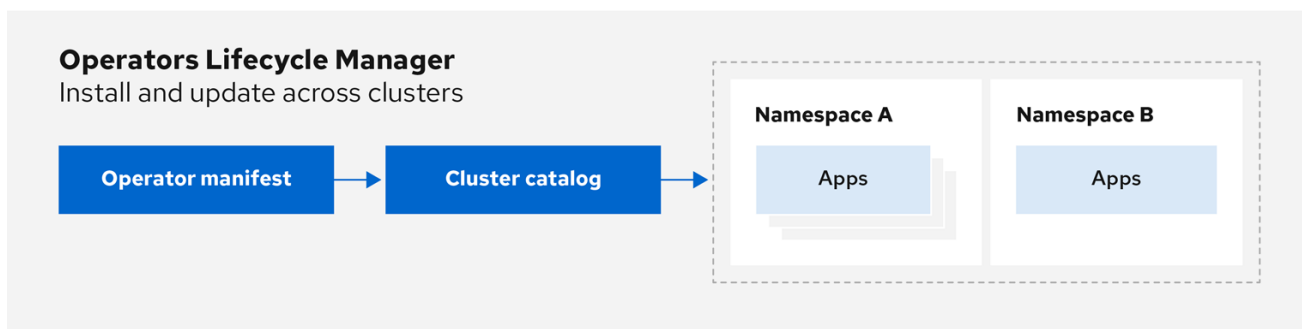
2.4.1. Operator Lifecycle Manager 概念和资源

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 workflow



OpenShift_43_1019

OLM 默认在 OpenShift Container Platform 4.15 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 2.2. 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** → **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 2.9. CatalogSource 对象示例

```
apiVersion: operators.coreos.com/v1alpha1
kind: CatalogSource
metadata:
  generation: 1
  name: example-catalog ①
  namespace: openshift-marketplace ②
  annotations:
    olm.catalogImageTemplate: ③
    "quay.io/example-org/example-catalog:v{kube_major_version}.{kube_minor_version}.
{kube_patch_version}"
spec:
  displayName: Example Catalog ④
  image: quay.io/example-org/example-catalog:v1 ⑤
  priority: -400 ⑥
  publisher: Example Org
  sourceType: grpc ⑦
  grpcPodConfig:
    securityContextConfig: <security_mode> ⑧
    nodeSelector: ⑨
    custom_label: <label>
    priorityClassName: system-cluster-critical ⑩
  tolerations: ⑪
    - key: "key1"
      operator: "Equal"
      value: "value1"
      effect: "NoSchedule"
  updateStrategy:
```



```

registryPoll: 12
  interval: 30m0s
status:
connectionState:
  address: example-catalog.openshift-marketplace.svc:50051
  lastConnect: 2021-08-26T18:14:31Z
  lastObservedState: READY 13
latestImageRegistryPoll: 2021-08-26T18:46:25Z 14
registryService: 15
  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 要创建目录的命名空间。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 可选：为避免集群升级可能会使 Operator 安装处于不受支持的状态或没有持续更新路径，您可以启用自动更改 Operator 目录的索引镜像版本作为集群升级的一部分。

将 **olm.catalogImageTemplate** 注解设置为索引镜像名称，并使用一个或多个 Kubernetes 集群版本变量，如为镜像标签构建模板时所示。该注解会在运行时覆盖 **spec.image** 字段。如需了解更多详细信息，请参阅“用于自定义目录源的镜像模板”。

- 4 在 Web 控制台和 CLI 中显示目录的名称。
- 5 目录的索引镜像。在使用 **olm.catalogImageTemplate** 注解时，也可以省略，该注解会在运行时设置 pull spec。
- 6 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。
- 7 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。
- 8 指定 **legacy** 或 **restricted** 的值。如果没有设置该字段，则默认值为 **legacy**。在以后的 OpenShift Container Platform 发行版本中，计划默认值为 **restricted**。如果您的目录无法使用 **restricted** 权限运行，建议您手动将此字段设置为 **legacy**。
- 9 可选：对于 **grpc** 类型目录源，请覆盖在 **spec.image** 中提供内容的 pod 的默认节点选择器（如果定义）。
- 10 可选：对于 **grpc** 类型目录源，请覆盖在 **spec.image** 中提供内容的 pod 的默认优先级类名称（如果定义）。Kubernetes 默认提供 **system-cluster-critical** 和 **system-node-critical** 优先级类。将字段设置为空 ("") 可为 pod 分配默认优先级。可以手动定义其他优先级类。

- 11 可选：对于 **grpc** 类型目录源，请覆盖 **spec.image** 中提供内容的 pod 的默认容限（如果定义）。
- 12 在指定的时间段内自动检查新版本以保持最新。
- 13 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](#)。

- 14 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。
- 15 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 2.10. 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
```

其他资源

- [了解 OperatorHub](#)
- [红帽提供的 Operator 目录](#)
- [在集群中添加目录源](#)
- [目录优先级](#)
- [使用 CLI 查看 Operator 目录源状态](#)
- [了解并管理 pod 安全准入](#)
- [目录源 pod 调度](#)

2.4.1.2.2.1. 自定义目录源的镜像模板

与底层集群的 Operator 兼容性可以通过目录源以各种方式表示。其中一种用于红帽默认提供的目录源的方法是识别为特定平台发行版本（如 OpenShift Container Platform 4.15）特别创建的索引镜像的镜像标签。

在集群升级过程中，默认红帽提供的目录源的索引镜像标签由 Cluster Version Operator (CVO) 自动更新，以便 Operator Lifecycle Manager (OLM) 拉取目录的更新版本。例如，在从 OpenShift Container Platform 4.14 升级到 4.15 的过程中，**redhat-operators** 目录的 **CatalogSource** 对象中的 **spec.image** 字段被更新：

```
registry.redhat.io/redhat/redhat-operator-index:v4.14
```

改为：

```
registry.redhat.io/redhat/redhat-operator-index:v4.15
```

但是，CVO 不会自动更新自定义目录的镜像标签。为确保用户在集群升级后仍然安装兼容并受支持的 Operator，还应更新自定义目录以引用更新的索引镜像。

从 OpenShift Container Platform 4.9 开始，集群管理员可以在自定义目录的 **CatalogSource** 对象中添加 **olm.catalogImageTemplate** 注解到包含模板的镜像引用。模板中支持使用以下 Kubernetes 版本变量：

- **kube_major_version**
- **kube_minor_version**
- **kube_patch_version**



注意

您必须指定 Kubernetes 集群版本，而不是 OpenShift Container Platform 集群版本，因为后者目前不适用于模板。

如果您已创建并推送了带有指定更新 Kubernetes 版本标签的索引镜像，设置此注解可使自定义目录中的索引镜像版本在集群升级后自动更改。注解值用于设置或更新 **CatalogSource** 对象的 **spec.image** 字段中的镜像引用。这有助于避免集群升级，从而避免在不受支持的状态或没有持续更新路径的情况下安装 Operator。



重要

您必须确保集群可在集群升级时访问带有更新标签的索引镜像（无论存储在哪一 registry 中）。

例 2.11. 带有镜像模板的目录源示例

```
apiVersion: operators.coreos.com/v1alpha1
kind: CatalogSource
metadata:
  generation: 1
  name: example-catalog
  namespace: openshift-marketplace
annotations:
  olm.catalogImageTemplate:
    "quay.io/example-org/example-catalog:v{kube_major_version}.{kube_minor_version}"
```

```
spec:
  displayName: Example Catalog
  image: quay.io/example-org/example-catalog:v1.28
  priority: -400
  publisher: Example Org
```



注意

如果设置了 **spec.image** 字段和 **olm.catalogImageTemplate** 注解，则 **spec.image** 字段会被注解中的解析值覆盖。如果注解没有解析为可用的 pull spec，目录源会回退到设置的 **spec.image** 值。

如果没有设置 **spec.image** 字段，且注解没有解析为可用的 pull spec，OLM 会停止目录源的协调，并将其设置为人类可读的错误条件。

对于使用 Kubernetes 1.28 的 OpenShift Container Platform 4.15 集群，上例中的 **olm.catalogImageTemplate** 注解会解析为以下镜像引用：

```
quay.io/example-org/example-catalog:v1.28
```

对于将来的 OpenShift Container Platform 版本，您可以为自定义目录创建更新的索引镜像，该镜像以更新的 Kubernetes 版本为目标，供以后的 OpenShift Container Platform 版本使用。升级前设置了 **olm.catalogImageTemplate** 注解，将集群升级到更新的 OpenShift Container Platform 版本也会自动更新目录的索引镜像。

2.4.1.2.2.2. 目录健康要求

集群上的 Operator 目录可从安装解析视角进行交换；**Subscription** 对象可能会引用特定目录，但依赖项会根据集群中的所有目录解决。

例如，如果 Catalog A 不健康，则引用 Catalog A 的订阅可能会解析 Catalog B 中的依赖项，集群管理员可能还没有预期，因为 B 通常具有比 A 更低的目录优先级。

因此，OLM 要求所有具有给定全局命名空间的目录（例如，默认的 **openshift-marketplace** 命名空间或自定义全局命名空间）都健康。当目录不健康时，其共享全局命名空间中的所有 Operator 或更新操作都将因为 **CatalogSourcesUnhealthy** 条件而失败。如果这些操作处于不健康状态，OLM 可能会做出对集群管理员意外的解析和安装决策。

作为集群管理员，如果您观察一个不健康的目录，并希望将目录视为无效并恢复 Operator 安装，请参阅“删除自定义目录”或“Disabling the default OperatorHub 目录源”部分，以了解有关删除不健康目录的信息。

其他资源

- [删除自定义目录](#)
- [禁用默认的 OperatorHub 目录源](#)

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。

其他资源

- [多租户和 Operator 共处](#)
- [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 2.12. 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
...

```

其他资源

- [多租户和 Operator 共处](#)
- [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。

其他资源

- [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 **Spec.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。



注意

默认情况下，**OperatorCondition** 对象中没有 **Spec.Conditions** 数组，直到由用户添加或使用自定义 Operator 逻辑的结果为止。

其他资源

- [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 2.3. 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 而需创建的资源计算列表。
CatalogSource	catalog	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 2.4. 由 OLM 和 Catalog Operator 创建的资源

资源	所有者
部署	OLM
ServiceAccounts	
(Cluster)Roles	
(Cluster)RoleBindings	

资源	所有者
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 CSV 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。

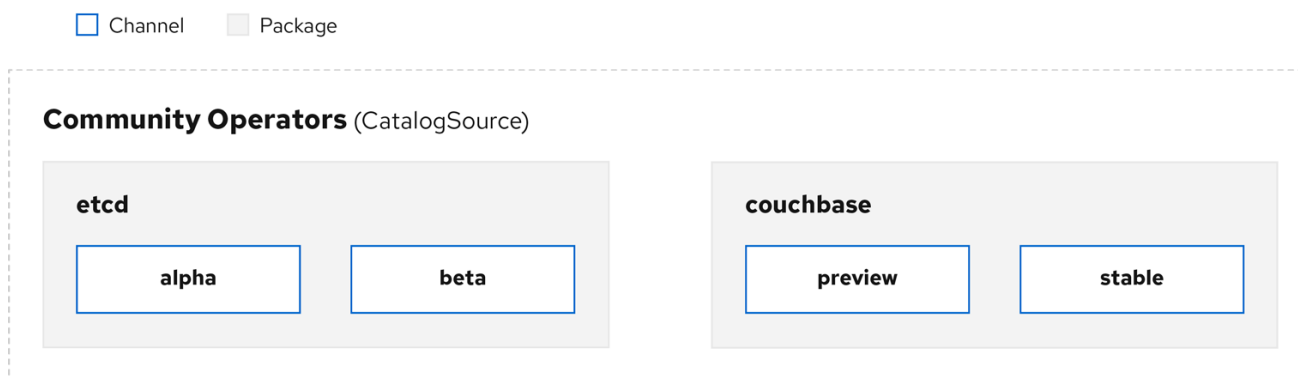
2.4.3.1. OLM 中的 Operator 安装和升级工作流

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。

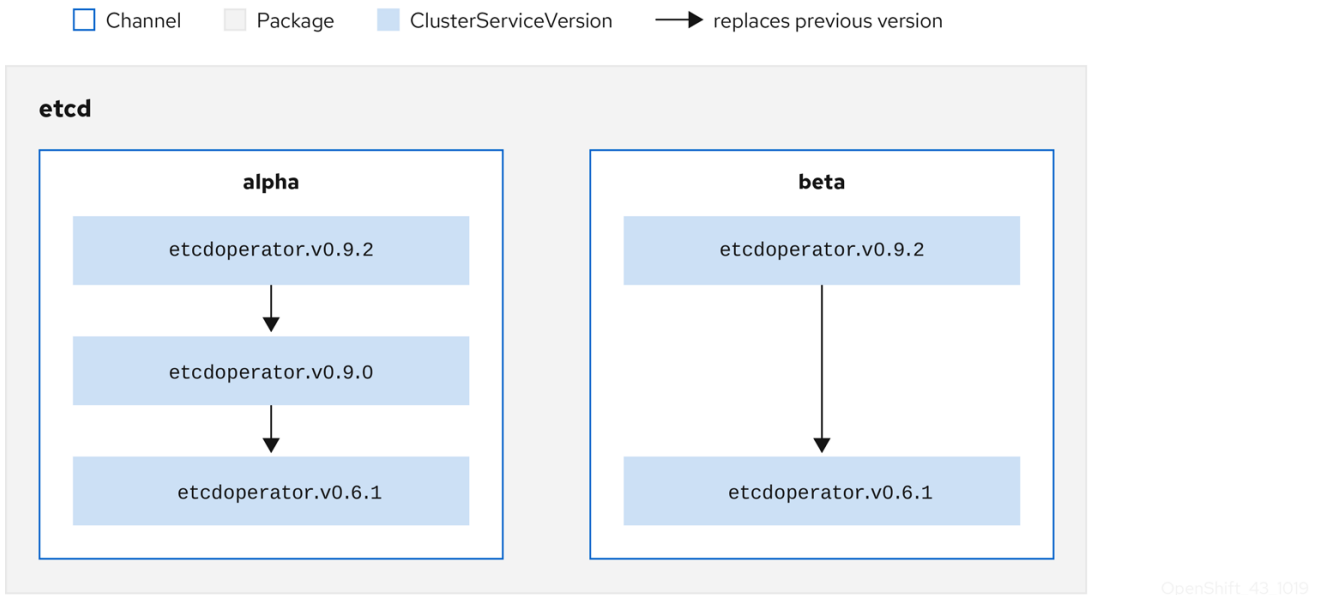
图 2.3. 目录源概述



OpenShift_43_1019

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。

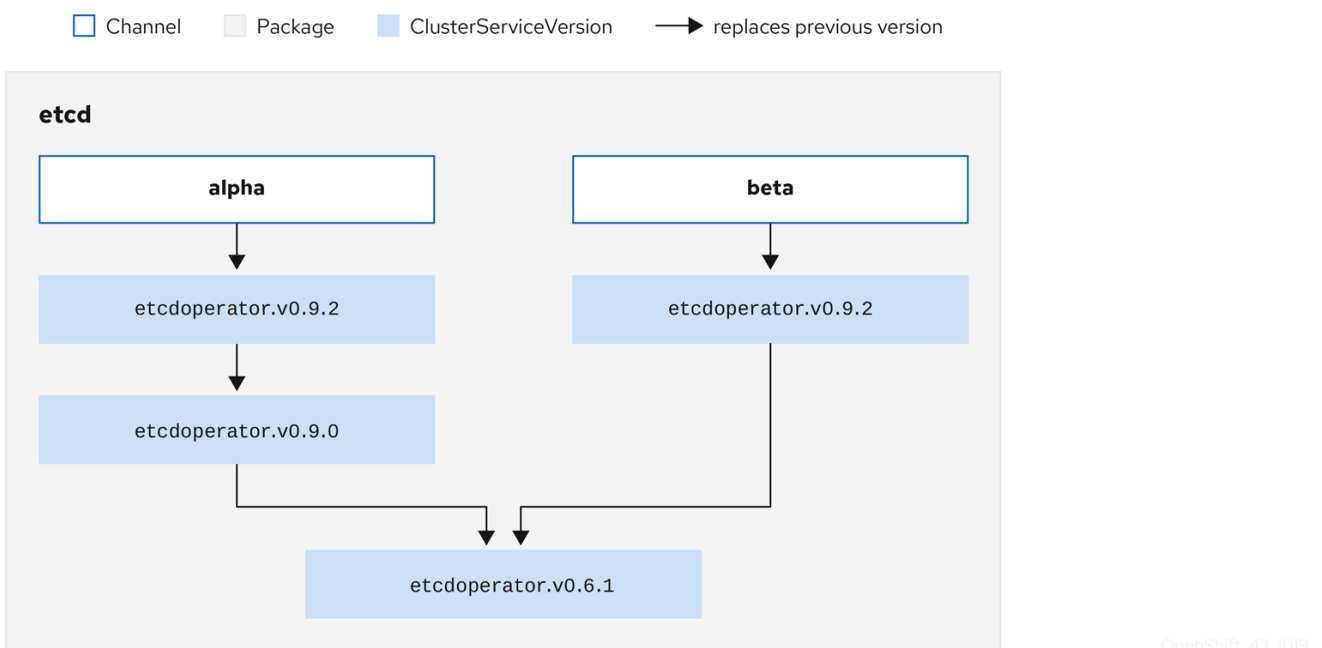


注意

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

```

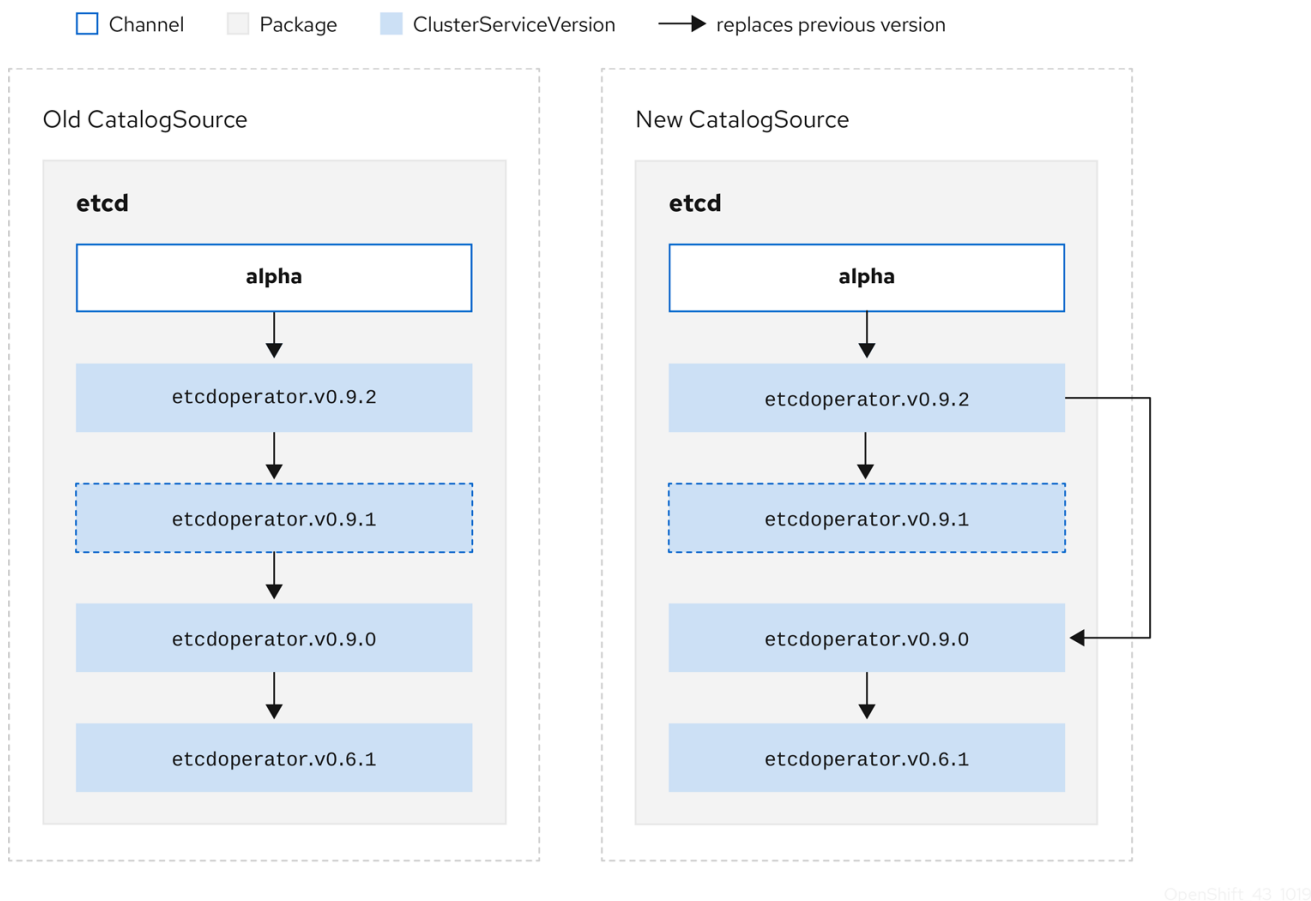
apiVersion: operators.coreos.com/v1alpha1
kind: ClusterServiceVersion
metadata:
  name: etcdoperator.v0.9.2
  namespace: placeholder
  annotations:
spec:
  displayName: etcd
  description: Etcd Operator

```

```
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



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

```

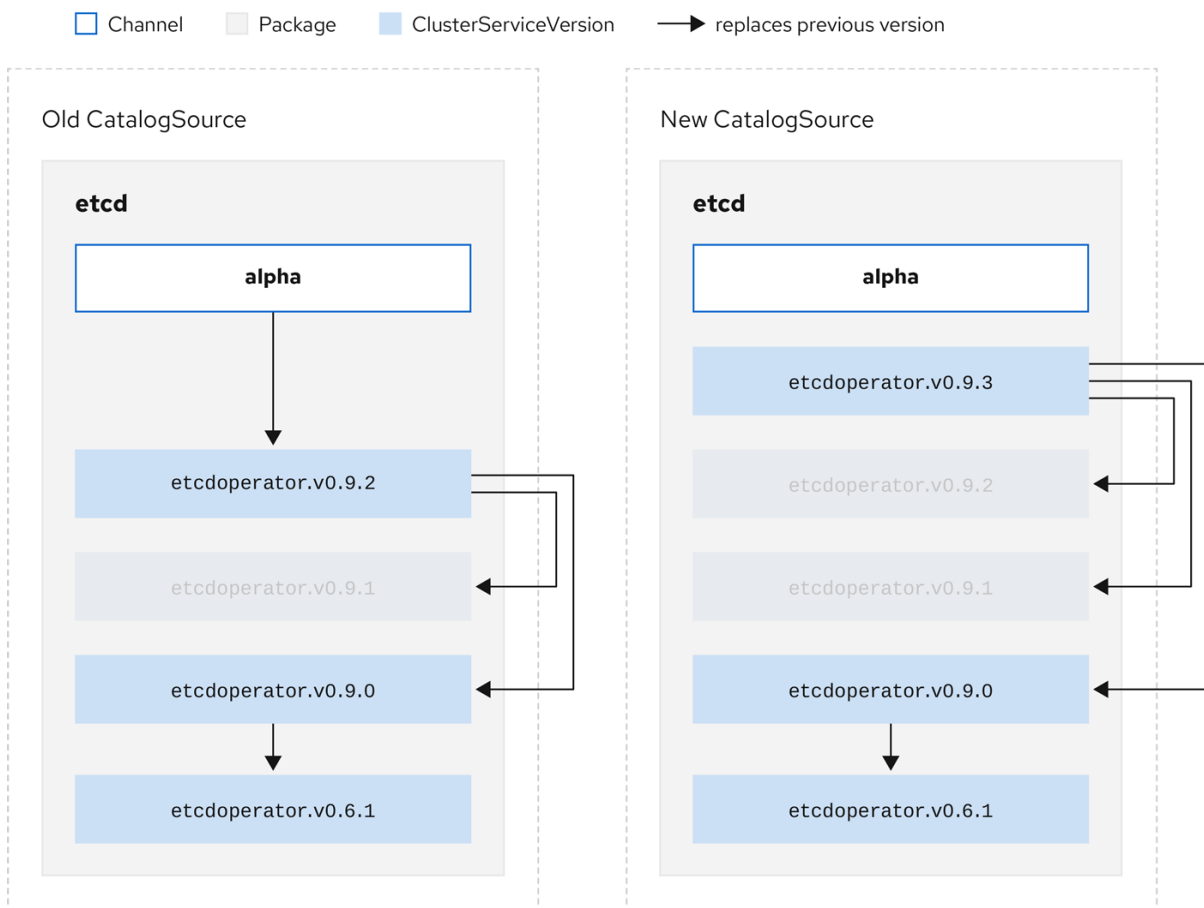
apiVersion: operators.coreos.com/v1alpha1
kind: ClusterServiceVersion
metadata:
  name: elasticsearch-operator.v4.1.2
  namespace: <namespace>
  annotations:
    olm.skipRange: '>=4.1.0 <4.1.2'
    
```

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，像在新 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

Operator Lifecycle Manager(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 的问题与其他系统或语言软件包管理器类似，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。

因此，OLM 不得创建以下情况：

- 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

这可以通过两种类型的数据：

Properties	在依赖项解析器中输入构成了公共接口的 Operator 元数据。示例包括 Operator 提供的 API 的 group/version/kind(GVK)，以及 Operator 的语义版本(semver)。
约束或依赖项	应该对可能或还没有在目标集群中安装的其他 Operator 满足 Operator 的要求。它们充当所有可用 Operator 的查询或过滤，并在依赖项解析和安装过程中限制选择。例如，需要特定的 API 在集群中可用，或希望安装带有特定版本的特定 Operator。

OLM 将这些属性和约束转换为布尔值公式系统，并将其传递给 SAT solver，SAT solver 是一个处理布尔值的程序，用于确定应该安装哪些 Operator。

2.4.4.2. Operator 属性

目录中的所有 Operator 均具有以下属性：

olm.package

包括软件包和 Operator 版本的名称

olm.gvk

集群服务版本(CSV)中每个提供的 API 的单个属性

Operator 作者也可以在 Operator 捆绑包的 `metadata/` 目录中包括 `properties.yaml` 文件来直接声明其他属性。

任意 (arbitrary) 属性示例

```
properties:
- type: olm.kubeversion
  value:
    version: "1.16.0"
```

2.4.4.2.1. 任意属性

Operator 作者可在 Operator 捆绑包的 `metadata/` 目录中的 `properties.yaml` 文件中声明任意属性。这些属性转换为映射数据结构，该结构用作运行时 Operator Lifecycle Manager(OLM)解析器的输入。

这些属性对解析器不理解属性而不理解这些属性，但可以针对这些属性评估通用限制，以确定约束是否可以满足给定的属性列表。

任意属性示例

```
properties:
- property:
  type: color
  value: red
- property:
  type: shape
  value: square
- property:
  type: olm.gvk
  value:
    group: olm.coreos.io
    version: v1alpha1
    kind: myresource
```

此结构可用于为通用限制构建通用表达式语言(CEL)表达式。

其他资源

- [常见表达式语言\(CEL\)约束](#)

2.4.4.3. Operator 依赖项

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。支持以下 Operator 依赖项：

`olm.package`

这个类型表示特定 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`

使用这个类型，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 的使用量。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。

olm.constraint

这个类型在任意 Operator 属性上声明通用限制。

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
```

2.4.4.4. 通用限制

olm.constraint 属性声明特定类型的依赖项约束，区分非约束和约束属性。其 **value** 字段是一个包含 **failureMessage** 字段的对象，其中包含约束消息的字符串表。如果约束在运行时不满意，则这一消息被作为信息性提供给用户使用。

以下键表示可用的约束类型：

gvk

其值及对其的解释与 **olm.gvk** 类型相同的类型

package

其值及对其的解释与 **olm.package** 类型相同的类型

cel

Operator Lifecycle Manager(OLM)解析程序通过任意捆绑包属性和集群信息在运行时评估的通用表达式语言(CEL)表达式

all, any, not

分别为 Conjunction, disjunction, 和 negation 约束，包括一个或多个 concrete 约束，如 **gvk** 或一个嵌套的 compound 约束

2.4.4.4.1. 常见表达式语言(CEL)约束

cel 约束类型支持将通用表达式语言(CEL)用作表达式语言。**cel struct** 有一个 **rule** 字段，其中包含在运行时针对 Operator 属性评估的 CEL 表达式字符串，以确定 Operator 是否满足约束。

cel 约束示例

```
type: olm.constraint
value:
  failureMessage: 'require to have "certified"'
  cel:
    rule: 'properties.exists(p, p.type == "certified")'
```

CEL 语法支持广泛的逻辑运算符，如 **AND** 和 **OR**。因此，单个 CEL 表达式可以具有多个规则，这些条件由这些逻辑运算符链接在一起。这些规则针对来自捆绑包或任何给定源的多个不同属性的数据评估，输出可以解决单一约束内满足所有这些规则的捆绑包或 Operator。

使用多个规则的 cel 约束示例

```
type: olm.constraint
value:
  failureMessage: 'require to have "certified" and "stable" properties'
  cel:
    rule: 'properties.exists(p, p.type == "certified") && properties.exists(p, p.type == "stable")'
```

2.4.4.4.2. Compound 约束 (all, any, not)

复合约束类型按照其逻辑定义进行评估。

以下是两个软件包的 conjunctive 约束 (**all**) 的示例，以及一个 GVK。这代表，安装捆绑包都必须满足它们：

all 约束示例

```
schema: olm.bundle
name: red.v1.0.0
properties:
- type: olm.constraint
  value:
    failureMessage: All are required for Red because...
    all:
      constraints:
      - failureMessage: Package blue is needed for...
        package:
          name: blue
          versionRange: '>=1.0.0'
      - failureMessage: GVK Green/v1 is needed for...
        gvk:
          group: greens.example.com
          version: v1
          kind: Green
```

以下是同一个 GVK 的三个版本的 disjunctive 约束 (**any**) 示例。这代表，安装捆绑包必须至少满足其中一项：

any 约束示例

```
schema: olm.bundle
name: red.v1.0.0
properties:
- type: olm.constraint
  value:
    failureMessage: Any are required for Red because...
    any:
      constraints:
      - gvk:
          group: blues.example.com
```

```

    version: v1beta1
    kind: Blue
  - gvk:
    group: blues.example.com
    version: v1beta2
    kind: Blue
  - gvk:
    group: blues.example.com
    version: v1
    kind: Blue

```

以下是 GVK 的一个版本的 negation 约束 (**not**) 的示例。这代表, 此 GVK 无法由结果集中的任何捆绑包提供:

not 约束示例

```

schema: olm.bundle
name: red.v1.0.0
properties:
- type: olm.constraint
  value:
    all:
      constraints:
      - failureMessage: Package blue is needed for...
        package:
          name: blue
          versionRange: '>=1.0.0'
      - failureMessage: Cannot be required for Red because...
        not:
          constraints:
          - gvk:
            group: greens.example.com
            version: v1alpha1
            kind: greens

```

对于 **not** 约束, 其中的负语义可能并不明确。这里的负语义代表指示解析器删除所有可能的解决方案, 这些解决方案包括特定 GVK、特点版本的软版本, 或满足结果集中的一些子复合约束。

not compound 约束不应该和 **all** 或 **any** 一起使用, 因为这里的负语言在没有先选择一组可能的依赖项时是并没有意义。

2.4.4.4.3. 嵌套复合限制

一个嵌套复合约束 (包括最少一个子复合约束以及零个或更多简单约束) 会从底向上的顺序被评估, 并根据每个前面描述的约束类型的过程进行。

以下是一个 disjunction 的 conjunctions 示例, 其中一个、另一个、或两者都能满足约束:

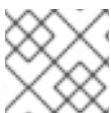
嵌套复合约束示例

```

schema: olm.bundle
name: red.v1.0.0
properties:
- type: olm.constraint
  value:

```

```
failureMessage: Required for Red because...
any:
  constraints:
  - all:
    constraints:
    - package:
      name: blue
      versionRange: '>=1.0.0'
    - gvk:
      group: blues.example.com
      version: v1
      kind: Blue
  - all:
    constraints:
    - package:
      name: blue
      versionRange: '<1.0.0'
    - gvk:
      group: blues.example.com
      version: v1beta1
      kind: Blue
```



注意

olm.constraint 类型的最大原始大小为 64KB，用于限制资源耗尽的情况。

2.4.4.5. 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。

2.4.4.5.1. 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
  grpcPodConfig:
    securityContextConfig: <security_mode> 1
  image: example.com/my/operator-index:v1
  displayName: "My Operators"
  priority: 100
```

1 指定 **legacy** 或 **restricted** 的值。如果没有设置该字段，则默认值为 **legacy**。在以后的 OpenShift Container Platform 发行版本中，计划默认值为 **restricted**。如果您的目录无法使用 **restricted** 权限运行，建议您手动将此字段设置为 **legacy**。

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。

2.4.4.5.2. 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。

2.4.4.5.3. 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。

2.4.4.5.4. 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。

2.4.4.5.4.1. 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。

2.4.4.5.4.2. 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。

2.4.4.5.5. 其他资源

- [目录健康要求](#)

2.4.4.6. 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](#)

2.4.4.7. 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](#)

2.4.4.8. 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。

2.4.4.9. 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 2.5. 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
```

Operator 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。

注解	描述
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
<code>olm.og.<operatorgroup_name>-admin- <hash_value></code>	<code>olm.opgroup.permissions/aggregate-to-admin: <operatorgroup_name></code>
<code>olm.og.<operatorgroup_name>-edit- <hash_value></code>	<code>olm.opgroup.permissions/aggregate-to-edit: <operatorgroup_name></code>
<code>olm.og.<operatorgroup_name>-view- <hash_value></code>	<code>olm.opgroup.permissions/aggregate-to-view: <operatorgroup_name></code>

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 2.6. 来自 CRD 的为每个 API 资源生成的集群角色

集群角色	设置
<code><kind>.<group>-<version>-admin</code>	<p><code><kind></code> 上的操作动词：</p> <ul style="list-style-type: none"> • * <p>聚合标签：</p> <ul style="list-style-type: none"> • <code>rbac.authorization.k8s.io/aggregate-to-admin: true</code> • <code>olm.opgroup.permissions/aggregate-to-admin: <operatorgroup_name></code>
<code><kind>.<group>-<version>-edit</code>	<p><code><kind></code> 上的操作动词：</p> <ul style="list-style-type: none"> • <code>create</code> • <code>update</code> • <code>patch</code> • <code>delete</code> <p>聚合标签：</p> <ul style="list-style-type: none"> • <code>rbac.authorization.k8s.io/aggregate-to-edit: true</code> • <code>olm.opgroup.permissions/aggregate-to-edit: <operatorgroup_name></code>

集群角色	设置
<code><kind>.<group>-<version>-view</code>	<p><code><kind></code> 上的操作动词：</p> <ul style="list-style-type: none"> ● <code>get</code> ● <code>list</code> ● <code>watch</code> <p>聚合标签：</p> <ul style="list-style-type: none"> ● <code>rbac.authorization.k8s.io/aggregate-to-view: true</code> ● <code>olm.opgroup.permissions/aggregate-to-view: <operatorgroup_name></code>
<code><kind>.<group>-<version>-view-crdview</code>	<p><code>apiextensions.k8s.io</code> <code>customresourcedefinitions <crd-name></code> 上的操作动词：</p> <ul style="list-style-type: none"> ● <code>get</code> <p>聚合标签：</p> <ul style="list-style-type: none"> ● <code>rbac.authorization.k8s.io/aggregate-to-view: true</code> ● <code>olm.opgroup.permissions/aggregate-to-view: <operatorgroup_name></code>

表 2.7. 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
<code><kind>.<group>-<version>-admin</code>	<p><code><kind></code> 上的操作动词：</p> <ul style="list-style-type: none"> ● <code>*</code> <p>聚合标签：</p> <ul style="list-style-type: none"> ● <code>rbac.authorization.k8s.io/aggregate-to-admin: true</code> ● <code>olm.opgroup.permissions/aggregate-to-admin: <operatorgroup_name></code>

集群角色	设置
<kind>.<group>-<version>-edit	<p><kind> 上的操作动词：</p> <ul style="list-style-type: none"> ● create ● update ● patch ● delete <p>聚合标签：</p> <ul style="list-style-type: none"> ● rbac.authorization.k8s.io/aggregate-to-edit: true ● olm.opgroup.permissions/aggregate-to-edit: <operatorgroup_name>
<kind>.<group>-<version>-view	<p><kind> 上的操作动词：</p> <ul style="list-style-type: none"> ● get ● list ● watch <p>聚合标签：</p> <ul style="list-style-type: none"> ● rbac.authorization.k8s.io/aggregate-to-view: true ● 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。

注意

默认情况下禁用 **disableCopiedCSVs** 字段。启用 **disableCopiedCSVs** 字段后，OLM 会删除集群中的现有复制的 CSV。当 **disableCopiedCSVs** 字段被禁用时，OLM 会再次添加复制的 CSV。

- 禁用 **disableCopiedCSVs** 字段：

```
$ cat << EOF | oc apply -f -
apiVersion: operators.coreos.com/v1
kind: OLMConfig
metadata:
  name: cluster
spec:
  features:
    disableCopiedCSVs: false
EOF
```

- 启用 **disableCopiedCSVs** 字段：

```
$ cat << EOF | oc apply -f -
apiVersion: operators.coreos.com/v1
kind: OLMConfig
metadata:
  name: cluster
spec:
  features:
    disableCopiedCSVs: true
EOF
```

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 支持在同一集群中安装不同版本的 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 是控制平面的扩展，因为它们使用了 **CustomResourceDefinition** 对象 (CRD)，它们是 Kubernetes 中的全局资源。一个 Operator 的不同主版本通常具有不兼容的 CRD。这使得它们不兼容，可以在集群中的不同命名空间中安装。

所有租户或命名空间共享同一集群的 control plane。因此，多租户集群中的租户也共享全局 CRD，这限制同一集群中可以并行使用同一 Operator 实例的不同 Operator 实例。

支持的场景包括：

- 提供相同 CRD 定义的不同版本的 Operator（如果版本化 CRD，则完全相同的版本）
- 没有提供 CRD 的不同版本的 Operator，并在 OperatorHub 上的单独捆绑包中提供它们的 CRD

不支持所有其他场景，因为如果不同 Operator 版本中的多个竞争或重叠 CRD 在同一集群中协调，则无法保证集群数据的完整性。

其他资源

- [Operator Lifecycle Manager \(OLM\)→ Multitenancy 和 Operator colocation](#)
- [多租户集群中的 Operator](#)
- [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

- 安装计划的命名空间必须只包含一个 Operator 组。当尝试在命名空间中生成集群服务版本 (CSV) 时，安装计划会认为一个 Operator 组在以下情况下无效：
 - 安装计划的命名空间中没有 Operator 组。
 - 安装计划的命名空间中存在多个 Operator 组。
 - 在 Operator 组中指定不正确或不存在的服务帐户名称。

如果安装计划遇到无效的 Operator 组，则不会生成 CSV，**InstallPlan** 资源将继续使用相关消息进行安装。例如，如果同一命名空间中存在多个 Operator 组，则会提供以下信息：

```
attenuated service account query failed - more than one operator group(s) are managing this namespace count=2
```

其中 **count=** 指定命名空间中的 Operator 组数量。

- 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. 多租户和 Operator 共处

本指南概述了 Operator Lifecycle Manager (OLM) 中的多租户和 Operator 共处。

2.4.6.1. 命名空间中的 Operator 共处

Operator Lifecycle Manager (OLM) 处理在同一命名空间中安装的 OLM 管理的 Operator，这意味着其 **Subscription** 资源与相关 Operator 位于同一个命名空间中。即使它们实际不相关，OLM 会在更新其中任何一个时考虑其状态，如它们的版本和更新策略。

这个默认行为清单可以通过两种方式：

- 待处理的更新的 **InstallPlan** 资源包括同一命名空间中的所有其他 Operator 的 **ClusterServiceVersion (CSV)** 资源。
- 同一命名空间中的所有 Operator 都共享相同的更新策略。例如，如果一个 Operator 设置为手动更新，则所有其他 Operator 更新策略也会设置为 manual。

这些场景可能会导致以下问题：

- 很难了解有关 Operator 更新安装计划的原因，因为它们中定义了除更新 Operator 以外的更多资源。
- 在命名空间更新中，无法自动更新一些 Operator，而其他 Operator 也无法手动更新，这对集群管理员来说很常见。

这些问题通常是在使用 OpenShift Container Platform Web 控制台安装 Operator 时，默认行为会将支持 **All namespaces** 安装模式的 Operator 安装到默认的 **openshift-operators** 全局命名空间中。

作为集群管理员，您可以使用以下工作流手动绕过此默认行为：

1. 为 Operator 的安装创建一个命名空间。
2. 创建自定义 *全局 Operator 组*，这是监视所有命名空间的 Operator 组。通过将此 Operator 组与您刚才创建的命名空间关联，从而使安装命名空间成为全局命名空间，从而使 Operator 在所有命名空间中都可用。
3. 在安装命名空间中安装所需的 Operator。

如果 Operator 具有依赖项，依赖项会在预先创建的命名空间中自动安装。因此，它对依赖项 Operator 有效，使其具有相同的更新策略和共享安装计划。具体步骤，请参阅“在自定义命名空间中安装全局 Operator”。

其他资源

- [在自定义命名空间中安装全局 Operator](#)
- [多租户集群中的 Operator](#)

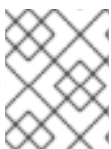
2.4.7. Operator 条件

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

2.4.7.1. 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 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 **Spec.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。



注意

默认情况下，**OperatorCondition** 对象中没有 **Spec.Conditions** 数组，直到由用户添加或使用自定义 Operator 逻辑的结果为止。

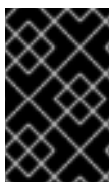
2.4.7.2. 支持的条件

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

2.4.7.2.1. 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。



重要

将 **Upgradeable** Operator 条件设置为 **False** 值不会避免 pod 中断。如果需要确保 pod 没有中断，请参阅“使用 pod 中断预算来指定必须在线的 pod 数量，以及“Additional resources”部分的“Graceful termination”。

Upgradeable Operator 条件

```
apiVersion: operators.coreos.com/v1
kind: OperatorCondition
metadata:
  name: my-operator
  namespace: operators
spec:
```

```

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。**False** 值不会阻止集群升级。

2.4.7.3. 其他资源

- [管理 Operator 条件](#)
- [启用 Operator 条件](#)
- [使用 pod 中断预算指定必须在线的 pod 数量](#)
- [恰当终止](#)

2.4.8. Operator Lifecycle Manager 指标数据

2.4.8.1. 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 2.8. OLM 公开的指标

名称	描述
<code>catalog_source_count</code>	目录源数量。
<code>catalogsource_ready</code>	目录源的状态。值 1 表示目录源处于 READY 状态。 0 表示目录源没有处于 READY 状态。
<code>csv_abnormal</code>	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。
<code>csv_count</code>	成功注册的 CSV 数量。
<code>csv_succeeded</code>	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。
<code>csv_upgrade_count</code>	CSV 升级的 Monotonic 计数。

名称	描述
install_plan_count	安装计划的数量。
installplan_warnings_total	由资源生成的警告数量（如已弃用资源）包含在安装计划中。
olm_resolution_duration_seconds	依赖项解析尝试的持续时间。
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。

2.4.9. 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 \(CSV\)](#)。

2.4.9.1. 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。
社区 Operator	由 redhat-openshift-ecosystem/community-operators-prod/operators GitHub 存储库中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](#)) 提供的认证工作流程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 workflow](#)
- [Red Hat Partner Connect](#)
- [Red Hat Marketplace](#)

2.6. 红帽提供的 OPERATOR 目录

红帽提供了一些默认包含在 OpenShift Container Platform 中的 Operator 目录。



重要

从 OpenShift Container Platform 4.11 开始，默认的红帽提供的 Operator 目录以基于文件的目录格式发布。通过以过时的 SQLite 数据库格式发布的 4.10，用于 OpenShift Container Platform 4.6 的默认红帽提供的 Operator 目录。

与 SQLite 数据库格式相关的 **opm** 子命令、标志和功能已被弃用，并将在以后的版本中删除。功能仍被支持，且必须用于使用已弃用的 SQLite 数据库格式的目录。

许多 **opm** 子命令和标志都用于 SQLite 数据库格式，如 **opm index prune**，它们无法使用基于文件的目录格式。有关使用基于文件的目录的更多信息，请参阅[管理自定义目录](#)、[Operator Framework 打包格式](#)，以及使用 **oc-mirror** 插件为断开连接的安装 **mirror** 镜像。

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。

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。

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

Kubernetes 定期弃用后续版本中删除的某些 API。因此，从使用删除 API 的 Kubernetes 版本的 OpenShift Container Platform 版本开始，Operator 无法使用删除 API 的 API。

如果您的集群使用自定义目录，请参阅[控制 Operator 与 OpenShift Container Platform 版本的兼容性](#)，以了解更多有关 Operator 作者如何更新其项目的详细信息，以帮助避免工作负载问题并防止不兼容的升级。



注意

OpenShift Container Platform 4.8 及之后的版本中删除了对 Operator 的传统软件包清单格式的支持，包括使用传统格式的自定义目录。

在创建自定义目录镜像时，在以前的 OpenShift Container Platform 4 版本中需要使用 **oc adm catalog build** 命令，这个命令已在多个版本中被弃用，现在已被删除。从 OpenShift Container Platform 4.6 开始，红帽提供的索引镜像可用后，目录构建器必须使用 **opm index** 命令来管理索引镜像。

其他资源

- [管理自定义目录](#)
- [打包格式](#)
- [在受限网络中使用 Operator Lifecycle Manager](#)

2.6.2. 关于红帽提供的 Operator 目录

在 **openshift-marketplace** 命名空间中默认安装红帽提供的目录源，从而使目录在所有命名空间中都可使用。

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:v4.15	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:v4.15	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:v4.15	可通过 Red Hat Marketplace 购买认证的软件。

目录	索引镜像	描述
community-operators	registry.redhat.io/redhat/community-operator-index:v4.15	由 redhat-openshift-ecosystem/community-operators-prod/operators GitHub 仓库中相关代表维护的软件。无官方支持。

在集群升级过程中，默认红帽提供的目录源的索引镜像标签由 Cluster Version Operator (CVO) 自动更新，以便 Operator Lifecycle Manager (OLM) 拉取目录的更新版本。例如，在从 OpenShift Container Platform 4.8 升级到 4.9 过程中，**redhat-operators** 目录的 **CatalogSource** 对象中的 **spec.image** 字段被更新：

```
registry.redhat.io/redhat/redhat-operator-index:v4.8
```

改为：

```
registry.redhat.io/redhat/redhat-operator-index:v4.9
```

2.7. 多租户集群中的 OPERATOR

Operator Lifecycle Manager (OLM) 的默认行为旨在简化 Operator 的安装过程。但是，此行为可能会缺少灵活性，特别是在多租户集群中。为了让 OpenShift Container Platform 集群上的多个租户使用 Operator，OLM 的默认行为要求管理员以 **All namespaces** 模式安装 Operator，这可能被视为违反最小特权的原则。

请考虑以下场景，以确定哪个 Operator 安装工作流最适合您的环境和要求。

其他资源

- [常见术语：多租户 \(Multitenant\)](#)
- [多租户 Operator 管理的限制](#)

2.7.1. 默认 Operator 安装模式和行为

当以管理员身份使用 Web 控制台安装 Operator 时，通常会根据 Operator 的功能，对安装模式有两个选择：

单个命名空间

在所选命名空间中安装 Operator，并发出 Operator 请求在该命名空间中提供的所有权限。

所有命名空间

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。进行所有命名空间中 Operator 请求的所有权限。在某些情况下，Operator 作者可以定义元数据，为用户授予该 Operator 建议的命名空间的第二个选项。

此选择还意味着受影响命名空间中的用户可以访问 Operator API，该 API 可以利用他们拥有的自定义资源 (CR)，具体取决于命名空间中的角色：

- **namespace-admin** 和 **namespace-edit** 角色可以对 Operator API 进行读/写，这意味着他们可以使用它们。
- **namespace-view** 角色可以读取该 Operator 的 CR 对象。

对于 **Single namespace** 模式，因为 Operator 本身安装在所选命名空间中，所以其 pod 和服务帐户也位于那里。对于 **All namespaces** 模式，Operator 的权限会自动提升到集群角色，这意味着 Operator 在所有命名空间中都有这些权限。

其他资源

- [在集群中添加 Operator](#)
- [安装模式类型](#)
- [设置建议的命名空间](#)

2.7.2. 多租户集群的建议解决方案

虽然 **Multinamespace** 安装模式存在，但只有少数 Operator 支持它。作为标准 **All namespaces** 和 **Single namespace** 安装模式之间的中间解决方案，您可以使用以下工作流安装同一 Operator 的多个实例，每个租户一个实例：

1. 为租户 Operator 创建命名空间，与租户的命名空间分开。
2. 为租户 Operator 创建 Operator 组，范围仅限于租户的命名空间。
3. 在租户 Operator 命名空间中安装 Operator。

因此，Operator 驻留在租户 Operator 命名空间中，并监视租户命名空间，但 Operator 的 pod 及其服务帐户都无法被租户可见或可用。

此解决方案以资源使用量成本提供更好的租户分离，以及确保满足约束的额外编配功能。如需详细步骤，请参阅“为多租户集群准备 Operator 的多个实例”。

限制和注意事项

只有在满足以下限制时，这个解决方案才可以正常工作：

- 同一 Operator 的所有实例都必须是相同的版本。
- Operator 无法依赖于其他 Operator。
- Operator 无法提供 CRD 转换 Webhook。



重要

您不能在同一集群中使用相同的 Operator 的不同版本。最后，当 Operator 的安装满足以下条件时，会阻断另一个 Operator 实例：

- 实例不是 Operator 的最新版本。
- 该实例提供了一个较老的 CRD 修订，它缺少新修订版本已在集群中使用的信息或版本。



警告

作为管理员，在允许非集群管理员自行安装 Operator 时请小心，如“允许非集群管理员安装 Operator”中所述。这些租户应只能访问已知没有依赖项的 Operator 策展目录。这些租户也必须强制使用 Operator 的同一版本行，以确保 CRD 不会改变。这需要命名空间范围的目录，并可能会禁用全局默认目录。

其他资源

- [为多租户集群准备多个 Operator 实例](#)
- [允许非集群管理员安装 Operator](#)
- [禁用默认的 OperatorHub 目录源](#)

2.7.3. Operator 共处和 Operator 组

Operator Lifecycle Manager (OLM) 处理在同一命名空间中安装的 OLM 管理的 Operator，这意味着其 **Subscription** 资源与相关 Operator 位于同一个命名空间中。即使它们实际不相关，OLM 会在更新其中任何一个时考虑其状态，如它们的版本和更新策略。

如需有关 Operator 共处和使用 Operator 组的更多信息，请参阅 [Operator Lifecycle Manager \(OLM\)→ Multitenancy](#) 和 [Operator colocation](#)。

2.8. CRD

2.8.1. 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。

2.8.1.1. 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。

2.8.1.2. 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 ❶
kind: CustomResourceDefinition
metadata:
  name: crontabs.stable.example.com ❷
spec:
  group: stable.example.com ❸
  versions:
    - name: v1 ❹
      served: true
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                cronSpec:
                  type: string
                image:
                  type: string
                replicas:
                  type: integer
  scope: Namespaced ❺
  names:
    plural: crontabs ❻
    singular: crontab ❼
    kind: CronTab ❽
    shortNames:
      - ct ❾
  
```

- 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。

2.8.1.3. 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到默认集群角色中。

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
```

2.8.1.4. 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" ❶
kind: CronTab ❷
metadata:
  name: my-new-cron-object ❸
  finalizers: ❹
  - finalizer.stable.example.com
spec: ❺
  cronSpec: "* * * * /5"
  image: my-awesome-cron-image
```

- ❶ 指定 CRD 中的组名称和 API 版本（名称/版本）。
- ❷ 指定 CRD 中的类型。
- ❸ 指定对象的名称。
- ❹ 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。
- ❺ 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
```

2.8.1.5. 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。

2.8.2. 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)。

2.8.2.1. 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。

2.8.2.2. 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" ❶
kind: CronTab ❷
metadata:
  name: my-new-cron-object ❸
  finalizers: ❹
  - finalizer.stable.example.com
spec: ❺
  cronSpec: "* * * * /5"
  image: my-awesome-cron-image
```

- ❶ 指定 CRD 中的组名称和 API 版本 (名称/版本)。
- ❷ 指定 CRD 中的类型。
- ❸ 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
```

2.8.2.3. 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 4.15 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. 在下一页中，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. 关于使用 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. 对于启用了令牌身份验证的云供应商上的集群：
 - 如果集群使用 AWS STS (Web 控制台中的**STS 模式**)，在 **role ARN** 字段中输入服务帐户的 AWS IAM 角色的 Amazon Resource Name (ARN)。

要创建角色的 ARN，请按照 [准备 AWS 帐户](#) 中所述的步骤进行操作。

- 如果集群使用 Microsoft Entra Workload ID (Web 控制台中的**Workload Identity / Federated Identity Mode**)，请在适当的项中添加客户端 ID、租户 ID 和订阅 ID。
- c. 如果有多个更新频道可用，请选择一个 **更新频道**。
 - d. 如前面所述，选择**自动或手动批准策略**。



重要

如果 web 控制台显示集群使用 AWS STS 或 Microsoft Entra Workload ID，您必须将 **Update approval** 设置为 **Manual**。

不建议使用具有自动更新批准的订阅，因为更新前可能会有权限更改。使用手动批准的订阅可确保管理员有机会验证更新版本的权限，并在更新前采取必要的操作。

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。
- 已安装 OpenShift CLI(**oc**)。

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。如果要安装的 Operator 使用 **AllNamespaces** 模式，**openshift-operators** 命名空间已有适当的 **global-operators** 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。
- 每个命名空间只能有一个 Operator 组。如需更多信息，请参阅 "Operator 组"。

- 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** 对于默认的 **AllNamespaces** 安装模式用法，请指定 **openshift-operators** 命名空间。另外，如果创建了自定义全局命名空间，您可以指定一个自定义全局命名空间。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 中必须存在的卷挂载列表。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. 对于启用了令牌身份验证的云供应商上的集群：

a. 确保 **Subscription** 对象被设置为手动更新批准：

```
kind: Subscription
# ...
spec:
  installPlanApproval: Manual 1
```

1 不建议使用具有自动更新批准的订阅，因为更新前可能会有权限更改。使用手动批准的订阅可确保管理员有机会验证更新版本的权限，并在更新前采取必要的操作。

b. 在 **Subscription** 对象的 **config** 部分包括相关的云供应商相关字段：

- 如果集群处于 AWS STS 模式，请包含以下字段：

```
kind: Subscription
# ...
spec:
  config:
    env:
      - name: ROLEARN
        value: "<role_arn>" 1
```

1 包含角色 ARN 详情。

- 如果集群处于 Microsoft Entra Workload ID 模式，请包括以下字段：

```
kind: Subscription
# ...
spec:
  config:
    env:
      - name: CLIENTID
        value: "<client_id>" 1
      - name: TENANTID
        value: "<tenant_id>" 2
      - name: SUBSCRIPTIONID
        value: "<subscription_id>" 3
```

1 包含客户端 ID。

2 包含租户 ID。

3 包括订阅 ID。

6. 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

- [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. 运行以下命令，查找您要安装的 Operator 的可用版本和频道：

命令语法

```
$ oc describe packagemanifests <operator_name> -n <catalog_namespace>
```

例如，以下命令从 OperatorHub 打印 Red Hat Quay Operator 可用频道和版本：

示例命令

```
$ oc describe packagemanifests quay-operator -n openshift-marketplace
```

例 3.1. 输出示例

```
Name:      quay-operator
Namespace: operator-marketplace
Labels:    catalog=redhat-operators
           catalog-namespace=openshift-marketplace
           hypershift.openshift.io/managed=true
           operatorframework.io/arch.amd64=supported
           operatorframework.io/os.linux=supported
           provider=Red Hat
           provider-url=
Annotations: <none>
API Version: packages.operators.coreos.com/v1
Kind:      PackageManifest
...
Current CSV: quay-operator.v3.7.11
...
Entries:
```

```

Name: quay-operator.v3.7.11
Version: 3.7.11
Name: quay-operator.v3.7.10
Version: 3.7.10
Name: quay-operator.v3.7.9
Version: 3.7.9
Name: quay-operator.v3.7.8
Version: 3.7.8
Name: quay-operator.v3.7.7
Version: 3.7.7
Name: quay-operator.v3.7.6
Version: 3.7.6
Name: quay-operator.v3.7.5
Version: 3.7.5
Name: quay-operator.v3.7.4
Version: 3.7.4
Name: quay-operator.v3.7.3
Version: 3.7.3
Name: quay-operator.v3.7.2
Version: 3.7.2
Name: quay-operator.v3.7.1
Version: 3.7.1
Name: quay-operator.v3.7.0
Version: 3.7.0
Name: stable-3.7
...
Current CSV: quay-operator.v3.8.5
...
Entries:
Name: quay-operator.v3.8.5
Version: 3.8.5
Name: quay-operator.v3.8.4
Version: 3.8.4
Name: quay-operator.v3.8.3
Version: 3.8.3
Name: quay-operator.v3.8.2
Version: 3.8.2
Name: quay-operator.v3.8.1
Version: 3.8.1
Name: quay-operator.v3.8.0
Version: 3.8.0
Name: stable-3.8
Default Channel: stable-3.8
Package Name: quay-operator

```

提示

您可以运行以下命令来以 YAML 格式输出 Operator 的版本和频道信息：

```
$ oc get packagemanifests <operator_name> -n <catalog_namespace> -o yaml
```

- 如果在命名空间中安装多个目录，请运行以下命令从特定目录中查找 Operator 的可用版本和频道：

```
$ oc get packagemanifest \
  --selector=catalog=<catalogsource_name> \
  --field-selector metadata.name=<operator_name> \
  -n <catalog_namespace> -o yaml
```

重要

如果没有指定 Operator 的目录，运行 **oc get packagemanifest** 和 **oc describe packagemanifest** 命令可能会在满足以下条件时从一个意料外的目录中返回一个软件包：

- 在同一命名空间中安装多个目录。
- 目录包含具有相同名称的相同 Operator 或 Operator。

2. 由 **OperatorGroup** 对象定义的 Operator 组选择目标命名空间，在其中为与 Operator 组相同的命名空间中的所有 Operator 生成所需的基于角色的访问控制 (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** 模式安装的 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：

- 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
```

3. 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 3.7.10：

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: stable-3.7  
  installPlanApproval: Manual ❶  
  name: quay-operator  
  source: redhat-operators  
  sourceNamespace: openshift-marketplace  
  startingCSV: quay-operator.v3.7.10 ❷
```

- ❶ 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。
- ❷ 设置 Operator CSV 的特定版本。

4. 创建 **Subscription** 对象：

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

5. 手动批准待处理的安装计划以完成 Operator 安装。

其他资源

- [手动批准待处理的 Operator 更新](#)

第 4 章 管理员任务

4.1. 在集群中添加 OPERATOR

通过使用 Operator Lifecycle Manager (OLM)，集群管理员可将基于 OLM 的 Operator 安装到 OpenShift Container Platform 集群。



注意

如需有关 OLM 如何处理在同一命名空间中并置安装的 Operator 的更新，以及使用自定义全局 Operator 组安装 Operator 的替代方法，请参阅[多租户和 Operator 共处](#)。

4.1.1. 关于使用 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。

其他资源

- [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. 对于启用了令牌身份验证的云供应商上的集群：
 - 如果集群使用 AWS STS (Web 控制台中的**STS 模式**)，在 **role ARN** 字段中输入服务帐户的 AWS IAM 角色的 Amazon Resource Name (ARN)。

要创建角色的 ARN，请按照 [准备 AWS 帐户](#) 中所述的步骤进行操作。

- 如果集群使用 Microsoft Entra Workload ID (Web 控制台中的**Workload Identity / Federated Identity Mode**)，请在适当的项中添加客户端 ID、租户 ID 和订阅 ID。
- d. 如果有多个更新频道可用，请选择一个 **更新频道**。
 - e. 如前面所述，选择**自动或手动批准策略**。



重要

如果 web 控制台显示集群使用 AWS STS 或 Microsoft Entra Workload ID，您必须将 **Update approval** 设置为 **Manual**。

不建议使用具有自动更新批准的订阅，因为更新前可能会有权限更改。使用手动批准的订阅可确保管理员有机会验证更新版本的权限，并在更新前采取必要的操作。

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。
- 已安装 OpenShift CLI(**oc**)。

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。如果要安装的 Operator 使用 **AllNamespaces** 模式，**openshift-operators** 命名空间已有适当的 **global-operators** 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。
- 每个命名空间只能有一个 Operator 组。如需更多信息，请参阅 "Operator 组"。

- 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** 对于默认的 **AllNamespaces** 安装模式用法，请指定 **openshift-operators** 命名空间。另外，如果创建了自定义全局命名空间，您可以指定一个自定义全局命名空间。否则，为 **SingleNamespaces** 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 中必须存在的卷挂载列表。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. 对于启用了令牌身份验证的云供应商上的集群：

- a. 确保 **Subscription** 对象被设置为手动更新批准：

```
kind: Subscription
# ...
spec:
  installPlanApproval: Manual 1
```

- 1 不建议使用具有自动更新批准的订阅，因为更新前可能会有权限更改。使用手动批准的订阅可确保管理员有机会验证更新版本的权限，并在更新前采取必要的操作。

- b. 在 **Subscription** 对象的 **config** 部分包括相关的云供应商相关字段：

- 如果集群处于 AWS STS 模式，请包含以下字段：

```
kind: Subscription
# ...
spec:
  config:
    env:
      - name: ROLEARN
        value: "<role_arn>" 1
```

- 1 包含角色 ARN 详情。

- 如果集群处于 Microsoft Entra Workload ID 模式，请包括以下字段：

```
kind: Subscription
# ...
spec:
  config:
    env:
      - name: CLIENTID
        value: "<client_id>" 1
      - name: TENANTID
        value: "<tenant_id>" 2
      - name: SUBSCRIPTIONID
        value: "<subscription_id>" 3
```

- 1 包含客户端 ID。

2 包含租户 ID。

3 包括订阅 ID。

6. 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. 运行以下命令，查找您要安装的 Operator 的可用版本和频道：

命令语法

```
$ oc describe packagemanifests <operator_name> -n <catalog_namespace>
```

例如，以下命令从 OperatorHub 打印 Red Hat Quay Operator 可用频道和版本：

示例命令

```
$ oc describe packagemanifests quay-operator -n openshift-marketplace
```

例 4.1. 输出示例

```
Name:      quay-operator
Namespace: operator-marketplace
Labels:    catalog=redhat-operators
           catalog-namespace=openshift-marketplace
           hypershift.openshift.io/managed=true
           operatorframework.io/arch.amd64=supported
           operatorframework.io/os.linux=supported
           provider=Red Hat
           provider-url=
Annotations: <none>
API Version: packages.operators.coreos.com/v1
```

```
Kind:      PackageManifest
...
Current CSV: quay-operator.v3.7.11
...
Entries:
  Name:     quay-operator.v3.7.11
  Version:  3.7.11
  Name:     quay-operator.v3.7.10
  Version:  3.7.10
  Name:     quay-operator.v3.7.9
  Version:  3.7.9
  Name:     quay-operator.v3.7.8
  Version:  3.7.8
  Name:     quay-operator.v3.7.7
  Version:  3.7.7
  Name:     quay-operator.v3.7.6
  Version:  3.7.6
  Name:     quay-operator.v3.7.5
  Version:  3.7.5
  Name:     quay-operator.v3.7.4
  Version:  3.7.4
  Name:     quay-operator.v3.7.3
  Version:  3.7.3
  Name:     quay-operator.v3.7.2
  Version:  3.7.2
  Name:     quay-operator.v3.7.1
  Version:  3.7.1
  Name:     quay-operator.v3.7.0
  Version:  3.7.0
  Name:     stable-3.7
...
Current CSV: quay-operator.v3.8.5
...
Entries:
  Name:     quay-operator.v3.8.5
  Version:  3.8.5
  Name:     quay-operator.v3.8.4
  Version:  3.8.4
  Name:     quay-operator.v3.8.3
  Version:  3.8.3
  Name:     quay-operator.v3.8.2
  Version:  3.8.2
  Name:     quay-operator.v3.8.1
  Version:  3.8.1
  Name:     quay-operator.v3.8.0
  Version:  3.8.0
  Name:     stable-3.8
Default Channel: stable-3.8
Package Name:  quay-operator
```

提示

您可以运行以下命令来以 YAML 格式输出 Operator 的版本和频道信息：

```
$ oc get packagemanifests <operator_name> -n <catalog_namespace> -o yaml
```

- 如果在命名空间中安装多个目录，请运行以下命令从特定目录中查找 Operator 的可用版本和频道：

```
$ oc get packagemanifest \
  --selector=catalog=<catalogsource_name> \
  --field-selector metadata.name=<operator_name> \
  -n <catalog_namespace> -o yaml
```



重要

如果没有指定 Operator 的目录，运行 **oc get packagemanifest** 和 **oc describe packagemanifest** 命令可能会在满足以下条件时从一个意料外的目录中返回一个软件包：

- 在同一命名空间中安装多个目录。
- 目录包含具有相同名称的相同 Operator 或 Operator。

2. 由 **OperatorGroup** 对象定义的 Operator 组选择目标命名空间，在其中为与 Operator 组相同的命名空间中的所有 Operator 生成所需的基于角色的访问控制 (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** 模式安装的 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：

- 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
```

3. 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 3.7.10：

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: stable-3.7
  installPlanApproval: Manual 1
  name: quay-operator
  source: redhat-operators
  sourceNamespace: openshift-marketplace
  startingCSV: quay-operator.v3.7.10 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。

4. 创建 **Subscription** 对象：

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

5. 手动批准待处理的安装计划以完成 Operator 安装。

其他资源

- [手动批准待处理的 Operator 更新](#)
- [在自定义命名空间中安装全局 Operator](#)

4.1.5. 在 web 控制台中安装 Operator 的特定版本

您可以使用 Web 控制台中的 OperatorHub 安装 Operator 的特定版本。您可以在其可能具有的任何频道中浏览 Operator 的各种版本，查看该频道和版本的元数据，然后选择您要安装的确切版本。

先决条件

- 您必须具有管理员特权。

流程

1. 在 Web 控制台中，点 **Operators → OperatorHub**。
2. 选择您要安装的 Operator。
3. 在所选的 Operator 中，您可以从列表中选择一个 **Channel** 和 **Version**。



注意

版本选择默认为所选频道的最新版本。如果选择了该频道的最新版本，则默认启用自动批准策略。如果没有为所选频道安装最新版本，则需要手动批准。

手动批准适用于命名空间中安装的所有 Operator。

使用手动批准安装 Operator 会导致命名空间中安装的所有 Operator 并使用 Manual 批准策略和所有 Operator 一起更新。将 Operator 安装到单独的命名空间中，以独立更新。

4. 点 Install

验证

- 安装 Operator 时，元数据会指示安装了哪个频道和版本。



注意

频道和版本下拉菜单仍可查看此目录上下文中的其他版本元数据。

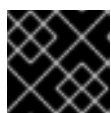
4.1.6. 为多租户集群准备多个 Operator 实例

作为集群管理员，您可以添加多个 Operator 实例以用于多租户集群。这是使用标准 **All namespaces** 安装模式的替代解决方案，可被视为违反最小特权的原则，或 **Multinamespace** 模式（没有被广泛采用）。如需更多信息，请参阅“多租户集群中的 Operator”。

在以下步骤中，*租户* 是为一组部署的工作负载共享通用访问和特权的用户或组。*租户 Operator* 是 Operator 实例，仅用于由该租户使用。

先决条件

- 您要安装的 Operator 的所有实例都必须在给定集群中相同。



重要

有关这个限制和其他限制的更多信息，请参阅“多租户集群中的 Operator”。

流程

1. 在安装 Operator 前，为租户 Operator 创建一个命名空间，该 Operator 与租户的命名空间分开。例如，如果租户的命名空间是 **team1**，您可以创建一个 **team1-operator** 命名空间：
 - a. 定义 **Namespace** 资源并保存 YAML 文件，如 **team1-operator.yaml**：

```
apiVersion: v1
kind: Namespace
metadata:
  name: team1-operator
```

- b. 运行以下命令创建命名空间：

```
$ oc create -f team1-operator.yaml
```


2. 为租户 Operator 创建 Operator 组，范围到租户的命名空间，只有 `spec.targetNamespaces` 列表中有一个命名空间条目：
 - a. 定义 **OperatorGroup** 资源并保存 YAML 文件，如 `team1-operatorgroup.yaml`：

```

apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: team1-operatorgroup
  namespace: team1-operator
spec:
  targetNamespaces:
    - team1 1

```

- 1 仅在 `spec.targetNamespaces` 列表中定义租户的命名空间。

- b. 运行以下命令来创建 Operator 组：

```
$ oc create -f team1-operatorgroup.yaml
```

后续步骤

- 在租户 Operator 命名空间中安装 Operator。通过在 Web 控制台使用 OperatorHub 而不是 CLI，可以更轻松地执行此任务；请参阅 [使用 Web 控制台从 OperatorHub 安装](#)。



注意

完成 Operator 安装后，Operator 驻留在租户 Operator 命名空间中，并监视租户命名空间，但 Operator 的 pod 及其服务帐户都无法被租户可见或可用。

其他资源

- [多租户集群中的 Operator](#)

4.1.7. 在自定义命名空间中安装全局 Operator

在使用 OpenShift Container Platform Web 控制台安装 Operator 时，默认行为会将支持 **All namespaces** 安装模式的 Operator 安装到默认的 **openshift-operators** 全局命名空间中。这可能导致与命名空间中所有 Operator 共享安装计划和更新策略相关的问题。有关这些限制的详情，请参阅 "Multitenancy 和 Operator colocation"。

作为集群管理员，您可以通过创建自定义全局命名空间并使用该命名空间安装单个或范围 Operator 及其依赖项来手动绕过此默认行为。

先决条件

- 您可以使用具有 **cluster-admin** 角色的用户访问集群。

流程

1. 在安装 Operator 前，为所需 Operator 安装创建一个命名空间。此安装命名空间将成为自定义全局命名空间：

- a. 定义 **Namespace** 资源并保存 YAML 文件，如 **global-operators.yaml**：

```
apiVersion: v1
kind: Namespace
metadata:
  name: global-operators
```

- b. 运行以下命令创建命名空间：

```
$ oc create -f global-operators.yaml
```

2. 创建自定义 *全局 Operator 组*，这是监视所有命名空间的 Operator 组：

- a. 定义 **OperatorGroup** 资源并保存 YAML 文件，如 **global-operatorgroup.yaml**。省略 **spec.selector** 和 **spec.targetNamespaces** 字段，使其成为一个 *全局 Operator 组*，该组选择所有命名空间：

```
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: global-operatorgroup
  namespace: global-operators
```



注意

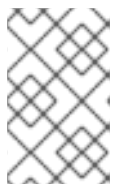
创建的全局 OperatorGroup 的 **status.namespace** 包含空字符串 ("")，而该字符串会向正在使用的 Operator 发出信号，要求其监视所有命名空间。

- b. 运行以下命令来创建 Operator 组：

```
$ oc create -f global-operatorgroup.yaml
```

后续步骤

- 在自定义全局命名空间中安装所需的 Operator。因为 Web 控制台没有在 Operator 安装过程中使用自定义全局命名空间填充 **Installed Namespace** 菜单，所以此任务只能使用 OpenShift CLI (**oc**) 执行。具体步骤请参阅[使用 CLI 从 OperatorHub 安装](#)。



注意

启动 Operator 安装时，如果 Operator 有依赖项，依赖项也会自动安装到自定义全局命名空间中。因此，它对依赖项 Operator 有效，使其具有相同的更新策略和共享安装计划。

其他资源

- [多租户和 Operator 共处](#)

4.1.8. 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，使用计算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，通过使用一个计算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，如以下部分所述。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。

其他资源

- [手动向节点或计算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](#)

4.1.9. 控制安装 Operator 的位置

默认情况下，当安装 Operator 时，OpenShift Container Platform 会随机将 Operator pod 安装到其中一个 worker 节点。然而，在某些情况下，您可能希望该 pod 调度到特定节点或一组节点上。

以下示例描述了您可能希望将 Operator pod 调度到特定节点或一组节点的情况：

- 如果 Operator 需要特定的平台，如 **amd64** 或 **arm64**
- 如果 Operator 需要特定的操作系统，如 Linux 或 Windows
- 如果您希望 Operator 在同一个主机上或位于同一机架的主机上工作
- 如果您希望 Operator 在整个基础架构中分散，以避免因为网络或硬件问题而停机

您可以通过在 Operator 的 **Subscription** 对象中添加节点关联性、pod 关联性或 pod 反关联性限制来控制 Operator pod 的安装位置。节点关联性是由调度程序用来确定 pod 的可放置位置的一组规则。pod 关联性允许您确保将相关的 pod 调度到同一节点。通过 Pod 反关联性，您可以防止 pod 调度到节点上。

以下示例演示了如何使用节点关联性或 pod 反关联性将自定义 Metrics Autoscaler Operator 实例安装到集群中的特定节点：

将 Operator pod 放置到特定节点的节点关联性示例

```

apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: openshift-custom-metrics-autoscaler-operator
  namespace: openshift-keda
spec:
  name: my-package
  source: my-operators
  sourceNamespace: operator-registries
  config:
    affinity:
      nodeAffinity: ❶
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: kubernetes.io/hostname
                operator: In
                values:
                  - ip-10-0-163-94.us-west-2.compute.internal
#...

```

- ❶ 要求 Operator 的 pod 调度到名为 **ip-10-0-163-94.us-west-2.compute.internal** 的节点关联性。

将 Operator pod 放置到带有特定平台的节点关联性示例

```

apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: openshift-custom-metrics-autoscaler-operator
  namespace: openshift-keda
spec:
  name: my-package
  source: my-operators
  sourceNamespace: operator-registries
  config:
    affinity:
      nodeAffinity: ❶
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: kubernetes.io/arch
                operator: In
                values:
                  - arm64
              - key: kubernetes.io/os
                operator: In
                values:
                  - linux
#...

```

- ❶ 要求 Operator 的 pod 调度到具有 **kubernetes.io/arch=arm64** 和 **kubernetes.io/os=linux** 标签的节点上。

将 Operator pod 放置到一个或多个特定节点的 Pod 关联性示例

```

apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: openshift-custom-metrics-autoscaler-operator
  namespace: openshift-keda
spec:
  name: my-package
  source: my-operators
  sourceNamespace: operator-registries
  config:
    affinity:
      podAffinity: ❶
        requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
                - key: app
                  operator: In
                  values:
                    - test
            topologyKey: kubernetes.io/hostname
#...
```

- ❶ 将 Operator 的 pod 放置到具有 **app=test** 标签的 pod 的节点上的 pod 关联性。

防止 Operator pod 来自一个或多个特定节点的 Pod 反关联性示例

```

apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: openshift-custom-metrics-autoscaler-operator
  namespace: openshift-keda
spec:
  name: my-package
  source: my-operators
  sourceNamespace: operator-registries
  config:
    affinity:
      podAntiAffinity: ❶
        requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
                - key: cpu
                  operator: In
                  values:
                    - high
            topologyKey: kubernetes.io/hostname
#...
```

- ❶ 一个 pod 反关联性，它可防止 Operator 的 pod 调度到具有 **cpu=high** 标签的 pod 的节点上。

流程

要控制 Operator pod 的放置，请完成以下步骤：

1. 照常安装 Operator。
2. 如果需要，请确保您的节点已标记为正确响应关联性。
3. 编辑 Operator **Subscription** 对象以添加关联性：

```

apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: openshift-custom-metrics-autoscaler-operator
  namespace: openshift-keda
spec:
  name: my-package
  source: my-operators
  sourceNamespace: operator-registries
  config:
    affinity: ❶
      nodeAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          nodeSelectorTerms:
            - matchExpressions:
                - key: kubernetes.io/hostname
                  operator: In
                  values:
                    - ip-10-0-185-229.ec2.internal
#...
```

- ❶ 添加 **nodeAffinity**、**podAffinity** 或 **podAntiAffinity**。有关创建关联性的详情，请参考下面的附加资源部分。

验证

- 要确保 pod 部署到特定的节点上，请运行以下命令：

```
$ oc get pods -o wide
```

输出示例

```

NAME                                READY STATUS  RESTARTS  AGE  IP
NODE                                NOMINATED NODE  READINESS GATES
custom-metrics-autoscaler-operator-5dcc45d656-bhshg  1/1   Running  0        50s
10.131.0.20 ip-10-0-185-229.ec2.internal <none>    <none>
```

其他资源

- [了解 pod 关联性](#)
- [了解节点关联性](#)
- [了解如何更新节点上的标签](#)

4.2. 更新安装的 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。



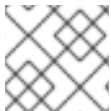
注意

如需有关 OLM 如何处理在同一命名空间中并置安装的 Operator 的更新，以及使用自定义全局 Operator 组安装 Operator 的替代方法，请参阅[多租户和 Operator 共处](#)。

4.2.1. 准备 Operator 更新

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。您可以更改更新频道，以开始跟踪并从更新频道接收更新。

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

您不能将已安装的 Operator 更改为比当前频道旧的频道。

红帽客户门户网站 Labs 包括以下应用程序，可帮助管理员准备更新其 Operator：

- [Red Hat OpenShift Container Platform Operator Update Information Checker](#)

您可以使用应用程序搜索基于 Operator Lifecycle Manager 的 Operator，并在不同版本的 OpenShift Container Platform 中验证每个更新频道的可用 Operator 版本。不包含基于 Cluster Version Operator 的 Operator。

4.2.2. 更改 Operator 的更新频道

您可以使用 OpenShift Container Platform Web 控制台更改 Operator 的更新频道。

提示

如果订阅中的批准策略被设置为 **Automatic**，则更新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更新。

先决条件

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

流程

1. 在 web 控制台的 **Administrator** 视角中，导航到 **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 **Update channel** 下的更新频道名称。
5. 点要更改的更新频道，然后点 **Save**。

- 对于带有 **自动批准策略** 的订阅，更新会自动开始。返回到 **Operators → Installed Operators** 页面，以监控更新的进度。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，您可以从 **Subscription** 选项卡中手动批准更新。

4.2.3. 手动批准待处理的 Operator 更新

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

- 在 OpenShift Container Platform Web 控制台的 **Administrator** 视角中，进入 **Operators → Installed Operators**。
- 处于待更新 Operator 会显示 **Upgrade available** 状态。点您要更新的 Operator 的名称。
- 点 **Subscription** 标签页。任何需要批准的更新都会在 **Upgrade status** 旁边显示。例如：它可能会显示 **1 requires approval**。
- 点 **1 requires approval**，然后点 **Preview Install Plan**。
- 检查列出可用于更新的资源。在满意后，点 **Approve**。
- 返回到 **Operators → Installed Operators** 页面，以监控更新的进度。完成后，状态会变为 **Succeeded** 和 **Up to date**。

4.2.4. 其他资源

- [在受限网络中使用 Operator Lifecycle Manager](#)

4.3. 从集群中删除 OPERATOR

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



重要

在尝试重新安装同一 Operator 前，您必须已成功并完全卸载了 Operator。没有正确地完全卸载 Operator 可能会留下一些资源，如项目或命名空间，处于 "Terminating" 状态，并导致尝试重新安装 Operator 时观察到 "error resolving resource" 消息。

如需更多信息，请参阅[卸载失败后重新安装 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

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

流程

1. 进入到 **Operators → Installed Operators** 页面。
2. 在 **Filter by name** 字段中滚动或输入关键字以查找您要删除的 Operator。然后点它。
3. 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。
4. 选择 **Uninstall** 来删除 Operator、Operator 部署和 pod。按照此操作，Operator 将停止运行，不再接收更新。



注意

此操作不会删除 Operator 管理的资源，包括自定义资源定义 (CRD) 和自定义资源 (CR)。Web 控制台和继续运行的集群资源启用的仪表板和导航项可能需要手动清理。要在卸载 Operator 后删除这些，您可能需要手动删除 Operator CRD。

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

- 可以使用具有 **cluster-admin** 权限的账户访问 OpenShift Container Platform 集群。
- OpenShift CLI (**oc**) 安装在您的工作站上。

流程

1. 确保在 **currentCSV** 字段中标识了订阅 Operator 的最新版本（如 **serverless-operator**）。

```
$ oc get subscription.operators.coreos.com serverless-operator -n openshift-serverless -o yaml | grep currentCSV
```

输出示例

```
currentCSV: serverless-operator.v1.28.0
```

2. 删除订阅（如 **serverless-operator**）：

```
$ oc delete subscription.operators.coreos.com serverless-operator -n openshift-serverless
```

输出示例

```
subscription.operators.coreos.com "serverless-operator" 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 serverless-operator.v1.28.0 -n openshift-serverless
```

输出示例

```
clusterserviceversion.operators.coreos.com "serverless-operator.v1.28.0" 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. 配置 OPERATOR LIFECYCLE MANAGER 功能

Operator Lifecycle Manager(OLM)控制器由名为 **cluster** 的 **OLMConfig** 自定义资源(CR)进行配置。集群管理员可以修改此资源以启用或禁用某些功能。

本文档概述了由 **OLMConfig** 资源配置的 OLM 当前支持的功能。

4.4.1. 禁用复制的 CSV

当 Operator Lifecycle Manager (OLM) 安装 Operator 时，默认情况下会在 Operator 配置为监视的每个命名空间中创建其集群服务版本 (CSV) 的简化副本。这些 CSV 称为 *复制的 CSV*，并告知用户控制器在给定的命名空间中主动协调资源事件。

当 Operator 配置为使用 **AllNamespaces** 安装模式时，与将单个或指定命名空间集为目标时，会在集群中的每个命名空间中创建 Operator 的复制的 CSV。在大型集群中，带有命名空间和安装的 Operator 可能存在于数百个或数千种 CSV 中，复制的 CSV 消耗大量资源，如 OLM 的内存用量、集群 etcd 限值和

网络带宽。

为了支持这些较大的集群，集群管理员可以为使用 **AllNamespaces** 模式全局安装的 Operator 禁用复制的 CSV。



注意

如果您禁用复制的 CSV，则在 **AllNamespaces** 模式中安装的 Operator 只将其 CSV 复制到 **openshift** 命名空间，而不是集群中的每个命名空间。在禁用复制的 CSV 模式中，Web 控制台和 CLI 的行为有所不同：

- 在 web 控制台中，默认行为会被修改为显示每个命名空间中的 **openshift** 命名空间中复制的 CSV，即使 CSV 不会实际复制到每个命名空间中。这允许常规用户仍可在其命名空间中查看这些 Operator 的详情，并创建相关的自定义资源 (CR)。
- 在 OpenShift CLI (**oc**) 中，常规用户可以使用 **oc get csvs** 命令查看直接安装在其命名空间中的 Operator，但 **openshift** 命名空间中的复制的 CSV 无法在其命名空间中可见。受此限制影响的 Operator 仍然可用，并继续协调用户命名空间中的事件。
要查看安装的全局 Operator 的完整列表，类似于 Web 控制台行为，所有经过身份验证的用户都可以运行以下命令：

```
$ oc get csvs -n openshift
```

流程

- 编辑名为 **cluster** 的 **OLMConfig** 对象，将 **spec.features.disableCopiedCSVs** 字段设置为 **true**：

```
$ oc apply -f - <<EOF
apiVersion: operators.coreos.com/v1
kind: OLMConfig
metadata:
  name: cluster
spec:
  features:
    disableCopiedCSVs: true ①
EOF
```

- ① 为 **AllNamespaces** 安装模式 Operator 禁用复制的 CSV

验证

- 当禁用复制的 CSV 时，OLM 会在 Operator 命名空间中捕获这些信息：

```
$ oc get events
```

输出示例

```
LAST SEEN   TYPE      REASON          OBJECT                                     MESSAGE
85s        Warning   DisabledCopiedCSVs  clusterserviceversion/my-csv.v1.0.0  CSV
copying disabled for operators/my-csv.v1.0.0
```

当 `spec.features.disableCopiedCSVs` 字段缺失或设置为 `false` 时，OLM 会为使用 `AllNamespaces` 模式安装的所有 Operator 重新创建复制的 CSV，并删除前面提到的事件。

其他资源

- [安装模式](#)

4.5. 在 OPERATOR LIFECYCLE MANAGER 中配置代理支持

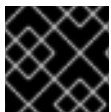
如果在 OpenShift Container Platform 集群中配置了全局代理，Operator Lifecycle Manager (OLM) 会自动配置使用集群范围代理管理的 Operator。但是，您也可以配置已安装的 Operator 来覆盖全局代理服务或注入自定义 CA 证书。

其他资源

- [配置集群范围代理](#)
- [配置自定义 PKI](#)（自定义 CA 证书）
- 开发支持 [Go](#)、[Ansible](#) 和 [Helm](#) 的代理设置的 Operator

4.5.1. 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，则所有三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
...

```

4.5.2. 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，这将会很有用。

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。



注意

Operator 的部署可能无法验证颁发机构，并显示 **x509 certificate signed by unknown authority** 错误。即使在使用 Operator 订阅时注入自定义 CA，也会发生这个错误。在这种情况下，您可以使用 Operator 的订阅将 **mountPath** 设置为 trusted-ca 的 **/etc/ssl/certs**。

4.6. 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。

4.6.1. 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。
ResolutionFailed	订阅的依赖项解析失败。



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。

其他资源

- [刷新失败的订阅](#)

4.6.2. 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

```
Name:      cluster-logging
Namespace: openshift-logging
Labels:    operators.coreos.com/cluster-logging.openshift-logging=
Annotations: <none>
API Version: operators.coreos.com/v1alpha1
Kind:      Subscription
# ...
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。

4.6.3. 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
Labels:    <none>
Annotations: operatorframework.io/managed-by: marketplace-operator
            target.workload.openshift.io/management: {"effect": "PreferredDuringScheduling"}
API Version: operators.coreos.com/v1alpha1
Kind:      CatalogSource
# ...
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。

- 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-vsxd/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-vsxd
  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。

其他资源

- [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](#)

4.7. 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。

4.7.1. 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。当存在时，**Spec.Overrides** 阵列中的 Operator 条件会覆盖 **Spec.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。



注意

默认情况下，**OperatorCondition** 对象中不存在 **Spec.Overrides** 数组，直到集群管理员添加为止。**Spec.Conditions** 数组还不存在，直到被用户添加或因为自定义 Operator 逻辑而添加为止。

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** 对象的 **Spec.Overrides** 阵列中添加 **type** 和 **status** 来覆盖 Operator 条件来实现。

先决条件

- 您可以使用具有 **cluster-admin** 角色的用户访问集群。
- 具有 **OperatorCondition** 对象的 Operator，使用 OLM 安装。

流程

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

```
$ 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: "upgradelsSafe"
    message: "This is a known issue with the Operator where it always reports that it cannot be upgraded."
  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**。

4.7.2. 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。有关将 Operator 条件设置为 Operator 作者的更多信息，请参阅[启用 Operator 条件](#)页面。

4.7.2.1. 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。

4.7.3. 其他资源

- [Operator 条件](#)

4.8. 允许非集群管理员安装 OPERATOR

集群管理员可以使用 *Operator 组* 来允许常规用户安装 Operator。

其他资源

- [operator 组](#)

4.8.1. 了解 Operator 安装策略

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。

为确保 Operator 无法获得集群范围的权限，并且用户无法使用 OLM 升级权限，集群管理员可在将 Operator 添加到集群前手动审核 Operator。集群管理员还可获得一些工具来决定和限制在使用服务帐户安装或升级 Operator 期间允许的操作。

集群管理员可以将 Operator 组与赋予了一组权限的服务帐户关联。服务帐户在 Operator 上设置策略，通过使用基于角色的访问控制 (RBAC) 规则来确保它们仅在预先确定的边界内运行。因此，Operator 无法执行这些规则未明确允许的任何操作。

通过使用 Operator 组，具有足够权限的用户可以安装具有有限范围的 Operator。因此，更多 Operator Framework 工具可以安全地提供给更多用户，为使用 Operator 构建应用程序提供更丰富的体验。



注意

Subscription 对象的基于角色的访问控制 (RBAC) 会自动授予命名空间中具有 **edit** 或 **admin** 角色的用户。但是，**OperatorGroup** 对象不存在 RBAC；没有什么情况可防止常规用户安装 Operator。预安装 Operator 组实际上会提供安装权限。

在将 Operator 组与服务帐户关联时请注意以下几点：

- **APIService** 和 **CustomResourceDefinition** 资源都由 OLM 使用 **cluster-admin** 角色来创建。不应向与 Operator 组相关联的服务帐户授予写入这些资源的权限。
- 与该 Operator 组相关联的所有 Operator 现已被限制在指定服务帐户获得的权限范围内。如果 Operator 请求了超出服务帐户范围的权限，安装会失败，并显示适当的错误，以便集群管理员能够排除故障并解决问题。

4.8.1.1. 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 组并指定了服务帐户。已安装与该 Operator 组关联的所有 Operator，并根据相应服务帐户获得的权限运行。
- 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。
- 对于未指定服务帐户的现有 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帐户获得的权限运行，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，或者现有服务帐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帐户获得的权限运行，与新 Operator 一样。
- 集群管理员从 Operator 组中删除服务帐户。默认行为保留，并允许安装和升级 Operator。

4.8.1.2. 安装 workflow

当 Operator 组与服务帐户绑定，并且安装或升级了 Operator 时，Operator Lifecycle Manager (OLM) 会使用以下 workflow：

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 在服务帐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 组中服务帐户的权限范围内。
5. OLM 新建一个服务帐户，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帐户运行。

4.8.2. 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。

“最佳实践”

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. 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帐户。

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> ❶
  sourceNamespace: <catalog_source_namespace> ❷
EOF
```

❶ 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。

❷ 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帐户授予的权限。如果 Operator 请求的权限超出服务帐户范围，安装会失败并显示相关错误。

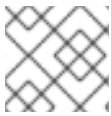
4.8.2.1. 细粒度权限

Operator Lifecycle Manager (OLM) 使用 Operator 组中指定的服务帐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。

4.8.3. Operator 目录访问控制

当在全局目录命名空间 **openshift-marketplace** 中创建 Operator 目录时，目录的 Operator 会提供给所有命名空间。在其他命名空间中创建的目录仅使其 Operator 在目录的同一命名空间中可用。

在非集群管理员用户委托了 Operator 安装权限的集群上，集群管理员可能需要进一步控制或限制允许用户安装的 Operator 集合。这可以通过以下操作来实现：

1. 禁用所有默认全局目录。
2. 在预安装相关 Operator 组的同一命名空间中启用自定义、策展的目录。

其他资源

- [禁用默认的 OperatorHub 目录源](#)
- [在集群中添加目录源](#)

4.8.4. 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。

4.9. 管理自定义目录

集群管理员和 Operator 目录维护人员可以使用 OpenShift Container Platform 的 Operator Lifecycle Manager (OLM) 上的[捆绑包格式](#)创建和管理打包的自定义目录。



重要

Kubernetes 定期弃用后续版本中删除的某些 API。因此，从使用删除 API 的 Kubernetes 版本的 OpenShift Container Platform 版本开始，Operator 无法使用删除 API 的 API。

如果您的集群使用自定义目录，请参阅[控制 Operator 与 OpenShift Container Platform 版本的兼容性](#)，以了解更多有关 Operator 作者如何更新其项目的详细信息，以帮助避免工作负载问题并防止不兼容的升级。

其他资源

- [红帽提供的 Operator 目录](#)

4.9.1. 先决条件

- 已安装 `opm` CLI。

4.9.2. 基于文件的目录

*基于文件的目录*是 Operator Lifecycle Manager (OLM) 中目录格式的最新迭代。它是基于纯文本（JSON 或 YAML）和早期 SQLite 数据库格式的声明式配置演变，并且完全向后兼容。



重要

从 OpenShift Container Platform 4.11 开始，默认的红帽提供的 Operator 目录以基于文件的目录格式发布。通过以过时的 SQLite 数据库格式发布的 4.10，用于 OpenShift Container Platform 4.6 的默认红帽提供的 Operator 目录。

与 SQLite 数据库格式相关的 **opm** 子命令、标志和功能已被弃用，并将在以后的版本中删除。功能仍被支持，且必须用于使用已弃用的 SQLite 数据库格式的目录。

许多 **opm** 子命令和标志都用于 SQLite 数据库格式，如 **opm index prune**，它们无法使用基于文件的目录格式。有关使用基于文件的目录的更多信息，请参阅 [Operator Framework 打包格式](#) 以及使用 [oc-mirror 插件为断开连接的安装 mirror 镜像](#)。

4.9.2.1. 创建基于文件的目录镜像

您可以使用 **opm** CLI 创建一个目录镜像，它使用纯文本（*基于文件的目录*）格式（JSON 或 YAML），替换已弃用的 SQLite 数据库格式。

先决条件

- 已安装 **opm** CLI。
- 您有 **podman** 版本 1.9.3+。
- 已构建捆绑包镜像并推送到支持 [Docker v2-2](#) 的 registry。

流程

1. 初始化目录：

- a. 运行以下命令，为目录创建一个目录：

```
$ mkdir <catalog_dir>
```

- b. 运行 **opm generate dockerfile** 命令生成可构建目录镜像的 Dockerfile：

```
$ opm generate dockerfile <catalog_dir> \
  -i registry.redhat.io/openshift4/ose-operator-registry:v4.15 1
```

- 1** 使用 **-i** 标志指定官方红帽基础镜像，否则 Dockerfile 使用默认的上游镜像。

Dockerfile 必须与您在上一步中创建的目录目录位于相同的父目录中：

目录结构示例

```
. 1
├── <catalog_dir> 2
└── <catalog_dir>.Dockerfile 3
```

- 1** 父目录
- 2** Catalog 目录

3 `opm generate dockerfile` 命令生成的 Dockerfile

c. 运行 `opm init` 命令，使用 Operator 的软件包定义填充目录：

```
$ opm init <operator_name> \ 1
  --default-channel=preview \ 2
  --description=./README.md \ 3
  --icon=./operator-icon.svg \ 4
  --output yml \ 5
  > <catalog_dir>/index.yaml 6
```

- 1 operator 或 package, name
- 2 在未指定时该订阅默认到的频道
- 3 Operator 的 **README.md** 或者其它文档的路径
- 4 到 Operator 图标的路径
- 5 输出格式：JSON 或 YAML
- 6 创建目录配置文件的路径

此命令在指定的目录配置文件中生成 `olm.package` 声明性配置 blob。

2. 运行 `opm render` 命令向目录添加捆绑包：

```
$ opm render <registry>/<namespace>/<bundle_image_name>:<tag> \ 1
  --output=yaml \
  >> <catalog_dir>/index.yaml 2
```

- 1 拉取捆绑包镜像的 spec
- 2 目录配置文件的路径



注意

频道必须至少包含一个捆绑包。

3. 为捆绑包添加频道条目。例如，根据您的规格修改以下示例，并将其添加到 `<catalog_dir>/index.yaml` 文件中：

频道条目示例

```
---
schema: olm.channel
package: <operator_name>
name: preview
entries:
  - name: <operator_name>.v0.1.0 1
```

- 1 确定在 `<operator_name>` 之后、版本 `v` 中包含句点 (`.`)。否则，条目无法传递 `opm validate` 命令。

4. 验证基于文件的目录：

- a. 针对目录目录运行 `opm validate` 命令：

```
$ opm validate <catalog_dir>
```

- b. 检查错误代码是否为 `0`：

```
$ echo $?
```

输出示例

```
0
```

5. 运行 `podman build` 命令构建目录镜像：

```
$ podman build . \
  -f <catalog_dir>.Dockerfile \
  -t <registry>/<namespace>/<catalog_image_name>:<tag>
```

6. 将目录镜像推送到 registry：

- a. 如果需要，运行 `podman login` 命令与目标 registry 进行身份验证：

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

- b. 运行 `podman push` 命令来推送目录镜像：

```
$ podman push <registry>/<namespace>/<catalog_image_name>:<tag>
```

其他资源

- [opm CLI 参考](#)

4.9.2.2. 更新或过滤基于文件的目录镜像

您可以使用 `opm` CLI 更新或过滤使用基于文件的目录格式的目录镜像。通过提取现有目录镜像的内容，您可以根据需要修改目录，例如：

- 添加软件包
- 删除软件包
- 更新现有软件包条目
- 详细说明每个软件包、频道和捆绑包的弃用信息

然后，您可以将镜像重新构建为目录的更新版本。



注意

或者，如果您已在镜像 registry 上已有目录镜像，您可以使用 `oc-mirror` CLI 插件在将其镜像到目标 registry 时自动从该目录镜像更新的源版本中修剪任何删除的镜像。

有关 `oc-mirror` 插件和此用例的更多信息，请参阅“更新您的镜像 registry 内容”部分，特别是“使用 `oc-mirror` 插件为断开连接的安装镜像镜像”部分。

先决条件

- 在您的工作站上有以下内容：
 - `opm` CLI。
 - `podman` 版本 1.9.3+。
 - 基于文件的目录镜像。
 - 最近在与此目录相关的工作站上初始化的目录结构。
如果您没有初始化的 `catalog` 目录，请创建目录并生成 `Dockerfile`。如需更多信息，请参阅“创建基于文件的目录镜像”中的“初始化目录”步骤。

流程

1. 以 YAML 格式将目录镜像的内容提取到 `catalog` 目录中的 `index.yaml` 文件中：

```
$ opm render <registry>/<namespace>/<catalog_image_name>:<tag> \
  -o yaml > <catalog_dir>/index.yaml
```



注意

或者，您可以使用 `-o json` 标志以 JSON 格式输出。

2. 将生成的 `index.yaml` 文件的内容修改为您的规格：



重要

在目录中发布捆绑包后，假设您安装了其中一个用户。确保之前发布目录中的所有捆绑包都具有到当前或更新频道头的更新路径，以避免安装该版本的用户。

- 要添加 Operator，请按照“创建基于文件的目录镜像”过程中创建软件包、捆绑包和频道条目的步骤进行操作。
- 要删除 Operator，请删除与软件包相关的 `olm.package`、`olm.channel` 和 `olm.bundle` blob 的集合。以下示例显示了一个需要删除的集合，才能从目录中删除 `example-operator` 软件包：

例 4.2. 删除条目示例

```
---
defaultChannel: release-2.7
icon:
  base64data: <base64_string>
  mediatype: image/svg+xml
name: example-operator
```

```

schema: olm.package
---
entries:
- name: example-operator.v2.7.0
  skipRange: '>=2.6.0 <2.7.0'
- name: example-operator.v2.7.1
  replaces: example-operator.v2.7.0
  skipRange: '>=2.6.0 <2.7.1'
- name: example-operator.v2.7.2
  replaces: example-operator.v2.7.1
  skipRange: '>=2.6.0 <2.7.2'
- name: example-operator.v2.7.3
  replaces: example-operator.v2.7.2
  skipRange: '>=2.6.0 <2.7.3'
- name: example-operator.v2.7.4
  replaces: example-operator.v2.7.3
  skipRange: '>=2.6.0 <2.7.4'
name: release-2.7
package: example-operator
schema: olm.channel
---
image: example.com/example-inc/example-operator-bundle@sha256:<digest>
name: example-operator.v2.7.0
package: example-operator
properties:
- type: olm.gvk
  value:
    group: example-group.example.io
    kind: MyObject
    version: v1alpha1
- type: olm.gvk
  value:
    group: example-group.example.io
    kind: MyOtherObject
    version: v1beta1
- type: olm.package
  value:
    packageName: example-operator
    version: 2.7.0
- type: olm.bundle.object
  value:
    data: <base64_string>
- type: olm.bundle.object
  value:
    data: <base64_string>
relatedImages:
- image: example.com/example-inc/example-related-image@sha256:<digest>
  name: example-related-image
schema: olm.bundle
---

```

- 要为 Operator 添加或更新弃用信息，请确保在与软件包的 **index.yaml** 文件相同的目录中有一个 **deprecations.yaml** 文件。有关 **deprecations.yaml** 文件格式的详情，请参考 "olm.deprecations schema"。

3. 保存您的更改。

4. 验证目录：

```
$ opm validate <catalog_dir>
```

5. 重建目录：

```
$ podman build . \
  -f <catalog_dir>.Dockerfile \
  -t <registry>/<namespace>/<catalog_image_name>:<tag>
```

6. 将更新的目录镜像推送到 registry：

```
$ podman push <registry>/<namespace>/<catalog_image_name>:<tag>
```

验证

1. 在 Web 控制台中，进入 **Administration** → **Cluster Settings** → **Configuration** 页面中的 OperatorHub 配置资源。
2. 添加目录源或更新现有目录源，以便将 pull spec 用于更新的目录镜像。如需更多信息，请参阅本节的“添加资源”中的“在集群中添加目录源”。
3. 在目录源处于 **READY** 状态后，进入 **Operators** → **OperatorHub** 页面，检查您所做的更改是否反映在 Operator 列表中。

其他资源

- [Packaging format](#) → [Schemas](#) → [olm.deprecations schema](#)
- [使用 oc-mirror 插件为断开连接的安装 mirror 镜像](#) → [Keeping your mirror registry 内容已更新](#)
- [在集群中添加目录源](#)

4.9.3. 基于 SQLite 的目录



重要

Operator 目录的 SQLite 数据库格式是一个弃用的功能。弃用的功能仍然包含在 OpenShift Container Platform 中，并将继续被支持。但是，这个功能会在以后的发行版本中被删除，且不建议在新的部署中使用。

有关 OpenShift Container Platform 中已弃用或删除的主要功能的最新列表，请参阅 OpenShift Container Platform 发行注记中 *已弃用和删除的功能* 部分。

4.9.3.1. 创建基于 SQLite 的索引镜像

您可以使用 **opm** CLI 根据 SQLite 数据库格式创建索引镜像。

先决条件

- 已安装 **opm** CLI。

- 您有 **podman** 版本 1.9.3+。
- 已构建捆绑包镜像并推送到支持 [Docker v2-2](#) 的 registry。

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>/<index_image_name>:<tag>
```

4.9.3.2. 更新基于 SQLite 的索引镜像

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

- 已安装 **opm** CLI。
- 您有 **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 **4.15**。

<updated_tag>

指定要应用到更新的索引镜像的镜像标签，如 **4.15.1**。

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:4.15 \
  --tag mirror.example.com/abc/abc-redhat-operator-index:4.15.1 \
  --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
```

4.9.3.3. 过滤基于 SQLite 的索引镜像

基于 Operator Bundle Format 的索引镜像是 Operator 目录的容器化快照。您可以过滤或 *prune*（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

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

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:v4.15
```

输出示例

```
Trying to pull registry.redhat.io/redhat/redhat-operator-index:v4.15...
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:v4.15 \ 1
-p advanced-cluster-management,jaeger-product,quay-operator \ 2
[-i registry.redhat.io/openshift4/ose-operator-registry:v4.9] \ 3
-t <target_registry>:<port>/<namespace>/redhat-operator-index:v4.15 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® 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:v4.15
```

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

4.9.4. 目录源和 pod 安全准入

OpenShift Container Platform 4.11 中引入了 *Pod 安全准入*，以确保 pod 安全标准。使用基于 SQLite 的目录格式构建的目录源以及在 OpenShift Container Platform 4.11 无法运行受限 pod 安全强制前发布的 **opm** CLI 工具的版本。

在 OpenShift Container Platform 4.15 中，命名空间默认没有受限 pod 安全强制，默认的目录源安全模式设置为 **legacy**。

计划在以后的 OpenShift Container Platform 发行版本中包括所有命名空间的默认限制强制。当发生受限强制时，目录源 pod 规格的安全上下文必须与受限 pod 安全标准匹配。如果您的目录源镜像需要不同的 pod 安全标准，则必须明确设置命名空间的 pod 安全准入标签。



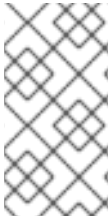
注意

如果您不想以受限方式运行基于 SQLite 的目录源 pod，则不需要在 OpenShift Container Platform 4.15 中更新目录源。

但是，建议您采取措施来确保目录源在受限 pod 安全强制下运行。如果您不采取措施来确保目录源在受限 pod 安全强制下运行，您的目录源可能不会在以后的 OpenShift Container Platform 版本中运行。

作为目录作者，您可以通过完成以下任一操作来启用与受限 pod 安全强制的兼容性：

- 将您的目录迁移到基于文件的目录格式。
- 使用 OpenShift Container Platform 4.11 或更高版本发布的 **opm** CLI 工具版本更新您的目录镜像。



注意

SQLite 数据库目录格式已弃用，但仍然被红帽支持。在以后的发行版本中，不支持 SQLite 数据库格式，目录将需要迁移到基于文件的目录格式。从 OpenShift Container Platform 4.11 开始，默认的红帽提供的 Operator 目录以基于文件的目录格式发布。基于文件的目录与受限 pod 安全强制兼容。

如果您不想更新 SQLite 数据库目录镜像，或将目录迁移到基于文件的目录格式，您可以将目录配置为使用升级的权限运行。

其他资源

- [了解并管理 pod 安全准入](#)

4.9.4.1. 将 SQLite 数据库目录迁移到基于文件的目录格式

您可以将已弃用的 SQLite 数据库格式目录更新为基于文件的目录格式。

先决条件

- SQLite 数据库目录源
- 您可以使用具有 **cluster-admin** 角色的用户访问集群。
- 工作站上 OpenShift Container Platform 4.15 发布的 **opm** CLI 工具的最新版本

流程

1. 运行以下命令，将 SQLite 数据库目录迁移到基于文件的目录：

```
$ opm migrate <registry_image> <fbc_directory>
```

2. 运行以下命令，为您的基于文件的目录生成 Dockerfile：

```
$ opm generate dockerfile <fbc_directory> \  
--binary-image \  
registry.redhat.io/openshift4/ose-operator-registry:v4.15
```

后续步骤

- 生成的 Dockerfile 可以构建、标记并推送到 registry。

其他资源

- [在集群中添加目录源](#)

4.9.4.2. 重建 SQLite 数据库目录镜像

您可以使用 OpenShift Container Platform 版本发布的 **opm** CLI 工具的最新版本重建 SQLite 数据库目录镜像。

先决条件

- SQLite 数据库目录源
- 您可以使用具有 **cluster-admin** 角色的用户访问集群。
- 工作站上 OpenShift Container Platform 4.15 发布的 **opm** CLI 工具的最新版本

流程

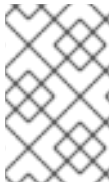
- 运行以下命令，使用 **opm** CLI 工具的最新版本重建目录：

```
$ opm index add --binary-image \
registry.redhat.io/openshift4/ose-operator-registry:v4.15 \
--from-index <your_registry_image> \
--bundles "" -t \<your_registry_image>
```

4.9.4.3. 配置目录以使用升级的权限运行

如果您不想更新 SQLite 数据库目录镜像，或将目录迁移到基于文件的目录格式，您可以执行以下操作以确保目录源在默认 pod 安全强制更改为受限时运行：

- 在目录源定义中手动将目录安全模式设置为 legacy。此操作可确保您的目录使用旧权限运行，即使默认目录安全模式更改为 restricted。
- 为基准或特权 pod 安全强制标记目录源命名空间。



注意

SQLite 数据库目录格式已弃用，但仍然被红帽支持。在以后的发行版本中，不支持 SQLite 数据库格式，目录将需要迁移到基于文件的目录格式。基于文件的目录与受限 pod 安全强制兼容。

先决条件

- SQLite 数据库目录源
- 您可以使用具有 **cluster-admin** 角色的用户访问集群。
- 支持运行带有升级 pod 安全准入标准 **baseline** 或 **privileged** 的 pod 的目标命名空间

流程

1. 通过将 **spec.grpcPodConfig.securityContextConfig** 标签设置为 **legacy** 来编辑 **CatalogSource** 定义，如下例所示：

CatalogSource 定义示例

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

```

name: my-catsrc
namespace: my-ns
spec:
  sourceType: grpc
  grpcPodConfig:
    securityContextConfig: legacy
  image: my-image:latest

```

提示

在 OpenShift Container Platform 4.15 中，**spec.grpcPodConfig.securityContextConfig** 字段默认设置为 **legacy**。在以后的发行版本中，计划默认设置将更改为 **restricted**。如果您的目录无法在受限强制下运行，建议您手动将此字段设置为 **legacy**。

2. 编辑 **<namespace>.yaml** 文件，将升级的 pod 安全准入标准添加到目录源命名空间中，如下例所示：

<namespace>.yaml 文件示例

```

apiVersion: v1
kind: Namespace
metadata:
  ...
  labels:
    security.openshift.io/scc.podSecurityLabelSync: "false" 1
    openshift.io/cluster-monitoring: "true"
    pod-security.kubernetes.io/enforce: baseline 2
  name: "<namespace_name>"

```

- 1** 通过在命名空间中添加 **security.openshift.io/scc.podSecurityLabelSync=false** 标签来关闭 pod 安全标签同步。
- 2** 应用 pod 安全准入 **pod-security.kubernetes.io/enforce** 标签。将标签设置为 **baseline** 或 **privileged**。使用 **baseline** pod 安全配置集，除非命名空间中的其他工作负载需要 **privileged** 配置集。

4.9.5. 在集群中添加目录源

将目录源添加到 OpenShift Container Platform 集群可为用户发现和安装 Operator。集群管理员可以创建一个 **CatalogSource** 对象来引用索引镜像。OperatorHub 使用目录源来填充用户界面。

提示

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

先决条件

- 构建并推送索引镜像到 registry。
- 您可以使用具有 **cluster-admin** 角色的用户访问集群。

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 ❶
  annotations:
    olm.catalogImageTemplate: ❷
    "<registry>/<namespace>/<index_image_name>:v{kube_major_version}.
{kube_minor_version}.{kube_patch_version}"
spec:
  sourceType: grpc
  grpcPodConfig:
    securityContextConfig: <security_mode> ❸
  image: <registry>/<namespace>/<index_image_name>:<tag> ❹
  displayName: My Operator Catalog
  publisher: <publisher_name> ❺
  updateStrategy:
    registryPoll: ❻
    interval: 30m

```

- ❶ 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。
- ❷ 可选：将 **olm.catalogImageTemplate** 注解设置为索引镜像名称，并使用一个或多个 Kubernetes 集群版本变量，如为镜像标签构建模板时所示。
- ❸ 指定 **legacy** 或 **restricted** 的值。如果没有设置该字段，则默认值为 **legacy**。在以后的 OpenShift Container Platform 发行版本中，计划默认值为 **restricted**。如果您的目录无法使用 **restricted** 权限运行，建议您手动将此字段设置为 **legacy**。
- ❹ 指定索引镜像。如果您在镜像名称后指定了标签，如 **:v4.15**，则目录源 Pod 会使用镜像 pull 策略 **Always**，这意味着 pod 始终在启动容器前拉取镜像。如果您指定了摘要，如 **@sha256:<id>**，则镜像拉取策略为 **IfNotPresent**，这意味着仅在节点上不存在的镜像时才拉取镜像。
- ❺ 指定发布目录的名称或机构名称。
- ❻ 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

- [Operator Lifecycle Manager 概念和资源 → Catalog 源](#)
- [从私有 registry 访问 Operator 的镜像](#)
- [镜像拉取 \(pull\) 策略](#)

4.9.6. 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：

索引镜像

CatalogSource 对象可以引用索引镜像，该镜像使用 Operator 捆绑包格式，并作为托管在镜像 registry 中的容器镜像的目录源打包。如果索引镜像托管在私有 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

- 您至少有一个托管在私有 registry 中的以下之一：
 - 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。
- 您可以使用具有 **cluster-admin** 角色的用户访问集群。

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 凭证文件可以在一个 registry 中存储多个 registry 或多个存储库的详细信息。确认您的文件的当前内容。例如：

为多个 registry 存储凭证的文件

```
{
  "auths": {
    "registry.redhat.io": {
      "auth": "FrNHNYdQXdzclNqdg=="
    },
    "quay.io": {
      "auth": "fegdsRib21iMQ=="
    }
  }
}
```

```

    },
    "https://quay.io/my-namespace/my-user/my-image": {
      "auth": "eWfjwsDdfsa221=="
    },
    "https://quay.io/my-namespace/my-user": {
      "auth": "feFweDdscw34rR=="
    },
    "https://quay.io/my-namespace": {
      "auth": "frwEews4fescyq=="
    }
  }
}

```

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: ❶
  - "<secret_name_1>"
  - "<secret_name_2>"
  grpcPodConfig:
    securityContextConfig: <security_mode> ❷
  image: <registry>:<port>/<namespace>/<image>:<tag>
  displayName: My Operator Catalog
  publisher: <publisher_name>
  updateStrategy:
    registryPoll:
      interval: 30m

```

- ❶ 添加 **spec.secrets** 部分并指定任何所需 secret。
 - ❷ 指定 **legacy** 或 **restricted** 的值。如果没有设置该字段，则默认值为 **legacy**。在以后的 OpenShift Container Platform 发行版本中，计划默认值为 **restricted**。如果您的目录无法使用 **restricted** 权限运行，建议您手动将此字段设置为 **legacy**。
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>"}' ❶
  > 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](#)。

4.9.7. 禁用默认的 OperatorHub 目录源

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** → **Configuration** → **OperatorHub** 页面中，点 **Sources** 选项卡，您可以在其中创建、更新、删除、禁用和启用单独的源。

4.9.8. 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。

先决条件

- 您可以使用具有 **cluster-admin** 角色的用户访问集群。

流程

1. 在 Web 控制台的 **Administrator** 视角中，导航到 **Administration** → **Cluster Settings**。
2. 点 **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**。

4.10. 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 在受限网络中成功运行仍取决于 Operator 本身满足以下条件：

- 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。

您可以通过使用以下选择过滤，在 [Red Hat Ecosystem Catalog](#) 上搜索软件以获取支持以断开连接模式运行的红帽 Operator 列表：

类型	容器化应用程序
部署方法	Operator
基础架构特性	断开连接

其他资源

- [红帽提供的 Operator 目录](#)
- [为受限网络环境启用 Operator](#)

4.10.1. 先决条件

- 以具有 **cluster-admin** 权限的用户身份登录 OpenShift Container Platform 集群。

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。

4.10.2. 禁用默认的 OperatorHub 目录源

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：

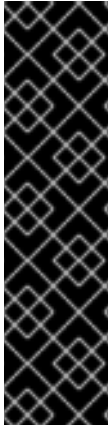
```
$ oc patch OperatorHub cluster --type json \
  -p [{"op": "add", "path": "/spec/disableAllDefaultSources", "value": true}]
```


提示

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

4.10.3. 对 Operator 目录进行镜像(mirror)

有关与断开连接的集群一起使用的 Operator 目录的说明，请参阅 [为断开连接的安装安装 → 镜像](#)。



重要

从 OpenShift Container Platform 4.11 开始，默认的红帽提供的 Operator 目录以基于文件的目录格式发布。通过以过时的 SQLite 数据库格式发布的 4.10，用于 OpenShift Container Platform 4.6 的默认红帽提供的 Operator 目录。

与 SQLite 数据库格式相关的 **opm** 子命令、标志和功能已被弃用，并将在以后的版本中删除。功能仍被支持，且必须用于使用已弃用的 SQLite 数据库格式的目录。

许多 **opm** 子命令和标志都用于 SQLite 数据库格式，如 **opm index prune**，它们无法使用基于文件的目录格式。有关使用基于文件的目录的更多信息，请参阅 [Operator Framework 打包格式](#)，[管理自定义目录](#)，以及使用 **oc-mirror** 插件为断开连接的安装 **mirror** 镜像。

4.10.4. 在集群中添加目录源

将目录源添加到 OpenShift Container Platform 集群可为用户发现和安装 Operator。集群管理员可以创建一个 **CatalogSource** 对象来引用索引镜像。OperatorHub 使用目录源来填充用户界面。

提示

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

先决条件

- 构建并推送索引镜像到 registry。
- 您可以使用具有 **cluster-admin** 角色的用户访问集群。

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 manifests 目录中生成的 **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
  grpcPodConfig:
    securityContextConfig: <security_mode> 3
```

```
image: <registry>/<namespace>/redhat-operator-index:v4.15 4
displayName: My Operator Catalog
publisher: <publisher_name> 5
updateStrategy:
  registryPoll: 6
    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 **legacy** 或 **restricted** 的值。如果没有设置该字段，则默认值为 **legacy**。在以后的 OpenShift Container Platform 发行版本中，计划默认值为 **restricted**。如果您的目录无法使用 **restricted** 权限运行，建议您手动将此字段设置为 **legacy**。
- 4 指定索引镜像。如果您在镜像名称后指定了标签，如 **:v4.15**，则目录源 Pod 会使用镜像 pull 策略 **Always**，这意味着 pod 始终在启动容器前拉取镜像。如果您指定了摘要，如 **@sha256:<id>**，则镜像拉取策略为 **IfNotPresent**，这意味着仅在节点上不存在的镜像时才拉取镜像。
- 5 指定发布目录的名称或机构名称。
- 6 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-6njsx6          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

- [从私有 registry 访问 Operator 的镜像](#)
- [自定义目录源的镜像模板](#)
- [镜像拉取 \(pull\) 策略](#)

4.10.5. 后续步骤

- [更新安装的 Operator](#)

4.11. 目录源 POD 调度

当源类型 **grpc** 的 Operator Lifecycle Manager (OLM) 目录源定义 **spec.image** 时，Catalog Operator 会创建一个提供定义的镜像内容的 pod。默认情况下，此 pod 在规格中定义以下内容：

- 只有 **kubernetes.io/os=linux** 节点选择器。
- 默认优先级类名称：**system-cluster-critical**。
- 没有容限。

作为管理员，您可以通过修改 **CatalogSource** 对象的可选 **spec.grpcPodConfig** 部分中的字段来覆盖这些值。

重要

Marketplace Operator **openshift-marketplace** 负责管理默认的 **OperatorHub** 自定义资源 (CR)。此 CR 管理 **CatalogSource** 对象。如果您试图修改 **CatalogSource** 对象的 **spec.grpcPodConfig** 部分中的字段，则 Marketplace Operator 会自动恢复这些修改。默认情况下，如果您修改了 **CatalogSource** 对象的 **spec.grpcPodConfig** 部分中的字段，则 Marketplace Operator 会自动恢复这些更改。

要将持久性更改应用到 **CatalogSource** 对象，您必须首先禁用一个默认的 **CatalogSource** 对象。

其他资源

- [OLM 概念和资源 → Catalog source](#)

4.11.1. 在本地级别禁用默认 CatalogSource 对象

您可以通过禁用默认的 **CatalogSource** 对象，对 **CatalogSource** 对象（如目录源 pod）应用到本地级

别。当默认 **CatalogSource** 对象的配置不符合您的机构需求时，请考虑默认配置。默认情况下，如果您修改 **CatalogSource** 对象的 **spec.grpcPodConfig** 部分中的字段，Marketplace Operator 会自动恢复这些更改。

Marketplace Operator **openshift-marketplace** 负责管理 **OperatorHub** 的默认自定义资源 (CR)。 **OperatorHub** 管理 **CatalogSource** 对象。

要将持久性更改应用到 **CatalogSource** 对象，您必须首先禁用一个默认的 **CatalogSource** 对象。

流程

- 要在本地级别禁用所有默认 **CatalogSource** 对象，请输入以下命令：

```
$ oc patch operatorhub cluster -p '{"spec": {"disableAllDefaultSources": true}}' --type=merge
```



注意

您还可以将默认 **OperatorHub** CR 配置为禁用所有 **CatalogSource** 对象或禁用特定对象。

其他资源

- [OperatorHub 自定义资源](#)
- [禁用默认的 OperatorHub 目录源](#)

4.11.2. 覆盖目录源 pod 的节点选择器

先决条件

- 源类型的 **CatalogSource** 对象，定义了 **spec.image**

流程

- 编辑 **CatalogSource** 对象并添加或修改 **spec.grpcPodConfig** 部分，使其包含以下内容：

```
grpcPodConfig:
  nodeSelector:
    custom_label: <label>
```

其中 **<label>** 是您希望目录源 pod 用于调度的节点选择器的标签。

其他资源

- [使用节点选择器将 pod 放置到特定节点](#)

4.11.3. 覆盖目录源 pod 的优先级类名称

先决条件

- 源类型的 **CatalogSource** 对象，定义了 **spec.image**

流程

- 编辑 **CatalogSource** 对象并添加或修改 **spec.grpcPodConfig** 部分，使其包含以下内容：

```
grpcPodConfig:
  priorityClassName: <priority_class>
```

其中 **<priority_class>** 是以下之一：

- Kubernetes 提供的默认优先级类之一：**system-cluster-critical** 或 **system-node-critical**
- 用于分配默认优先级的空集合 ("")
- 预先存在的和自定义优先级类

注意

在以前的版本中，唯一可以被覆盖的 pod 调度参数是 **priorityClassName**。这可以通过将 **operatorframework.io/priorityclass** 注解添加到 **CatalogSource** 对象来实现。例如：

```
apiVersion: operators.coreos.com/v1alpha1
kind: CatalogSource
metadata:
  name: example-catalog
  namespace: openshift-marketplace
  annotations:
    operatorframework.io/priorityclass: system-cluster-critical
```

如果 **CatalogSource** 对象同时定义了注解和 **spec.grpcPodConfig.priorityClassName**，注解优先于配置参数。

其他资源

- [Pod 优先级类](#)

4.11.4. 覆盖目录源 pod 的容限

先决条件

- 源类型的 **CatalogSource** 对象，定义了 **spec.image**

流程

- 编辑 **CatalogSource** 对象并添加或修改 **spec.grpcPodConfig** 部分，使其包含以下内容：

```
grpcPodConfig:
  tolerations:
    - key: "<key_name>"
      operator: "<operator_type>"
      value: "<value>"
      effect: "<effect>"
```

其他资源

- [了解污点和容限](#)

4.12. 管理平台 OPERATOR（技术预览）

平台 Operator 是一个基于 OLM 的 Operator，可在 OpenShift Container Platform 集群的第 0 天操作期间或之后安装，并参与集群的生命周期。作为集群管理员，您可以使用 **PlatformOperator** API 管理平台 Operator。



重要

平台 Operator 类型只是一个技术预览功能。技术预览功能不受红帽产品服务等级协议（SLA）支持，且功能可能并不完整。红帽不推荐在生产环境中使用它们。这些技术预览功能可以使用户提早试用新的功能，并有机会在开发阶段提供反馈意见。

有关红帽技术预览功能支持范围的更多信息，请参阅[技术预览功能支持范围](#)。

4.12.1. 关于平台 Operator

Operator Lifecycle Manager (OLM) 引入了一个新的 Operator 类型，名为 *平台 Operator*。平台 Operator 是一个基于 OLM 的 Operator，可在 OpenShift Container Platform 集群的第 0 天操作期间或之后安装，并参与集群的生命周期。作为集群管理员，您可以使用平台 Operator 来进一步自定义 OpenShift Container Platform 安装，以满足您的要求和用例。

使用 OpenShift Container Platform 中的现有集群功能功能，集群管理员可能已经禁用了基于 Cluster Version Operator (CVO) 的部分组件，因为对于集群安装前的初始负载，它们被视为是非必须的。平台 Operator 通过提供额外的自定义选项来迭代此模型。通过平台 Operator 机制，依赖于 RukPak 组件中的资源，现在可在集群安装时安装基于 OLM 的 Operator，如果 Operator 无法成功安装，则可阻止集群推出部署。

在 OpenShift Container Platform 4.12 中，这个技术预览版本侧重于基本平台 Operator 机制，并构建在即将发布的版本中扩展概念的基础。您可以使用集群范围的 **PlatformOperator** API 在启用了 **TechPreviewNoUpgrade** 功能集的集群中或之后配置 Operator。

其他资源

- [RukPak 组件和打包格式](#)
- [集群功能](#)

4.12.1.1. 平台 Operator 的技术预览限制

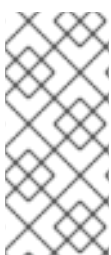
在 OpenShift Container Platform 4.12 中，平台 Operator 功能的技术预览版本中，以下限制决定是否可以通过平台 Operator 机制安装 Operator：

- Kubernetes 清单必须使用 Operator Lifecycle Manager (OLM) **registry+v1** 捆绑包格式进行打包。
- Operator 无法声明软件包或 group/version/kind (GVK) 依赖项。
- Operator 无法指定 **AllNamespaces** 以外的集群服务版本 (CSV) 安装模式
- Operator 无法指定任何 **Webhook** 或 **APIService** 定义。
- 所有软件包捆绑包都必须位于 **redhat-operators** 目录源中。

在考虑这些限制后，可以成功安装以下 Operator：

表 4.2. 基于 OLM 的 Operator 可作为平台 Operator 安装

3scale-operator	amq-broker-rhel8
amq-online	amq-streams
ansible-cloud-addons-operator	apicast-operator
container-security-operator	eap
file-integrity-operator	gatekeeper-operator-product
integration-operator	jws-operator
kiali-ossm	node-healthcheck-operator
odf-csi-addons-operator	odr-hub-operator
openshift-custom-metrics-autoscaler-operator	openshift-gitops-operator
openshift-pipelines-operator-rh	quay-operator
red-hat-camel-k	rhpm-kogito-operator
service-registry-operator	servicemeshoperator
skupper-operator	



注意

在此技术预览版本中不提供以下功能：

- 集群推出部署后自动升级平台 Operator 软件包
- 扩展平台 Operator 机制以支持任何基于 CVO 的可选组件

4.12.2. 先决条件

- 使用具有 **cluster-admin** 权限的账户访问 OpenShift Container Platform 集群。
- 在集群中启用 **TechPreviewNoUpgrades** 功能集



警告

启用 **TechPreviewNoUpgrade** 功能集将无法撤消，并防止次版本更新。不建议在生产环境集群中使用这些功能集。

- 仅在集群中启用 **redhat-operators** 目录源。这是技术预览版本期间的一个限制。
- **oc** 命令在工作站上安装。

其他资源

- [使用功能门启用功能](#)
- [禁用默认的 OperatorHub 目录源](#)

4.12.3. 在集群创建过程中安装平台 Operator

作为集群管理员，您可以通过在创建集群过程中提供 **FeatureGate** 和 **PlatformOperator** 清单来安装平台 Operator。

流程

1. 从受支持的基于 OLM 的 Operator 集合中选择一个平台 Operator。有关此集合列表以及当前限制的详情，请参阅“平台 Operator 的技术预览限制”。
2. 选择集群安装方法并按照创建 **install-config.yaml** 文件进行操作。有关准备集群安装的详情，请参阅“选择集群安装方法并为用户准备它”。
3. 创建 **install-config.yaml** 文件并完成对其所做的任何修改后，切换到包含安装程序的目录并创建清单：

```
$ ./openshift-install create manifests --dir <installation_directory> 1
```

- 1** 对于 **<installation_directory>**，请指定包含集群的 **install-config.yaml** 文件的目录名称。

4. 在 **<installation_directory>/manifests/** 目录中创建一个 **FeatureGate** 对象 YAML 文件，该文件启用了 **TechPreviewNoUpgrade** 功能集，例如 **feature-gate.yaml** 文件：

feature-gate.yaml 文件示例

```
apiVersion: config.openshift.io/v1
kind: FeatureGate
metadata:
  annotations:
    include.release.openshift.io/self-managed-high-availability: "true"
    include.release.openshift.io/single-node-developer: "true"
    release.openshift.io/create-only: "true"
```



```
name: cluster
spec:
  featureSet: TechPreviewNoUpgrade 1
```

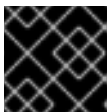
- 1 启用 **TechPreviewNoUpgrade** 功能集。

5. 在 `<installation_directory>/manifests/` 目录中为所选平台 Operator 创建 **PlatformOperator** 对象 YAML 文件，如 Red Hat OpenShift Service Mesh Operator 的 **service-mesh-po.yaml** 文件：

service-mesh-po.yaml 文件示例

```
apiVersion: platform.openshift.io/v1alpha1
kind: PlatformOperator
metadata:
  name: service-mesh-po
spec:
  package:
    name: servicemeshoperator
```

6. 当您准备好完成集群安装时，请参阅您选择的安装方法并继续运行 **openshift-install create cluster** 命令。
在集群创建过程中，提供的清单用于启用 **TechPreviewNoUpgrade** 功能集并安装所选平台 Operator。



重要

无法成功安装平台 Operator 将阻止集群安装过程。

验证

1. 运行以下命令，检查 **service-mesh-po** platform Operator 的状态：

```
$ oc get platformoperator service-mesh-po -o yaml
```

输出示例

```
...
status:
  activeBundleDeployment:
    name: service-mesh-po
  conditions:
  - lastTransitionTime: "2022-10-24T17:24:40Z"
    message: Successfully applied the service-mesh-po BundleDeployment resource
    reason: InstallSuccessful
    status: "True" 1
    type: Installed
```

- 1 等待 **Installed** 状态报告为 **True**。

2. 验证 **platform-operators-aggregated** 集群 Operator 是否已报告 **Available=True** 状态：

■

```
$ oc get clusteroperator platform-operators-aggregated -o yaml
```

输出示例

```
...
status:
  conditions:
  - lastTransitionTime: "2022-10-24T17:43:26Z"
    message: All platform operators are in a successful state
    reason: AsExpected
    status: "False"
    type: Progressing
  - lastTransitionTime: "2022-10-24T17:43:26Z"
    message: All platform operators are in a successful state
    reason: AsExpected
    status: "True"
    type: Available
```

其他资源

- [选择集群安装方法并为用户准备它](#)
- [平台 Operator 的技术预览限制](#)

4.12.4. 在集群创建后安装平台 Operator

作为集群管理员，您可以在使用集群范围的 **PlatformOperator** API 启用 **TechPreviewNoUpgrades** 功能集的集群中安装平台 Operator。

流程

1. 从受支持的基于 OLM 的 Operator 集合中选择一个平台 Operator。有关此集合列表以及当前限制的详情，请参阅“平台 Operator 的技术预览限制”。
2. 为所选平台 Operator 创建 **PlatformOperator** 对象 YAML 文件，如 Red Hat OpenShift Service Mesh Operator 的 **service-mesh-po.yaml** 文件：

service-mesh-po.yaml 文件示例

```
apiVersion: platform.openshift.io/v1alpha1
kind: PlatformOperator
metadata:
  name: service-mesh-po
spec:
  package:
    name: servicemeshoperator
```

3. 运行以下命令来创建 **PlatformOperator** 对象：

```
$ oc apply -f service-mesh-po.yaml
```



注意

如果您的集群没有启用 **TechPreviewNoUpgrade** 功能集，则对象创建会失败，并显示以下信息：

```
error: resource mapping not found for name: "service-mesh-po" namespace:
"" from "service-mesh-po.yaml": no matches for kind "PlatformOperator" in
version "platform.openshift.io/v1alpha1"
ensure CRDs are installed first
```

验证

1. 运行以下命令，检查 **service-mesh-po** platform Operator 的状态：

```
$ oc get platformoperator service-mesh-po -o yaml
```

输出示例

```
...
status:
  activeBundleDeployment:
    name: service-mesh-po
  conditions:
  - lastTransitionTime: "2022-10-24T17:24:40Z"
    message: Successfully applied the service-mesh-po BundleDeployment resource
    reason: InstallSuccessful
    status: "True" ①
    type: Installed
```

- ① 等待 **Installed** 状态报告为 **True**。

2. 验证 **platform-operators-aggregated** 集群 Operator 是否已报告 **Available=True** 状态：

```
$ oc get clusteroperator platform-operators-aggregated -o yaml
```

输出示例

```
...
status:
  conditions:
  - lastTransitionTime: "2022-10-24T17:43:26Z"
    message: All platform operators are in a successful state
    reason: AsExpected
    status: "False"
    type: Progressing
  - lastTransitionTime: "2022-10-24T17:43:26Z"
    status: "False"
    type: Degraded
  - lastTransitionTime: "2022-10-24T17:43:26Z"
    message: All platform operators are in a successful state
```

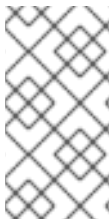
```
reason: AsExpected
status: "True"
type: Available
```

其他资源

- [平台 Operator 的技术预览限制](#)

4.12.5. 删除平台 Operator

作为集群管理员，您可以删除现有的平台 Operator。Operator Lifecycle Manager (OLM) 执行级联删除。首先，OLM 会删除平台 Operator 的捆绑包部署，然后删除 **registry+v1** 类型捆绑包中引用的任何对象。



注意

平台 Operator Manager 和捆绑包部署置备程序只管理捆绑包中引用的对象，而不管理任何捆绑包工作负载本身部署的对象。例如，如果捆绑包工作负载会创建一个命名空间，且 Operator 没有配置为在删除 Operator 前进行清理，则它超出了 OLM 范围，以便在平台 Operator 删除过程中删除命名空间。

流程

1. 获取已安装的平台 Operator 列表，找到您要删除的 Operator 的名称：

```
$ oc get platformoperator
```

2. 删除所选 Operator 的 **PlatformOperator** 资源，如 Quay Operator：

```
$ oc delete platformoperator quay-operator
```

输出示例

```
platformoperator.platform.openshift.io "quay-operator" deleted
```

验证

1. 验证平台 Operator 的命名空间最终已被删除，例如 Quay Operator：

```
$ oc get ns quay-operator-system
```

输出示例

```
Error from server (NotFound): namespaces "quay-operator-system" not found
```

2. 验证 **platform-operators-aggregated** 集群 Operator 继续报告 **Available=True** 状态：

```
$ oc get co platform-operators-aggregated
```

输出示例

NAME	VERSION	AVAILABLE	PROGRESSING	DEGRADED	SINCE
platform-operators-aggregated	4.15.0-0	True	False	False	70s

4.13. TROUBLESHOOTING OPERATOR 的问题

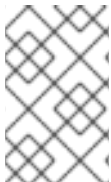
如果遇到 Operator 问题，请验证 Operator 订阅状态。检查集群中的 Operator pod 健康状况，并收集 Operator 日志以进行诊断。

4.13.1. operator 订阅状况类型

订阅可报告以下状况类型：

表 4.3. 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。
ResolutionFailed	订阅的依赖项解析失败。



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。

其他资源

- [目录健康要求](#)

4.13.2. 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

```
Name:      cluster-logging
Namespace: openshift-logging
Labels:    operators.coreos.com/cluster-logging.openshift-logging=
Annotations: <none>
API Version: operators.coreos.com/v1alpha1
Kind:      Subscription
# ...
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。

4.13.3. 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
Labels: <none>
Annotations: operatorframework.io/managed-by: marketplace-operator
             target.workload.openshift.io/management: {"effect": "PreferredDuringScheduling"}
API Version: operators.coreos.com/v1alpha1
Kind: CatalogSource
# ...
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-vsxd/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-vsxd
  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。

其他资源

- [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](#)

4.13.4. 查询 Operator pod 状态

您可以列出集群中的 Operator pod 及其状态。您还可以收集详细的 Operator pod 概述。

先决条件

- 您可以使用具有 **cluster-admin** 角色的用户访问集群。
- API 服务仍然可以正常工作。
- 已安装 OpenShift CLI (**oc**)。

流程

1. 列出集群中运行的 Operator。输出包括 Operator 版本、可用性和运行时间信息：


```
$ oc get clusteroperators
```

- 列出在 Operator 命名空间中运行的 Operator pod，以及 pod 状态、重启和年龄：

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

- 输出详细的 Operator pod 概述：

```
$ oc describe pod <operator_pod_name> -n <operator_namespace>
```

- 如果 Operator 问题特定于某个节点，则在该节点上查询 Operator 容器状态。

- 为节点启动 debug pod：

```
$ oc debug node/my-node
```

- 将 **/host** 设为 debug shell 中的根目录。debug pod 在 pod 中的 **/host** 中挂载主机的 root 文件系统。将根目录改为 **/host**，您可以运行主机可执行路径中包含的二进制文件：

```
# chroot /host
```



注意

运行 Red Hat Enterprise Linux CoreOS(RHCOS)的 OpenShift Container Platform 4.15 集群节点不可变，它依赖于 Operator 来应用集群更改。不建议使用 SSH 访问集群节点。但是，如果 OpenShift Container Platform API 不可用，或 kubelet 在目标节点上无法正常工作，**oc** 操作将会受到影响。在这种情况下，可以使用 **ssh core@<node>.<cluster_name>.<base_domain>** 来访问节点。

- 列出节点容器的详细信息，包括状态和关联的 pod ID:

```
# crictl ps
```

- 列出节点上特定 Operator 容器的信息。以下示例列出了 **network-operator** 容器的信息：

```
# crictl ps --name network-operator
```

- 退出 debug shell。

4.13.5. 收集 Operator 日志

如果遇到 Operator 问题，您可以从 Operator pod 日志中收集详细诊断信息。

先决条件

- 您可以使用具有 **cluster-admin** 角色的用户访问集群。
- API 服务仍然可以正常工作。
- 已安装 OpenShift CLI(**oc**)。

- 您有 control plane 或 control plane 机器的完全限定域名。

流程

1. 列出在 Operator 命名空间中运行的 Operator pod，以及 pod 状态、重启和年龄：

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

2. 检查 Operator pod 的日志：

```
$ oc logs pod/<pod_name> -n <operator_namespace>
```

如果 Operator pod 具有多个容器，则上述命令将会产生一个错误，其中包含每个容器的名称。从独立容器查询日志：

```
$ oc logs pod/<operator_pod_name> -c <container_name> -n <operator_namespace>
```

3. 如果 API 无法正常工作，请使用 SSH 来查看每个 control plane 节点上的 Operator pod 和容器日志。将 **<master-node>.<cluster_name>.<base_domain>** 替换为适当的值。

- a. 列出每个 control plane 节点上的 pod：

```
$ ssh core@<master-node>.<cluster_name>.<base_domain> sudo crictl pods
```

- b. 对于任何未显示 **Ready** 状态的 Operator pod，详细检查 Pod 的状态。将 **<operator_pod_id>** 替换为上一命令输出中列出的 Operator pod ID:

```
$ ssh core@<master-node>.<cluster_name>.<base_domain> sudo crictl inspectp <operator_pod_id>
```

- c. 列出与 Operator pod 相关的容器：

```
$ ssh core@<master-node>.<cluster_name>.<base_domain> sudo crictl ps --pod= <operator_pod_id>
```

- d. 对于任何未显示 **Ready** 状态的 Operator 容器，请详细检查容器的状态。将 **<container_id>** 替换为上一命令输出中列出的容器 ID:

```
$ ssh core@<master-node>.<cluster_name>.<base_domain> sudo crictl inspect <container_id>
```

- e. 检查任何未显示 **Ready** 状态的 Operator 容器的日志。将 **<container_id>** 替换为上一命令输出中列出的容器 ID:

```
$ ssh core@<master-node>.<cluster_name>.<base_domain> sudo crictl logs -f <container_id>
```



注意

运行 Red Hat Enterprise Linux CoreOS (RHCOS) 的 OpenShift Container Platform 4.15 集群节点不可变，它依赖于 Operator 来应用集群更改。不建议使用 SSH 访问集群节点。在尝试通过 SSH 收集诊断数据前，请运行 **oc adm must gather** 和其他 **oc** 命令看它们是否可以提供足够的信息。但是，如果 OpenShift Container Platform API 不可用，或 kubelet 在目标节点上无法正常工作，**oc** 操作将会受到影响。在这种情况下，可以使用 **ssh core@<node>.<cluster_name>.<base_domain>** 来访问节点。

4.13.6. 禁用 Machine Config Operator 自动重新引导

当 Machine Config Operator (MCO) 进行配置更改时，Red Hat Enterprise Linux CoreOS (RHCOS) 必须重启才能使更改生效。无论配置更改是自动还是手动的，RHCOS 节点都会自动重启，除非它已被暂停。



注意

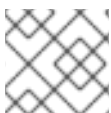
以下修改不会触发节点重新引导：

- 当 MCO 检测到以下任何更改时，它会在不排空或重启节点的情况下应用更新：
 - 在机器配置的 **spec.config.passwd.users.sshAuthorizedKeys** 参数中更改 SSH 密钥。
 - 在 **openshift-config** 命名空间中更改全局 pull secret 或 pull secret。
 - Kubernetes API Server Operator 自动轮转 **/etc/kubernetes/kubelet-ca.crt** 证书颁发机构 (CA)。
- 当 MCO 检测到对 **/etc/containers/registries.conf** 文件的更改时，如添加或编辑 **ImageDigestMirrorSet**、**ImageTagMirrorSet** 或 **ImageContentSourcePolicy** 对象，它会排空对应的节点，应用更改并取消记录节点。对于以下更改，节点排空不会发生：
 - 增加了一个 registry，带有为每个镜像 (mirror) 设置了 **pull-from-mirror = "digest-only"** 参数。
 - 增加了一个镜像 (mirror)，带有在一个 registry 中设置的 **pull-from-mirror = "digest-only"** 参数。
 - 在 **unqualified-search-registries** 列表中添加项目。

为了避免不必要的中断，您可以修改机器配置池 (MCP) 以防止在 Operator 更改机器配置后自动重启。

4.13.6.1. 使用控制台禁用 Machine Config Operator 自动重新引导

为了避免对 Machine Config Operator (MCO) 所做的更改造成不必要的中断，您可以使用 OpenShift Container Platform Web 控制台修改机器配置池 (MCP)，以防止 MCO 在那个池中对节点进行任何更改。这会防止任何通常属于 MCO 更新过程一部分的重启。



注意

请参阅第二个有关 [禁用 Machine Config Operator 自动重新引导](#) 的备注。

先决条件

- 您可以使用具有 **cluster-admin** 角色的用户访问集群。

流程

要暂停或取消暂停自动 MCO 更新重新引导：

- 暂停自动引导过程：
 1. 以具有 **cluster-admin** 角色的用户身份登录到 OpenShift Container Platform web 控制台。
 2. 点 **Compute** → **MachineConfigPools**。
 3. 在 **MachineConfigPools** 页面中，点击 **master** 或 **worker**，具体取决于您要暂停重新引导的节点。
 4. 在 **master** 或 **worker** 页面中，点 **YAML**。
 5. 在 YAML 中，将 **spec.paused** 字段更新为 **true**。

MachineConfigPool 对象示例

```
apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfigPool
# ...
spec:
# ...
  paused: true ①
# ...
```

- ① 将 **spec.paused** 字段更新为 **true** 以暂停重新引导。

6. 要验证 MCP 是否已暂停，请返回到 **MachineConfigPools** 页面。
在 **MachineConfigPools** 页面中，您修改的 MCP 报告了 **Paused** 列中为 **True**。

如果 MCP 在暂停时有待处理的变化，**Updated** 列为 **False**，**Updating** 为 **False**。当 **Updated** 为 **True** 且 **Updating** 为 **False** 时，代表没有待处理的更改。



重要

如果有尚未进行的更改（**Updated** 和 **Updating** 字段都是 **False**），建议您尽快调度一个维护窗口用于重启。使用以下步骤取消暂停自动引导过程，以应用上一次重启后排队的更改。

- 取消暂停自动引导过程：
 1. 以具有 **cluster-admin** 角色的用户身份登录到 OpenShift Container Platform web 控制台。
 2. 点 **Compute** → **MachineConfigPools**。
 3. 在 **MachineConfigPools** 页面中，点击 **master** 或 **worker**，具体取决于您要暂停重新引导的节点。
 4. 在 **master** 或 **worker** 页面中，点 **YAML**。

- 在 YAML 中，将 `spec.paused` 字段更新为 `false`。

MachineConfigPool 对象示例

```
apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfigPool
# ...
spec:
# ...
  paused: false ❶
# ...
```

- ❶ 将 `spec.paused` 字段更新为 `false` 以允许重启。



注意

通过取消暂停 MCP，MCO 应用所有暂停的更改，根据需要重启 Red Hat Enterprise Linux CoreOS (RHCOS)。

- 要验证 MCP 是否已暂停，请返回到 **MachineConfigPools** 页面。在 **MachineConfigPools** 页面中，您修改的 MCP 报告 **Paused** 列为 **False**。

如果 MCP 正在应用任何待处理的更改，**Updated** 列为 **False**，**Updating** 列为 **True**。当 **Updated** 为 **True** 且 **Updating** 为 **False** 时，不会再进行任何更改。

4.13.6.2. 使用 CLI 禁用 Machine Config Operator 自动重新引导

为了避免对 Machine Config Operator (MCO) 所做的更改造成不必要的中断，您可以使用 OpenShift CLI (`oc`) 来修改机器配置池 (MCP)，以防止 MCO 在那个池中对节点进行任何更改。这会防止任何通常属于 MCO 更新过程一部分的重启。



注意

请参阅第二个有关 [禁用 Machine Config Operator 自动重新引导](#) 的备注。

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

要暂停或取消暂停自动 MCO 更新重新引导：

- 暂停自动引导过程：
 - 更新 **MachineConfigPool** 自定义资源，将 `spec.paused` 字段设置为 `true`。

Control plane (master) 节点

```
$ oc patch --type=merge --patch='{"spec":{"paused":true}}' machineconfigpool/master
```

Worker 节点

```
$ oc patch --type=merge --patch='{"spec":{"paused":true}}' machineconfigpool/worker
```

2. 验证 MCP 是否已暂停：

Control plane (master) 节点

```
$ oc get machineconfigpool/master --template='{{.spec.paused}}'
```

Worker 节点

```
$ oc get machineconfigpool/worker --template='{{.spec.paused}}'
```

输出示例

```
true
```

spec.paused 字段为 **true**，MCP 暂停。

3. 确定 MCP 是否有待处理的更改：

```
# oc get machineconfigpool
```

输出示例

NAME	CONFIG	UPDATED	UPDATING
master	rendered-master-33cf0a1254318755d7b48002c597bf91	True	False
worker	rendered-worker-e405a5bdb0db1295acea08bccca33fa60	False	False

如果 **UPDATED** 列是 **False**，**UPDATING** 为 **False**，则有待处理的更改。当 **UPDATED** 为 **True** 且 **UPDATING** 为 **False** 时，没有待处理的更改。在上例中，worker 节点有待处理的变化。control plane 节点没有任何待处理的更改。



重要

如果有尚未进行的更改（**Updated** 和 **Updating** 字段都是 **False**），建议您尽快调度一个维护窗口用于重启。使用以下步骤取消暂停自动引导过程，以应用上一次重启后排队的更改。

- 取消暂停自动引导过程：

1. 更新 **MachineConfigPool** 自定义资源，将 **spec.paused** 字段设置为 **false**。

Control plane (master) 节点

```
$ oc patch --type=merge --patch='{"spec":{"paused":false}}' machineconfigpool/master
```

Worker 节点

```
$ oc patch --type=merge --patch='{"spec":{"paused":false}}' machineconfigpool/worker
```



注意

通过取消暂停 MCP，MCO 应用所有暂停的更改，根据需要重启 Red Hat Enterprise Linux CoreOS (RHCOS)。

2. 验证 MCP 是否已取消暂停：

Control plane (master) 节点

```
$ oc get machineconfigpool/master --template='{{.spec.paused}}'
```

Worker 节点

```
$ oc get machineconfigpool/worker --template='{{.spec.paused}}'
```

输出示例

```
false
```

spec.paused 字段为 **false**，MCP 被取消暂停。

3. 确定 MCP 是否有待处理的更改：

```
$ oc get machineconfigpool
```

输出示例

```
NAME          CONFIG                                UPDATED UPDATING
master        rendered-master-546383f80705bd5aeaba93 True    False
worker        rendered-worker-b4c51bb33ccaae6fc4a6a5 False   True
```

如果 MCP 正在应用任何待处理的更改，**UPDATED** 列为 **False**，**UPDATING** 列为 **True**。当 **UPDATED** 为 **True** 且 **UPDATING** 为 **False** 时，没有进一步的更改。在上例中，MCO 正在更新 worker 节点。

4.13.7. 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>
```

4.13.8. 卸载失败后重新安装 Operator

在尝试重新安装同一 Operator 前，您必须已成功并完全卸载了 Operator。无法正确卸载 Operator 可以正确地保留资源，如项目或命名空间，处于"Terminating"状态，并导致"错误解析资源"信息。例如：

Project 资源描述示例

```
...
message: 'Failed to delete all resource types, 1 remaining: Internal error occurred:
error resolving resource'
...
```

这些类型的问题可能会阻止 Operator 成功重新安装。



警告

强制删除命名空间可能无法解决 "Terminating" 状态问题，并可能导致不稳定或无法预计的集群行为，因此最好尝试查找可能会阻止命名空间被删除的相关资源。如需更多信息，请参阅[红帽知识库解决方案 #4165791](#)，请专注于注意和警告部分。

以下流程演示了如何在无法重新安装 Operator 时进行故障排除，因为之前安装 Operator 中的自定义资源定义 (CRD) 会阻止相关的命名空间成功删除。

流程

1. 检查是否有与 Operator 相关的命名空间是否处于"Terminating"状态：

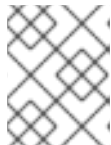
```
$ oc get namespaces
```

输出示例

```
operator-ns-1          Terminating
```

2. 检查卸载失败后是否存在与 Operator 相关的 CRD :

```
$ oc get crds
```



注意

CRD 是全局集群定义；与 CRD 相关的实际自定义资源 (CR) 实例可能位于其他命名空间中，也可以是全局集群实例。

3. 如果有任何 CRD 由 Operator 提供或管理，并在卸载后删除 CRD :

```
$ oc delete crd <crd_name>
```

4. 检查卸载后是否仍然存在与 Operator 相关的剩余 CR 实例，如果存在，请删除 CR :

- a. 要搜索的 CR 类型可能很难确定卸载后，需要了解 Operator 管理哪些 CRD。例如，如果您要对提供 **EtcdCluster** CRD 的 etcd Operator 卸载进行故障排除，您可以在命名空间中搜索剩余的 **EtcdCluster** CR :

```
$ oc get EtcdCluster -n <namespace_name>
```

另外，您还可以在所有命名空间中搜索 :

```
$ oc get EtcdCluster --all-namespaces
```

- b. 如果有任何剩余的 CR 应该被删除，请删除实例 :

```
$ oc delete <cr_name> <cr_instance_name> -n <namespace_name>
```

5. 检查命名空间删除是否已成功解决 :

```
$ oc get namespace <namespace_name>
```



重要

如果命名空间或其他 Operator 资源仍然没有完全卸载，请联系红帽支持。

6. 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>
```

其他资源

- [从集群中删除 Operator](#)
- [在集群中添加 Operator](#)

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、Java 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 4.15 支持 Operator SDK 1.31.0。

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](#)。

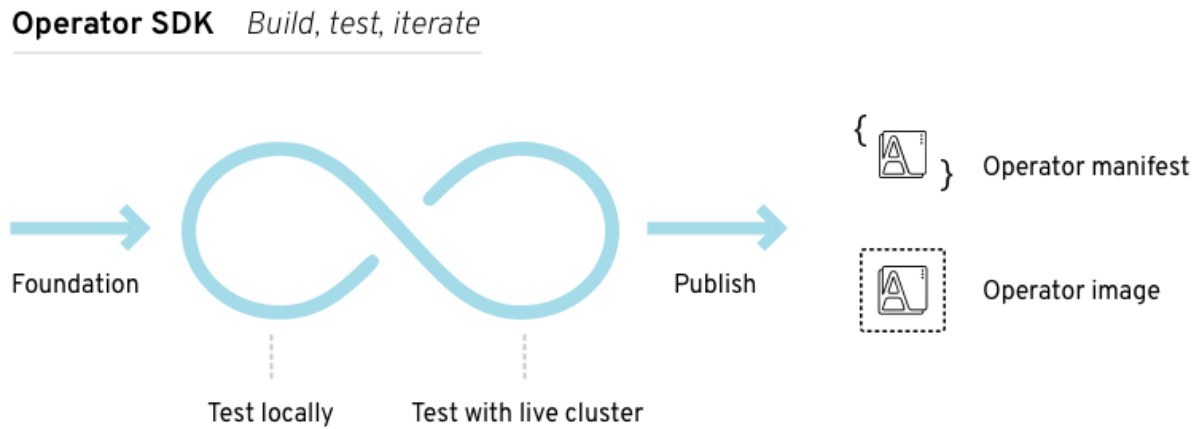
5.1.2. 开发 workflow

Operator SDK 提供以下 workflow 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 workflow



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。

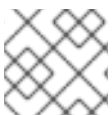
5.1.3. 其他资源

- [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。

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、Java 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 4.15 支持 Operator SDK 1.31.0。

5.2.1. 在 Linux 上安装 Operator SDK CLI

您可以在 Linux 上安装 OpenShift SDK CLI 工具。

先决条件

- [Go](#) v1.19+
- **docker** v17.03+、**podman** v1.9.3+ 或 **buildah** v1.7+

流程

1. 进入到 [OpenShift 镜像站点](#)。
2. 从最新的 4.15 目录中，下载适用于 Linux 的 tarball 的最新版本。
3. 解包存档：

```
$ tar xvf operator-sdk-v1.31.0-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: "v1.31.0-ocp", ...
```

5.2.2. 在 macOS 上安装 Operator SDK CLI

您可以在 macOS 上安装 OpenShift SDK CLI 工具。

先决条件

- [Go](#) v1.19+
- **docker** v17.03+、**podman** v1.9.3+ 或 **buildah** v1.7+

流程

1. 对于 **amd64** 和 **arm64** 架构，分别进入 [amd64 集成架构的 OpenShift 镜像站点](#) 和 [arm64 基础架构的 OpenShift 镜像站点](#)。
2. 从最新的 4.15 目录中，下载 macOS 的 tarball 的最新版本。
3. 运行以下命令，为 **amd64** 架构解包 Operator SDK 归档：

```
$ tar xvf operator-sdk-v1.31.0-ocp-darwin-x86_64.tar.gz
```

- 运行以下命令，为 **arm64** 架构解包 Operator SDK 归档：

```
$ tar xvf operator-sdk-v1.31.0-ocp-darwin-aarch64.tar.gz
```

- 运行以下命令使文件可执行：

```
$ chmod +x operator-sdk
```

- 运行以下命令，将提取的 **operator-sdk** 二进制文件移到 **PATH** 上的目录中：

提示

运行以下命令检查 **PATH**：

```
$ echo $PATH
```

```
$ sudo mv ./operator-sdk /usr/local/bin/operator-sdk
```

验证

- 安装 Operator SDK CLI 后，运行以下命令验证是否可用：

```
$ operator-sdk version
```

输出示例

```
operator-sdk version: "v1.31.0-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**) 4.15+
- [Go](#) 1.21+
- 使用具有 **cluster-admin** 权限的 **oc** 登录到 OpenShift Container Platform 4.15 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

其他资源

- [安装 Operator SDK CLI](#)
- [OpenShift CLI 入门](#)

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。

- 创建您的项目目录：

```
$ mkdir memcached-operator
```

- 切换到项目所在的目录：

```
$ cd memcached-operator
```

- 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。

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。

- 安装 CRD：

```
$ make install
```

- 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. 删除 CR。

运行以下命令来删除 CR：

```
$ oc delete -f config/samples/cache_v1_memcached.yaml -n memcached-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
```

5.3.1.3. 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**) 4.15+
- [Go](#) 1.21+

- 使用具有 **cluster-admin** 权限的 **oc** 登录到 OpenShift Container Platform 4.15 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

其他资源

- [安装 Operator SDK CLI](#)
- [OpenShift CLI 入门](#)

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。

5.3.2.2.1. 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"
plugins:
  manifests.sdk.operatorframework.io/v2: {}
  scorecard.sdk.operatorframework.io/v2: {}
  sdk.x-openshift.io/v1: {}
```

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"
    ctrllog "sigs.k8s.io/controller-runtime/pkg/log"

    cachev1 "github.com/example-inc/memcached-operator/api/v1"

```

`)`

// 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;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;

// 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)
    log := ctrllog.FromContext(ctx)
    // Fetch the Memcached instance
    memcached := &cachev1.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 *cachev1.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(&cachev1.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来直接进入 [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循环。它负责实现协调循环的...

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)
    ...
}
```

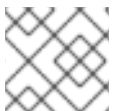
根据返回值、结果和错误，请求可能会重新排序，协调循环可能会再次触发：

```
// 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;update;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;
```

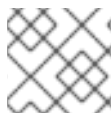
```
func (r *MemcachedReconciler) Reconcile(ctx context.Context, req ctrl.Request) (ctrl.Result, error) {
    ...
}
```

`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. 启用代理支持

Operator 作者可开发支持网络代理的 Operator。集群管理员配置对 Operator Lifecycle Manager (OLM) 处理的环境变量的代理支持。要支持代理集群，Operator 必须检查以下标准代理变量的环境，并将值传递给 Operands：

- `HTTP_PROXY`
- `HTTPS_PROXY`
- `NO_PROXY`



注意

本教程使用 `HTTP_PROXY` 作为示例环境变量。

先决条件

- 启用了集群范围的出口代理的集群。

流程

1. 编辑 `controllers/memcached_controller.go` 文件，使其包含以下项：
 - a. 从 `operator-lib` 库导入 `proxy` 软件包：

```
import (
    ...
    "github.com/operator-framework/operator-lib/proxy"
)
```

- b. 将 `proxy.ReadProxyVarsFromEnv` helper 功能添加到协调循环中，并将结果附加到 Operand 环境：

```
for i, container := range dep.Spec.Template.Spec.Containers {
    dep.Spec.Template.Spec.Containers[i].Env = append(container.Env,
        proxy.ReadProxyVarsFromEnv(...)...)
}
...
```

2. 通过在 `config/manager/manager.yaml` 文件中添加以下内容来设置 Operator 部署上的环境变量：

```
containers:
  - args:
    - --leader-elect
```

```

- --leader-election-id=ansible-proxy-demo
image: controller:latest
name: manager
env:
- name: "HTTP_PROXY"
  value: "http_proxy_test"

```

5.3.2.6. 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。

5.3.2.6.1. 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}

```

5.3.2.6.2. 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

- 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。

- 构建镜像：

```
$ make docker-build IMG=<registry>/<user>/<image_name>:<tag>
```

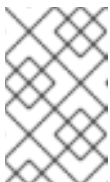


注意

由 SDK 为 Operator 生成的 Dockerfile 需要为 **go build** 明确引用 **GOARCH=amd64**。这可以在非 AMD64 构架中使用 **GOARCH=\$TARGETARCH**。Docker 自动将环境变量设置为 **-platform** 指定的值。对于 Buildah，需要使用 **-build-arg** 来实现这一目的。如需更多信息，请参阅[多个架构](#)。

- 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。

- 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。

- 运行以下命令验证 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
```

5.3.2.6.3. 捆绑 Operator 并使用 Operator Lifecycle Manager 进行部署

5.3.2.6.3.1. 捆绑 Operator

Operator 捆绑包格式是 Operator SDK 和 Operator Lifecycle Manager (OLM) 的默认打包方法。您可以使用 Operator SDK 来构建和推送 Operator 项目作为捆绑包镜像，使 Operator 可供 OLM 使用。

先决条件

- 在开发工作站上安装 operator SDK CLI
- 已安装 OpenShift CLI (**oc**) v4.15+
- 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 需要为 **go build** 明确引用 **GOARCH=amd64**。这可以在非 AMD64 构架中使用 **GOARCH=\$TARGETARCH**。Docker 自动将环境变量设置为 **-platform** 指定的值。对于 Buildah，需要使用 **-build-arg** 来实现这一目的。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_IMG**：

```
$ 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>
```

5.3.2.6.3.2. 使用 Operator Lifecycle Manager 部署 Operator

Operator Lifecycle Manager (OLM) 可帮助您在 Kubernetes 集群中安装、更新和管理 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。您可以使用 Operator SDK 在 OLM 上快速运行捆绑包镜像，以确保它正确运行。

先决条件

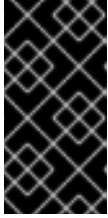
- 在开发工作站上安装 operator SDK CLI
- 构建并推送到 registry 的 Operator 捆绑包镜像
- OLM 安装在一个基于 Kubernetes 的集群上（如果使用 **apiextensions.k8s.io/v1** CRD，则为 v1.16.0 或更新版本，如 OpenShift Container Platform 4.15）
- 使用具有 **cluster-admin** 权限的账户使用 **oc** 登录到集群
- 如果 Operator 是基于 Go 的，则必须更新您的项目以使用支持的镜像在 OpenShift Container Platform 上运行

流程

- 输入以下命令在集群中运行 Operator：

```
$ operator-sdk run bundle \ ❶
-n <namespace> \ ❷
<registry>/<user>/<bundle_image_name>:<tag> ❸
```

- ❶ **run bundle** 命令创建基于文件的有效目录，并使用 OLM 在集群中安装 Operator 捆绑包。
- ❷ 可选：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。
- ❸ 如果没有指定镜像，该命令使用 **quay.io/operator-framework/opm:latest** 作为默认索引镜像。如果指定了镜像，该命令会使用捆绑包镜像本身作为索引镜像。



重要

自 OpenShift Container Platform 4.11 起，**run bundle** 命令默认支持 Operator 目录基于文件的目录格式。Operator 目录已弃用的 SQLite 数据库格式仍被支持，但将在以后的发行版本中删除。建议 Operator 作者将其工作流迁移到基于文件的目录格式。

这个命令执行以下操作：

- 创建引用捆绑包镜像的索引镜像。索引镜像不透明且具有临时性，但准确反映了如何将捆绑包添加到生产中的目录中。
- 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 **OperatorGroup**、**Subscription**、**InstallPlan** 和所有其他所需资源（包括 RBAC），将 Operator 部署到集群中。

5.3.2.7. 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. 运行以下命令来删除 CR :

```
$ oc delete -f config/samples/cache_v1_memcached.yaml
```

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>
```

5.3.2.8. 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。
- 如果配置了[集群范围的出口代理](#), 集群管理员可以[覆盖代理设置](#), 或为 Operator Lifecycle Manager(OLM)上运行的特定 Operator 注入自定义 CA 证书。

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。

文件或目录	用途
manifests/	Kubernetes 清单用于注册 CRD、设置 RBAC 和将 Operator 部署为部署。

5.3.4. 为较新的 Operator SDK 版本更新基于 Go 的 Operator 项目

OpenShift Container Platform 4.15 支持 Operator SDK 1.31.0。如果您已在工作站上安装了 1.28.0 CLI，您可以通过[安装最新版本](#)将 CLI 更新至 1.31.0。

但是，要确保现有 Operator 项目保持与 Operator SDK 1.31.0 的兼容性，需要执行更新的相关步骤才能解决从 1.28.0 以来引入的相关破坏更改。您必须在之前使用 1.28.0 创建或维护的任何 Operator 项目中手动执行更新步骤。

5.3.4.1. 为 Operator SDK 1.31.0 更新基于 Go 的 Operator 项目

以下流程更新了基于 Go 的 Operator 项目，以便与 1.31.0 兼容。

先决条件

- 已安装 operator SDK 1.31.0
- 使用 Operator SDK 1.28.0 创建或维护的 Operator 项目

流程

- 编辑 Operator 项目的 makefile，将 Operator SDK 版本更新至 1.31.0，如下例所示：

makefile 示例

```
# Set the Operator SDK version to use. By default, what is installed on the system is used.
# This is useful for CI or a project to utilize a specific version of the operator-sdk toolkit.
OPERATOR_SDK_VERSION ?= v1.31.0 1
```

- 1** 将版本从 **1.28.0** 更改为 **1.31.0**。

5.3.4.2. 其他资源

- [将软件包清单项目迁移到捆绑包格式](#)
- [为 Operator SDK 1.16.0 升级项目](#)
- [升级 Operator SDK v1.10.1 的项目](#)
- [针对 Operator SDK v1.8.0 升级项目](#)

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**) 4.15+
- [Ansible 2.15.0](#)
- [Ansible Runner 2.3.3+](#)
- [Ansible Runner HTTP Event Emitter plugin 1.0.0+](#)
- [Python 3.9+](#)
- [Python Kubernetes 客户端](#)
- 使用具有 **cluster-admin** 权限的 **oc** 登录到 OpenShift Container Platform 4.15 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

其他资源

- [安装 Operator SDK CLI](#)
- [OpenShift CLI 入门](#)

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 ❶
```

❶ 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 ledelection.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. 删除 CR。

运行以下命令来删除 CR：

```
$ oc delete -f config/samples/cache_v1_memcached.yaml -n memcached-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
```

5.4.1.3. 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**) 4.15+
- [Ansible](#) 2.15.0
- [Ansible Runner](#) 2.3.3+
- [Ansible Runner HTTP Event Emitter plugin](#) 1.0.0+

- [Python 3.9+](#)
- [Python Kubernetes 客户端](#)
- 使用具有 **cluster-admin** 权限的 **oc** 登录到 OpenShift Container Platform 4.15 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

其他资源

- [安装 Operator SDK CLI](#)
- [OpenShift CLI 入门](#)

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
plugins:
  manifests.sdk.operatorframework.io/v2: {}
  scorecard.sdk.operatorframework.io/v2: {}
  sdk.x-openshift.io/v1: {}
projectName: memcached-operator
version: "3"
```

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

- 用下列结构更新 **roles/memcached/tasks/main.yml** 文件：

```
---
- name: start memcached
  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:
  labels:
    app.kubernetes.io/name: memcached
    app.kubernetes.io/instance: memcached-sample
    app.kubernetes.io/part-of: memcached-operator
    app.kubernetes.io/managed-by: kustomize
    app.kubernetes.io/created-by: memcached-operator
  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. 启用代理支持

Operator 作者可开发支持网络代理的 Operator。集群管理员配置对 Operator Lifecycle Manager (OLM) 处理的环境变量的代理支持。要支持代理集群，Operator 必须检查以下标准代理变量的环境，并将值传递给 Operands：

- **HTTP_PROXY**

- **HTTPS_PROXY**
- **NO_PROXY**



注意

本教程使用 **HTTP_PROXY** 作为示例环境变量。

先决条件

- 启用了集群范围的出口代理的集群。

流程

1. 通过使用以下内容更新 **roles/memcached/tasks/main.yml** 文件，将环境变量添加到部署中：

```
...
env:
  - name: HTTP_PROXY
    value: '{{ lookup("env", "HTTP_PROXY") | default("", True) }}'
  - name: http_proxy
    value: '{{ lookup("env", "HTTP_PROXY") | default("", True) }}'
...
```

2. 通过在 **config/manager/manager.yaml** 文件中添加以下内容来设置 Operator 部署上的环境变量：

```
containers:
  - args:
    - --leader-elect
    - --leader-election-id=ansible-proxy-demo
    image: controller:latest
    name: manager
    env:
      - name: "HTTP_PROXY"
        value: "http_proxy_test"
```

5.4.2.6. 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。

5.4.2.6.1. 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 `~/kubernetes/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":"Memcached"}
{"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}
```

5.4.2.6.2. 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

- 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。
 - 构建镜像：

```
$ make docker-build IMG=<registry>/<user>/<image_name>:<tag>
```



注意

由 SDK 为 Operator 生成的 Dockerfile 需要为 **go build** 明确引用 **GOARCH=amd64**。这可以在非 AMD64 构架中使用 **GOARCH=\$TARGETARCH**。Docker 自动将环境变量设置为 **-platform** 指定的值。对于 Buildah，需要使用 **-build-arg** 来实现这一目的。如需更多信息，请参阅[多个架构](#)。

- 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。

- 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。

- 运行以下命令验证 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
```

5.4.2.6.3. 捆绑 Operator 并使用 Operator Lifecycle Manager 进行部署

5.4.2.6.3.1. 捆绑 Operator

Operator 捆绑包格式是 Operator SDK 和 Operator Lifecycle Manager (OLM) 的默认打包方法。您可以使用 Operator SDK 来构建和推送 Operator 项目作为捆绑包镜像，使 Operator 可供 OLM 使用。

先决条件

- 在开发工作站上安装 operator SDK CLI
- 已安装 OpenShift CLI (**oc**) v4.15+
- 使用 Operator SDK 初始化 operator 项目

流程

- 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。

- 构建镜像 :

```
$ make docker-build IMG=<registry>/<user>/<operator_image_name>:<tag>
```



注意

由 SDK 为 Operator 生成的 Dockerfile 需要为 **go build** 明确引用 **GOARCH=amd64**。这可以在非 AMD64 构架中使用 **GOARCH=\$TARGETARCH**。Docker 自动将环境变量设置为 **-platform** 指定的值。对于 Buildah，需要使用 **-build-arg** 来实现这一目的。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_IMG**：

```
$ 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>
```

5.4.2.6.3.2. 使用 Operator Lifecycle Manager 部署 Operator

Operator Lifecycle Manager（OLM）可帮助您在 Kubernetes 集群中安装、更新和管理 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。您可以使用 Operator SDK 在 OLM 上快速运行捆绑包镜像，以确保它正确运行。

先决条件

- 在开发工作stations上安装 operator SDK CLI
- 构建并推送到 registry 的 Operator 捆绑包镜像
- 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 4.15）
- 使用具有 **cluster-admin** 权限的账户使用 **oc** 登录到集群

流程

- 输入以下命令在集群中运行 Operator :

```
$ operator-sdk run bundle \ ❶
-n <namespace> \ ❷
<registry>/<user>/<bundle_image_name>:<tag> ❸
```

- ❶ **run bundle** 命令创建基于文件的有效目录，并使用 OLM 在集群中安装 Operator 捆绑包。
- ❷ 可选：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。
- ❸ 如果没有指定镜像，该命令使用 `quay.io/operator-framework/opm:latest` 作为默认索引镜像。如果指定了镜像，该命令会使用捆绑包镜像本身作为索引镜像。



重要

自 OpenShift Container Platform 4.11 起，**run bundle** 命令默认支持 Operator 目录基于文件的目录格式。Operator 目录已弃用的 SQLite 数据库格式仍被支持，但将在以后的发行版本中删除。建议 Operator 作者将其工作流迁移到基于文件的目录格式。

这个命令执行以下操作：

- 创建引用捆绑包镜像的索引镜像。索引镜像不透明且具有临时性，但准确反映了如何将捆绑包添加到生产中的目录中。
- 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 **OperatorGroup**、**Subscription**、**InstallPlan** 和所有其他所需资源（包括 RBAC），将 Operator 部署到集群中。

5.4.2.7. 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. 运行以下命令来删除 CR：

```
$ oc delete -f config/samples/cache_v1_memcached.yaml
```

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>
```

5.4.2.8. 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。
- 如果配置了[集群范围的出口代理](#)，集群管理员可以[覆盖代理设置](#)，或为 [Operator Lifecycle Manager\(OLM\)](#)上运行的特定 Operator 注入自定义 CA 证书。

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。
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.yaml	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. 为较新的 Operator SDK 版本更新项目

OpenShift Container Platform 4.15 支持 Operator SDK 1.31.0。如果您已在工作站上安装了 1.28.0 CLI，您可以通过[安装最新版本](#)将 CLI 更新至 1.31.0。

但是，要确保现有 Operator 项目保持与 Operator SDK 1.31.0 的兼容性，需要执行更新的相关步骤才能解决从 1.28.0 以来引入的相关破坏更改。您必须在之前使用 1.28.0 创建或维护的任何 Operator 项目中手动执行更新步骤。

5.4.4.1. 为 Operator SDK 1.31.0 更新基于 Ansible 的 Operator 项目

以下流程更新了基于 Ansible 的 Operator 项目，以便与 1.31.0 兼容。

先决条件

- 已安装 operator SDK 1.31.0
- 使用 Operator SDK 1.28.0 创建或维护的 Operator 项目

流程

1. 对 Operator 的 Dockerfile 进行以下更改：

- a. 将 **ansible-operator-2.11-preview** 基础镜像替换为 **ansible-operator** 基础镜像，并将版本更新至 1.31.0，如下例所示：

Dockerfile 示例

```
FROM quay.io/operator-framework/ansible-operator:v1.31.0
```

- b. Ansible Operator 版本 1.30.0 中的 Ansible 2.15.0 的更新删除了以下预安装的 Python 模块：

- **ipaddress**
- **openshift**
- **jmespath**
- **cryptography**
- **oauthlib**

如果您的 Operator 依赖于其中一个删除的 Python 模块，请更新 Dockerfile 以使用 **pip install** 命令安装所需的模块。

2. 编辑 Operator 项目的 makefile，将 Operator SDK 版本更新至 1.31.0，如下例所示：

makefile 示例

```
# Set the Operator SDK version to use. By default, what is installed on the system is used.
# This is useful for CI or a project to utilize a specific version of the operator-sdk toolkit.
OPERATOR_SDK_VERSION ?= v1.31.0 ❶
```

- ❶ 将版本从 **1.28.0** 更改为 **1.31.0**。

3. 更新 **requirements.yaml** 和 **requirements.go** 文件，以删除 **community.kubernetes** 集合，并将 **operator_sdk.util** 集合更新至版本 **0.5.0**，如下例所示：

requirements.yaml 文件示例

```
collections:
- - name: community.kubernetes ❶
  - version: "2.0.1"
- - name: operator_sdk.util
  - version: "0.4.0"
+ - version: "0.5.0" ❶
```

```
- name: kubernetes.core
  version: "2.4.0"
- name: cloud.common
```

- 1 删除 **community.kubernetes** 集合
- 2 将 **operator_sdk.util** 集合更新至 **0.5.0** 版本。

4. 从 **molecule/kind/molecule.yml** 和 **molecule/default/molecule.yml** 文件中删除 **lint** 字段的所有实例，如下例所示：

```
---
dependency:
  name: galaxy
driver:
  name: delegated
- lint: |
- set -e
- yamllint -d "{extends: relaxed, rules: {line-length: {max: 120}}}" .
platforms:
  - name: cluster
  groups:
- k8s
provisioner:
  name: ansible
- lint: |
- set -e
  ansible-lint
inventory:
  group_vars:
all:
  namespace: ${TEST_OPERATOR_NAMESPACE:-osdk-test}
  host_vars:
localhost:
  ansible_python_interpreter: '{{ ansible_playbook_python }}'
  config_dir: ${MOLECULE_PROJECT_DIRECTORY}/config
  samples_dir: ${MOLECULE_PROJECT_DIRECTORY}/config/samples
  operator_image: ${OPERATOR_IMAGE:-""}
  operator_pull_policy: ${OPERATOR_PULL_POLICY:-"Always"}
  kustomize: ${KUSTOMIZE_PATH:-kustomize}
  env:
    K8S_AUTH_KUBECONFIG: ${KUBECONFIG:-"~/.kube/config"}
verifier:
  name: ansible
- lint: |
- set -e
- ansible-lint
```

5.4.4.2. 其他资源

- [为 Operator SDK v1.25.4 升级项目](#)
- [为 Operator SDK v1.22.0 升级项目](#)

- [为 Operator SDK v1.16.0 升级项目](#)
- [升级 Operator SDK v1.10.1 的项目](#)
- [针对 Operator SDK v1.8.0 升级项目](#)
- [将软件包清单项目迁移到捆绑包格式](#)

5.4.5. Operator SDK 中的 Ansible 支持

5.4.5.1. 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"
```

5.4.5.2. 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 <code>/opt/ansible/roles/</code> 中，角色名为 busybox ，则该值应为 <code>/opt/ansible/roles/busybox</code> 。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。

5.4.5.2.1. 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
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
```

5.4.5.3. 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 }}"
```

5.4.5.4. 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](#)。

5.4.6. 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。

5.4.6.1. 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 2.15+ :

```
$ sudo dnf install ansible
```

2. 安装 [Python Kubernetes 客户端软件包](#) :

```
$ pip install kubernetes
```

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。默认情况下，此文件会安装 **community.kubernetes** 集合以及 **operator_sdk.util** 集合，它为特定于 Operator 的功能提供模块和插件。

安装来自 **requirements.yml** 文件的依赖模块 :

```
$ ansible-galaxy collection install -r requirements.yml
```

5.4.6.2. 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: <operator_namespace> ❶
state: "{{ state }}"
ignore_errors: true ❷

```

❶ 指定要创建配置映射的命名空间。

❷ 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
```

5.4.6.3. 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](#)。

5.4.7. 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。

5.4.7.1. 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"
```

5.4.7.2. 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 v2.3.3+](#)

- [Ansible Runner HTTP Event Emitter plugin](#) 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":"v1.10.1","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":fals
e}
{"level":"info","ts":1612739148.3490646,"logger":"ansible-controller","msg":"Watching
resource","Options.Group":"cache.example.com","Options.Version":"v1","Options.Kind":"Memc
ached"}
{"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
```

5.4.7.3. 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 需要为 **go build** 明确引用 **GOARCH=amd64**。这可以在非 AMD64 构架中使用 **GOARCH=\$TARGETARCH**。Docker 自动将环境变量设置为 **-platform** 指定的值。对于 Buildah，需要使用 **-build-arg** 来实现这一目的。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. 运行以下命令验证 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
```

5.4.7.4. 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。

5.4.7.4.1. 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": "v1.10.1",
  "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": false}
{"level": "info", "ts": 1612732107.7688901, "logger": "ansible-controller", "msg": "Watching resource", "Options.Group": "cache.example.com", "Options.Version": "v1", "Options.Kind": "Memcached"}
{"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}
```

5.4.7.4.2. 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"
```

5.4.7.4.3. 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
```

5.4.8. 自定义资源状态管理

5.4.8.1. 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。

5.4.8.2. 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**) 4.15+
- 使用具有 **cluster-admin** 权限的 **oc** 登录到 OpenShift Container Platform 4.15 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

其他资源

- [安装 Operator SDK CLI](#)
- [OpenShift CLI 入门](#)

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. 删除 CR。

运行以下命令来删除 CR：

```
$ oc delete -f config/samples/demo_v1_nginx.yaml -n nginx-operator-system
```

8. 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**) 4.15+
- 使用具有 **cluster-admin** 权限的 **oc** 登录到 OpenShift Container Platform 4.15 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

其他资源

- [安装 Operator SDK CLI](#)
- [OpenShift CLI 入门](#)

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
```

- 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。

- 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
<code><repo_name>/<chart_name></code>	从名为 <code><repo_name></code> 的 helm chart 中获取名为 <code><chart_name></code> 的 Helm chart，如 <code>\$HELM_HOME/repositories/repositories.yaml</code> 文件指定。使用 <code>helm repo add</code> 命令来配置此文件。
<code><url></code>	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
<code><chart_name></code>	在由 <code>--helm-chart-repo</code> URL 值指定的 Helm Chart 仓库中获取名为 <code><chart_name></code> 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
plugins:
  manifests.sdk.operatorframework.io/v2: {}
  scorecard.sdk.operatorframework.io/v2: {}
  sdk.x-openshift.io/v1: {}
projectName: nginx-operator
resources:
- api:
  crdVersion: v1
  namespaced: true
  domain: example.com
  group: demo
  kind: Nginx
  version: v1
  version: "3"
```

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行版本的实用信息。

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. 启用代理支持

Operator 作者可开发支持网络代理的 Operator。集群管理员配置对 Operator Lifecycle Manager (OLM) 处理的环境变量的代理支持。要支持代理集群，Operator 必须检查以下标准代理变量的环境，并将值传递给 Operands：

- **HTTP_PROXY**
- **HTTPS_PROXY**
- **NO_PROXY**



注意

本教程使用 **HTTP_PROXY** 作为示例环境变量。

先决条件

- 启用了集群范围的出口代理的集群。

流程

1. 通过添加 **overrideValues** 字段来编辑 **watches.yaml** 文件，使其包含基于环境变量的覆盖：

```

...
- group: demo.example.com
  version: v1alpha1
  kind: Nginx
  chart: helm-charts/nginx
  overrideValues:
    proxy.http: $HTTP_PROXY
...

```

2. 在 **helm-charts/nginx/values.yaml** 文件中添加 **proxy.http** 值：

```

...
proxy:
  http: ""
  https: ""
  no_proxy: ""

```

3. 要确保 Chart 模板支持使用变量，请编辑 **helm-charts/nginx/templates/deployment.yaml** 文件中的 chart 模板，使其包含以下内容：

```

containers:

```



```

- name: {{ .Chart.Name }}
  securityContext:
    - toYaml {{ .Values.securityContext | nindent 12 }}
  image: "{{ .Values.image.repository }}:{{ .Values.image.tag | default .Chart.AppVersion }}"
  imagePullPolicy: {{ .Values.image.pullPolicy }}
  env:
    - name: http_proxy
      value: "{{ .Values.proxy.http }}"

```

4. 通过在 **config/manager/manager.yaml** 文件中添加以下内容来设置 Operator 部署上的环境变量：

```

containers:
  - args:
    - --leader-elect
    - --leader-election-id=ansible-proxy-demo
    image: controller:latest
    name: manager
    env:
      - name: "HTTP_PROXY"
        value: "http_proxy_test"

```

5.5.2.5. 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。

5.5.2.5.1. 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":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}
```

5.5.2.5.2. 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 需要为 **go build** 明确引用 **GOARCH=amd64**。这可以在非 AMD64 构架中使用 **GOARCH=\$TARGETARCH**。Docker 自动将环境变量设置为 **-platform** 指定的值。对于 Buildah，需要使用 **-build-arg** 来实现这一目的。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. 运行以下命令验证 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

```

5.5.2.5.3. 捆绑 Operator 并使用 Operator Lifecycle Manager 进行部署

5.5.2.5.3.1. 捆绑 Operator

Operator 捆绑包格式是 Operator SDK 和 Operator Lifecycle Manager (OLM) 的默认打包方法。您可以使用 Operator SDK 来构建和推送 Operator 项目作为捆绑包镜像，使 Operator 可供 OLM 使用。

先决条件

- 在开发工作stations上安装 operator SDK CLI
- 已安装 OpenShift CLI (**oc**) v4.15+
- 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 需要为 **go build** 明确引用 **GOARCH=amd64**。这可以在非 AMD64 构架中使用 **GOARCH=\$TARGETARCH**。Docker 自动将环境变量设置为 **-platform** 指定的值。对于 Buildah，需要使用 **-build-arg** 来实现这一目的。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_IMG**：

```
$ 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>
```

5.5.2.5.3.2. 使用 Operator Lifecycle Manager 部署 Operator

Operator Lifecycle Manager (OLM) 可帮助您在 Kubernetes 集群中安装、更新和管理 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。您可以使用 Operator SDK 在 OLM 上快速运行捆绑包镜像，以确保它正确运行。

先决条件

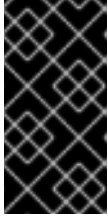
- 在开发工作站上安装 operator SDK CLI
- 构建并推送到 registry 的 Operator 捆绑包镜像
- OLM 安装在一个基于 Kubernetes 的集群上（如果使用 **apiextensions.k8s.io/v1** CRD，则为 v1.16.0 或更新版本，如 OpenShift Container Platform 4.15）
- 使用具有 **cluster-admin** 权限的账户使用 **oc** 登录到集群

流程

- 输入以下命令在集群中运行 Operator：

```
$ operator-sdk run bundle \ 1
-n <namespace> \ 2
<registry>/<user>/<bundle_image_name>:<tag> 3
```

- 1** **run bundle** 命令创建基于文件的有效目录，并使用 OLM 在集群中安装 Operator 捆绑包。
- 2** 可选：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。
- 3** 如果没有指定镜像，该命令使用 **quay.io/operator-framework/opm:latest** 作为默认索引镜像。如果指定了镜像，该命令会使用捆绑包镜像本身作为索引镜像。



重要

自 OpenShift Container Platform 4.11 起，**run bundle** 命令默认支持 Operator 目录基于文件的目录格式。Operator 目录已弃用的 SQLite 数据库格式仍被支持，但将在以后的发行版本中删除。建议 Operator 作者将其工作流迁移到基于文件的目录格式。

这个命令执行以下操作：

- 创建引用捆绑包镜像的索引镜像。索引镜像不透明且具有临时性，但准确反映了如何将捆绑包添加到生产中的目录中。
- 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 **OperatorGroup**、**Subscription**、**InstallPlan** 和所有其他所需资源（包括 RBAC），将 Operator 部署到集群中。

5.5.2.6. 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

- 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. 运行以下命令来删除 CR :

```
$ oc delete -f config/samples/demo_v1_nginx.yaml
```

9. 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>
```

5.5.2.7. 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。
- 如果配置了[集群范围的出口代理](#), 集群管理员可以[覆盖代理设置](#), 或为 [Operator Lifecycle Manager\(OLM\)](#)上运行的特定 Operator 注入自定义 CA 证书。

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. 为较新的 Operator SDK 版本更新基于 Helm 的项目

OpenShift Container Platform 4.15 支持 Operator SDK 1.31.0。如果您已在工作站上安装了 1.28.0 CLI，您可以通过[安装最新版本](#)将 CLI 更新至 1.31.0。

但是，要确保现有 Operator 项目保持与 Operator SDK 1.31.0 的兼容性，需要执行更新的相关步骤才能解决从 1.28.0 以来引入的相关破坏更改。您必须在之前使用 1.28.0 创建或维护的任何 Operator 项目中手动执行更新步骤。

5.5.4.1. 为 Operator SDK 1.31.0 更新基于 Helm 的 Operator 项目

以下流程更新了基于 Helm 的 Operator 项目，以便与 1.31.0 兼容。

先决条件

- 已安装 operator SDK 1.31.0
- 使用 Operator SDK 1.28.0 创建或维护的 Operator 项目

流程

1. 编辑 Operator 的 Dockerfile，将 Helm Operator 版本更新至 1.31.0，如下例所示：

Dockerfile 示例

```
FROM quay.io/operator-framework/helm-operator:v1.31.0 1
```

- 1** 将 Helm Operator 版本从 **1.28.0** 更新至 **1.31.0**

2. 编辑 Operator 项目的 makefile，将 Operator SDK 更新至 1.31.0，如下例所示：

makefile 示例

```
# Set the Operator SDK version to use. By default, what is installed on the system is used.
# This is useful for CI or a project to utilize a specific version of the operator-sdk toolkit.
OPERATOR_SDK_VERSION ?= v1.31.0 1
```

- 1** 将版本从 **1.28.0** 更改为 **1.31.0**。

3. 如果使用自定义服务帐户进行部署，请定义以下角色来需要对 secret 资源进行监视操作，如下例所示：

config/rbac/role.yaml 文件示例

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: <operator_name>-admin
subjects:
- kind: ServiceAccount
  name: <operator_name>
  namespace: <operator_namespace>
```



```

roleRef:
  kind: ClusterRole
  name: cluster-admin
  apiGroup: ""
rules: ❶
  - apiGroups:
    - ""
  resources:
    - secrets
  verbs:
    - watch

```

❶ 添加 **rules** 小节，为您的 secret 资源创建监视操作。

5.5.4.2. 其他资源

- [将软件包清单项目迁移到捆绑包格式](#)
- [为 Operator SDK 1.16.0 升级项目](#)
- [升级 Operator SDK v1.10.1 的项目](#)
- [针对 Operator SDK v1.8.0 升级项目](#)

5.5.5. Operator SDK 中的 Helm 支持

5.5.5.1. 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
```

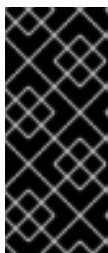
不要求使用 Helm CLI 或安装 Tiller；基于 Helm 的 Operator 会从 Helm 项目中导入代码。您要做的只是运行一个 Operator 实例，并使用自定义资源定义 (CRD) 注册 CR。因其遵循 RBAC，所以可以更容易防止生产环境改变。

5.5.6. Hybrid Helm Operators Operator SDK 教程

与基于 Go 和基于 Ansible 的 Operator 相比（它们已实现 [Operator maturity model](#) 中的 Auto Pilot capability (level V)），Operator SDK 中对标准的基于 Helm 的 Operator 支持有一定限制。

Hybrid Helm Operator 通过 Go API 增强了基于 Helm 的现有支持能力。借助此混合的 Helm 和 Go 方法，Operator SDK 可让 Operator 作者使用以下流程：

- 在与 Helm 相同的项目中为 Go API 生成一个默认的结构或 (*scaffold*)。
- 通过 Hybrid Helm Operator 提供的库，在项目的 `main.go` 文件中配置 Helm reconciler。



重要

Hybrid Helm Operator 只是一个技术预览功能。技术预览功能不受红帽产品服务等级协议 (SLA) 支持，且功能可能并不完整。红帽不推荐在生产环境中使用它们。这些技术预览功能可以使用户提早试用新的功能，并有机会在开发阶段提供反馈意见。

有关红帽技术预览功能支持范围的更多信息，请参阅[技术预览功能支持范围](#)。

本教程介绍了使用 Hybrid Helm Operator 的以下过程：

- 如果不存在，使用 Helm chart 创建一个 **Memcached**
- 确保部署大小与 **Memcached** 自定义资源 (CR) spec 指定的大小相同
- 使用 Go API 创建 **MemcachedBackup** 部署

5.5.6.1. 先决条件

- 已安装 operator SDK CLI
- 已安装 OpenShift CLI (`oc`) 4.15+
- 使用具有 **cluster-admin** 权限的 `oc` 登录到 OpenShift Container Platform 4.15 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

其他资源

- [安装 Operator SDK CLI](#)
- [OpenShift CLI 入门](#)

5.5.6.2. 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/github.com/example/memcached-operator
```

2. 进入该目录：

```
$ cd $HOME/github.com/example/memcached-operator
```

3. 运行 **operator-sdk init** 命令以初始化项目。使用域 **example.com**，所有 API 组都为 **<group>.example.com**：

```
$ operator-sdk init \
  --plugins=hybrid.helm.sdk.operatorframework.io \
  --project-version="3" \
  --domain example.com \
  --repo=github.com/example/memcached-operator
```

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 **config/rbac/role.yaml** 文件中生成的规则是否满足您的 Operator 的权限要求。

其他资源

- 此流程会创建一个与 Helm 和 Go API 兼容的项目结构。要了解更多有关项目目录结构的信息，请参阅[项目布局](#)。

5.5.6.3. 创建 Helm 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 Helm API：

```
$ operator-sdk create api \
  --plugins helm.sdk.operatorframework.io/v1 \
  --group cache \
  --version v1 \
  --kind Memcached
```

注意

此流程还将您的 Operator 项目配置为监视 API 版本 **v1** 的 **Memcached** 资源，并构建 boilerplate Helm Chart。除了通过由 Operator SDK 构建的 boilerplate Helm chart 创建项目，也可以使用一个您的本地文件系统或远程 chart 仓库中的现有的 chart 进行创建。

有关基于现有或新 chart 创建 Helm API 的详情和示例，请运行以下命令：

```
$ operator-sdk create api --plugins helm.sdk.operatorframework.io/v1 --help
```



其他资源

- [现有 Helm chart](#)

5.5.6.3.1. Helm API 的 Operator 逻辑

默认情况下，您的构建 Operator 项目会监视 **Memcached** 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。

例 5.2. watches.yaml 文件示例

```
# Use the 'create api' subcommand to add watches to this file.
- group: cache.my.domain
  version: v1
  kind: Memcached
  chart: helm-charts/memcached
  #+kubebuilder:scaffold:watch
```

其他资源

- 有关通过 chart 自定义 Helm Operator 逻辑的详细文档，请参阅[了解 Operator 逻辑](#)。

5.5.6.3.2. 使用所提供的库 API 的自定义 Helm reconciler 配置

现有基于 Helm 的 Operator 的缺点是无法配置 Helm reconciler，因为它是从用户中提取的。对于基于 Helm 的 Operator 可访问 Seam 无升级功能（级别 II 及更新的版本），它重复使用已存在的 Helm Chart，在 Go 和 Helm Operator 类型间的混合会添加值。

helm-operator-plugins 库中提供的 API 允许 Operator 作者进行以下配置：

- 根据集群状态自定义值映射
- 通过配置协调器的事件记录程序在特定事件中执行代码
- 自定义协调器的日志记录器
- 设置 **Install**, **Upgrade**, 和 **Uninstall** 注解，以便根据协调器监视的自定义资源中的注解启用 Helm 的操作
- 配置协调器以使用 **Pre** 和 **Post** hook 运行

以上对协调器的配置可在 **main.go** 文件中完成：

main.go 文件示例

```
// Operator's main.go
// With the help of helpers provided in the library, the reconciler can be
// configured here before starting the controller with this reconciler.
reconciler := reconciler.New(
  reconciler.WithChart(*chart),
  reconciler.WithGroupVersionKind(gvk),
)
```

```
if err := reconciler.SetupWithManager(mgr); err != nil {
    panic(fmt.Sprintf("unable to create reconciler: %s", err))
}
```

5.5.6.4. 创建 Go API

使用 Operator SDK CLI 创建 Go API。

流程

1. 运行以下命令，使用组 **cache**、版本 **v1** 和 kind **MemcachedBackup** 创建 Go API：

```
$ operator-sdk create api \
  --group=cache \
  --version v1 \
  --kind MemcachedBackup \
  --resource \
  --controller \
  --plugins=go/v3
```

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
```

此流程在 **api/v1/memcachedbackup_types.go** 和 **controllers/memcachedbackup_controller.go** 中生成 **MemcachedBackup** 资源 API。

5.5.6.4.1. 定义 API

定义 **MemcachedBackup** 自定义资源(CR)的 API。

通过定义 **MemcachedBackup** 类型来表示此 Go API，它具有 **MemcachedBackupSpec.Size** 字段，用于设置要部署的 Memcached 备份实例(CR)数量，以及用于存储 CR 的 pod 名称的 **MemcachedBackupStatus.Nodes** 字段。



注意

Node 字段用于说明 **Status** 字段的一个示例。

流程

1. 通过修改 **api/v1/memcachedbackup_types.go** 文件中的 Go 类型定义来定义 **MemcachedBackup** CR 的 API，使其具有以下 **spec** 和 **status**：

例 5.3. api/v1/memcachedbackup_types.go 文件示例

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

```

//+kubebuilder:validation:Minimum=0
// Size is the size of the memcached deployment
Size int32 `json:"size"`
}

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

3. 使用 `spec` 和 `status` 字段和 CRD 验证标记定义 API 后，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.my.domain_memcachedbackups.yaml` 文件中生成 CRD 清单。

5.5.6.4.2. 控制器实现

本教程中的控制器执行以下操作：

- 如果尚无 **Memcached** 部署，请创建一个。
- 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。

有关如何配置控制器来执行上述操作的详情，请参考使用基于 Go 的 Operator 在 Operator SDK 指南中的 [实施控制器](#)。

5.5.6.4.3. main.go 的不同

对于基于 Go 的标准 Operator 和混合 Helm Operator，`main.go` 文件处理构建 Go API 的 **Manager** 程序初始化和运行。但是，对于 Hybrid Helm Operator，`main.go` 文件也会公开加载 `watches.yaml` 文件和配置 Helm reconciler 的逻辑。

例 5.4. main.go 文件示例

```

...
for _, w := range ws {
// Register controller with the factory

```

```

reconcilePeriod := defaultReconcilePeriod
if w.ReconcilePeriod != nil {
    reconcilePeriod = w.ReconcilePeriod.Duration
}

maxConcurrentReconciles := defaultMaxConcurrentReconciles
if w.MaxConcurrentReconciles != nil {
    maxConcurrentReconciles = *w.MaxConcurrentReconciles
}

r, err := reconciler.New(
    reconciler.WithChart(*w.Chart),
    reconciler.WithGroupVersionKind(w.GroupVersionKind),
    reconciler.WithOverrideValues(w.OverrideValues),
    reconciler.SkipDependentWatches(w.WatchDependentResources != nil &&
!*w.WatchDependentResources),
    reconciler.WithMaxConcurrentReconciles(maxConcurrentReconciles),
    reconciler.WithReconcilePeriod(reconcilePeriod),
    reconciler.WithInstallAnnotations(annotation.DefaultInstallAnnotations...),
    reconciler.WithUpgradeAnnotations(annotation.DefaultUpgradeAnnotations...),
    reconciler.WithUninstallAnnotations(annotation.DefaultUninstallAnnotations...),
)
...

```

管理器使用 **Helm** 和 **Go** 协调器初始化：

例 5.5. Helm 和 Go reconcilers 示例

```

...
// Setup manager with Go API
if err = (&controllers.MemcachedBackupReconciler{
    Client: mgr.GetClient(),
    Scheme: mgr.GetScheme(),
}).SetupWithManager(mgr); err != nil {
    setupLog.Error(err, "unable to create controller", "controller", "MemcachedBackup")
    os.Exit(1)
}

...
// Setup manager with Helm API
for _, w := range ws {

    ...
    if err := r.SetupWithManager(mgr); err != nil {
        setupLog.Error(err, "unable to create controller", "controller", "Helm")
        os.Exit(1)
    }
    setupLog.Info("configured watch", "gvk", w.GroupVersionKind, "chartPath", w.ChartPath,
    "maxConcurrentReconciles", maxConcurrentReconciles, "reconcilePeriod", reconcilePeriod)
}

// Start the manager
if err := mgr.Start(ctrl.SetupSignalHandler()); err != nil {

```

```

    setupLog.Error(err, "problem running manager")
    os.Exit(1)
}

```

5.5.6.4.4. 权限和 RBAC 清单

控制器需要特定的基于角色的访问控制(RBAC)权限与它管理的资源交互。对于 Go API，它们通过 RBAC 标记来指定，如基于 Go 的标准 Operator 的 Operator SDK 教程所示。

对于 Helm API，在 **roles.yaml** 中默认构建权限。目前，由于在构建 Go API 时存在一个已知问题，Helm API 的权限会被覆盖。因此，请确保 **roles.yaml** 中定义的权限与您的要求匹配。



注意

这个已知问题会在 <https://github.com/operator-framework/helm-operator-plugins/issues/142> 中进行跟踪。

以下是 Memcached Operator 的 **role.yaml** 示例：

例 5.6. Helm 和 Go reconcilers 示例

```

---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: manager-role
rules:
- apiGroups:
  - ""
  resources:
  - namespaces
  verbs:
  - get
- apiGroups:
  - apps
  resources:
  - deployments
  - daemonsets
  - replicaset
  - statefulsets
  verbs:
  - create
  - delete
  - get
  - list
  - patch
  - update
  - watch
- apiGroups:
  - cache.my.domain
  resources:
  - memcachedbackups
  verbs:

```



```
- create
- delete
- get
- list
- patch
- update
- watch
- apiGroups:
- cache.my.domain
resources:
- memcachedbackups/finalizers
verbs:
- create
- delete
- get
- list
- patch
- update
- watch
- apiGroups:
- ""
resources:
- pods
- services
- services/finalizers
- endpoints
- persistentvolumeclaims
- events
- configmaps
- secrets
- serviceaccounts
verbs:
- create
- delete
- get
- list
- patch
- update
- watch
- apiGroups:
- cache.my.domain
resources:
- memcachedbackups/status
verbs:
- get
- patch
- update
- apiGroups:
- policy
resources:
- events
- poddisruptionbudgets
verbs:
- create
- delete
- get
```

```
- list
- patch
- update
- watch
- apiGroups:
- cache.my.domain
resources:
- memcacheds
- memcacheds/status
- memcacheds/finalizers
verbs:
- create
- delete
- get
- list
- patch
- update
- watch
```

其他资源

- [基于 Go 的 Operator 的 RBAC 标记](#)

5.5.6.5. 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
```

5.5.6.6. 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 需要为 **go build** 明确引用 **GOARCH=amd64**。这可以在非 AMD64 构架中使用 **GOARCH=\$TARGETARCH**。Docker 自动将环境变量设置为 **-platform** 指定的值。对于 Buildah，需要使用 **-build-arg** 来实现这一目的。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. 运行以下命令验证 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
```

5.5.6.7. 创建自定义资源

安装 Operator 后，您可以通过创建由 Operator 在集群中提供的自定义资源(CR)来测试它。

流程

1. 切换到安装 Operator 的命名空间：

```
$ oc project <project_name>-system
```

2. 通过将 **replicaCount** 字段更新为 **3**，更新 **config/samples/cache_v1_memcached.yaml** 文件中的 **Memcached** CR 清单示例：

例 5.7. config/samples/cache_v1_memcached.yaml 文件示例

```
apiVersion: cache.my.domain/v1
kind: Memcached
metadata:
  name: memcached-sample
```

```

spec:
  # Default values copied from <project_dir>/helm-charts/memcached/values.yaml
  affinity: {}
  autoscaling:
    enabled: false
    maxReplicas: 100
    minReplicas: 1
    targetCPUUtilizationPercentage: 80
  fullnameOverride: ""
  image:
    pullPolicy: IfNotPresent
    repository: nginx
    tag: ""
  imagePullSecrets: []
  ingress:
    annotations: {}
    className: ""
    enabled: false
    hosts:
      - host: chart-example.local
    paths:
      - path: /
        pathType: ImplementationSpecific
    tls: []
  nameOverride: ""
  nodeSelector: {}
  podAnnotations: {}
  podSecurityContext: {}
  replicaCount: 3
  resources: {}
  securityContext: {}
  service:
    port: 80
    type: ClusterIP
  serviceAccount:
    annotations: {}
    create: true
    name: ""
  tolerations: []

```

3. 创建 **Memcached** 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 pods
```

输出示例

NAME	READY	STATUS	RESTARTS	AGE
memcached-sample-6fd7c98d8-7dqdr	1/1	Running	0	18m
memcached-sample-6fd7c98d8-g5k7v	1/1	Running	0	18m

```
memcached-sample-6fd7c98d8-m7vn7 1/1 Running 0 18m
```

- 通过将 **size** 更新至 **2**，更新 **config/samples/cache_v1_memcachedbackup.yaml** 文件中的 **MemcachedBackup** CR 清单示例：

例 5.8. config/samples/cache_v1_memcachedbackup.yaml 文件示例

```
apiVersion: cache.my.domain/v1
kind: MemcachedBackup
metadata:
  name: memcachedbackup-sample
spec:
  size: 2
```

- 创建 **MemcachedBackup** CR：

```
$ oc apply -f config/samples/cache_v1_memcachedbackup.yaml
```

- 确保 **memcachedbackup** pod 的数量与 CR 中指定的相同：

```
$ oc get pods
```

输出示例

```
NAME                                READY STATUS RESTARTS AGE
memcachedbackup-sample-8649699989-4bbzg 1/1 Running 0 22m
memcachedbackup-sample-8649699989-mq6mx 1/1 Running 0 22m
```

- 您可以更新以上每个 CR 中的 **spec**，然后再次应用它们。控制器再次进行协调，并确保按照相应 CR 的 **spec** 中指定 pod 的大小。
- 清理本教程中创建的资源：

- 删除 **Memcached** 资源：

```
$ oc delete -f config/samples/cache_v1_memcached.yaml
```

- 删除 **MemcachedBackup** 资源：

```
$ oc delete -f config/samples/cache_v1_memcachedbackup.yaml
```

- 如果使用 **make deploy** 命令来测试 Operator，请运行以下命令：

```
$ make undeploy
```

5.5.6.8. 项目布局

混合 Helm Operator 的构建是自定义的，以匹配 Helm 和 Go API。

文件/文件夹	用途
Docker	容器引擎使用 make docker-build 命令构建 Operator 镜像的说明。
Makefile	构建包含帮助程序目标的文件，以帮助您操作项目。
PROJECT	包含 Operator 元数据信息的 YAML 文件。代表项目的配置，用于跟踪 CLI 和插件的有用信息。
bin/	包含有用的二进制文件，如管理器(manager)用于在本地运行您的项目，以及用于项目配置的 kustomize 工具。
config/	<p>包含配置文件，其中包括所有 Kustomize 清单，以便在集群上启动 Operator 项目。插件可能会使用它来提供功能。例如，对于 Operator SDK 可以帮助创建 Operator 捆绑包，CLI 会查找在这个目录中构建的 CRD 和 CR。</p> <p>config/crd/ 包含自定义资源定义(CRD)。</p> <p>config/default/ 包含一个 Kustomize 基础，用于在标准配置中启动控制器。</p> <p>config/manager/ 包含清单，作为集群中的 pod 启动 Operator 项目。</p> <p>config/manifests/ 包含要在 bundle/ 目录中生成 OLM 清单的基础。</p> <p>config/prometheus/ 包含启用项目为 Prometheus 提供指标（如 ServiceMonitor 资源）所需的清单。</p> <p>config/scorecard/ 包含允许使用 scorecard 工具测试项目所需的清单。</p> <p>config/rbac/ 包含运行项目所需的 RBAC 权限。</p> <p>config/samples/ 包含自定义资源的示例。</p>
api/	包含 Go API 定义。
controllers/	包含 Go API 的控制器。
hack/	包含实用程序文件，如用于构建项目文件的许可证标头的文件。
main.go	Operator 的主要程序。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。
helm-charts/	包含 Helm chart，可以使用带有 Helm 插件的 create api 命令来指定。
watches.yaml	包含 group/version/kind (GVK) 和 Helm Chart 位置。用于配置 Helm 监视。

5.5.7. 为较新的 Operator SDK 版本更新基于 Helm 的混合项目

OpenShift Container Platform 4.15 支持 Operator SDK 1.31.0。如果您已在工作站上安装了 1.28.0 CLI，您可以通过[安装最新版本](#)将 CLI 更新至 1.31.0。

但是，要确保现有 Operator 项目保持与 Operator SDK 1.31.0 的兼容性，需要执行更新的相关步骤才能解决从 1.28.0 以来引入的相关破坏更改。您必须在之前使用 1.28.0 创建或维护的任何 Operator 项目中手动执行更新步骤。

5.5.7.1. 为 Operator SDK 1.31.0 更新基于 Helm 的 Operator 项目

以下流程更新了基于 Helm 的现有 Operator 项目，以便与 1.31.0 兼容。

先决条件

- 已安装 operator SDK 1.31.0
- 使用 Operator SDK 1.28.0 创建或维护的 Operator 项目

流程

- 编辑 Operator 项目的 makefile，将 Operator SDK 版本更新至 1.31.0，如下例所示：

makefile 示例

```
# Set the Operator SDK version to use. By default, what is installed on the system is used.
# This is useful for CI or a project to utilize a specific version of the operator-sdk toolkit.
OPERATOR_SDK_VERSION ?= v1.31.0 1
```

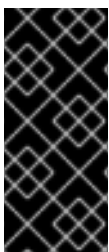
- 1** 将版本从 **1.28.0** 更改为 **1.31.0**。

5.5.7.2. 其他资源

- [将软件包清单项目迁移到捆绑包格式](#)
- [为 Operator SDK 1.16.0 升级项目](#)
- [升级 Operator SDK v1.10.1 的项目](#)
- [针对 Operator SDK v1.8.0 升级项目](#)

5.6. 基于 JAVA 的 OPERATOR

5.6.1. 基于 Java 的 Operator 的 Operator SDK 入门



重要

基于 Java 的 Operator SDK 只是一个技术预览功能。技术预览功能不受红帽产品服务等级协议（SLA）支持，且功能可能并不完整。红帽不推荐在生产环境中使用它们。这些技术预览功能可以使用户提早试用新的功能，并有机会在开发阶段提供反馈意见。

有关红帽技术预览功能支持范围的更多信息，请参阅[技术预览功能支持范围](#)。

如需演示使用 Operator SDK 提供的工具和库来设置和运行基于 Java 的 Operator 的基本知识，Operator 开发人员可以为 Memcached 构建基于 Java 的 Operator 示例，一个分布式键值存储，并将它部署到集群中。

5.6.1.1. 先决条件

- 已安装 operator SDK CLI
- 已安装 OpenShift CLI (**oc**) 4.15+
- [Java 11+](#)
- [Maven 3.6.3+](#)
- 使用具有 **cluster-admin** 权限的 **oc** 登录到 OpenShift Container Platform 4.15 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

其他资源

- [安装 Operator SDK CLI](#)
- [OpenShift CLI 入门](#)

5.6.1.2. 创建并部署基于 Java 的 Operator

您可以使用 Operator SDK 为 Memcached 构建和部署简单的基于 Java 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 **quarkus** 插件运行 **operator-sdk init** 命令以初始化项目：

```
$ operator-sdk init \  
  --plugins=quarkus \  
  --domain=example.com \  
  --project-name=memcached-operator
```

2. 创建 API。

创建简单的 Memcached API：

```
$ operator-sdk create api \  
  --plugins quarkus \  
  --group cache \  
  --name memcached
```



```
--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. 删除 CR。

运行以下命令来删除 CR：

```
$ oc delete -f config/samples/cache_v1_memcached.yaml -n memcached-operator-system
```

7. 清理。

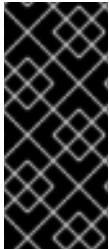
运行以下命令清理在此流程中创建的资源：

```
$ make undeploy
```

5.6.1.3. 后续步骤

- 如需更深入地了解如何构建基于 Java 的 Operator，请参阅[基于 Java 的 Operator SDK 指南](#)。

5.6.2. 基于 Java 的 Operator 的 operator SDK 指南



重要

基于 Java 的 Operator SDK 只是一个技术预览功能。技术预览功能不受红帽产品服务等级协议 (SLA) 支持，且功能可能并不完整。红帽不推荐在生产环境中使用它们。这些技术预览功能可以使用户提早试用新的功能，并有机会在开发阶段提供反馈意见。

有关红帽技术预览功能支持范围的更多信息，请参阅[技术预览功能支持范围](#)。

Operator 开发人员可以利用 Operator SDK 中的 Java 编程语言支持，为 Memcached 构建基于 Java 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 **java-operator-sdk** library 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 Java 的 Operator 开始使用 Operator SDK](#)的内容更详细。

5.6.2.1. 先决条件

- 已安装 operator SDK CLI
- 已安装 OpenShift CLI (**oc**) 4.15+
- [Java](#) 11+
- [Maven](#) 3.6.3+
- 使用具有 **cluster-admin** 权限的 **oc** 登录到 OpenShift Container Platform 4.15 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

其他资源

- [安装 Operator SDK CLI](#)
- [OpenShift CLI 入门](#)

5.6.2.2. 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 **quarkus** 插件运行 **operator-sdk init** 命令以初始化项目：

```
$ operator-sdk init \
  --plugins=quarkus \
  --domain=example.com \
  --project-name=memcached-operator
```

5.6.2.2.1. 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，并注意到项目类型为 Java。例如：

```
domain: example.com
layout:
- quarkus.javaoperatorsdk.io/v1-alpha
projectName: memcached-operator
version: "3"
```

5.6.2.3. 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. 运行以下命令来创建 API：

```
$ operator-sdk create api \
  --plugins=quarkus \ ①
  --group=cache \ ②
  --version=v1 \ ③
  --kind=Memcached ④
```

- ① 将插件标志设置为 **quarkus**。
- ② 将 group 标志设置为 **cache**。
- ③ 将 version 标志设置为 **v1**。
- ④ 将 kind 标志设置为 **Memcached**。

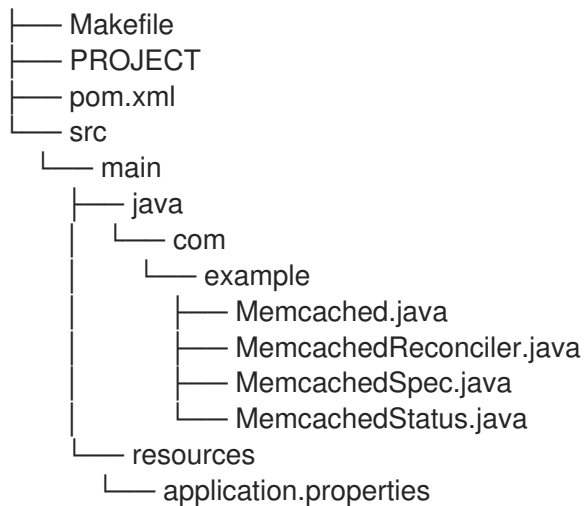
验证

1. 运行 **tree** 命令来查看文件结构：

```
$ tree
```

输出示例

```
.
```



6 directories, 8 files

5.6.2.3.1. 定义 API

定义 **Memcached** 自定义资源 (CR) 的 API。

流程

- 编辑作为 **create api** 进程一部分生成的以下文件：
 - a. 更新 **MemcachedSpec.java** 文件中的以下属性，以定义 **Memcached** CR 的所需状态：

```

public class MemcachedSpec {

    private Integer size;

    public Integer getSize() {
        return size;
    }

    public void setSize(Integer size) {
        this.size = size;
    }
}

```

- b. 更新 **MemcachedStatus.java** 文件中的以下属性，以定义 **Memcached** CR 的观察状态：



注意

以下示例中显示了一个 Node 状态字段。建议您在实践中使用典型的属性。

```

import java.util.ArrayList;
import java.util.List;

public class MemcachedStatus {

    // Add Status information here
    // Nodes are the names of the memcached pods

```

```

private List<String> nodes;

public List<String> getNodes() {
    if (nodes == null) {
        nodes = new ArrayList<>();
    }
    return nodes;
}

public void setNodes(List<String> nodes) {
    this.nodes = nodes;
}
}

```

- c. 更新 **Memcached.java** 文件，以定义 Memcached API 的 Schema，该 API 扩展至 **MemcachedSpec.java** 和 **MemcachedStatus.java** 文件。

```

@Version("v1")
@Group("cache.example.com")
public class Memcached extends CustomResource<MemcachedSpec,
MemcachedStatus> implements Namespaced {}

```

5.6.2.3.2. 生成 CRD 清单

在使用 **MemcachedSpec** 和 **MemcachedStatus** 文件定义 API 后，您可以生成 CRD 清单。

流程

- 从 **memcached-operator** 目录运行以下命令，以生成 CRD：

```
$ mvn clean install
```

验证

- 验证 **target/kubernetes/memcacheds.cache.example.com-v1.yml** 文件中的 CRD 内容，如下例所示：

```
$ cat target/kubernetes/memcacheds.cache.example.com-v1.yaml
```

输出示例

```

# Generated by Fabric8 CRDGenerator, manual edits might get overwritten!
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: memcacheds.cache.example.com
spec:
  group: cache.example.com
  names:
    kind: Memcached
    plural: memcacheds
    singular: memcached
  scope: Namespaced

```

```

versions:
- name: v1
  schema:
    openAPIV3Schema:
      properties:
        spec:
          properties:
            size:
              type: integer
          type: object
        status:
          properties:
            nodes:
              items:
                type: string
              type: array
            type: object
          type: object
      served: true
      storage: true
      subresources:
        status: {}

```

5.6.2.3.3. 创建自定义资源

生成 CRD 清单后，您可以创建自定义资源(CR)。

流程

- 创建名为 **memcached-sample.yaml** 的 Memcached CR：

```

apiVersion: cache.example.com/v1
kind: Memcached
metadata:
  name: memcached-sample
spec:
  # Add spec fields here
  size: 1

```

5.6.2.4. 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

1. 将以下依赖项附加到 **pom.xml** 文件中：

```

<dependency>
  <groupId>commons-collections</groupId>
  <artifactId>commons-collections</artifactId>
  <version>3.2.2</version>
</dependency>

```

2. 在本例中，将生成的控制器文件 **MemcachedReconciler.java** 替换为以下示例实现：

例 5.9. MemcachedReconciler.java 示例

```

package com.example;

import io.fabric8.kubernetes.client.KubernetesClient;
import io.javaoperatorsdk.operator.api.reconciler.Context;
import io.javaoperatorsdk.operator.api.reconciler.Reconciler;
import io.javaoperatorsdk.operator.api.reconciler.UpdateControl;
import io.fabric8.kubernetes.api.model.ContainerBuilder;
import io.fabric8.kubernetes.api.model.ContainerPortBuilder;
import io.fabric8.kubernetes.api.model.LabelSelectorBuilder;
import io.fabric8.kubernetes.api.model.ObjectMetaBuilder;
import io.fabric8.kubernetes.api.model.OwnerReferenceBuilder;
import io.fabric8.kubernetes.api.model.Pod;
import io.fabric8.kubernetes.api.model.PodSpecBuilder;
import io.fabric8.kubernetes.api.model.PodTemplateSpecBuilder;
import io.fabric8.kubernetes.api.model.apps.Deployment;
import io.fabric8.kubernetes.api.model.apps.DeploymentBuilder;
import io.fabric8.kubernetes.api.model.apps.DeploymentSpecBuilder;
import org.apache.commons.collections.CollectionUtils;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class MemcachedReconciler implements Reconciler<Memcached> {
    private final KubernetesClient client;

    public MemcachedReconciler(KubernetesClient client) {
        this.client = client;
    }

    // TODO Fill in the rest of the reconciler

    @Override
    public UpdateControl<Memcached> reconcile(
        Memcached resource, Context context) {
        // TODO: fill in logic
        Deployment deployment = client.apps()
            .deployments()
            .inNamespace(resource.getMetadata().getNamespace())
            .withName(resource.getMetadata().getName())
            .get();

        if (deployment == null) {
            Deployment newDeployment = createMemcachedDeployment(resource);
            client.apps().deployments().create(newDeployment);
            return UpdateControl.noUpdate();
        }

        int currentReplicas = deployment.getSpec().getReplicas();
        int requiredReplicas = resource.getSpec().getSize();

        if (currentReplicas != requiredReplicas) {
            deployment.getSpec().setReplicas(requiredReplicas);
            client.apps().deployments().createOrReplace(deployment);
        }
    }

```

```

        return UpdateControl.noUpdate();
    }

    List<Pod> pods = client.pods()
        .inNamespace(resource.getMetadata().getNamespace())
        .withLabels(labelsForMemcached(resource))
        .list()
        .getItems();

    List<String> podNames =
        pods.stream().map(p -> p.getMetadata().getName()).collect(Collectors.toList());

    if (resource.getStatus() == null
        || !CollectionUtils.isEqualCollection(podNames,
        resource.getStatus().getNodes())) {
        if (resource.getStatus() == null) resource.setStatus(new MemcachedStatus());
        resource.getStatus().setNodes(podNames);
        return UpdateControl.updateResource(resource);
    }

    return UpdateControl.noUpdate();
}

private Map<String, String> labelsForMemcached(Memcached m) {
    Map<String, String> labels = new HashMap<>();
    labels.put("app", "memcached");
    labels.put("memcached_cr", m.getMetadata().getName());
    return labels;
}

private Deployment createMemcachedDeployment(Memcached m) {
    Deployment deployment = new DeploymentBuilder()
        .withMetadata(
            new ObjectMetaBuilder()
                .withName(m.getMetadata().getName())
                .withNamespace(m.getMetadata().getNamespace())
                .build()
        )
        .withSpec(
            new DeploymentSpecBuilder()
                .withReplicas(m.getSpec().getSize())
                .withSelector(
                    new
LabelSelectorBuilder().withMatchLabels(labelsForMemcached(m)).build()
                )
                .withTemplate(
                    new PodTemplateSpecBuilder()
                        .withMetadata(
                            new ObjectMetaBuilder().withLabels(labelsForMemcached(m)).build()
                        )
                        .withSpec(
                            new PodSpecBuilder()
                                .withContainers(
                                    new ContainerBuilder()
                                        .withImage("memcached:1.4.36-alpine")
                                        .withName("memcached")
                                        .withCommand("memcached", "-m=64", "-o", "modern", "-v")
                                        .withPorts(

```



```

        new ContainerPortBuilder()
            .withContainerPort(11211)
            .withName("memcached")
            .build()
        .build()
    .build()
    .build()
    .build();
deployment.addOwnerReference(m);
return deployment;
}
}

```

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，请创建一个。
- 确保部署大小与 **Memcached** CR spec 指定的大小匹配。
- 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来直接进入[运行 Operator](#)。

5.6.2.4.1. 协调循环

1. 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 **Deployment** 参数，如下例所示：

```

Deployment deployment = client.apps()
    .deployments()
    .inNamespace(resource.getMetadata().getNamespace())
    .withName(resource.getMetadata().getName())
    .get();

```

2. 如以下示例所示，如果 **Deployment** 是 **null**，则需要创建部署。创建 **Deployment** 后，您可以确定是否需要协调。如果不需要协调，返回 **UpdateControl.noUpdate()** 的值，否则返回 **'UpdateControl.updateStatus(resource)** 的值：

```

if (deployment == null) {
    Deployment newDeployment = createMemcachedDeployment(resource);
    client.apps().deployments().create(newDeployment);
    return UpdateControl.noUpdate();
}

```

3. 获取 **Deployment** 后，获取当前和所需的副本，如下例所示：

```

int currentReplicas = deployment.getSpec().getReplicas();
int requiredReplicas = resource.getSpec().getSize();

```

4. 如果 **currentReplicas** 与 **requiredReplicas** 不匹配，您必须更新 **Deployment**，如下例所示：

```

if (currentReplicas != requiredReplicas) {
    deployment.getSpec().setReplicas(requiredReplicas);
    client.apps().deployments().createOrReplace(deployment);
    return UpdateControl.noUpdate();
}

```

5. 以下示例演示了如何获取 pod 列表及其名称：

```

List<Pod> pods = client.pods()
    .inNamespace(resource.getMetadata().getNamespace())
    .withLabels(labelsForMemcached(resource))
    .list()
    .getItems();

List<String> podNames =
    pods.stream().map(p -> p.getMetadata().getName()).collect(Collectors.toList());

```

6. 检查资源是否已创建并使用 Memcached 资源验证 pod 名称。如果这两个条件之一都存在不匹配，请执行协调，如下例所示：

```

if (resource.getStatus() == null
    || !CollectionUtils.isEqualCollection(podNames, resource.getStatus().getNodes())) {
    if (resource.getStatus() == null) resource.setStatus(new MemcachedStatus());
    resource.getStatus().setNodes(podNames);
    return UpdateControl.updateResource(resource);
}

```

5.6.2.4.2. 定义 labelsForMemcached

LabelsForMemcached 是一个返回要附加到资源的标签映射的工具：

```

private Map<String, String> labelsForMemcached(Memcached m) {
    Map<String, String> labels = new HashMap<>();
    labels.put("app", "memcached");
    labels.put("memcached_cr", m.getMetadata().getName());
    return labels;
}

```

5.6.2.4.3. 定义 createMemcachedDeployment

createMemcachedDeployment 方法使用 **fabric8 DeploymentBuilder** 类：

```

private Deployment createMemcachedDeployment(Memcached m) {
    Deployment deployment = new DeploymentBuilder()
        .withMetadata(
            new ObjectMetaBuilder()
                .withName(m.getMetadata().getName())
                .withNamespace(m.getMetadata().getNamespace())
                .build())
        .withSpec(
            new DeploymentSpecBuilder()
                .withReplicas(m.getSpec().getSize())
                .withSelector(

```

```

        new LabelSelectorBuilder().withMatchLabels(labelsForMemcached(m)).build())
    .withTemplate(
        new PodTemplateSpecBuilder()
            .withMetadata(
                new ObjectMetaBuilder().withLabels(labelsForMemcached(m)).build())
            .withSpec(
                new PodSpecBuilder()
                    .withContainers(
                        new ContainerBuilder()
                            .withImage("memcached:1.4.36-alpine")
                            .withName("memcached")
                            .withCommand("memcached", "-m=64", "-o", "modern", "-v")
                            .withPorts(
                                new ContainerPortBuilder()
                                    .withContainerPort(11211)
                                    .withName("memcached")
                                    .build())
                                .build())
                            .build())
                    .build())
            .build())
        .build();
    deployment.addOwnerReference(m);
    return deployment;
}

```

5.6.2.5. 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。

5.6.2.5.1. 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

1. 运行以下命令来编译 Operator :

```
$ mvn clean install
```

输出示例

```

[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 11.193 s
[INFO] Finished at: 2021-05-26T12:16:54-04:00
[INFO] -----

```

- 运行以下命令，将 CRD 安装到 default 命名空间：

```
$ oc apply -f target/kubernetes/memcacheds.cache.example.com-v1.yml
```

输出示例

```
customresourcedefinition.apiextensions.k8s.io/memcacheds.cache.example.com created
```

- 如下例所示，名为 **rbac.yaml** 的文件：

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: memcached-operator-admin
subjects:
- kind: ServiceAccount
  name: memcached-quarkus-operator-operator
  namespace: <operator_namespace>
roleRef:
  kind: ClusterRole
  name: cluster-admin
  apiGroup: ""
```

- 运行以下命令通过应用 **rbac.yaml** 文件向 **memcached-quarkus-operator-operator** 授予 **cluster-admin** 权限：

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

- 输入以下命令来运行 Operator：

```
$ java -jar target/quarkus-app/quarkus-run.jar
```



注意

java 命令将运行 Operator 并保持运行，直到您结束进程。您需要另一个终端来完成这些命令的其余部分。

- 使用以下命令应用 **memcached-sample.yaml** 文件：

```
$ kubectl apply -f memcached-sample.yaml
```

输出示例

```
memcached.cache.example.com/memcached-sample created
```

验证

- 运行以下命令，确认 pod 已启动：

```
$ oc get all
```

输出示例

```
NAME                                READY STATUS RESTARTS AGE
pod/memcached-sample-6c765df685-mfqnz    1/1   Running 0      18s
```

5.6.2.5.2. 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 需要为 **go build** 明确引用 **GOARCH=amd64**。这可以在非 AMD64 构架中使用 **GOARCH=\$TARGETARCH**。Docker 自动将环境变量设置为 **-platform** 指定的值。对于 Buildah，需要使用 **-build-arg** 来实现这一目的。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，将 CRD 安装到 default 命名空间：

```
$ oc apply -f target/kubernetes/memcacheds.cache.example.com-v1.yml
```

输出示例

```
customresourcedefinition.apiextensions.k8s.io/memcacheds.cache.example.com created
```

3. 如下例所示，名为 **rbac.yaml** 的文件：

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: memcached-operator-admin
subjects:
```

```
- kind: ServiceAccount
  name: memcached-quarkus-operator-operator
  namespace: <operator_namespace>
  roleRef:
    kind: ClusterRole
    name: cluster-admin
  apiGroup: ""
```



重要

rbac.yaml 文件将在以后的步骤中应用。

- 运行以下命令来部署 Operator :

```
$ make deploy IMG=<registry>/<user>/<image_name>:<tag>
```

- 运行以下命令通过应用上一步中创建的 **rbac.yaml** 文件向 **memcached-quarkus-operator-operator** 授予 **cluster-admin** 权限 :

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

- 运行以下命令验证 Operator 是否正在运行 :

```
$ oc get all -n default
```

输出示例

```
NAME                                READY UP-TO-DATE AVAILABLE AGE
pod/memcached-quarkus-operator-operator-7db86ccf58-k4mlm 0/1    Running 0
18s
```

- 运行以下命令以应用 **memcached-sample.yaml** 并创建 **memcached-sample** pod :

```
$ oc apply -f memcached-sample.yaml
```

输出示例

```
memcached.cache.example.com/memcached-sample created
```

验证

- 运行以下命令，确认 pod 已启动 :

```
$ oc get all
```

输出示例

```
NAME                                READY STATUS RESTARTS AGE
pod/memcached-quarkus-operator-operator-7b766f4896-kxnzt 1/1    Running 1    79s
pod/memcached-sample-6c765df685-mfqnz                    1/1    Running 0    18s
```

5.6.2.5.3. 捆绑 Operator 并使用 Operator Lifecycle Manager 进行部署

5.6.2.5.3.1. 捆绑 Operator

Operator 捆绑包格式是 Operator SDK 和 Operator Lifecycle Manager (OLM) 的默认打包方法。您可以使用 Operator SDK 来构建和推送 Operator 项目作为捆绑包镜像，使 Operator 可供 OLM 使用。

先决条件

- 在开发工作站上安装 operator SDK CLI
- 已安装 OpenShift CLI (**oc**) v4.15+
- 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 需要为 **go build** 明确引用 **GOARCH=amd64**。这可以在非 AMD64 构架中使用 **GOARCH=\$TARGETARCH**。Docker 自动将环境变量设置为 **-platform** 指定的值。对于 Buildah，需要使用 **-build-arg** 来实现这一目的。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_IMG**：

```
$ 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>
```

5.6.2.5.3.2. 使用 Operator Lifecycle Manager 部署 Operator

Operator Lifecycle Manager (OLM) 可帮助您在 Kubernetes 集群中安装、更新和管理 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。您可以使用 Operator SDK 在 OLM 上快速运行捆绑包镜像，以确保它正确运行。

先决条件

- 在开发工作stations上安装 operator SDK CLI
- 构建并推送到 registry 的 Operator 捆绑包镜像
- 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 4.15）
- 使用具有 **cluster-admin** 权限的账户使用 **oc** 登录到集群

流程

- 输入以下命令在集群中运行 Operator：

```
$ operator-sdk run bundle \ 1
-n <namespace> \ 2
<registry>/<user>/<bundle_image_name>:<tag> 3
```

- 1 **run bundle** 命令创建基于文件的有效目录，并使用 OLM 在集群中安装 Operator 捆绑包。
- 2 可选：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。
- 3 如果没有指定镜像，该命令使用 **quay.io/operator-framework/opm:latest** 作为默认索引镜像。如果指定了镜像，该命令会使用捆绑包镜像本身作为索引镜像。



重要

自 OpenShift Container Platform 4.11 起，**run bundle** 命令默认支持 Operator 目录基于文件的目录格式。Operator 目录已弃用的 SQLite 数据库格式仍被支持，但将在以后的发行版本中删除。建议 Operator 作者将其 workflows 迁移到基于文件的目录格式。

这个命令执行以下操作：

- 创建引用捆绑包镜像的索引镜像。索引镜像不透明且具有临时性，但准确反映了如何将捆绑包添加到生产中的目录中。
- 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 **OperatorGroup**、**Subscription**、**InstallPlan** 和所有其他所需资源（包括 RBAC），将 Operator 部署到集群中。

5.6.2.6. 其他资源

- 请参阅[基于 Java 的 Operator 的项目布局](#)，以了解 Operator SDK 创建的目录结构。
- 如果配置了集群范围的出口代理，集群管理员可以覆盖代理设置，或为 Operator Lifecycle Manager(OLM)上运行的特定 Operator 注入自定义 CA 证书。

5.6.3. 基于 Java 的 Operator 的项目布局



重要

基于 Java 的 Operator SDK 只是一个技术预览功能。技术预览功能不受红帽产品服务等级协议（SLA）支持，且功能可能并不完整。红帽不推荐在生产环境中使用它们。这些技术预览功能可以使用户提早试用新的功能，并有机会在开发阶段提供反馈意见。

有关红帽技术预览功能支持范围的更多信息，请参阅[技术预览功能支持范围](#)。

operator-sdk CLI 可为每个 Operator 项目生成或 *scaffold* 多个软件包和文件。

5.6.3.1. 基于 Java 的项目布局

由 **operator-sdk init** 命令生成的基于 Java 的 Operator 项目包含以下文件和目录：

文件或目录	用途
pom.xml	包含运行 Operator 所需的依赖项的文件。
<domain>/	包含代表 API 的文件的目录。如果域是 example.com ，这个文件夹名为 example/ 。
MemcachedReconciler.java	定义控制器实现的 Java 文件。
MemcachedSpec.java	定义 Memcached CR 所需状态的 Java 文件。

文件或目录	用途
MemcachedStat us.java	定义 Memcached CR 观察状态的 Java 文件。
Memcached.jav a	定义 Memcached API 的 Schema 的 Java 文件。
target/kubernet es/	包含 CRD yaml 文件的目录。

5.6.4. 为较新的 Operator SDK 版本更新项目

OpenShift Container Platform 4.15 支持 Operator SDK 1.31.0。如果您已在工作站上安装了 1.28.0 CLI，您可以通过[安装最新版本](#)将 CLI 更新至 1.31.0。

但是，要确保现有 Operator 项目保持与 Operator SDK 1.31.0 的兼容性，需要执行更新的相关步骤才能解决从 1.28.0 以来引入的相关破坏更改。您必须在之前使用 1.28.0 创建或维护的任何 Operator 项目中手动执行更新步骤。

5.6.4.1. 为 Operator SDK 1.31.0 更新基于 Java 的 Operator 项目

以下流程更新了基于 Java 的 Operator 项目，以便与 1.31.0 兼容。

先决条件

- 已安装 operator SDK 1.31.0
- 使用 Operator SDK 1.28.0 创建或维护的 Operator 项目

流程

- 编辑 Operator 项目的 makefile，将 Operator SDK 版本更新至 1.31.0，如下例所示：

makefile 示例

```
# Set the Operator SDK version to use. By default, what is installed on the system is used.
# This is useful for CI or a project to utilize a specific version of the operator-sdk toolkit.
OPERATOR_SDK_VERSION ?= v1.31.0 ①
```

- ① 将版本从 **1.28.0** 更改为 **1.31.0**。

5.6.4.2. 其他资源

- [将软件包清单项目迁移到捆绑包格式](#)

5.7. 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。

5.7.1. 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

- 如需了解包括生成捆绑包和 CSV 的完整流程，请参阅[捆绑 Operator](#)。

5.7.1.1. 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。

5.7.1.2. 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。

5.7.2. 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. 必需的 CSV 字段

字段	描述
<code>metadata.name</code>	该 CSV 的唯一名称。Operator 版本应包含在名称中，以保证唯一性，如 <code>app-operator.v0.1.1</code> 。
<code>metadata.capabilities</code>	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 。
<code>spec.displayName</code>	用于标识 Operator 的公共名称。
<code>spec.description</code>	有关 Operator 功能的简短描述。
<code>spec.keywords</code>	描述 Operator 的关键词。
<code>spec.maintainers</code>	维护 Operator 的个人或组织实体，含名称和电子邮件地址。
<code>spec.provider</code>	Operator 的供应商（通常是机构），含名称。
<code>spec.labels</code>	供 Operator 内部使用的键值对。
<code>spec.version</code>	Operator 的语义版本，如 <code>0.1.1</code> 。

字段	描述
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： <ul style="list-style-type: none"> ● description : CRD 描述。 ● resources : CRD 利用的任何 Kubernetes 资源，如 Pod 和 StatefulSet 对象。 ● specDescriptors : 用于 Operator 输入和输出的 UI 提示。

表 5.8. 可选的 CSV 字段

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](#)

5.7.3. Operator 元数据注解

Operator 开发人员可以在集群服务版本(CSV)的元数据中设置某些注解，以启用功能或在用户界面(UI)中突出显示功能，如 OperatorHub 或 [Red Hat Ecosystem Catalog](#)。通过在 CSV YAML 文件中设置 **metadata.annotations** 字段来手动定义 Operator 元数据注解。

5.7.3.1. 基础架构功能注解

features.operators.openshift.io 组中的注解详细说明 Operator 可能支持的基础架构功能，通过设置 **"true"** 或 **"false"** 值来指定。在 web 控制台或 [红帽生态系统目录](#) 中通过 OperatorHub 发现 Operator 时，用户可以查看和过滤这些功能。OpenShift Container Platform 4.10 及更新的版本支持这些注解。



重要

features.operators.openshift.io 基础架构功能注解弃用早期版本的 OpenShift Container Platform 中使用的 **operators.openshift.io/infrastructure-features** 注解。如需更多信息，请参阅“过时的基础架构功能注解”。

表 5.9. 基础架构功能注解

注解	描述	有效值[1]
features.operators.openshift.io/disconnected	指定 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 利用 spec.relatedImages CSV 字段来根据其摘要引用任何相关镜像。	"true" 或 "false"
features.operators.openshift.io/fips-compliant	指定 Operator 是否接受底层平台的 FIPS-140 配置，并可用于引导到 FIPS 模式的节点。在这个模式中，Operator 及其管理（操作）的任何工作负载都只调用为 FIPS-140 验证提交的 Red Hat Enterprise Linux (RHEL) 加密库。	"true" 或 "false"
features.operators.openshift.io/proxy-aware	通过接受标准 HTTP_PROXY 和 HTTPS_PROXY 代理环境变量来指定 Operator 支持在代理后面的集群中运行。如果适用，Operator 会将此信息传递给它管理的工作负载（操作）。	"true" 或 "false"
features.operators.openshift.io/tls-profiles	指定 Operator 是否实现已知的可调项，以修改 Operator 使用的 TLS 密码套件；如果适用，它管理的任何工作负载（操作）。	"true" 或 "false"
features.operators.openshift.io/token-auth-aws	使用 Cloud Credential Operator (CCO) 指定 Operator 支持通过 AWS Secure Token Service (STS)使用 AWS API 进行 tokenized 身份验证配置。	"true" 或 "false"
features.operators.openshift.io/token-auth-azure	指定 Operator 支持使用 Cloud Credential Operator (CCO) 通过 Azure Managed Identity 通过 Azure API 进行 tokenized 身份验证配置。	"true" 或 "false"
features.operators.openshift.io/token-auth-gcp	指定 Operator 支持使用 Cloud Credential Operator (CCO) 通过 GCP Workload Identity Foundation (WIF)通过 GCP Workload Identity Foundation (WIF) 进行 tokenized 身份验证的配置。	"true" 或 "false"
features.operators.openshift.io/cnf	指定 Operator 是否提供 Cloud-Native Network Function (CNF) Kubernetes 插件。	"true" 或 "false"

注解	描述	有效值 ^[1]
features.operators.openshift.io/cni	指定 Operator 是否提供 Container Network Interface (CNI) Kubernetes 插件。	"true" 或 "false"
features.operators.openshift.io/csi	指定 Operator 是否提供 Container Storage Interface (CSI) Kubernetes 插件。	"true" 或 "false"

1. 这里我们有意在有效值中使用了双引号，因为 Kubernetes 注解必须是字符串。

具有基础架构功能注解的 CSV 示例

```

apiVersion: operators.coreos.com/v1alpha1
kind: ClusterServiceVersion
metadata:
  annotations:
    features.operators.openshift.io/disconnected: "true"
    features.operators.openshift.io/fips-compliant: "false"
    features.operators.openshift.io/proxy-aware: "false"
    features.operators.openshift.io/tls-profiles: "false"
    features.operators.openshift.io/token-auth-aws: "false"
    features.operators.openshift.io/token-auth-azure: "false"
    features.operators.openshift.io/token-auth-gcp: "false"

```

其他资源

- [为受限网络环境启用 Operator](#)（断开连接模式）
- [支持 FIPS 加密](#)

5.7.3.2. 弃用的基础架构功能注解

从 OpenShift Container Platform 4.14 开始，注解的 **operators.openshift.io/infrastructure-features** 组页弃用，由带有 **features.operators.openshift.io** 命名空间的注解组替代。虽然我们推荐使用较新的注解，但当前可以并行使用这两个组。

这些注解详细介绍了 Operator 支持的基础架构功能。在 [web 控制台](#) 或 [红帽生态系统目录](#) 中通过 OperatorHub 发现 Operator 时，用户可以查看和过滤这些功能。

表 5.10. 弃用的 **operators.openshift.io/infrastructure-features** 注解

有效注解值	描述
断开连接	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。
cnf	Operator 提供了一个 Cloud-native Network Function (CNF) Kubernetes 插件。

有效注解值	描述
cni	Operator 提供了一个 Container Network Interface (CNI) Kubernetes 插件。
csi	Operator 提供了一个 Container Storage Interface (CSI) Kubernetes 插件。
fips	<p>Operator 接受底层平台的 FIPS 模式，并可用于引导到 FIPS 模式的节点。</p> <div style="display: flex; align-items: flex-start;">  <div> <p>重要</p> <p>当以 FIPS 模式运行 Red Hat Enterprise Linux (RHEL) 或 Red Hat Enterprise Linux CoreOS (RHCOS) 时，OpenShift Container Platform 核心组件使用 RHEL 加密库，在 x86_64、ppc64le 和 s390x 架构上提交到 NIST FIPS 140-2/140-3 Validation。</p> </div> </div>
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。

支持断开连接的和代理支持的 CSV 示例

```
apiVersion: operators.coreos.com/v1alpha1
kind: ClusterServiceVersion
metadata:
  annotations:
    operators.openshift.io/infrastructure-features: ["disconnected", "proxy-aware"]
```

5.7.3.3. 其他可选注解

以下 Operator 注解是可选的。

表 5.11. 其他可选注解

注解	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	通过在 Operator 安装过程中将 operatorframework.io/initialization-resource 注解添加到集群服务版本 (CSV) 来指定所需的自定义资源。然后，系统会提示您通过 CSV 中提供的模板创建自定义资源。必须包含带有完整 YAML 定义的模板。

注解	描述
<code>operatorframework.io/suggested-namespace</code>	设置部署 Operator 的建议命名空间。
<code>operatorframework.io/suggested-namespace-template</code>	为 Namespace 对象设置清单，为指定命名空间使用默认节点选择器。
<code>operators.openshift.io/valid-subscription</code>	用于列出使用 Operator 所需的任何特定订阅的空闲数组。例如， <code>["3Scale Commercial License", "Red Hat Managed Integration"]</code> 。
<code>operators.operatorframework.io/internal-objects</code>	在 UI 中隐藏不用于用户操作的 CRD。

具有 OpenShift Container Platform 许可证要求的 CSV 示例

```
apiVersion: operators.coreos.com/v1alpha1
kind: ClusterServiceVersion
metadata:
  annotations:
    operators.openshift.io/valid-subscription: ["OpenShift Container Platform"]
```

具有 3scale 许可证要求的 CSV 示例

```
apiVersion: operators.coreos.com/v1alpha1
kind: ClusterServiceVersion
metadata:
  annotations:
    operators.openshift.io/valid-subscription: ["3Scale Commercial License", "Red Hat Managed Integration"]
```

其他资源

- [CRD 模板](#)
- [初始化所需的自定义资源](#)
- [设置建议的命名空间](#)
- [使用默认节点选择器设置建议的命名空间](#)
- [隐藏内部对象](#)

5.7.4. 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

- 使用环境变量替换硬编码的镜像引用。

- 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。

先决条件

- 包含 CSV 的 Operator 项目。以下流程使用 Memcached Operator 作为基于 Go-、Ansible 和 Helm 的项目的示例。

流程

1. 为 Operator 在 `config/manager/manager.yaml` 文件中被 Operator 使用的额外镜像引用设置环境变量：

例 5.10. config/manager/manager.yaml 文件示例

```
...
spec:
  ...
  spec:
    ...
    containers:
    - command:
      - /manager
      ...
      env:
      - name: <related_image_environment_variable> ❶
        value: "<related_image_reference_with_tag>" ❷
```

- ❶ 定义环境变量，如 `RELATED_IMAGE_MEMCACHED`。
- ❷ 设置相关的镜像引用和标签，如 `docker.io/memcached:1.4.36-alpine`。

2. 将硬编码镜像引用替换为 Operator 项目类型的相关文件中的环境变量：

- 对于基于 Go 的 Operator 项目，将环境变量添加到 `controllers/memcached_controller.go` 文件中，如下例所示：

例 5.11. controllers/memcached_controller.go 文件示例

```
// deploymentForMemcached returns a memcached Deployment object
...
Spec: corev1.PodSpec{
  Containers: []corev1.Container{{
- Image: "memcached:1.4.36-alpine",
+ Image: os.Getenv("<related_image_environment_variable>"),
```

```
Name: "memcached",
Command: []string{"memcached", "-m=64", "-o", "modern", "-v"},
Ports: []corev1.ContainerPort{{
```

...

- ❶ 删除镜像引用和标签。
- ❷ 使用 `os.Getenv` 函数调用 `<related_image_environment_variable>`。



注意

如果未设置变量，则 `os.Getenv` 函数会返回空字符串。在更改文件前设置 `<related_image_environment_variable>`。

- 对于基于 Ansible 的 Operator 项目，将环境变量添加到 `roles/memcached/tasks/main.yml` 文件，如下例所示：

例 5.12. `roles/memcached/tasks/main.yml` 文件示例

```
spec:
  containers:
    - name: memcached
      command:
        - memcached
        - -m=64
        - -o
        - modern
        - -v
      - image: "docker.io/memcached:1.4.36-alpine" ❶
      + image: "{{ lookup('env', '<related_image_environment_variable>') }}" ❷
      ports:
        - containerPort: 11211
  ...
```

- ❶ 删除镜像引用和标签。
- ❷ 使用 `lookup` 功能调用 `<related_image_environment_variable>`。

- 对于基于 Helm 的 Operator 项目，将 `overrideValues` 字段添加到 `watches.yaml` 文件中，如下例所示：

例 5.13. `watches.yaml` 文件示例

```
...
- group: demo.example.com
  version: v1alpha1
  kind: Memcached
```

```
chart: helm-charts/memcached
overrideValues: ❶
  relatedImage: ${<related_image_environment_variable>} ❷
```

- ❶ 添加 **overrideValues** 字段。
- ❷ 使用 **<related_image_environment_variable>** 来定义 **overrideValues** 字段，如 **RELATED_IMAGE_MEMCACHED**。

- a. 将 **overrideValues** 字段的值添加到 **helm-charts/memcached/values.yaml** 文件中，如下例所示：

helm-charts/memcached/values.yaml 文件示例

```
...
relatedImage: ""
```

- b. 编辑 **helm-charts/memcached/templates/deployment.yaml** 文件中的 chart 模板，如下例所示：

例 5.14. helm-charts/memcached/templates/deployment.yaml 文件示例

```
containers:
  - name: {{ .Chart.Name }}
  securityContext:
    - toYaml {{ .Values.securityContext | nindent 12 }}
  image: "{{ .Values.image.pullPolicy }}"
  env: ❶
    - name: related_image ❷
      value: "{{ .Values.relatedImage }}" ❸
```

- ❶ 添加 **env** 字段。
- ❷ 命名环境变量。
- ❸ 定义环境变量的值。

3. 使用以下更改将 **BUNDLE_GEN_FLAGS** 变量定义添加到 **Makefile** 中：

Makefile 示例

```
BUNDLE_GEN_FLAGS ?= -q --overwrite --version $(VERSION)
$(BUNDLE_METADATA_OPTS)

# USE_IMAGE_DIGESTS defines if images are resolved via tags or digests
# You can enable this value if you would like to use SHA Based Digests
# To enable set flag to true
USE_IMAGE_DIGESTS ?= false
ifeq ($(USE_IMAGE_DIGESTS), true)
  BUNDLE_GEN_FLAGS += --use-image-digests
endif
```

```

...
- $(KUSTOMIZE) build config/manifests | operator-sdk generate bundle -q --overwrite --
version $(VERSION) $(BUNDLE_METADATA_OPTS) ❶
+ $(KUSTOMIZE) build config/manifests | operator-sdk generate bundle
$(BUNDLE_GEN_FLAGS) ❷
...

```

- ❶ 删除 **Makefile** 中的这一行。
 - ❷ 将上面的行替换为这一行。
4. 要将 Operator 镜像更新为使用摘要(SHA)而不是标签，请运行 **make bundle** 命令，并将 **USE_IMAGE_DIGESTS** 设置为 **true**：

```
$ make bundle USE_IMAGE_DIGESTS=true
```

5. 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。

5.7.5. 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 ❶
  operatorframework.io/os.<os>: supported ❷

```

- ❶ 将 **<arch>** 设置为受支持的字符串。
- ❷ 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。

5.7.5.1. 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 5.12. OpenShift Container Platform 支持的架构

架构	字符串
AMD64	amd64
ARM64	arm64
IBM Power®	ppc64le
IBM Z®	s390x

表 5.13. 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。

5.7.6. 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
```

- 设置建议的命名空间。

5.7.7. 使用默认节点选择器设置建议的命名空间

有些 Operator 只在 control plane 节点上运行，这可以通过由 Operator 本身在 **Pod** 规格中设置 `nodeSelector` 来完成。

为了避免重复重复且可能冲突集群范围的默认 `nodeSelector`，您可以在运行 Operator 的命名空间上设置默认节点选择器。默认节点选择器优先于集群默认值，因此集群默认不会应用到 Operator 命名空间中的 pod。

当使用 OperatorHub 将 Operator 添加到集群时，Web 控制台会在安装过程中为集群管理员自动填充推荐的命名空间。建议的命名空间使用 YAML 中的命名空间清单创建，该清单包含在集群服务版本 (CSV) 中。

流程

- 在 CSV 中，使用 `Namespace` 对象的清单设置 `operatorframework.io/suggested-namespace-template`。以下示例是指定了命名空间默认节点选择器的示例命名空间的清单：

```
metadata:
```

```

annotations:
  operatorframework.io/suggested-namespace-template: ❶
  {
    "apiVersion": "v1",
    "kind": "Namespace",
    "metadata": {
      "name": "vertical-pod-autoscaler-suggested-template",
      "annotations": {
        "openshift.io/node-selector": ""
      }
    }
  }
}

```

❶ 设置建议的命名空间。



注意

如果 CSV 中存在 **recommendations-namespace** 和 **recommended-namespace-template** 注解，则 **recommend-namespace-template** 应该优先使用。

5.7.8. 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 OLM 创建的 **OperatorCondition** CR，并能够完成以下任务：

- 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**：

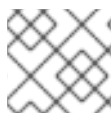
Get

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 1.31.0 支持 **operator-lib** v0.11.0。

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 1.19

require (
    k8s.io/apimachinery v0.26.0
    k8s.io/client-go v0.26.0
    sigs.k8s.io/controller-runtime v0.14.1
    operator-framework/operator-lib v0.11.0
)
```

2. 在 Operator 逻辑中编写您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](#)

5.7.9. 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`。

- [Webhook 准入插件类型](#)
- Kubernetes 文档：
 - [验证准入 webhook](#)
 - [变异准入 webhook](#)
 - [webhook 转换](#)

5.7.9.1. 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。

5.7.10. 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。

5.7.10.1. 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 5.14. 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
Resources	<p>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。</p> <p>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。</p>	选填

字段	描述	必需/可选
SpecDescriptors 、 StatusDescriptors 和 ActionDescriptors	<p>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。</p> <p>共有以下三类描述符：</p> <ul style="list-style-type: none"> ● SpecDescriptors：引用对象 spec 块中的字段。 ● StatusDescriptors：引用对象 status 块中的字段。 ● ActionDescriptors：引用对象上可执行的操作。 <p>所有描述符都接受以下字段：</p> <ul style="list-style-type: none"> ● DisplayName: Spec、Status 或 Action 的人类可读名称。 ● Description: 有关 Spec、Status 或 Action 以及 Operator 如何使用它的简短描述。 ● Path：描述符描述的对象上字段的点分隔路径。 ● 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。 <p>有关描述符的更多一般信息，请参见 openshift/console 项目。</p>	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.

```

5.7.10.2. 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 5.15. 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
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.
```

5.7.10.3. 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。

5.7.10.3.1. 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。

5.7.10.3.2. 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 ❶
    storage: true
```

- ❶ 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 ❶
  - name: v1beta1
    served: true
    storage: true ❷
```

- ❶ ❷ 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。

5.7.10.4. 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":"<operator_namespace>"},"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>"}]}]}

```

5.7.10.5. 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。

5.7.10.6. 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，您可以通过在 Operator 安装过程中将 **operatorframework.io/initialization-resource** 添加到集群服务版本 (CSV) 来指定单个所需的自定义资源。然后，系统会提示您通过 CSV 中提供的模板创建自定义资源。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": {}
  }
]
}
...

```

5.7.11. 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*。

5.7.11.1. 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 5.16. 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
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。 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

字段	描述	必需/可选
SpecDescriptors 、 StatusDescriptors 和 ActionDescriptors	与拥有的 CRD 基本相同。	选填

5.7.11.1.1. 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。

5.7.11.1.2. 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。

5.7.11.2. 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 5.17. 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
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

5.8. 使用捆绑包镜像

您可以使用 Operator SDK 在 Operator Lifecycle Manager (OLM) 中以捆绑包格式 (Bundle Format) 打包、部署和升级 Operator。

5.8.1. 捆绑 Operator

Operator 捆绑包格式是 Operator SDK 和 Operator Lifecycle Manager (OLM) 的默认打包方法。您可以使用 Operator SDK 来构建和推送 Operator 项目作为捆绑包镜像，使 Operator 可供 OLM 使用。

先决条件

- 在开发工作站上安装 operator SDK CLI
- 已安装 OpenShift CLI (**oc**) v4.15+
- 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 需要为 **go build** 明确引用 **GOARCH=amd64**。这可以在非 AMD64 构架中使用 **GOARCH=\$TARGETARCH**。Docker 自动将环境变量设置为 **-platform** 指定的值。对于 Buildah，需要使用 **-build-arg** 来实现这一目的。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。

- 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。
 - 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_IMG`：

```
$ make bundle-build BUNDLE_IMG=<registry>/<user>/<bundle_image_name>:<tag>
```

- 推送捆绑包镜像：

```
$ docker push <registry>/<user>/<bundle_image_name>:<tag>
```

5.8.2. 使用 Operator Lifecycle Manager 部署 Operator

Operator Lifecycle Manager (OLM) 可帮助您在 Kubernetes 集群中安装、更新和管理 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。您可以使用 Operator SDK 在 OLM 上快速运行捆绑包镜像，以确保它正确运行。

先决条件

- 在开发工作stations上安装 operator SDK CLI
- 构建并推送到 registry 的 Operator 捆绑包镜像
- 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 4.15）
- 使用具有 `cluster-admin` 权限的账户使用 `oc` 登录到集群
- 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

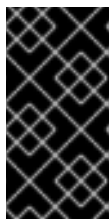
流程

- 输入以下命令在集群中运行 Operator：

```
$ operator-sdk run bundle \ 1
-n <namespace> \ 2
<registry>/<user>/<bundle_image_name>:<tag> 3
```

- 1 `run bundle` 命令创建基于文件的有效目录，并使用 OLM 在集群中安装 Operator 捆绑包。
- 2 可选：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。

- 3 如果没有指定镜像，该命令使用 `quay.io/operator-framework/opm:latest` 作为默认索引镜像。如果指定了镜像，该命令会使用捆绑包镜像本身作为索引镜像。



重要

自 OpenShift Container Platform 4.11 起，`run bundle` 命令默认支持 Operator 目录基于文件的目录格式。Operator 目录已弃用的 SQLite 数据库格式仍被支持，但将在以后的发行版本中删除。建议 Operator 作者将其工作流迁移到基于文件的目录格式。

这个命令执行以下操作：

- 创建引用捆绑包镜像的索引镜像。索引镜像不透明且具有临时性，但准确反映了如何将捆绑包添加到生产中的目录中。
- 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 **OperatorGroup**、**Subscription**、**InstallPlan** 和所有其他所需资源（包括 RBAC），将 Operator 部署到集群中。

其他资源

- Operator Framework 打包格式的[基于文件的目录](#)
- 管理自定义目录的[基于文件的目录](#)
- [捆绑包格式](#)

5.8.3. 发布包含捆绑 Operator 的目录

要安装和管理 Operator，Operator Lifecycle Manager (OLM) 要求 Operator 捆绑包列在索引镜像中，该镜像由集群中的目录引用。作为 Operator 作者，您可以使用 Operator SDK 为 Operator 及其所有依赖项创建一个包含捆绑包的索引。这可用于测试远程集群并发布到容器 registry。



注意

Operator SDK 使用 `opm` CLI 来简化索引镜像的创建。不要求具备 `opm` 命令相关经验。对于高级用例，可以直接使用 `opm` 命令，而不是 Operator SDK。

先决条件

- 在开发工作stations上安装 operator SDK CLI
- 构建并推送到 registry 的 Operator 捆绑包镜像
- 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 4.15）
- 使用具有 `cluster-admin` 权限的账户使用 `oc` 登录到集群

流程

1. 在 Operator 项目目录中运行以下 `make` 命令，以构建包含 Operator 捆绑包的索引镜像：

■


```
$ make catalog-build CATALOG_IMG=<registry>/<user>/<index_image_name>:<tag>
```

其中 **CATALOG_IMG** 参数引用您有权访问的存储库。您可以获取在存储库站点（如 Quay.io）存储容器的帐户。

2. 将构建的索引镜像推送到存储库：

```
$ make catalog-push CATALOG_IMG=<registry>/<user>/<index_image_name>:<tag>
```

提示

如果您要同时按顺序执行多个操作，您可以使用 Operator SDK **make** 命令。例如，如果您还没有为 Operator 项目构建捆绑包镜像，您可以使用以下语法构建和推送捆绑包镜像和索引镜像：

```
$ make bundle-build bundle-push catalog-build catalog-push \
  BUNDLE_IMG=<bundle_image_pull_spec> \
  CATALOG_IMG=<index_image_pull_spec>
```

另外，您可以将 **Makefile** 中的 **IMAGE_TAG_BASE** 字段设置为现有的存储库：

```
IMAGE_TAG_BASE=quay.io/example/my-operator
```

然后，您可以使用以下语法使用自动生成的名称构建和推送镜像，例如捆绑包镜像 **quay.io/example/my-operator-bundle:v0.0.1** 和 **quay.io/example/my-operator-catalog:v0.0.1** 作为索引镜像：

```
$ make bundle-build bundle-push catalog-build catalog-push
```

3. 定义一个 **CatalogSource** 对象来引用您刚才生成的索引镜像，然后使用 **oc apply** 命令或 Web 控制台创建对象：

CatalogSource YAML 示例

```
apiVersion: operators.coreos.com/v1alpha1
kind: CatalogSource
metadata:
  name: cs-memcached
  namespace: <operator_namespace>
spec:
  displayName: My Test
  publisher: Company
  sourceType: grpc
  grpcPodConfig:
    securityContextConfig: <security_mode> 1
  image: quay.io/example/memcached-catalog:v0.0.1 2
  updateStrategy:
    registryPoll:
      interval: 10m
```

- 1 指定 **legacy** 或 **restricted** 的值。如果没有设置该字段，则默认值为 **legacy**。在以后的 OpenShift Container Platform 发行版本中，计划默认值为 **restricted**。如果您的目录无法使用 **restricted** 权限运行，建议您手动将此字段设置为 **legacy**。

- 2 将 **image** 设置为您之前与 **CATALOG_IMG** 参数搭配使用的镜像拉取规格。

4. 检查目录源：

```
$ oc get catalogsource
```

输出示例

```
NAME          DISPLAY  TYPE  PUBLISHER  AGE
cs-memcached  My Test  grpc  Company    4h31m
```

验证

1. 使用您的目录安装 Operator:
 - a. 定义 **OperatorGroup** 对象并使用 **oc apply** 命令或 Web 控制台创建它：

OperatorGroup YAML 示例

```
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: my-test
  namespace: <operator_namespace>
spec:
  targetNamespaces:
  - <operator_namespace>
```

- b. 定义 **Subscription** 对象并使用 **oc apply** 命令或 Web 控制台创建它：

Subscription YAML 示例

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: catalogtest
  namespace: <catalog_namespace>
spec:
  channel: "alpha"
  installPlanApproval: Manual
  name: catalog
  source: cs-memcached
  sourceNamespace: <operator_namespace>
  startingCSV: memcached-operator.v0.0.1
```

2. 验证已安装的 Operator 是否正在运行：

- a. 检查 Operator 组：

```
$ oc get og
```

输出示例

```
-
```

```
NAME          AGE
my-test       4h40m
```

- b. 检查集群服务版本（CSV）：

```
$ oc get csv
```

输出示例

```
NAME                DISPLAY VERSION  REPLACES  PHASE
memcached-operator.v0.0.1  Test    0.0.1    Succeeded
```

- c. 检查 Operator 的 pod:

```
$ oc get pods
```

输出示例

```
NAME                                                    READY  STATUS   RESTARTS  AGE
9098d908802769fbde8bd45255e69710a9f8420a8f3d814abe88b68f8ervdj6  0/1
Completed 0      4h33m
catalog-controller-manager-7fd5b7b987-69s4n           2/2    Running  0
4h32m
cs-memcached-7622r                                     1/1    Running  0      4h33m
```

其他资源

- 如需了解更多高级用例，请参阅[管理自定义目录](#)以了解有关 **opm** CLI 直接使用的详情。

5.8.4. 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** 子命令或传统的 OLM 安装安装 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 Operator 尚未安装 OLM，请使用 **run bundle** 子命令或传统的 OLM 安装安装较早的版本。



注意

如果通过传统方式使用 OLM 安装捆绑包的早期版本，则您要升级到的较新的捆绑包不能存在于目录源引用的索引镜像中。否则，运行 **run bundle-upgrade** 子命令将导致 registry pod 失败，因为较新的捆绑包已被提供软件包和集群服务版本的索引引用。

例如，您可以通过指定更早的捆绑包镜像，为 Memcached Operator 使用以下 **run bundle** 子命令：

```
$ operator-sdk run bundle <registry>/<user>/memcached-operator:v0.0.1
```

输出示例

```
INFO[0006] Creating a File-Based Catalog of the bundle "quay.io/demo/memcached-operator:v0.0.1"
INFO[0008] Generated a valid File-Based Catalog
INFO[0012] Created registry pod: quay-io-demo-memcached-operator-v1-0-1
INFO[0012] Created CatalogSource: memcached-operator-catalog
INFO[0012] OperatorGroup "operator-sdk-og" created
INFO[0012] Created Subscription: memcached-operator-v0-0-1-sub
INFO[0015] Approved InstallPlan install-h9666 for the Subscription: memcached-operator-v0-0-1-sub
INFO[0015] Waiting for ClusterServiceVersion "my-project/memcached-operator.v0.0.1" to reach 'Succeeded' phase
INFO[0015] Waiting for ClusterServiceVersion ""my-project/memcached-operator.v0.0.1" to appear
INFO[0026] Found ClusterServiceVersion "my-project/memcached-operator.v0.0.1" phase: Pending
INFO[0028] Found ClusterServiceVersion "my-project/memcached-operator.v0.0.1" phase: Installing
INFO[0059] Found ClusterServiceVersion "my-project/memcached-operator.v0.0.1" phase: Succeeded
INFO[0059] OLM has successfully installed "memcached-operator.v0.0.1"
```

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[0008] Generated a valid Upgraded File-Based Catalog
INFO[0009] 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[0019] Found ClusterServiceVersion "my-project/memcached-operator.v0.0.2" phase:
Pending
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
```

其他资源

- [使用 OLM 安装传统的 Operator](#)

5.8.5. 控制与 OpenShift Container Platform 版本的 Operator 兼容性



重要

Kubernetes 定期弃用后续版本中删除的某些 API。如果您的 Operator 使用已弃用的 API，则在 OpenShift Container Platform 集群升级到已删除 API 的 Kubernetes 版本后，它可能无法正常工作。

作为 Operator 作者，强烈建议您查阅 Kubernetes 文档中的[已弃用 API 迁移指南](#)，并保持您的 Operator 项目最新状态以避免使用已弃用和删除的 API。理想情况下，您应该在更新的 OpenShift Container Platform 版本发行之前更新 Operator，从而避免 Operator 不兼容的问题。

当从 OpenShift Container Platform 版本中删除 API 时，在该集群版本上运行的仍使用删除的 API 的 Operator 将不再正常工作。作为 Operator 作者，您应该计划更新 Operator 项目，以适应 API 弃用和删除情况，以避免 Operator 用户中断。

提示

您可以检查 Operator 的事件警报，以查找有关当前是否正在使用 API 的警告。以下警报在检测到正在使用的 API 会在下一发行版本中会被删除时发出一个警告：

APIRemovedInNextReleaseInUse

将在下一个 OpenShift Container Platform 发行版本中删除的 API。

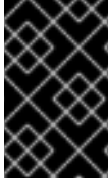
APIRemovedInNextEUSReleaseInUse

将在下一个 OpenShift Container Platform [延长更新支持 \(EUS\) 发行版本](#)中删除的 API。

如果集群管理员安装了 Operator，在升级到下一个 OpenShift Container Platform 版本前，必须确保安装与下一集群版本兼容的 Operator 版本。虽然建议您将 Operator 项目更新为不再使用已弃用或删除的 API，但如果您仍需要发布带有已删除 API 的 Operator 捆绑包，以便在早期版本的 OpenShift Container Platform 上继续使用，请确保正确配置了捆绑包。

以下流程可帮助管理员在不兼容的 OpenShift Container Platform 版本上安装 Operator 版本。这些步骤还可防止管理员升级到与当前在集群中安装的 Operator 版本不兼容的 OpenShift Container Platform 的更新版本。

当您知道当前版本的 Operator 因任何原因无法在特定的 OpenShift Container Platform 版本上正常工作时，此过程也很有用。通过定义应分发 Operator 的集群版本，可确保 Operator 不出现在允许范围内的集群版本目录中。



重要

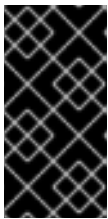
当集群管理员升级到不再支持 API 的未来 OpenShift Container Platform 版本时，使用已弃用 API 的 Operator 可能会对关键工作负载造成负面影响。如果您的 Operator 使用已弃用的 API，则应该尽快在 Operator 项目中配置以下设置。

先决条件

- 现有 Operator 项目

流程

1. 如果您知道特定 Operator 捆绑包不受支持，且早于特定集群版本无法在 OpenShift Container Platform 上正常工作，请配置 Operator 兼容的最大 OpenShift Container Platform 版本。在 Operator 项目的集群服务版本 (CSV) 中，设置 **olm.maxOpenShiftVersion** 注解以防止管理员在将已安装的 Operator 升级到兼容版本前升级其集群：



重要

只有在 Operator 捆绑包版本稍后无法工作时，才必须使用 **olm.maxOpenShiftVersion** 注解。请注意，集群管理员无法使用安装的解决方案升级其集群。如果没有提供更新的版本和有效的升级路径，管理员可以卸载 Operator，并可升级集群版本。

带有 olm.maxOpenShiftVersion 注解的 CSV 示例

```
apiVersion: operators.coreos.com/v1alpha1
kind: ClusterServiceVersion
metadata:
  annotations:
    "olm.properties": [{"type": "olm.maxOpenShiftVersion", "value": "<cluster_version>"}] 1
```

- 1** 指定 Operator 兼容的最大 OpenShift Container Platform 集群版本。例如，当在集群中安装这个捆绑包时，将 **value** 设为 **4.9** 可防止集群升级到 OpenShift Container Platform 4.9 之后的版本。

2. 如果您的捆绑包旨在红帽提供的 Operator 目录中发布，请通过设置以下属性为 Operator 配置兼容版本的 OpenShift Container Platform。此配置可确保您的 Operator 只包含在以兼容 OpenShift Container Platform 版本为目标的目录中：



注意

仅当在红帽提供的目录中发布 Operator 时，才需要这个步骤。如果您的捆绑包只用于在自定义目录中分发，您可以跳过这一步。如需了解更多详细信息，请参阅“红帽提供的 Operator 目录”。

- a. 在项目的 `bundle/metadata/annotations.yaml` 文件中设置 `com.redhat.openshift.versions` 注解：

兼容版本的 `bundle/metadata/annotations.yaml` 文件示例

```
com.redhat.openshift.versions: "v4.7-v4.9" 1
```

- 1** 设置一个范围或一个版本。

- b. 为防止您的捆绑包被传输到不兼容的 OpenShift Container Platform 版本，请确保索引镜像使用 Operator 捆绑包镜像中的正确 `com.redhat.openshift.versions` 标签生成。例如，如果您的项目是使用 Operator SDK 生成的，请更新 `bundle.Dockerfile` 文件：

与兼容版本的 `bundle.Dockerfile` 示例

```
LABEL com.redhat.openshift.versions="<versions>" 1
```

- 1** 设置为范围或单个版本，如 `v4.7-v4.9`。通过这个设置，您可以定义应分发 Operator 的集群版本，Operator 不会出现在范围之外的集群版本目录中。

现在，您可以捆绑 Operator 的新版本，并将更新的版本发布到目录以进行分发。

其他资源

- [认证的 Operator 构建指南中的管理 OpenShift 版本](#)
- [更新安装的 Operator](#)
- [红帽提供的 Operator 目录](#)

5.8.6. 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](#)。

5.9. 遵守 POD 安全准入

`Pod 安全准入` 是 [Kubernetes pod 安全标准的实现](#)。`Pod 安全准入` 限制 pod 的行为。不遵循全局或命名空间级别定义的 pod 安全准入的 Pod 不会被接受到集群且无法运行。

如果 Operator 项目不需要升级的权限才能运行，您可以确保您的工作负载在将命名空间设置为 `restricted` pod 安全级别。如果 Operator 项目需要升级的权限才能运行，您必须设置以下安全上下文配置：

- Operator 命名空间允许的 pod 安全准入级别
- 工作负载服务帐户允许的安全性上下文约束 (SCC)

如需更多信息，请参阅[了解和管理 pod 安全准入](#)。

5.9.1. 关于 pod 安全准入

OpenShift Container Platform 包括 [Kubernetes pod 安全准入](#)。不遵循全局或命名空间级别定义的 pod 安全准入的 Pod 不会被接受到集群且无法运行。

在全局范围内，会强制 **privileged** 配置集，**restricted** 配置集用于警告和审核。

您还可以在命名空间级别配置 pod 安全准入设置。



重要

不要在默认项目中运行工作负载或共享对默认项目的访问权限。为运行核心集群组件保留默认项目。

以下默认项目被视为具有高度特权：**default, kube-public, kube-system, openshift, openshift-infra, openshift-node**，其他系统创建的项目的标签 **openshift.io/run-level** 被设置为 **0** 或 **1**。依赖于准入插件（如 pod 安全准入、安全性上下文约束、集群资源配额和镜像引用解析）的功能无法在高特权项目中工作。

5.9.1.1. Pod 安全准入模式

您可以为命名空间配置以下 pod 安全准入模式：

表 5.18. Pod 安全准入模式

模式	标签	描述
enforce	pod-security.kubernetes.io/enforce	如果 pod 不符合集合配置集，则拒绝 pod 来自准入
audit	pod-security.kubernetes.io/audit	如果 pod 不符合集合配置集，日志审计事件
warn	pod-security.kubernetes.io/warn	如果 pod 不符合集合配置集，则会显示警告

5.9.1.2. Pod 安全准入配置集

您可以将每个 pod 安全准入模式设置为以下配置集之一：

表 5.19. Pod 安全准入配置集

profile	描述
privileged	最低限制策略；允许已知特权升级
baseline	最低限制策略；防止已知特权升级

profile	描述
restricted	最严格的策略；遵循当前的 pod 强化最佳实践

5.9.1.3. 特权命名空间

以下系统命名空间总是设置为 **privileged** pod 安全准入配置集：

- **default**
- **kube-public**
- **kube-system**

您无法更改这些特权命名空间的 pod 安全配置集。

5.9.2. 关于 pod 安全准入同步

除了全局 pod 安全准入控制配置外，还存在一个控制器，它会根据给定命名空间中的服务帐户的 SCC 权限将 pod 安全准入控制 **warn** 和 **audit** 标签应用到命名空间。

控制器检查 **ServiceAccount** 对象权限，以便在每个命名空间中使用安全性上下文约束。安全性上下文约束 (SCC) 根据其字段值映射到 Pod 安全配置集，控制器使用这些翻译配置集。Pod 安全准入 **warn** 和 **audit** 标签被设置为命名空间中的最特权 pod 安全配置集，以防止在创建 pod 时显示警告和日志记录审计事件。

命名空间标签基于对命名空间本地服务帐户权限的考虑。

直接应用 pod 可能会使用运行 Pod 的用户的 SCC 特权。但是，在自动标记过程中不会考虑用户权限。

5.9.2.1. Pod 安全准入同步命名空间排除

在大多数系统创建的命名空间中永久禁用 Pod 安全准入同步。同步最初在用户创建的带有 **openshift** 前缀的命名空间上被禁用，但您可以在以后启用同步。

重要

如果 pod 安全准入标签 (pod-security.kubernetes.io/<mode>) 被来自一个标签同步的命名空间中自动进行标签的值进行了修改，则对此标签的同步会被禁用。

如果需要，您可以使用以下方法之一再次启用同步：

- 从命名空间中删除修改后的 pod 安全准入标签
- 将 **security.openshift.io/scc.podSecurityLabelSync** 标签设置为 **true** 如果您通过添加此标签强制同步，则任何修改过的 pod 安全准入标签都会被覆盖。

永久禁用命名空间

定义为集群有效负载一部分的命名空间会永久禁用 pod 安全准入同步。以下命名空间被永久禁用：

- **default**

- **kube-node-lease**
- **kube-system**
- **kube-public**
- **openshift**
- 所有带有 **openshift-** 前缀的系统创建命名空间，但 **openshift-operators** 除外

最初禁用命名空间

默认情况下，具有 **openshift-** 前缀的所有命名空间最初都禁用了 pod 安全准入同步。您可以为用户创建的 **openshift-*** 命名空间和 **openshift-operators** 命名空间启用同步。



注意

除了 **openshift-operators** 之外，您无法为任何系统创建的 **openshift-*** 命名空间启用同步。

如果在用户创建的 **openshift-*** 命名空间中安装了 Operator，则在命名空间中创建集群服务版本 (CSV) 后会自动启用同步。同步标签派生自命名空间中服务帐户的权限。

5.9.3. 确保 Operator 工作负载在命名空间中运行，设置为受限 pod 安全级别

为确保 Operator 项目可以在各种部署和环境中运行，请将 Operator 的工作负载配置为在命名空间中运行，设置为 **restricted** pod 安全级别。

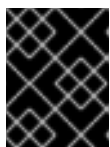


警告

您必须将 **runAsUser** 字段留空。如果您的镜像需要特定用户，则无法在受限安全性上下文约束 (SCC) 和受限 Pod 安全强制下运行。

流程

- 要将 Operator 工作负载配置为在设置为 **restricted** pod 安全级别的命名空间中运行，请编辑类似以下示例的 Operator 命名空间定义：



重要

建议您在 Operator 的命名空间定义中设置 **seccomp** 配置集。但是，OpenShift Container Platform 4.10 不支持设置 **seccomp** 配置集。

- 对于必须只在 OpenShift Container Platform 4.11 及之后的版本中运行的 Operator 项目，请编辑类似以下示例的 Operator 命名空间定义：

config/manager/manager.yaml 文件示例

```
...
spec:
```

```

securityContext:
  seccompProfile:
    type: RuntimeDefault ❶
  runAsNonRoot: true
containers:
- name: <operator_workload_container>
  securityContext:
    allowPrivilegeEscalation: false
  capabilities:
    drop:
      - ALL
...

```

❶ 通过将 seccomp 配置集类型设置为 **RuntimeDefault**，SCC 默认为命名空间的 pod 安全配置集。

- 对于必须在 OpenShift Container Platform 4.10 中运行的 Operator 项目，编辑 Operator 的命名空间定义类似以下示例：

config/manager/manager.yaml 文件示例

```

...
spec:
  securityContext: ❶
    runAsNonRoot: true
  containers:
- name: <operator_workload_container>
  securityContext:
    allowPrivilegeEscalation: false
  capabilities:
    drop:
      - ALL
...

```

❶ 不设置 seccomp 配置集类型可确保您的 Operator 项目可以在 OpenShift Container Platform 4.10 中运行。

其他资源

- [管理安全性上下文约束](#)

5.9.4. 为需要升级权限的 Operator 工作负载管理 pod 安全准入

如果 Operator 项目需要升级的权限才能运行，您必须编辑 Operator 的集群服务版本 (CSV)。

流程

- 将安全上下文配置设置为 Operator CSV 中所需的权限级别，如下例所示：

具有网络管理员特权的 <operator_name>.clusterserviceversion.yaml 文件示例

```

...

```

```

containers:
  - name: my-container
    securityContext:
      allowPrivilegeEscalation: false
    capabilities:
      add:
        - "NET_ADMIN"
  ...

```

2. 设置服务帐户权限，允许 Operator 工作负载使用所需的安全性上下文约束 (SCC)，如下例所示：

<operator_name>.clusterserviceversion.yaml 文件示例

```

...
install:
  spec:
    clusterPermissions:
      - rules:
          - apiGroups:
              - security.openshift.io
            resourceNames:
              - privileged
            resources:
              - securitycontextconstraints
            verbs:
              - use
        serviceAccountName: default
  ...

```

3. 编辑 Operator 的 CSV 描述，以说明 Operator 项目需要升级的权限，如下例所示：

<operator_name>.clusterserviceversion.yaml 文件示例

```

...
spec:
  apiservicedefinitions: {}
  ...
description: The <operator_name> requires a privileged pod security admission label set on the Operator's namespace. The Operator's agents require escalated permissions to restart the node if the node needs remediation.

```

5.9.5. 其他资源

- [了解并管理 pod 安全准入](#)

5.10. 令牌身份验证

5.10.1. 云供应商上的 Operator 的令牌身份验证

许多云提供商可以通过使用提供短期、有限权限安全凭据的帐户令牌来启用身份验证。

OpenShift Container Platform 包含 Cloud Credential Operator (CCO)，用于将云供应商凭证作为自定义资源定义(CRD)进行管理。**CredentialsRequest** 自定义资源(CR)的 CCO 同步，允许 OpenShift Container Platform 组件使用所需的特定权限请求云供应商凭证。

在以前的版本中，在 CCO 处于 *手动模式* 的集群中，由 Operator Lifecycle Manager (OLM)管理的 Operator 通常会在 OperatorHub 中提供详细说明，供用户手动置备任何所需的云凭证。

从 OpenShift Container Platform 4.14 开始，CCO 可以在启用的集群中检测它，以便在某些云供应商中使用短期凭证。然后，它可以半自动置备某些凭证，只要 Operator 作者启用了其 Operator 来支持更新的 CCO。

其他资源

- [关于 Cloud Credential Operator](#)
- [使用 AWS STS 的 OLM 管理的 Operator 基于 CCO 的工作流](#)
- [使用 Microsoft Entra Workload ID 为 OLM 管理的 Operator 基于 CCO 的工作流](#)

5.10.2. 使用 AWS STS 的 OLM 管理的 Operator 基于 CCO 的工作流

当在 AWS 上运行的 OpenShift Container Platform 集群处于 Security Token Service (STS) 模式时，这意味着集群会利用 AWS 和 OpenShift Container Platform 的功能在应用程序级别使用 IAM 角色。STS 使应用程序能够提供可假定 IAM 角色的 JSON Web Token (JWT)。

JWT 包含用于 **sts:AssumeRoleWithWebIdentity** IAM 操作的 Amazon 资源名称 (ARN)，以允许服务帐户临时获得权限。JWT 包含 AWS IAM 可验证的 **ProjectedServiceAccountToken** 的签名密钥。服务帐户令牌本身（已签名）用作假定 AWS 角色所需的 JWT。

Cloud Credential Operator (CCO) 是在云供应商上运行的 OpenShift Container Platform 集群中默认安装的集群 Operator。对于 STS，CCO 提供以下功能：

- 检测它是否在启用了 STS 的集群中运行的
- 检查 **CredentialsRequest** 对象中是否存在字段，该对象提供授予 Operator 对 AWS 资源访问权限所需的信息

即使处于手动模式，CCO 也会执行此检测。正确配置后，CCO 将带有所需访问信息的 **Secret** 对象项目到 Operator 命名空间中。

从 OpenShift Container Platform 4.14 开始，CCO 可以通过扩大使用 **CredentialsRequest** 对象来自动处理此任务，该对象可请求创建包含 STS 工作流所需的信息的 **Secret**。通过 Web 控制台或 CLI 安装 Operator 时，用户可以提供角色 ARN。



注意

不建议使用具有自动更新批准的订阅，因为更新前可能会有权限更改。使用手动批准的订阅可确保管理员有机会验证更新版本的权限，并在更新前采取必要的操作。

作为 Operator 作者准备一个 Operator 以用于 OpenShift Container Platform 4.14 或更高版本中更新的 CCO，您应该指示用户并添加代码来处理之前 CCO 版本的比较，除了处理 STS 令牌身份验证外（如果您的 Operator 还没有启用 STS）。推荐的方法是提供 **CredentialsRequest** 对象提供正确填充的 STS 相关字段，并让 CCO 为您的 Secret 创建 **Secret**。



重要

如果您计划支持早于版本 4.14 的 OpenShift Container Platform 集群，请考虑为用户提供有关如何使用 CCO 实用程序(**ccoctl**)手动创建带有 STS 额外信息的 **secret** 的说明。早期 CCO 版本不知道集群中的 STS 模式，且无法为您创建 **secret**。

您的代码应检查永远不会出现的 **secret**，并警告用户以遵循您提供的回退指令。如需更多信息，请参阅"Alternative method"子部分。

其他资源

- [OLM 管理的 Operator 支持使用 AWS STS 进行身份验证](#)
- [使用 Web 控制台从 OperatorHub 安装](#)
- [使用 CLI 从 OperatorHub 安装](#)

5.10.2.1. 启用 Operator 以支持使用 AWS STS 的基于 CCO 的工作流

作为 Operator 作者设计在 Operator Lifecycle Manager (OLM)上运行的项目，您可以通过自定义项目来支持 Cloud Credential Operator (CCO)，使 Operator 能够对启用了 STS 的 OpenShift Container Platform 集群上的 AWS 进行身份验证。

使用此方法时，Operator 负责创建 **CredentialsRequest** 对象，这意味着 Operator 需要 RBAC 权限来创建这些对象。然后，Operator 必须能够读取生成的 **Secret** 对象。



注意

默认情况下，与 Operator 部署相关的 pod 会挂载 **serviceAccountToken** 卷，以便在生成的 **Secret** 对象中引用服务帐户令牌。

先决条件

- OpenShift Container Platform 4.14 或更高版本
- 处于 STS 模式的集群
- 基于 OLM 的 Operator 项目

流程

1. 更新 Operator 项目的 **ClusterServiceVersion (CSV)**对象：
 - a. 确保 Operator 有 RBAC 权限来创建 **CredentialsRequests** 对象：

例 5.15. clusterPermissions 列表示例

```
# ...
install:
spec:
  clusterPermissions:
  - rules:
    - apiGroups:
      - "cloudcredential.openshift.io"
    resources:
      - credentialsrequests
```

```

verbs:
- create
- delete
- get
- list
- patch
- update
- watch

```

- b. 添加以下注解来声明对使用 AWS STS 的基于 CCO 工作流的方法的支持：

```

# ...
metadata:
  annotations:
    features.operators.openshift.io/token-auth-aws: "true"

```

2. 更新 Operator 项目代码：

- a. 从 pod 上由 **Subscription** 对象设置的环境变量获取角色 ARN。例如：

```

// Get ENV var
roleARN := os.Getenv("ROLEARN")
setupLog.Info("getting role ARN", "role ARN = ", roleARN)
webIdentityTokenPath := "/var/run/secrets/openshift/serviceaccount/token"

```

- b. 确保具有 **CredentialsRequest** 对象已准备好修补并应用。例如：

例 5.16. CredentialsRequest 对象创建示例

```

import (
  minterv1 "github.com/openshift/cloud-credential-
operator/pkg/apis/cloudcredential/v1"
  corev1 "k8s.io/api/core/v1"
  metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
)

var in = minterv1.AWSPProviderSpec{
  StatementEntries: []minterv1.StatementEntry{
    {
      Action: []string{
        "s3:*",
      },
      Effect: "Allow",
      Resource: "arn:aws:s3:*:*:*:*",
    },
  },
  STSIAMRoleARN: "<role_arn>",
}

var codec = minterv1.Codec
var ProviderSpec, _ = codec.EncodeProviderSpec(in.DeepCopyObject())

const (
  name = "<credential_request_name>"

```

```

    namespace = "<namespace_name>"
  )

  var CredentialsRequestTemplate = &minterv1.CredentialsRequest{
    ObjectMeta: metav1.ObjectMeta{
      Name:    name,
      Namespace: "openshift-cloud-credential-operator",
    },
    Spec: minterv1.CredentialsRequestSpec{
      ProviderSpec: ProviderSpec,
      SecretRef: corev1.ObjectReference{
        Name:    "<secret_name>",
        Namespace: namespace,
      },
      ServiceAccountNames: []string{
        "<service_account_name>",
      },
      CloudTokenPath: "",
    },
  }
}

```

另外，如果您从 YAML 表单的 **CredentialsRequest** 对象开始（例如，作为 Operator 项目代码的一部分），您可以以不同的方式处理它：

例 5.17. 以 YAML 格式创建 **CredentialsRequest** 对象示例

```

// CredentialsRequest is a struct that represents a request for credentials
type CredentialsRequest struct {
  APIVersion string `yaml:"apiVersion"`
  Kind       string `yaml:"kind"`
  Metadata   struct {
    Name       string `yaml:"name"`
    Namespace  string `yaml:"namespace"`
  } `yaml:"metadata"`
  Spec struct {
    SecretRef struct {
      Name       string `yaml:"name"`
      Namespace  string `yaml:"namespace"`
    } `yaml:"secretRef"`
    ProviderSpec struct {
      APIVersion  string `yaml:"apiVersion"`
      Kind        string `yaml:"kind"`
      StatementEntries []struct {
        Effect string `yaml:"effect"`
        Action []string `yaml:"action"`
        Resource string `yaml:"resource"`
      } `yaml:"statementEntries"`
      STSIAMRoleARN string `yaml:"stsiAMRoleARN"`
    } `yaml:"providerSpec"`

    // added new field
    CloudTokenPath string `yaml:"cloudTokenPath"`
  } `yaml:"spec"`
}

```



```

// ConsumeCredsRequestAddingTokenInfo is a function that takes a YAML filename
and two strings as arguments
// It unmarshals the YAML file to a CredentialsRequest object and adds the token
information.
func ConsumeCredsRequestAddingTokenInfo(fileName, tokenString, tokenPath
string) (*CredentialsRequest, error) {
    // open a file containing YAML form of a CredentialsRequest
    file, err := os.Open(fileName)
    if err != nil {
        return nil, err
    }
    defer file.Close()

    // create a new CredentialsRequest object
    cr := &CredentialsRequest{}

    // decode the yaml file to the object
    decoder := yaml.NewDecoder(file)
    err = decoder.Decode(cr)
    if err != nil {
        return nil, err
    }

    // assign the string to the existing field in the object
    cr.Spec.CloudTokenPath = tokenPath

    // return the modified object
    return cr, nil
}

```



注意

目前不支持在 Operator 捆绑包中添加 **CredentialsRequest** 对象。

- c. 在凭证请求中添加角色 ARN 和 Web 身份令牌路径，并在 Operator 初始化过程中应用它：

例 5.18. 在 Operator 初始化过程中应用 CredentialsRequest 对象示例

```

// apply credentialsRequest on install
credReq := credreq.CredentialsRequestTemplate
credReq.Spec.CloudTokenPath = webIdentityTokenPath

c := mgr.GetClient()
if err := c.Create(context.TODO(), credReq); err != nil {
    if !errors.IsAlreadyExists(err) {
        setupLog.Error(err, "unable to create CredRequest")
        os.Exit(1)
    }
}
}

```

- d. 确保 Operator 可以等待 **Secret** 对象从 CCO 显示，如下例所示，以及您在 Operator 中协调的其他项目：

例 5.19. 等待 Secret 对象示例

```

// WaitForSecret is a function that takes a Kubernetes client, a namespace, and a v1
// "k8s.io/api/core/v1" name as arguments
// It waits until the secret object with the given name exists in the given namespace
// It returns the secret object or an error if the timeout is exceeded
func WaitForSecret(client kubernetes.Interface, namespace, name string)
(*v1.Secret, error) {
    // set a timeout of 10 minutes
    timeout := time.After(10 * time.Minute) ❶

    // set a polling interval of 10 seconds
    ticker := time.NewTicker(10 * time.Second)

    // loop until the timeout or the secret is found
    for {
        select {
        case <-timeout:
            // timeout is exceeded, return an error
            return nil, fmt.Errorf("timed out waiting for secret %s in namespace %s", name,
namespace)
            // add to this error with a pointer to instructions for following a manual path to a
Secret that will work on STS
        case <-ticker.C:
            // polling interval is reached, try to get the secret
            secret, err := client.CoreV1().Secrets(namespace).Get(context.Background(),
name, metav1.GetOptions{})
            if err != nil {
                if errors.IsNotFound(err) {
                    // secret does not exist yet, continue waiting
                    continue
                } else {
                    // some other error occurred, return it
                    return nil, err
                }
            } else {
                // secret is found, return it
                return secret, nil
            }
        }
    }
}

```

- ❶ **timeout** 值基于 CCO 检测添加的 **CredentialsRequest** 对象并生成 **Secret** 对象的速度。您可能会考虑降低时间或为集群管理员创建自定义反馈，这可能会导致 Operator 尚未访问云资源的原因。

- e. 通过从凭证请求中读取 CCO 创建的 secret 并设置 AWS 配置，并创建包含该 secret 数据的 AWS 配置文件：

例 5.20. AWS 配置创建示例

```

func SharedCredentialsFileFromSecret(secret *corev1.Secret) (string, error) {
    var data []byte

```

```

switch {
case len(secret.Data["credentials"]) > 0:
    data = secret.Data["credentials"]
default:
    return "", errors.New("invalid secret for aws credentials")
}

f, err := ioutil.TempFile("", "aws-shared-credentials")
if err != nil {
    return "", errors.Wrap(err, "failed to create file for shared credentials")
}
defer f.Close()
if _, err := f.Write(data); err != nil {
    return "", errors.Wrapf(err, "failed to write credentials to %s", f.Name())
}
return f.Name(), nil
}

```

重要

secret 被假定为存在，但在使用此 secret 时，您的 Operator 代码应等待和重试，以提供 CCO 创建 secret 的时间。

另外，等待周期最终应该超时，并警告用户 OpenShift Container Platform 集群版本，因此 CCO 可能会是一个较早的版本，它不支持使用 STS 检测的 **CredentialsRequest** 对象工作流。在这种情况下，指示用户必须使用其他方法添加 secret。

f. 配置 AWS SDK 会话，例如：

例 5.21. AWS SDK 会话配置示例

```

sharedCredentialsFile, err := SharedCredentialsFileFromSecret(secret)
if err != nil {
    // handle error
}
options := session.Options{
    SharedConfigState: session.SharedConfigEnable,
    SharedConfigFiles: []string{sharedCredentialsFile},
}

```

5.10.2.2. 角色规格

Operator 描述应包含在安装前创建的角色具体信息，最好是管理员可以运行的脚本的形式。例如：

例 5.22. 角色创建脚本示例

```

#!/bin/bash
set -x

AWS_ACCOUNT_ID=$(aws sts get-caller-identity --query "Account" --output text)

```

```

OIDC_PROVIDER=$(oc get authentication cluster -ojson | jq -r .spec.serviceAccountIssuer | sed -
e "s/^https:\\\\//" )
NAMESPACE=my-namespace
SERVICE_ACCOUNT_NAME="my-service-account"
POLICY_ARN_STRINGS="arn:aws:iam::aws:policy/AmazonS3FullAccess"

read -r -d " TRUST_RELATIONSHIP <<EOF
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Federated": "arn:aws:iam::${AWS_ACCOUNT_ID}:oidc-provider/${OIDC_PROVIDER}"
      },
      "Action": "sts:AssumeRoleWithWebIdentity",
      "Condition": {
        "StringEquals": {
          "${OIDC_PROVIDER}:sub":
"system:serviceaccount:${NAMESPACE}:${SERVICE_ACCOUNT_NAME}"
        }
      }
    }
  ]
}
EOF

echo "${TRUST_RELATIONSHIP}" > trust.json

aws iam create-role --role-name "${SERVICE_ACCOUNT_NAME}" --assume-role-policy-document
file://trust.json --description "role for demo"

while IFS= read -r POLICY_ARN; do
  echo -n "Attaching $POLICY_ARN ... "
  aws iam attach-role-policy \
    --role-name "${SERVICE_ACCOUNT_NAME}" \
    --policy-arn "${POLICY_ARN}"
  echo "ok."
done <<< "$POLICY_ARN_STRINGS"

```

5.10.2.3. 故障排除

5.10.2.3.1. 身份验证失败

如果身份验证不成功，请确保您可以使用提供给 Operator 的令牌假设具有 Web 身份的角色。

流程

1. 从 pod 中提取令牌：

```

$ oc exec operator-pod -n <namespace_name> \
  -- cat /var/run/secrets/openshift/serviceaccount/token

```

2. 从 pod 中提取角色 ARN :

```
$ oc exec operator-pod -n <namespace_name> \
  -- cat /<path>/<to>/<secret_name> ❶
```

❶ 不要将 root 用于路径。

3. 尝试使用 Web 身份令牌假定角色 :

```
$ aws sts assume-role-with-web-identity \
  --role-arn $ROLEARN \
  --role-session-name <session_name> \
  --web-identity-token $TOKEN
```

5.10.2.3.2. Secret 无法正确挂载

以非 root 用户身份运行的 Pod 无法写入默认存在 AWS 共享凭证文件的 **/root** 目录。如果 secret 没有正确挂载到 AWS 凭证文件路径, 请考虑将 secret 挂载到不同的位置, 并在 AWS SDK 中启用共享凭证文件选项。

5.10.2.4. 其它方法

作为 Operator 作者的替代方法, 您可以指定用户在安装 Operator 前负责为 Cloud Credential Operator (CCO) 创建 **CredentialsRequest** 对象。

Operator 指令必须向用户指示以下内容 :

- 通过在说明中内联提供 YAML 或将用户指向下载位置, 提供 **CredentialsRequest** 对象的 YAML 版本
- 指示用户创建 **CredentialsRequest** 对象

在 OpenShift Container Platform 4.14 及更高版本中, 当 **CredentialsRequest** 对象出现在添加了适当 STS 信息的集群上后, Operator 可以读取 CCO 生成的 **Secret** 或挂载它, 并在集群服务版本(CSV)中定义挂载。

对于早期版本的 OpenShift Container Platform, Operator 指令还必须向用户指示以下内容 :

- 使用 CCO 实用程序(ccoctl)从 **CredentialsRequest** 对象生成 **Secret** YAML 对象
- 将 **Secret** 对象应用到适当的命名空间中的集群

Operator 仍然必须能够使用生成的 secret 与云 API 通信。因为在这种情况下, 用户会在安装 Operator 前创建 secret, 所以 Operator 可以执行以下操作之一 :

- 在 CSV 中的 **Deployment** 对象中定义显式挂载
- 从 API 服务器以编程方式读取 **Secret** 对象, 如推荐的"启用 Operator 以支持使用 AWS STS 的基于 CCO 的工作流"方法所示

5.10.3. 使用 Microsoft Entra Workload ID 为 OLM 管理的 Operator 基于 CCO 的工作流

当在 Azure 上运行的 OpenShift Container Platform 集群处于 **Workload Identity / Federated Identity** 模式时，这意味着集群会利用 Azure 和 OpenShift Container Platform 的功能在应用程序级别应用 *用户分配的管理的身份* 或 Microsoft Entra Workload ID 中的 *app 注册*。

Cloud Credential Operator (CCO) 是在云供应商上运行的 OpenShift Container Platform 集群中默认安装的集群 Operator。从 OpenShift Container Platform 4.14.8 开始，CCO 支持使用 Workload ID 的 OLM 管理的 Operator 工作流。

对于 Workload ID，CCO 提供以下功能：

- 检测在启用了 Workload ID 的集群中运行的时间
- 检查 **CredentialsRequest** 对象中是否存在字段，该对象提供授予 Operator 对 Azure 资源访问权限所需的信息

CCO 可以通过扩展 **CredentialsRequest** 对象来自动处理这个过程，该对象可以请求创建包含 Workload ID 工作流所需的信息的 **Secret**。



注意

不建议使用具有自动更新批准的订阅，因为更新前可能会有权限更改。使用手动批准的订阅可确保管理员有机会验证更新版本的权限，并在更新前采取必要的操作。

作为 Operator 作者准备一个 Operator 以用于 OpenShift Container Platform 4.14 及之后的版本中的更新的 CCO，您应该指示用户并添加代码来处理早期 CCO 版本的划分，除了处理 Workload ID token 身份验证（如果您的 Operator 还没有启用）。推荐的方法是使用正确的 Workload ID 相关字段提供 **CredentialsRequest** 对象，并让 CCO 为您创建 **Secret** 对象。



重要

如果您计划支持早于版本 4.14 的 OpenShift Container Platform 集群，请考虑为用户提供有关如何使用 CCO 实用程序(**ccoctl**)手动创建带有 Workload ID 启用信息的 secret 的说明。早期 CCO 版本不知道集群中的 Workload ID 模式，且无法为您创建 secret。

您的代码应检查永远不会出现的 secret，并警告用户以遵循您提供的回退指令。

使用 Workload ID 进行身份验证需要以下信息：

- **azure_client_id**
- **azure_tenant_id**
- **azure_region**
- **azure_subscription_id**
- **azure_federated_token_file**

Web 控制台中的 **Install Operator** 页面允许集群管理员在安装时提供此信息。然后，此信息会作为 Operator pod 上的环境变量传播到 **Subscription** 对象中。

其他资源

- [OLM 管理的 Operator 支持使用 Microsoft Entra Workload ID 进行身份验证](#)

- [使用 Web 控制台从 OperatorHub 安装](#)
- [使用 CLI 从 OperatorHub 安装](#)

5.10.3.1. 启用 Operator 以支持使用 Microsoft Entra Workload ID 的基于 CCO 的工作流

作为 Operator 作者设计在 Operator Lifecycle Manager (OLM) 上运行的项目，您可以通过自定义项目来支持 Cloud Credential Operator (CCO)，使 Operator 能够对启用了 Microsoft Entra Workload ID 的 OpenShift Container Platform 集群进行身份验证。

使用此方法时，Operator 负责创建 **CredentialsRequest** 对象，这意味着 Operator 需要 RBAC 权限来创建这些对象。然后，Operator 必须能够读取生成的 **Secret** 对象。



注意

默认情况下，与 Operator 部署相关的 pod 会挂载 **serviceAccountToken** 卷，以便在生成的 **Secret** 对象中引用服务帐户令牌。

先决条件

- OpenShift Container Platform 4.14 或更高版本
- 处于 Workload ID 模式的集群
- 基于 OLM 的 Operator 项目

流程

1. 更新 Operator 项目的 **ClusterServiceVersion (CSV)** 对象：
 - a. 确保 Operator 有 RBAC 权限来创建 **CredentialsRequests** 对象：

例 5.23. clusterPermissions 列表示例

```
# ...
install:
  spec:
    clusterPermissions:
      - rules:
        - apiGroups:
          - "cloudcredential.openshift.io"
          resources:
            - credentialsrequests
          verbs:
            - create
            - delete
            - get
            - list
            - patch
            - update
            - watch
```

- b. 添加以下注解来声明对使用 Workload ID 的基于 CCO 工作流的方法的支持：

```
# ...
metadata:
  annotations:
    features.operators.openshift.io/token-auth-azure: "true"
```

2. 更新 Operator 项目代码：

- a. 从由 **Subscription** 对象设置的环境变量中获取客户端 ID、租户 ID 和订阅 ID。例如：

```
// Get ENV var
clientID := os.Getenv("CLIENTID")
tenantID := os.Getenv("TENANTID")
subscriptionID := os.Getenv("SUBSCRIPTIONID")
azureFederatedTokenFile := "/var/run/secrets/openshift/serviceaccount/token"
```

- b. 确保具有 **CredentialsRequest** 对象已准备好修补并应用。



注意

目前不支持在 Operator 捆绑包中添加 **CredentialsRequest** 对象。

- c. 在凭证请求中添加 Azure 凭证信息和 Web 身份令牌路径，并在 Operator 初始化过程中应用它：

例 5.24. 在 Operator 初始化过程中应用 **CredentialsRequest** 对象示例

```
// apply credentialsRequest on install
credReqTemplate.Spec.AzureProviderSpec.AzureClientID = clientID
credReqTemplate.Spec.AzureProviderSpec.AzureTenantID = tenantID
credReqTemplate.Spec.AzureProviderSpec.AzureRegion = "centralus"
credReqTemplate.Spec.AzureProviderSpec.AzureSubscriptionID = subscriptionID
credReqTemplate.CloudTokenPath = azureFederatedTokenFile

c := mgr.GetClient()
if err := c.Create(context.TODO(), credReq); err != nil {
  if !errors.IsAlreadyExists(err) {
    setupLog.Error(err, "unable to create CredRequest")
    os.Exit(1)
  }
}
```

- d. 确保 Operator 可以等待 **Secret** 对象从 CCO 显示，如下例所示，以及您在 Operator 中协调的其他项目：

例 5.25. 等待 **Secret** 对象示例

```
// WaitForSecret is a function that takes a Kubernetes client, a namespace, and a v1
// "k8s.io/api/core/v1" name as arguments
// It waits until the secret object with the given name exists in the given namespace
// It returns the secret object or an error if the timeout is exceeded
func WaitForSecret(client kubernetes.Interface, namespace, name string)
(*v1.Secret, error) {
  // set a timeout of 10 minutes
  timeout := time.After(10 * time.Minute) 1
```



```

// set a polling interval of 10 seconds
ticker := time.NewTicker(10 * time.Second)

// loop until the timeout or the secret is found
for {
    select {
    case <-timeout:
        // timeout is exceeded, return an error
        return nil, fmt.Errorf("timed out waiting for secret %s in namespace %s", name,
namespace)
        // add to this error with a pointer to instructions for following a manual path to a
Secret that will work on STS
    case <-ticker.C:
        // polling interval is reached, try to get the secret
        secret, err := client.CoreV1().Secrets(namespace).Get(context.Background(),
name, metav1.GetOptions{})
        if err != nil {
            if errors.IsNotFound(err) {
                // secret does not exist yet, continue waiting
                continue
            } else {
                // some other error occurred, return it
                return nil, err
            }
        } else {
            // secret is found, return it
            return secret, nil
        }
    }
}
}
}

```

- 1** **timeout** 值基于 CCO 检测添加的 **CredentialsRequest** 对象并生成 **Secret** 对象的速度。您可能会考虑降低时间或为集群管理员创建自定义反馈，这可能会导致 Operator 尚未访问云资源的原因。

- e. 从 **CredentialsRequest** 对象读取 CCO 创建的 secret，以与 Azure 进行身份验证并接收必要的凭证。

5.11. 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

5.11.1. 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 workflow

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
```

5.11.2. 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:v1.31.0
  entrypoint:
  - scorecard-test
  - basic-check-spec
labels:
  suite: basic
  test: basic-check-spec-test
- image: quay.io/operator-framework/scorecard-test:v1.31.0
```

```

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

5.11.3. 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 5.20. 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 5.21. 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

测试	描述	短名称
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

5.11.4. 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>
```

5.11.5. Scorecard 输出

scorecard 命令的 **--output** 标志指定 scorecard 结果输出格式：**text** 或 **json**。

例 5.26. 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:v1.31.0",
        "entrypoint": [
          "scorecard-test",
          "olm-bundle-validation"
        ],
        "labels": {
          "suite": "olm",
          "test": "olm-bundle-validation-test"
        }
      }
    }
  ]
}
```


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)'
```

5.11.7. 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:v1.31.0
    labels:
      suite: basic
      test: basic-check-spec-test
  - entrypoint:
    - scorecard-test
    - olm-bundle-validation
    image: quay.io/operator-framework/scorecard-test:v1.31.0
    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。

5.11.8. 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 5.28. 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},
    }
}

```

5.12. 验证 OPERATOR 捆绑包

作为 Operator 作者，您可以在 Operator SDK 中运行 **bundle validate** 命令来验证 Operator 捆绑包的内容和格式。您可以在远程 Operator 捆绑包镜像或本地 Operator 捆绑包目录上运行该命令。

5.12.1. 关于 bundle validate 命令

虽然 Operator SDK **scorecard** 命令可以根据配置文件和测试镜像在 Operator 上运行测试，但 **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。

bundle validate 命令语法

```
$ operator-sdk bundle validate <bundle_dir_or_image> <flags>
```



注意

当您使用 **make bundle** 命令构建捆绑包时，**bundle validate** 命令会自动运行。

捆绑包镜像从远程 registry 中拉取，并在验证前在本地构建。本地捆绑包目录必须包含 Operator 元数据和清单。捆绑包元数据和清单必须具有类似以下捆绑包布局的结构：

捆绑包布局示例

```

./bundle
├── manifests
│   ├── cache.my.domain_memcacheds.yaml
│   └── memcached-operator.clusterserviceversion.yaml
├── metadata
└── annotations.yaml

```

如果检测到错误，捆绑包测试会传递验证，并使用退出代码 **0** 完成。

输出示例

```
INFO[0000] All validation tests have completed successfully
```

如果检测到错误，则测试失败的验证，并使用退出代码 **1** 完成。

输出示例

```
ERRO[0000] Error: Value cache.example.com/v1alpha1, Kind=Memcached: CRD
"cache.example.com/v1alpha1, Kind=Memcached" is present in bundle "" but not defined in CSV
```

在未检测错误时，导致警告的捆绑测试仍可使用退出代码 **0** 传递验证。测试只在错误时失败。

输出示例

```
WARN[0000] Warning: Value : (memcached-operator.v0.0.1) annotations not found
INFO[0000] All validation tests have completed successfully
```

有关 **bundle validate** 子命令的更多信息，请运行：

```
$ operator-sdk bundle validate -h
```

5.12.2. 内置捆绑包验证测试

Operator SDK 附带了预定义的验证器组合到套件中。如果您在没有指定验证器的情况下运行 **bundle validate** 命令，则默认测试会运行。默认测试会验证捆绑包是否遵循 Operator Framework 社区定义的规格。如需更多信息，请参阅“Bundle Format”。

您可以运行可选的验证器来测试 OperatorHub 兼容性或已弃用的 Kubernetes API 等问题。可选验证器总是在默认测试之外运行。

用于可选测试集的 `bundle validate` 命令语法

```
$ operator-sdk bundle validate <bundle_dir_or_image>
--select-optional <test_label>
```

表 5.22. 额外的 `bundle validate` 验证

Name	描述	标签
Operator Framework	这个验证器会根据 Operator Framework 提供的整个验证器套件测试 Operator 捆绑包。	suite=operatorframework
OperatorHub	这个验证器会测试 Operator 捆绑包以便与 OperatorHub 的兼容性。	name=operatorhub
最佳实践	此验证程序测试 Operator 捆绑包是否遵循 Operator Framework 定义的良好实践。它检查是否有问题，如空 CRD 描述或不支持的 Operator Lifecycle Manager(OLM) 资源。	name=good-practices

其他资源

- [捆绑包格式](#)

5.12.3. 运行 `bundle validate` 命令

每次进入 `bundle validate` 命令时，默认验证器都会运行测试。您可以使用 `--select-optional` 标志来运行可选验证器。可选验证器除默认测试外还运行测试。

先决条件

- 使用 Operator SDK 生成的 operator 项目

流程

1. 如果要针对本地捆绑包目录运行默认验证器，请从 Operator 项目目录中输入以下命令：

```
$ operator-sdk bundle validate ./bundle
```

2. 如果要针对远程 Operator 捆绑包镜像运行默认验证器，请输入以下命令：

```
$ operator-sdk bundle validate \
  <bundle_registry>/<bundle_image_name>:<tag>
```

其中：

`<bundle_registry>`

指定托管捆绑包的 registry，如 `quay.io/example`。

<bundle_image_name>

指定捆绑包镜像的名称，如 **memcached-operator**。

<tag>

指定捆绑包镜像的标签，如 **v1.31.0**。



注意

如果要验证 Operator 捆绑包镜像，则必须在远程 registry 中托管您的镜像。Operator SDK 在运行测试前拉取(pull)镜像并在本地构建。**bundle validate** 命令不支持测试本地捆绑包镜像。

3. 如果要针对 Operator 捆绑包运行附加验证器，请输入以下命令：

```
$ operator-sdk bundle validate \  
<bundle_dir_or_image> \  
--select-optional <test_label>
```

其中：

<bundle_dir_or_image>

指定本地捆绑包目录或远程捆绑包镜像，如 **~/projects/memcached** 或 **quay.io/example/memcached-operator:v1.31.0**。

<test_label>

指定您要运行的验证器的名称，如 **name=good-practices**。

输出示例

```
ERRO[0000] Error: Value apiextensions.k8s.io/v1, Kind=CustomResource: unsupported  
media type registry+v1 for bundle object  
WARN[0000] Warning: Value k8sevent.v0.0.1: owned CRD  
"k8sevents.k8s.k8sevent.com" has an empty description
```

5.12.4. 验证 Operator 的多平台就绪状态

您可以通过运行 **bundle validate** 命令来验证 Operator 的多平台就绪状态。该命令验证您的 Operator 项目是否满足以下条件：

- Operator 的管理器镜像支持集群服务版本(CSV)文件中标记的平台。
- 您的 Operator 的 CSV 具有 Operator Lifecycle Manager (OLM)和 OperatorHub 支持的平台标签。

流程

- 运行以下命令，以验证 Operator 项目是否有多个架构就绪：

```
$ operator-sdk bundle validate ./bundle \  
--select-optional name=multiarch
```

验证消息示例

INFO[0020] All validation tests have completed successfully

Manager 镜像中缺少 CSV 标签的错误消息示例

ERRO[0016] Error: Value test-operator.v0.0.1: not all images specified are providing the support described via the CSV labels. Note that (SO.architecture): (linux.ppc64le) was not found for the image(s) [quay.io/example-org/test-operator:v1alpha1]

ERRO[0016] Error: Value test-operator.v0.0.1: not all images specified are providing the support described via the CSV labels. Note that (SO.architecture): (linux.s390x) was not found for the image(s) [quay.io/example-org/test-operator:v1alpha1]

ERRO[0016] Error: Value test-operator.v0.0.1: not all images specified are providing the support described via the CSV labels. Note that (SO.architecture): (linux.amd64) was not found for the image(s) [quay.io/example-org/test-operator:v1alpha1]

ERRO[0016] Error: Value test-operator.v0.0.1: not all images specified are providing the support described via the CSV labels. Note that (SO.architecture): (linux.arm64) was not found for the image(s) [quay.io/example-org/test-operator:v1alpha1]

缺少 OperatorHub 标记的错误消息示例

WARN[0014] Warning: Value test-operator.v0.0.1: check if the CSV is missing the label (operatorframework.io/arch.<value>) for the Arch(s): ["amd64" "arm64" "ppc64le" "s390x"]. Be aware that your Operator manager image ["quay.io/example-org/test-operator:v1alpha1"] provides this support. Thus, it is very likely that you want to provide it and if you support more than amd64 architectures, you MUST use the required labels for all which are supported. Otherwise, your solution cannot be listed on the cluster for these architectures

其他资源

- [为多平台支持配置 Operator 项目](#)

5.13. 高可用性或单节点集群检测和支持

OpenShift Container Platform 集群能够以高可用性 (HA) 模式配置，该模式使用多个节点，或者在非 HA 模式中使用单一节点。单节点集群（也称为单节点 OpenShift）可能会有更保守的资源约束。因此，在单一节点集群中安装 Operator 务必要进行相应调整，并且仍然运行良好。

通过访问 OpenShift Container Platform 中提供的集群高可用性模式 API，Operator 作者可使用 Operator SDK 来让 Operator 检测集群的基础架构拓扑，不论是 HA 模式还是非 HA 模式。可以开发使用检测到的集群拓扑的自定义 Operator 逻辑，以自动将 Operator 及其管理的任何 Operands 或工作负载的资源要求切换到最适合拓扑的配置集。

5.13.1. 关于集群高可用性模式 API

OpenShift Container Platform 提供了一个集群高可用性模式 API，可供 Operator 用于帮助检测基础架构拓扑。基础架构 API 包含有关基础架构的集群范围信息。由 Operator Lifecycle Manager (OLM) 管理的操作员如果需要根据高可用性模式以不同的方式配置 Operand 或受管理的工作负载，则可以使用 Infrastructure API。

在 Infrastructure API 中，**infrastructureTopology** 状态表达了对未在 control plane 节点上运行的基础架构服务的期望，通常由节点选择器针对 **master** 以外的 **role** 值表示。**controlPlaneTopology** 状态表达了通常在 control plane 节点上运行的 Operand 的预期。

两个状态的默认设置都是 **HighlyAvailable**，它代表 Operator 在多个节点集群中具有的行为。**SingleReplica** 设置在单节点集群中（也称为单节点 OpenShift）中使用，表示 Operator 不应该为高可用性操作配置 Operands。

OpenShift Container Platform 安装程序根据以下规则，根据集群创建的副本数设置 **controlPlaneTopology** 和 **infrastructureTopology** 状态字段：

- 当 control plane 副本数小于 3 时，**controlPlaneTopology** 状态被设置为 **SingleReplica**。否则，它被设置为 **HighlyAvailable**。
- 当 worker 副本数为 0 时，control plane 节点也会配置为 worker。因此，**infrastructureTopology** 状态将与 **controlPlaneTopology** 状态相同。
- 当 worker 副本数为 1 时，**infrastructureTopology** 被设置为 **SingleReplica**。否则，它被设置为 **HighlyAvailable**。

5.13.2. Operator 项目中的 API 使用量示例

作为 Operator 作者，您可以使用普通的 Kubernetes 构造和 **controller-runtime** 库更新 Operator 项目以访问 Infrastructure API，如下例所示：

controller-runtime 库示例

```
// Simple query
nn := types.NamespacedName{
    Name: "cluster",
}
infraConfig := &configv1.Infrastructure{}
err = crClient.Get(context.Background(), nn, infraConfig)
if err != nil {
    return err
}
fmt.Printf("using crclient: %v\n", infraConfig.Status.ControlPlaneTopology)
fmt.Printf("using crclient: %v\n", infraConfig.Status.InfrastructureTopology)
```

Kubernetes 构造示例

```
operatorConfigInformer := configinformer.NewSharedInformerFactoryWithOptions(configClient,
2*time.Second)
infrastructureLister = operatorConfigInformer.Config().V1().Infrastructures().Lister()
infraConfig, err := configClient.ConfigV1().Infrastructures().Get(context.Background(), "cluster",
metav1.GetOptions{})
if err != nil {
    return err
}
// fmt.Printf("%v\n", infraConfig)
fmt.Printf("%v\n", infraConfig.Status.ControlPlaneTopology)
fmt.Printf("%v\n", infraConfig.Status.InfrastructureTopology)
```

5.14. 使用 PROMETHEUS 配置内置监控

本指南描述了 Operator SDK 使用 Prometheus Operator 提供的内置监控支持，以及使用基于 Go 和 Ansible 的 Operator 作者的详细信息。

5.14.1. 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。

5.14.2. 为基于 Go 的 Operator 公开自定义指标

作为 Operator 作者，您可以使用 **controller-runtime/pkg/metrics** 库中的全局 Prometheus registry 发布自定义指标。

先决条件

- 使用 Operator SDK 生成基于 Go 的 Operator
- Prometheus Operator，默认部署在 OpenShift Container Platform 集群中

流程

1. 在 Operator SDK 项目中，取消注释 **config/default/kustomization.yaml** 文件中的以下行：

```
../prometheus
```

2. 创建自定义控制器类，以便从 Operator 发布其他指标。以下示例将 **widgets** 和 **widgetFailures** 收集器声明为全局变量，然后将它们注册到控制器的软件包中的 **init()** 函数：

例 5.29. controllers/memcached_controller_test_metrics.go 文件

```
package controllers

import (
    "github.com/prometheus/client_golang/prometheus"
    "sigs.k8s.io/controller-runtime/pkg/metrics"
)

var (
    widgets = prometheus.NewCounter(
        prometheus.CounterOpts{
            Name: "widgets_total",
            Help: "Number of widgets processed",
        },
    )
    widgetFailures = prometheus.NewCounter(
        prometheus.CounterOpts{
            Name: "widget_failures_total",
            Help: "Number of failed widgets",
        },
    )
)

func init() {
```

```

// Register custom metrics with the global prometheus registry
metrics.Registry.MustRegister(widgets, widgetFailures)
}

```

- 记录这些收集器来自 **main** 控制器类中协调循环的任何部分，这决定了指标的业务逻辑：

例 5.30. controllers/memcached_controller.go 文件

```

func (r *MemcachedReconciler) Reconcile(ctx context.Context, req ctrl.Request)
(ctrl.Result, error) {
...
...
// Add metrics
widgets.Inc()
widgetFailures.Inc()

return ctrl.Result{}, nil
}

```

- 构建和推送 Operator：

```
$ make docker-build docker-push IMG=<registry>/<user>/<image_name>:<tag>
```

- 部署 Operator：

```
$ make deploy IMG=<registry>/<user>/<image_name>:<tag>
```

- 创建角色和角色绑定定义，以允许 OpenShift Container Platform 集群的 Prometheus 实例提取 Operator 的服务监控。
必须分配角色，以便服务帐户具有提取命名空间指标的权限：

例 5.31. config/prometheus/role.yaml 角色

```

apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: prometheus-k8s-role
  namespace: memcached-operator-system
rules:
- apiGroups:
  - ""
  resources:
  - endpoints
  - pods
  - services
  - nodes
  - secrets
verbs:
- get
- list
- watch

```


例 5.32. config/prometheus/rolebinding.yaml 角色绑定

```

apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: prometheus-k8s-rolebinding
  namespace: memcached-operator-system
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: prometheus-k8s-role
subjects:
  - kind: ServiceAccount
    name: prometheus-k8s
    namespace: openshift-monitoring

```

7. 为部署的 Operator 应用角色和绑定：

```
$ oc apply -f config/prometheus/role.yaml
```

```
$ oc apply -f config/prometheus/rolebinding.yaml
```

8. 为您要提取的命名空间设置标签，为该命名空间启用 OpenShift 集群监控：

```
$ oc label namespace <operator_namespace> openshift.io/cluster-monitoring="true"
```

验证

- 查询并查看 OpenShift Container Platform Web 控制台中的指标。您可以使用自定义控制器类中设置的名称，如 **widgets_total** 和 **widget_failures_total**。

5.14.3. 为基于 Ansible 的 Operator 公开自定义指标

作为创建基于 Ansible 的 Operator 的 Operator 作者，您可以使用 Operator SDK 的 **osdk_metrics** 模块来公开自定义 Operator 和 Operand 指标、发出事件和支持日志记录。

先决条件

- 使用 Operator SDK 生成的基于 Ansible 的 Operator
- Prometheus Operator，默认部署在 OpenShift Container Platform 集群中

流程

1. 生成基于 Ansible 的 Operator。这个示例使用 **testmetrics.com** 域：

```

$ operator-sdk init \
  --plugins=ansible \
  --domain=testmetrics.com

```

2. 创建 **metrics** API。这个示例使用名为 **Testmetrics** 的 **kind**：

-

```
$ operator-sdk create api \
  --group metrics \
  --version v1 \
  --kind Testmetrics \
  --generate-role
```

3. 编辑 `roles/testmetrics/tasks/main.yml` 文件，并使用 `osdk_metrics` 模块为 Operator 项目创建自定义指标：

例 5.33. roles/testmetrics/tasks/main.yml 文件示例

```
---
# tasks file for Memcached
- name: start k8sstatus
  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

- osdk_metric:
  name: my_thing_counter
  description: This metric counts things
  counter: {}

- osdk_metric:
  name: my_counter_metric
  description: Add 3.14 to the counter
  counter:
    increment: yes

- osdk_metric:
  name: my_gauge_metric
```

```

description: Create my gauge and set it to 2.
gauge:
  set: 2

- osdk_metric:
  name: my_histogram_metric
  description: Observe my histogram
  histogram:
    observe: 2

- osdk_metric:
  name: my_summary_metric
  description: Observe my summary
  summary:
    observe: 2

```

验证

1. 在集群上运行 Operator。例如，使用 "run 作为部署" 方法：

- a. 构建 Operator 镜像并将其推送到 registry：

```
$ make docker-build docker-push IMG=<registry>/<user>/<image_name>:<tag>
```

- b. 在集群上安装 Operator:

```
$ make install
```

- c. 部署 Operator：

```
$ make deploy IMG=<registry>/<user>/<image_name>:<tag>
```

2. 创建 **Testmetrics** 自定义资源(CR)：

- a. 定义 CR spec：

例 5.34. config/samples/metrics_v1_testmetrics.yaml 文件示例

```

apiVersion: metrics.testmetrics.com/v1
kind: Testmetrics
metadata:
  name: testmetrics-sample
spec:
  size: 1

```

- b. 创建对象：

```
$ oc create -f config/samples/metrics_v1_testmetrics.yaml
```

3. 获取 pod 详情：

```
$ oc get pods
```

-

输出示例

```

NAME                                READY STATUS  RESTARTS  AGE
ansiblemetrics-controller-manager- 2/2   Running  0         149m
testmetrics-sample-memcached-      1/1   Running  0         147m

```

4. 获取端点详情：

```
$ oc get ep
```

输出示例

```

NAME                                ENDPOINTS          AGE
ansiblemetrics-controller-manager-metrics-service 10.129.2.70:8443 150m

```

5. 请求自定义指标令牌：

```
$ token=`oc create token prometheus-k8s -n openshift-monitoring`
```

6. 检查指标值：

- a. 检查 **my_counter_metric** 值：

```
$ oc exec ansiblemetrics-controller-manager- -- curl -k -H "Authorization: Bearer $token" 'https://10.129.2.70:8443/metrics' | grep my_counter
```

输出示例

```

HELP my_counter_metric Add 3.14 to the counter
TYPE my_counter_metric counter
my_counter_metric 2

```

- b. 检查 **my_gauge_metric** 值：

```
$ oc exec ansiblemetrics-controller-manager- -- curl -k -H "Authorization: Bearer $token" 'https://10.129.2.70:8443/metrics' | grep gauge
```

输出示例

```
HELP my_gauge_metric Create my gauge and set it to 2.
```

- c. 检查 **my_histogram_metric** 和 **my_summary_metric** 值：

```
$ oc exec ansiblemetrics-controller-manager- -- curl -k -H "Authorization: Bearer $token" 'https://10.129.2.70:8443/metrics' | grep Observe
```

输出示例

```

HELP my_histogram_metric Observe my histogram
HELP my_summary_metric Observe my summary

```

5.15. 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，您可以在领导 pod 上指定 `node.kubernetes.io/unreachable` 和 `node.kubernetes.io/not-ready` 容限，并使用 `tolerationSeconds` 值来指定领导 pod 从节点删除所需的时间，并缩减。这些容限默认添加到 pod 的准入中，`tolerationSeconds` 值为 5 分钟。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。

5.15.1. 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。

5.15.1.1. 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。

5.15.1.2. 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。

5.16. 为多平台支持配置 OPERATOR 项目

与只支持单个平台的 Operator 项目相比，支持多架构和操作系统的 Operator 项目或平台可在更多的 Kubernetes 和 OpenShift Container Platform 集群上运行。示例构架包括 **amd64**, **arm64**, **ppc64le**, 和 **s390x**。示例操作系统包括 Linux 和 Windows。

执行以下操作以确保 Operator 项目可在多个 OpenShift Container Platform 平台上运行：

- 构建指定 Operator 支持的平台的清单列表。
- 设置 Operator 的节点关联性，以支持多架构计算机。

5.16.1. 构建 Operator 支持的平台清单列表

您可以使用 **make docker-buildx** 命令构建 Operator 和操作对象支持的平台清单列表。清单列表引用一个或多个架构的特定镜像清单。镜像清单指定镜像支持的平台。

如需更多信息，请参阅 [OpenContainers Image Index Spec](#) 或 [Image Manifest v2, Schema 2](#)。



重要

如果您的 Operator 项目部署了应用程序或其他工作负载资源，以下步骤会假定应用程序在应用程序发行过程中构建应用程序的多平台镜像。

先决条件

- 使用 Operator SDK 版本 1.31.0 或更高版本构建的 Operator 项目
- 已安装 Docker

流程

1. 检查 Operator 和操作对象的镜像清单，以查找 Operator 项目可以支持哪些平台。运行以下命令来检查镜像清单：

```
$ docker manifest inspect <image_manifest> ❶
```

- ❶ 指定镜像清单，如 **redhat/ubi9:latest**。

Operator 和操作对象相互支持的平台决定了 Operator 项目的平台兼容性。

输出示例

```
{
  "manifests": [
    {
      "digest":
"sha256:c0669ef34cdc14332c0f1ab0c2c01acb91d96014b172f1a76f3a39e63d1f0bda",
      "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
      "platform": {
        "architecture": "amd64",
        "os": "linux"
      },
      "size": 528
    },
    ...
    {
      "digest":
"sha256:30e6d35703c578ee703230b9dc87ada2ba958c1928615ac8a674fcbbcbb0f281",
      "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
      "platform": {
        "architecture": "arm64",
        "os": "linux",
        "variant": "v8"
      },
      "size": 528
    },
    ...
  ]
}
```

2. 如果上一命令没有输出平台信息，则指定的基础镜像可能是单个镜像，而不是镜像清单。您可以运行以下命令来查找镜像支持哪些架构：

```
$ docker inspect <image>
```

3. 对于基于 Go 的 Operator 项目，Operator SDK 会显式引用项目的 Dockerfile 中的 **amd64** 架构。对 Dockerfile 进行以下更改，将环境变量设置为 platform 标记指定的值：

Dockerfile 示例

```
FROM golang:1.19 as builder
ARG TARGETOS
ARG TARGETARCH
...
RUN CGO_ENABLED=0 GOOS=${TARGETOS:-linux} GOARCH=${TARGETARCH} go
build -a -o manager main.go ❶
```

- ❶ 将 **GOARCH** 字段从 **amd64** 更改为 **\$TARGETARCH**。

- 您的 Operator 项目的 makefile 定义 **PLATFORMS** 环境变量。如果 Operator 的镜像不支持默认设置的所有平台，请编辑变量以指定支持的平台。以下示例将支持的平台定义为 **linux/arm64** 和 **linux/amd64**：

makefile 示例

```
# ...
PLATFORMS ?= linux/arm64,linux/amd64 1
.PHONY: docker-buildx
# ...
```

- 以下 **PLATFORMS** 值默认设置：**linux/arm64**, **linux/amd64**, **linux/s390x**, 和 **linux/ppc64le**。

当您运行 **make docker buildx** 命令来生成清单列表时，Operator SDK 会为 **PLATFORMS** 变量指定的每个平台创建一个镜像清单。

- 从 Operator 项目目录运行以下命令，以构建管理器镜像。运行该命令构建具有多平台支持的管理器镜像，并将清单列表推送到 registry。

```
$ make docker-buildx \
  IMG=<image_registry>/<organization_name>/<repository_name>:<version_or_sha>
```

5.16.2. 关于多架构计算机器和 Operator 工作负载的节点关联性规则

您必须设置节点关联性规则，以确保 Operator 工作负载可以在多架构计算机器上运行。节点关联性是由调度程序用来定义 pod 的放置的一组规则。设置节点关联性规则可确保将 Operator 工作负载调度到具有兼容架构的计算机器。

如果 Operator 在特定构架上更好地执行，您可以设置首选的节点关联性规则，以将 pod 调度到具有指定架构的机器。

如需更多信息，请参阅“使用多架构计算机器打开集群”和“使用节点关联性规则控制节点上的 pod 放置”。

其他资源

- [使用节点关联性规则控制节点上的 pod 放置](#)
- [使用节点关联性来控制安装 Operator 的位置](#)
- [关于带有多架构计算机器的集群](#)

5.16.2.1. 使用所需的节点关联性规则来支持 Operator 项目的多架构计算机器

如果您希望 Operator 支持多架构计算机器，您必须定义 Operator 所需的节点关联性规则。

先决条件

- 使用 Operator SDK 1.31.0 或更高版本创建或维护的 Operator 项目。
- 定义 Operator 支持的平台的清单列表。

流程

1. 为定义 pod spec 和 pod 模板 spec 对象的 Kubernetes 清单搜索 Operator 项目。



重要

因为 YAML 文件中没有声明对象类型名称，请查找 Kubernetes 清单中的强制 **containers** 字段。在同时指定 pod spec 和 pod 模板 spec 对象时，需要 **containers** 字段。

您必须在定义 pod spec 或 pod 模板 spec 的所有 Kubernetes 清单中设置节点关联性规则，包括 **Pod**、**Deployment**、**DaemonSet** 和 **StatefulSet** 等对象。

Kubernetes 清单示例

```
apiVersion: v1
kind: Pod
metadata:
  name: s1
spec:
  containers:
    - name: <container_name>
      image: docker.io/<org>/<image_name>
```

2. 在 Kubernetes 清单中设置所需的节点关联性规则，以定义 pod spec 和 pod 模板 spec 对象，如下例所示：

Kubernetes 清单示例

```
apiVersion: v1
kind: Pod
metadata:
  name: s1
spec:
  containers:
    - name: <container_name>
      image: docker.io/<org>/<image_name>
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution: 1
        nodeSelectorTerms: 2
          - matchExpressions: 3
            - key: kubernetes.io/arch 4
              operator: In
              values:
                - amd64
                - arm64
                - ppc64le
                - s390x
            - key: kubernetes.io/os 5
              operator: In
              values:
                - linux
```

- 1** 定义必要规则。

- 2 如果您指定了多个与 **nodeAffinity** 类型关联的 **nodeSelectorTerms**，那么其中一个 **nodeSelectorTerms** 满足时 pod 就能调度到节点上。
 - 3 如果您指定了多个与 **nodeSelectorTerms** 关联的 **matchExpressions**，那么只有所有 **matchExpressions** 都满足时 pod 才能调度到节点上。
 - 4 指定清单列表中定义的架构。
 - 5 指定清单列表中定义的操作系统。
3. 使用动态创建工作负载的基于 Go 的 Operator 项目可能会将 pod spec 和 pod 模板 spec 对象嵌入到 Operator 的逻辑中。
如果您的项目在 Operator 的逻辑中嵌入 pod spec 或 pod 模板 spec 对象，请编辑类似以下示例的 Operator 逻辑。以下示例演示了如何使用 Go API 更新 **PodSpec** 对象：

```

Template: corev1.PodTemplateSpec{
  ...
  Spec: corev1.PodSpec{
    Affinity: &corev1.Affinity{
      NodeAffinity: &corev1.NodeAffinity{
        RequiredDuringSchedulingIgnoredDuringExecution: &corev1.NodeSelector{
          NodeSelectorTerms: []corev1.NodeSelectorTerm{
            {
              MatchExpressions: []corev1.NodeSelectorRequirement{
                {
                  Key:   "kubernetes.io/arch",
                  Operator: "In",
                  Values: []string{"amd64","arm64","ppc64le","s390x"},
                },
                {
                  Key:   "kubernetes.io/os",
                  Operator: "In",
                  Values: []string{"linux"},
                },
              },
            },
          },
        },
      },
    },
    SecurityContext: &corev1.PodSecurityContext{
      ...
    },
    Containers: []corev1.Container{{
      ...
    }},
  },
}

```

其中：

RequiredDuringSchedulingIgnoredDuringExecution

定义必要规则。

NodeSelectorTerms

如果您指定了多个与 **nodeAffinity** 类型关联的 **nodeSelectorTerms**，那么其中一个

如果您指定了多个与 `nodeAffinity` 关联的 `nodeSelectorTerms`，那么只有所有 `nodeSelectorTerms` 满足时 pod 就能调度到节点上。

matchExpressions

如果您指定了多个与 `nodeSelectorTerms` 关联的 `matchExpressions`，那么只有所有 `matchExpressions` 都满足时 pod 才能调度到节点上。

kubernetes.io/arch

指定清单列表中定义的架构。

kubernetes.io/os

指定清单列表中定义的操作系统。



警告

如果您没有设置节点关联性规则，且容器被调度到具有不兼容架构的计算机器，pod 会失败并触发以下事件之一：

CrashLoopBackOff

当镜像清单的入口点运行失败时，并在日志中包括 `exec format error` 信息。

ImagePullBackOff

当清单列表不包含调度 pod 的架构或节点关联性术语设置为错误的值时，会发生。

其他资源

- [配置节点关联性必要规则](#)
- [节点关联性规则示例](#)

5.16.2.2. 使用首选节点关联性规则为 Operator 项目配置对多架构计算机器的支持

如果 Operator 在特定构架上更好地执行，您可以配置首选的节点关联性规则，将 pod 调度到指定的架构。

先决条件

- 使用 Operator SDK 1.31.0 或更高版本创建或维护的 Operator 项目。
- 定义 Operator 支持的平台的清单列表。
- 为您的 Operator 项目设置所需的节点关联性规则。

流程

1. 为定义 pod spec 和 pod 模板 spec 对象的 Kubernetes 清单搜索 Operator 项目。

Kubernetes 清单示例

```
apiVersion: v1
kind: Pod
```

```

metadata:
  name: s1
spec:
  containers:
  - name: <container_name>
    image: docker.io/<org>/<image_name>

```

- 在 Kubernetes 清单中设置 Operator 的首选节点关联性规则，以定义 pod spec 和 pod 模板 spec 对象，如下例所示：

Kubernetes 清单示例

```

apiVersion: v1
kind: Pod
metadata:
  name: s1
spec:
  containers:
  - name: <container_name>
    image: docker.io/<org>/<image_name>
  affinity:
    nodeAffinity:
      preferredDuringSchedulingIgnoredDuringExecution: 1
      - preference:
          matchExpressions: 2
          - key: kubernetes.io/arch 3
            operator: In 4
            values:
            - amd64
            - arm64
          weight: 90 5

```

- 1** 定义偏好规则。
- 2** 如果您指定了多个与 **nodeSelectorTerms** 关联的 **matchExpressions**，那么只有所有 **matchExpressions** 都满足时 pod 才能调度到节点上。
- 3** 指定清单列表中定义的架构。
- 4** 指定一个 **operator**。Operator 可以是 **In**、**NotIn**、**Exists** 或 **DoesNotExist**。例如，使用 **In** 值要求节点上存在该标签。
- 5** 为节点指定权重，有效值为 **1-100**。优先选择权重最高的节点。

其他资源

- [配置首选的节点关联性规则](#)

5.16.3. 后续步骤

- [为 Operator Lifecycle Manager \(OLM\) 支持的平台添加标签](#)
- [捆绑 Operator 并使用 OLM 部署](#)

- 基于 Go 的 Operator 项目
- 基于 Ansible 的 Operator 项目
- 基于 Helm 的 Operator 项目
- 验证 Operator 的多平台就绪状态

5.17. 基于 GO 的 OPERATOR 的对象修剪工具

operator-lib 修剪工具使基于 Go 的 Operator 清理或修剪对象（当不再需要时）。Operator 作者也可以使用实用程序创建自定义 hook 和策略。

5.17.1. 关于 operator-lib 修剪工具

对象（如作业或 pod）作为 Operator 生命周期的一个普通部分创建。如果集群管理员或 Operator 没有删除这些对象，它们可以保留在集群中，并消耗资源。

在以前的版本中，可以使用以下选项修剪不必要的对象：

- Operator 作者必须为其 Operator 创建唯一的修剪解决方案。
- 集群管理员必须自行清理对象。

operator-lib pruning（修剪）工具 可为给定的命名空间从 Kubernetes 集群中删除对象。该程序库被添加到 **operator-lib** 库的 **0.9.0** 版本中，作为 Operator Framework 的一部分。

5.17.2. 修剪工具配置

operator-lib 修剪工具使用 Go 编写，包括基于 Go 的 Operator 的通用修剪策略。

配置示例

```
cfg = Config{
  log:      logf.Log.WithName("prune"),
  DryRun:   false,
  Clientset: client,
  LabelSelector: "app=<operator_name>",
  Resources: []schema.GroupVersionKind{
    {Group: "", Version: "", Kind: PodKind},
  },
  Namespaces: []string{"<operator_namespace>"},
  Strategy: StrategyConfig{
    Mode:      MaxCountStrategy,
    MaxCountSetting: 1,
  },
  PreDeleteHook: myhook,
}
```

修剪工具配置文件通过使用以下字段定义修剪操作：

配置字段	描述
------	----

配置字段	描述
log	用于处理库日志消息的日志记录器。
DryRun	确定是否应删除资源的布尔值。如果设置为 true ，则实用程序将运行，但不会删除资源。
Clientset	用于 Kubernetes API 调用的 client-go Kubernetes ClientSet。
LabelSelector	用于查找要修剪的资源的 Kubernetes 标签选择器表达式。
Resources	Kubernetes 资源类型。 PodKind 和 JobKind 目前被支持。
命名空间	用于搜索资源的 Kubernetes 命名空间列表。
策略	要运行的策略。
Strategy.Mode	目前支持 MaxCountStrategy 、 MaxAgeStrategy 或 CustomStrategy 。
Strategy.MaxCountSetting	MaxCountStrategy 的整数值，用于指定修剪实用程序运行后应保留多少个资源。
Strategy.MaxAgeSetting	Go time.Duration 字符串值，如 48h ，用于指定要修剪的资源的年龄。
Strategy.CustomSettings	制作可传递到自定义策略函数的值映射。
PreDeleteHook	可选：在修剪资源前调用 Go 功能。
CustomStrategy	可选：实现自定义修剪策略的 Go 功能

修剪执行

您可以通过在修剪配置上运行 `execute` 功能来调用修剪操作。

```
err := cfg.Execute(ctx)
```

您还可以使用 `cron` 软件包或通过触发器的事件调用修剪程序来调用修剪操作。

5.18. 将软件包清单项目迁移到捆绑包格式

OpenShift Container Platform 4.8 及更高版本中删除了对 Operator 的传统 *软件包清单格式* 的支持。如果您有一个最初使用软件包清单格式创建的 Operator 项目，您可以使用 Operator SDK 将项目迁移到捆绑包格式。从 OpenShift Container Platform 4.6 开始，捆绑包格式是 Operator Lifecycle Manager (OLM) 的首选打包格式。

5.18.1. 关于打包格式迁移

Operator SDK **pkgman-to-bundle** 命令有助于将 Operator Lifecycle Manager (OLM) 软件包清单迁移到捆绑包。命令采用输入软件包清单目录，并为输入目录中存在的每个清单版本生成捆绑包。然后，您还可以为每个生成的捆绑包构建捆绑包镜像。

例如，以软件包清单格式考虑项目的以下 **packagemanifests/** 目录：

软件包清单格式布局示例

```
packagemanifests/
├── etcd
│   ├── 0.0.1
│   │   ├── etcdcluster.crd.yaml
│   │   └── etcdoperator.clusterserviceversion.yaml
│   ├── 0.0.2
│   │   ├── etcdbackup.crd.yaml
│   │   ├── etcdcluster.crd.yaml
│   │   ├── etcdoperator.v0.0.2.clusterserviceversion.yaml
│   │   └── etcdrestore.crd.yaml
│   └── etcd.package.yaml
```

运行迁移后，**bundle/** 目录中会生成以下捆绑包：

捆绑包格式布局示例

```
bundle/
├── bundle-0.0.1
│   ├── bundle.Dockerfile
│   ├── manifests
│   │   ├── etcdcluster.crd.yaml
│   │   └── etcdoperator.clusterserviceversion.yaml
│   ├── metadata
│   │   └── annotations.yaml
│   ├── tests
│   │   └── scorecard
│   └── config.yaml
├── bundle-0.0.2
│   ├── bundle.Dockerfile
│   ├── manifests
│   │   ├── etcdbackup.crd.yaml
│   │   ├── etcdcluster.crd.yaml
│   │   ├── etcdoperator.v0.0.2.clusterserviceversion.yaml
│   │   └── etcdrestore.crd.yaml
│   ├── metadata
│   │   └── annotations.yaml
│   ├── tests
│   │   └── scorecard
│   └── config.yaml
```

根据生成的布局，两个捆绑包的捆绑包镜像也使用以下名称构建：

- **quay.io/example/etcd:0.0.1**
- **quay.io/example/etcd:0.0.2**

其他资源

- [Operator Framework 打包格式](#)

5.18.2. 迁移软件包清单项目到捆绑包格式

Operator 作者可以使用 Operator SDK 将软件包清单格式 Operator 项目迁移到捆绑包格式项目。

先决条件

- 已安装 operator SDK CLI
- Operator 项目最初使用 Operator SDK 以软件包清单格式生成

流程

- 使用 Operator SDK 将软件包清单项目迁移到捆绑包格式并生成捆绑包镜像：

```
$ operator-sdk pkgman-to-bundle <package_manifests_dir> \ 1
  [--output-dir <directory>] \ 2
  --image-tag-base <image_name_base> 3
```

- 1 指定项目的软件包清单目录的位置，如 **packagemanifests/** 或 **manifests/**。
- 2 可选：默认情况下，生成的捆绑包在本地写入磁盘到 **bundle/** 目录。您可以使用 **--output-dir** 标志来指定备选位置。
- 3 设置 **--image-tag-base** 标志，以提供镜像名称的基础，如 **quay.io/example/etcd**，它将用于捆绑包。提供一个没有标签的名称，因为镜像的标签将根据捆绑包版本进行设置。例如，生成完整捆绑包镜像名称的格式为 **<image_name_base>:<bundle_version>**。

验证

- 验证生成的捆绑包镜像是否成功运行：

```
$ operator-sdk run bundle <bundle_image_name>:<tag>
```

输出示例

```
INFO[0025] Successfully created registry pod: quay-io-my-etcd-0-9-4
INFO[0025] Created CatalogSource: etcd-catalog
INFO[0026] OperatorGroup "operator-sdk-og" created
INFO[0026] Created Subscription: etcdoperator-v0-9-4-sub
INFO[0031] Approved InstallPlan install-5t58z for the Subscription: etcdoperator-v0-9-4-sub
INFO[0031] Waiting for ClusterServiceVersion "default/etcdoperator.v0.9.4" to reach
'Succeeded' phase
INFO[0032] Waiting for ClusterServiceVersion "default/etcdoperator.v0.9.4" to appear
INFO[0048] Found ClusterServiceVersion "default/etcdoperator.v0.9.4" phase: Pending
INFO[0049] Found ClusterServiceVersion "default/etcdoperator.v0.9.4" phase: Installing
INFO[0064] Found ClusterServiceVersion "default/etcdoperator.v0.9.4" phase: Succeeded
INFO[0065] OLM has successfully installed "etcdoperator.v0.9.4"
```

5.19. 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。

5.19.1. bundle

operator-sdk bundle 命令管理 Operator 捆绑包元数据。

5.19.1.1. validate

bundle validate 子命令会验证 Operator 捆绑包。

表 5.23. 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 。
--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。

5.19.2. 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 5.24. 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 。

5.19.3. completion

operator-sdk completion 命令生成 shell completion，以便更迅速、更轻松发出 CLI 命令。

表 5.25. completion 子命令

子命令	描述
bash	生成 bash completion。
zsh	生成 zsh completion。

表 5.26. 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
```

5.19.4. create

operator-sdk create 命令用于创建或 *scaffold* Kubernetes API。

5.19.4.1. 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 5.27. create api 标记

标记	描述
-h, --help	run bundle 子命令的帮助输出。

5.19.5. generate

operator-sdk generate 命令调用特定的生成器来生成代码或清单。

5.19.5.1. 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 5.28. **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)	CustomResourceDefinition 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
--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** 命令调用 **generate bundle** 子命令的完整流程，请参阅[捆绑 Operator](#)。

5.19.5.2. kustomize

generate kustomize 子命令包含为 Operator 生成 [Kustomize](#) 数据的子命令。

5.19.5.2.1. 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成部分，除非基础已存在或设置了 **--interactive=false** 标志。

表 5.29. 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。
--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。

5.19.6. 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 5.30. `init` 标记

标记	描述
<code>--help, -h</code>	<code>init</code> 命令的帮助输出。
<code>--plugins</code> (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 <code>ansible.sdk.operatorframework.io/v1</code> 、 <code>go.kubebuilder.io/v2</code> 、 <code>go.kubebuilder.io/v3</code> 和 <code>helm.sdk.operatorframework.io/v1</code> 。
<code>--project-version</code>	项目版本。可用值为 2 和 3-alpha (默认值)。

5.19.7. run

`operator-sdk run` 命令提供可在各种环境中启动 Operator 的选项。

5.19.7.1. bundle

`run bundle` 子命令使用 Operator Lifecycle Manager (OLM) 以捆绑包格式部署 Operator。

表 5.31. `run bundle` 标记

标记	描述
<code>--index-image</code> (字符串)	在其中注入捆绑包的索引镜像。默认镜像为 <code>quay.io/operator-framework/upstream-opm-builder:latest</code> 。
<code>--install-mode</code> <code><install_mode_value></code>	安装 Operator 的集群服务版本 (CSV) 支持的模式，如 <code>AllNamespaces</code> 或 <code>SingleNamespace</code> 。
<code>--timeout <duration></code>	安装超时。默认值为 <code>2m0s</code> 。
<code>--kubeconfig</code> (string)	用于 CLI 请求的 <code>kubeconfig</code> 文件的路径。
<code>-n, --namespace</code> (字符串)	如果存在，代表在其中运行 CLI 请求的命名空间。
<code>--security-context-config</code> <code><security_context></code>	指定用于目录 pod 的安全上下文。允许的值包括 <code>restricted</code> 和 <code>legacy</code> 。默认值为 <code>legacy</code> 。 ^[1]
<code>-h, --help</code>	<code>run bundle</code> 子命令的帮助输出。

1. `restricted` 安全上下文与 `default` 命名空间不兼容。要在生产环境中配置 Operator 的 pod 安全准入，请参阅"Complying with pod 安全准入"。如需有关 pod 安全准入的更多信息，请参阅"了解和管理 pod 安全准入"。

其他资源

- 有关可能安装模式的详情，请参阅 [Operator 组成员资格](#)。
- [遵守 pod 安全准入](#)
- [了解并管理 pod 安全准入](#)

5.19.7.2. 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 5.32. 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。
--security-context-config <security_context>	指定用于目录 pod 的安全上下文。允许的值包括 restricted 和 legacy 。默认值为 legacy 。 ^[1]
-h, --help	run bundle 子命令的帮助输出。

1. **restricted** 安全上下文与 **default** 命名空间不兼容。要在生产环境中配置 Operator 的 pod 安全准入，请参阅“Complying with pod 安全准入”。如需有关 pod 安全准入的更多信息，请参阅“了解和 管理 pod 安全准入”。

其他资源

- [遵守 pod 安全准入](#)
- [了解并管理 pod 安全准入](#)

5.19.8. 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 5.33. 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 。

标记	描述
-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 。
--pod-security <security_context>	使用指定安全上下文运行 scorecard 的选项。允许的值包括 restricted 和 legacy 。默认值为 legacy 。 ^[1]
-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 。

1. **restricted** 安全上下文与 **default** 命名空间不兼容。要在生产环境中配置 Operator 的 pod 安全准入，请参阅“Complying with pod 安全准入”。如需有关 pod 安全准入的更多信息，请参阅“了解和管理 pod 安全准入”。

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](#)。
- [遵守 pod 安全准入](#)
- [了解并管理 pod 安全准入](#)

第 6 章 集群 OPERATOR 参考

本指南对红帽提供的集群 *Operator* 进行了索引，该 *Operator* 充当 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。

安装前可以禁用以下一些集群 *Operator*。如需更多信息，请参阅[集群功能](#)。

6.1. CLUSTER BAREMETAL OPERATOR



注意

Cluster Baremetal Operator 是一个可选的集群功能，可在安装过程中由集群管理员禁用。有关可选集群功能的更多信息，请参阅 [安装中的“集群功能”](#)。

用途

Cluster Baremetal Operator (CBO) 会部署使裸机服务器成为一个可完全正常工作的节点以运行 OpenShift Container Platform 计算节点所需的所有组件。CBO 确保 metal3 部署（由 Bare Metal Operator (BMO) 和 Ironic 容器组成）在 OpenShift Container Platform 集群内的一个 control plane 节点上运行。CBO 还会侦听 OpenShift Container Platform 对资源的更新，它会监视并采取适当的操作。

项目

[cluster-baremetal-operator](#)

其他资源

- [裸机功能](#)

6.2. 裸机事件中继

用途

OpenShift Bare Metal Event Relay 管理 Bare Metal Event Relay 的生命周期。Bare Metal 事件 Relay 允许您配置使用 Redfish 硬件事件监控的集群事件类型。

Configuration objects

您可以在安装后使用这个命令来编辑配置：例如 webhook 端口。您可以使用以下方法编辑配置对象：

```
$ oc -n [namespace] edit cm hw-event-proxy-operator-manager-config
```

```
apiVersion: controller-runtime.sigs.k8s.io/v1alpha1
kind: ControllerManagerConfig
health:
  healthProbeBindAddress: :8081
```



```

metrics:
  bindAddress: 127.0.0.1:8080
webhook:
  port: 9443
leaderElection:
  leaderElect: true
  resourceName: 6e7a703c.redhat-cne.org

```

项目

[hw-event-proxy-operator](#)

CRD

代理使在裸机集群中运行的应用程序可以快速响应 Redfish 硬件更改和故障，如违反温度阈值、故障、磁盘丢失、电源中断和内存故障，使用 HardwareEvent CR 报告。

hardwareevents.event.redhat-cne.org:

- Scope: Namespaced
- CR: HardwareEvent
- Validation: Yes

6.3. 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 ("")。

项目

[openshift-cloud-credential-operator](#)

CRD

- **credentialsrequests.cloudcredential.openshift.io**
 - Scope: Namespaced
 - CR : **CredentialsRequest**
 - Validation: Yes

Configuration objects

不需要配置。

其他资源

- [关于 Cloud Credential Operator](#)
- [CredentialsRequest 自定义资源](#)

6.4. 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](#)

6.5. 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。

6.6. CLUSTER CLOUD CONTROLLER MANAGER OPERATOR

用途



注意

对于 Amazon Web Services (AWS), Google Cloud Platform (GCP), IBM Cloud®, global Microsoft Azure, Microsoft Azure Stack Hub, Nutanix, Red Hat OpenStack Platform (RHOSP), 和 VMware vSphere, 这个 Operator 的状态是正式发布 (GA)。

对于 Alibaba Cloud 和 IBM Power® Virtual Server, 这个 Operator 是 [技术预览](#)。

Cluster Cloud Controller Manager Operator 管理并更新在 OpenShift Container Platform 上部署的云控制器管理器。Operator 基于 Kubebuilder 框架和 **controller-runtime** 库。它通过 Cluster Version Operator (CVO) 安装。

它包含以下组件：

- Operator
- 云配置观察

默认情况下，Operator 通过 **metrics** 服务公开 Prometheus 指标数据。

项目

[cluster-cloud-controller-manager-operator](#)

6.7. CLUSTER CAPI OPERATOR



注意

对于 Amazon Web Services (AWS)和 Google Cloud Platform (GCP)集群，这个 Operator 作为[技术预览](#)提供。

用途

Cluster CAPI Operator 维护 Cluster API 资源的生命周期。此 Operator 负责在 OpenShift Container Platform 集群中部署 Cluster API 项目的所有管理任务。

项目

[cluster-capi-operator](#)

CRD

- **awsmachines.infrastructure.cluster.x-k8s.io**
 - Scope: Namespaced
 - CR: **awsmachine**
 - Validation: No
- **gcpmachines.infrastructure.cluster.x-k8s.io**
 - Scope: Namespaced
 - CR: **gcpmachine**
 - Validation: No
- **awsmachinetemplates.infrastructure.cluster.x-k8s.io**
 - Scope: Namespaced
 - CR: **awsmachinetemplate**
 - Validation: No
- **gcpmachinetemplates.infrastructure.cluster.x-k8s.io**
 - Scope: Namespaced
 - CR: **gcpmachinetemplate**
 - Validation: No

6.8. 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。](#)

项目

[cluster-config-operator](#)

6.9. CLUSTER CSI SNAPSHOT CONTROLLER OPERATOR



注意

Cluster CSI Snapshot Controller Operator 是一个可选集群功能，集群管理员可在安装过程中禁用。有关可选集群功能的更多信息，请参阅 [安装中的"集群功能"](#)。

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。

项目

[cluster-csi-snapshot-controller-operator](#)

其他资源

- [CSI 快照控制器功能](#)

6.10. CLUSTER IMAGE REGISTRY OPERATOR

用途

Cluster Image Registry Operator 管理 OpenShift 镜像 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](#)

6.11. 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。

项目

[cluster-machine-approver-operator](#)

6.12. 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
```

6.13. CLUSTER NETWORK OPERATOR

用途

Cluster Network Operator 在 OpenShift Container Platform 集群上安装并升级网络组件。

6.14. CLUSTER SAMPLES OPERATOR



注意

Cluster Samples Operator 是一个可选集群功能，集群管理员可在安装过程中禁用。有关可选集群功能的更多信息，请参阅 [安装中的"集群功能"](#)。

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。开始时，OpenShift 镜像 registry 中的镜像流导入逻辑和 API 服务器会使用安装 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](#)

其他资源

- [OpenShift 示例功能](#)

6.15. CLUSTER STORAGE OPERATOR



注意

Cluster Storage Operator 是一个可选集群功能，集群管理员可在安装过程中禁用。有关可选集群功能的更多信息，请参阅 [安装中的"集群功能"](#)。

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 OpenShift Container Platform 集群存在默认存储类。它还安装 Container Storage Interface (CSI) 驱动程序，使集群能够使用各种存储后端。

项目

[cluster-storage-operator](#)

Configuration

不需要配置。

备注

- Operator 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，这个存储类就无法被删除。

其他资源

- [存储功能](#)

6.16. 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，以根据图中的当前组件版本和信息来查看有效的更新和更新路径，方法是收集集群版本及其集群 Operator 的状态。此状态包括条件类型，它告知您 OpenShift Container Platform 集群的健康状态和当前状态。

如需有关集群版本状况类型的更多信息，请参阅“了解集群版本状况类型”。

项目

[cluster-version-operator](#)

其他资源

- [了解集群版本状况类型](#)

6.17. CONSOLE OPERATOR



注意

Console Operator 是一个可选集群功能，集群管理员可在安装过程中禁用。如果您在安装时禁用了 Console Operator，您的集群仍被支持并可升级。有关可选集群功能的更多信息，请参阅安装中的“集群功能”。

用途

Console Operator 在集群中安装和维护 OpenShift Container Platform web 控制台。Console Operator 会被默认安装，并自动维护控制台。

项目

[console-operator](#)

其他资源

- [Web 控制台功能](#)

6.18. CONTROL PLANE MACHINE SET OPERATOR



注意

此 Operator 可用于 Amazon Web Services (AWS)、Google Cloud Platform (GCP)、Microsoft Azure、Nutanix 和 VMware vSphere。

用途

Control Plane Machine Set Operator 自动管理 OpenShift Container Platform 集群中的 control plane 机器资源。

项目

[cluster-control-plane-machine-set-operator](#)

CRD

- **controlplanemachineset.machine.openshift.io**
 - Scope: Namespaced
 - CR: **ControlPlaneMachineSet**
 - Validation: Yes

其他资源

- [关于 control plane 机器集](#)
- [ControlPlaneMachineSet 自定义资源](#)

6.19. 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**。
- 尚不支持配置 CoreDNS Corefile 或 Kubernetes 插件。

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](#)

6.20. 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
```

6.21. INGRESS OPERATOR

用途

Ingress Operator 配置并管理 OpenShift Container Platform 路由。

项目

[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 Controller 在只纯 IPv6 模式下运行。

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]
```

6.22. INSIGHTS OPERATOR



注意

Insights Operator 是一个可选集群功能，集群管理员可在安装过程中禁用。有关可选集群功能的更多信息，请参阅 [安装中的“集群功能”](#)。

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。

项目

[insights-operator](#)

配置

不需要配置。

备注

Insights Operator 补充 OpenShift Container Platform Telemetry。

其他资源

- [Insights 功能](#)
- 如需有关 Insights Operator 和 Telemetry 的详细信息，请参阅 [关于远程健康监控](#)。

6.23. 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 Container Platform **library-go** 框架，它与 Cluster Version Operator (CVO) 一起安装。

项目

[openshift-kube-apiserver-operator](#)

CRD

- [kubeadpiservers.operator.openshift.io](#)

- Scope: Cluster
- CR: **kubeapiserver**
- Validation: Yes

Configuration objects

```
$ oc edit kubeapiserver
```

6.24. 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 Container Platform **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](#)

6.25. 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](https://github.com/openshift/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。

6.26. 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。

项目

[cluster-kube-storage-version-migrator-operator](#)

6.27. 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**

6.28. MACHINE CONFIG OPERATOR

用途

Machine Config 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。



重要

目前，不支持阻止或限制机器配置服务器端点。机器配置服务器必须公开给网络，以便新置备的机器没有现有配置或状态，才能获取其配置。在这个模型中，信任的根是证书签名请求 (CSR) 端点，即 kubelet 发送其证书签名请求以批准加入集群。因此，机器配置不应用于分发敏感信息，如 secret 和证书。

为确保机器配置服务器端点，端口 22623 和 22624 在裸机场景中是安全的，客户必须配置正确的网络策略。

其他资源

- [关于 OpenShift SDN 网络插件。](#)

项目

[openshift-machine-config-operator](#)

6.29. MARKETPLACE OPERATOR



注意

Marketplace Operator 是一个可选集群功能，如果不需要，集群管理员可以禁用它。有关可选集群功能的更多信息，请参阅 [安装中的“集群功能”](#)。

用途

Marketplace Operator 通过使用集群中的一组默认 Operator Lifecycle Manager (OLM) 目录简化了将非集群 Operator 引入集群的过程。安装 Marketplace Operator 时，它会创建 **openshift-marketplace** 命名空间。OLM 确保在 **openshift-marketplace** 命名空间中安装的目录源可用于集群中的所有命名空间。

项目

[operator-marketplace](#)

其他资源

- [Marketplace 功能](#)

6.30. 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，并使用 Performance Profile 控制器获得低延迟性能。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。

Node Tuning Operator 使用 Performance Profile 控制器来实现自动性能优化，从而实现 OpenShift Container Platform 应用程序的低延迟性能。

集群管理员配置了性能配置集以定义节点级别的设置，例如：

- 将内核更新至 kernel-rt。
- 为内务选择 CPU。
- 为运行工作负载选择 CPU。



注意

目前，cgroup v2 不支持禁用 CPU 负载均衡。因此，如果您启用了 cgroup v2，则可能无法从性能配置集中获取所需的行为。如果您使用 `executeace` 配置集，则不建议启用 cgroup v2。

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。



注意

在早期版本的 OpenShift Container Platform 中，Performance Addon Operator 用来实现自动性能优化，以便为 OpenShift 应用程序实现低延迟性能。在 OpenShift Container Platform 4.11 及更新的版本中，这个功能是 Node Tuning Operator 的一部分。

项目

[cluster-node-tuning-operator](#)

其他资源

- [关于低延迟](#)

6.31. OPENSIFT 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

6.32. OPENSIFT 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
```

项目

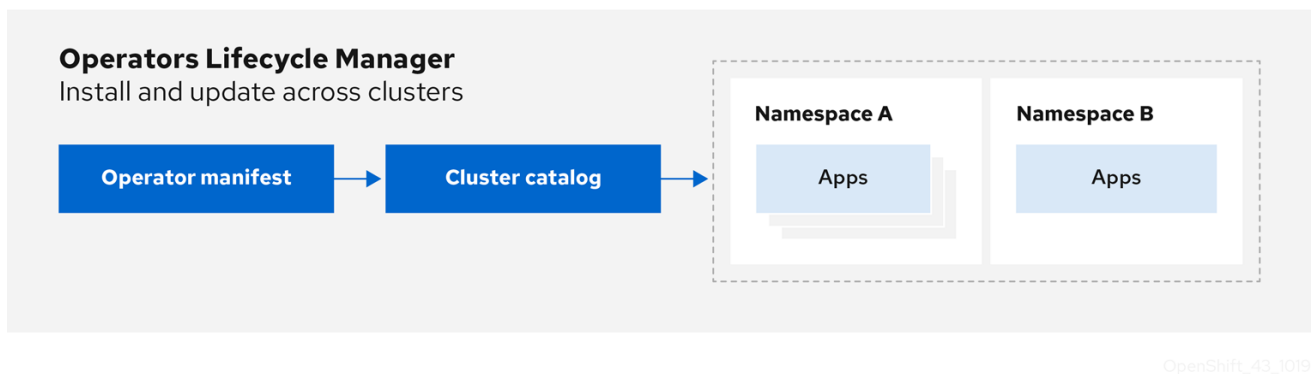
[cluster-openshift-controller-manager-operator](#)

6.33. 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 workflow



OpenShift_43_1019

OLM 默认在 OpenShift Container Platform 4.15 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	catalog	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	

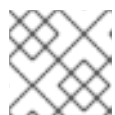
OLM Operator

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 CSV 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

- 如需更多信息，请参阅[了解 Operator Lifecycle Manager \(OLM\)](#) 的章节。

6.34. OPENSIFT SERVICE CA OPERATOR

用途

OpenShift Service CA Operator 管理 Kubernetes 服务的证书。

项目

[openshift-service-ca-operator](#)

6.35. 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。

配置

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。

其他资源

- 如需了解更多详细信息，请参阅[使用 vSphere Problem Detector Operator](#)。

第 7 章 OLM 1.0 (技术预览)

7.1. 关于 OPERATOR LIFECYCLE MANAGER 1.0 (技术预览)

自 OpenShift Container Platform 4 初始发行以来，Operator Lifecycle Manager (OLM) 已包含在 OpenShift Container Platform 4 中。OpenShift Container Platform 4.14 引入了用于 OLM 的下一代迭代组件作为技术预览功能，在这个阶段称为 *OLM 1.0*。此更新的框架改变了很多属于以前版本的 OLM 的概念，并添加了新功能。



重要

OLM 1.0 只是一个技术预览功能。技术预览功能不受红帽产品服务等级协议 (SLA) 支持，且功能可能并不完整。红帽不推荐在生产环境中使用它们。这些技术预览功能可以使用户提早试用新的功能，并有机会在开发阶段提供反馈意见。

有关红帽技术预览功能支持范围的更多信息，请参阅[技术预览功能支持范围](#)。

在 OpenShift Container Platform 4.14 中 OLM 1.0 的技术预览阶段，管理员可以探索以下功能：

支持 GitOps 工作流的全声明性模型

OLM 1.0 通过两个 API 简化了 Operator 管理：

- 一个新的 **Operator API** `operator.operators.operatorframework.io` 由新的 Operator Controller 组件提供，通过将面向用户的 API 整合到单个对象来简化已安装的 Operator 的管理。这让管理员和 SRE 能够使用 GitOps 原则自动化进程并定义所需的状态。
- **Catalog API** 由新 `catalogd` 组件提供，充当 OLM 1.0 的基础，为 on-cluster 客户端解包目录，使用户可以发现可安装的内容，如 Operator 和 Kubernetes 扩展。这可让您提高所有可用 Operator 捆绑包版本的可见性，包括它们的详情、频道和更新边缘。

如需更多信息，请参阅 [Operator Controller](#) 和 [Catalogd](#)。

改进了对 Operator 更新的控制

通过改进对目录内容的了解，管理员可以指定用于安装和更新的目标版本。这可让管理员对 Operator 更新的目标版本进行更多控制。如需更多信息，请参阅[更新 Operator](#)。

灵活的 Operator 打包格式

管理员可以使用基于文件的目录来安装和管理以下类型的内容：

- 基于 OLM 的 Operator，类似于现有的 OLM 体验
- **普通捆绑包**，它们是任意 Kubernetes 清单的静态集合

另外，捆绑包大小不再受 `etcd` 值大小限制。如需更多信息，请参阅从 [目录安装 Operator](#) 和 [管理普通捆绑包](#)。

7.1.1. 用途

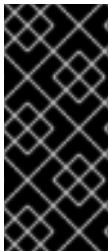
Operator Lifecycle Manager (OLM) 的任务是集中管理集群扩展的生命周期，并在 Kubernetes 集群上以声明性的方式管理。其目的是在整个基础集群生命周期中，对集群管理员和平台即服务(PaaS)管理员的安装、运行和更新功能扩展变得简单、安全且可重复生成。

由 OpenShift Container Platform 4 启动的 OLM 的初始版本，默认包含在内，专注于为特定类型的集群扩展（称为 Operator）提供特殊支持。Operator 被归类为一个或多个 Kubernetes 控制器，与一个或多个 API 扩展(**CustomResourceDefinition** (CRD)) 一起提供，以为集群提供额外的功能。

在生产环境集群中运行了多个版本后，OLM 的下一代产品旨在管理集群扩展（不仅限于 Operator）的生命周期。

7.2. 组件和架构

7.2.1. OLM 1.0 组件概述（技术预览）



重要

OLM 1.0 只是一个技术预览功能。技术预览功能不受红帽产品服务等级协议 (SLA) 支持，且功能可能并不完整。红帽不推荐在生产环境中使用它们。这些技术预览功能可以使用户提早试用新的功能，并有机会在开发阶段提供反馈意见。

有关红帽技术预览功能支持范围的更多信息，请参阅[技术预览功能支持范围](#)。

Operator Lifecycle Manager (OLM) 1.0 包含以下组件项目：

Operator 控制器

Operator Controller 是 OLM 1.0 的核心组件，通过 API 扩展 Kubernetes，用户可以安装和管理 Operator 和扩展的生命周期。它消耗来自以下每个组件的信息。

RukPak

RukPak 是一个可插拔式解决方案，用于打包和分发云原生内容。它支持安装、更新和策略的高级策略。

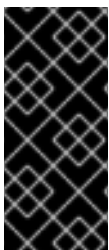
RukPak 提供用于在 Kubernetes 集群上安装各种工件的内容生态系统。工件示例包括 Git 仓库、Helm chart 和 OLM 捆绑包。然后，RukPak 可以以安全的方式管理、扩展和升级这些工件，以启用强大的集群扩展。

Catalogd

Catalogd 是一个 Kubernetes 扩展，它解包基于文件的目录(FBC)内容，由容器镜像打包并提供，供集群客户端使用。作为 OLM 1.0 微服务架构的组件，用于由扩展作者打包的 Kubernetes 扩展的目录主机元数据，因此可帮助用户发现可安装的内容。

7.2.2. Operator 控制器（技术预览）

Operator Controller 是 Operator Lifecycle Manager (OLM) 1.0 的核心组件，并消耗其他 OLM 1.0 组件、RukPak 和 catalogd。它使用一个 API 扩展 Kubernetes，用户可以安装 Operator 和扩展。



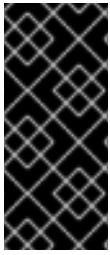
重要

OLM 1.0 只是一个技术预览功能。技术预览功能不受红帽产品服务等级协议 (SLA) 支持，且功能可能并不完整。红帽不推荐在生产环境中使用它们。这些技术预览功能可以使用户提早试用新的功能，并有机会在开发阶段提供反馈意见。

有关红帽技术预览功能支持范围的更多信息，请参阅[技术预览功能支持范围](#)。

7.2.2.1. Operator API

Operator Controller 提供了一个新的 **Operator** API 对象，它是一个代表已安装 Operator 实例的单个资源。此 **operator.operators.operatorframework.io** API 通过将面向用户的 API 整合到单个对象来简化已安装的 Operator 管理。



重要

在 OLM 1.0 中，**Operator** 对象是集群范围的。这与早期的 OLM 版本不同，Operator 可以是命名空间范围的或集群范围的，具体取决于其相关 **Subscription** 和 **OperatorGroup** 对象的配置。

如需有关之前行为的更多信息，请参阅 [多租户和 Operator 共处](#)。

Operator 对象示例

```
apiVersion: operators.operatorframework.io/v1alpha1
kind: Operator
metadata:
  name: <operator_name>
spec:
  packageName: <package_name>
  channel: <channel_name>
  version: <version_number>
```



注意

当使用 OpenShift CLI (**oc**) 时，由在此技术预览阶段中的 OLM 1.0 所提供的 **Operator** 资源需要指定完整的 **<resource>.<group>** 格式：

operator.operators.operatorframework.io。例如：

```
$ oc get operator.operators.operatorframework.io
```

如果您在没有 API 组的情况下只指定 **Operator** 资源，CLI 会为早期的 API (**operator.operators.coreos.com**) 返回的结果与 OLM 1.0 不相关。

其他资源

- [Operator Lifecycle Manager \(OLM\)→ Multitenancy 和 Operator colocation](#)

7.2.2.1.1. 指定目标版本的自定义资源(CR)示例

在 Operator Lifecycle Manager (OLM) 1.0 中，集群管理员可以在自定义资源(CR)中声明性地设置 Operator 或扩展的目标版本。

您可以通过指定以下字段来定义目标版本：

- Channel
- 版本号
- 版本范围

如果您在 CR 中指定频道，OLM 1.0 会安装可在指定频道中解析的 Operator 或扩展的最新版本。当向指定的频道发布更新时，OLM 1.0 会自动更新至可以从频道解析的最新发行版本。

带有指定频道的 CR 示例

```
apiVersion: operators.operatorframework.io/v1alpha1
kind: Operator
metadata:
  name: pipelines-operator
spec:
  packageName: openshift-pipelines-operator-rh
  channel: latest ❶
```

- ❶ 安装可从指定频道解析的最新版本。对频道的更新会自动安装。

如果在 CR 中指定 Operator 或扩展的目标版本，OLM 1.0 将安装指定的版本。当在 CR 中指定目标版本时，OLM 1.0 在向目录发布更新时不会更改目标版本。

如果要更新集群中安装的 Operator 版本，您必须手动编辑 Operator 的 CR。指定 Operator 的目标版本将 Operator 的版本固定到指定的发行版本。

指定了目标版本的 CR 示例

```
apiVersion: operators.operatorframework.io/v1alpha1
kind: Operator
metadata:
  name: pipelines-operator
spec:
  packageName: openshift-pipelines-operator-rh
  version: 1.11.1 ❶
```

- ❶ 指定目标版本。如果要更新安装的 Operator 或扩展版本，您必须手动将 CR 更新至所需的目标版本。

如果要为 Operator 或扩展定义可接受的版本范围，您可以使用比较字符串指定版本范围。当您指定版本范围时，OLM 1.0 会安装可由 Operator Controller 解析的 Operator 或扩展的最新版本。

指定了版本范围的 CR 示例

```
apiVersion: operators.operatorframework.io/v1alpha1
kind: Operator
metadata:
  name: pipelines-operator
spec:
  packageName: openshift-pipelines-operator-rh
  version: >1.11.1 ❶
```

- ❶ 指定所需的版本范围大于 1.11.1 版本。如需更多信息，请参阅“支持版本范围”。

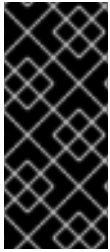
创建或更新 CR 后，运行以下命令来应用配置文件：

命令语法

```
$ oc apply -f <extension_name>.yaml
```

7.2.3. Rukpak (技术预览)

Operator Lifecycle Manager (OLM) 1.0 使用 RukPak 组件及其资源来管理云原生内容。



重要

OLM 1.0 只是一个技术预览功能。技术预览功能不受红帽产品服务等级协议 (SLA) 支持，且功能可能并不完整。红帽不推荐在生产环境中使用它们。这些技术预览功能可以使用户提早试用新的功能，并有机会在开发阶段提供反馈意见。

有关红帽技术预览功能支持范围的更多信息，请参阅[技术预览功能支持范围](#)。

7.2.3.1. 关于 RukPak

RukPak 是一个可插拔式解决方案，用于打包和分发云原生内容。它支持安装、更新和策略的高级策略。

RukPak 提供用于在 Kubernetes 集群上安装各种工件的内容生态系统。工件示例包括 Git 仓库、Helm chart 和 OLM 捆绑包。然后，RukPak 可以以安全的方式管理、扩展和升级这些工件，以启用强大的集群扩展。

在其核心上，RukPak 是一组 API 和控制器。API 打包为自定义资源定义 (CRD)，用于表达要在集群中安装的内容以及如何创建运行的内容。控制器会监视 API。

常见术语

捆绑包 (Bundle)

定义要部署到集群的内容的 Kubernetes 清单集合

捆绑包镜像

在其文件系统中包含捆绑包的容器镜像

捆绑包 Git 存储库

在目录中包含捆绑包的 Git 存储库

Provisioner

在 Kubernetes 集群上安装和管理内容的控制器

捆绑包部署

生成捆绑包部署的实例

7.2.3.2. 关于置备程序

RukPak 由一系列控制器组成，称为 *provisioners*，后者在 Kubernetes 集群上安装和管理内容。RukPak 还提供两个主要 API：**Bundle** 和 **BundleDeployment**。这些组件协同工作，将内容整合到集群中，并安装它，并在集群中生成资源。

目前，两个置备程序被实施并捆绑到 RukPak：*plain provisioner source* 并解包 **plain+v0** 捆绑包；*registry provisioner source* Operator Lifecycle Manager (OLM) **registry+v1** 捆绑包。

每个置备程序都会被分配一个唯一 ID，它负责协调与该特定 ID 匹配的 **spec.provisionerClassName** 字段的 **Bundle** 和 **BundleDeployment** 对象。例如，普通置备程序可以解包给定 **plain+v0** 捆绑包到集群中，然后实例化它，使捆绑包的内容在集群中可用。

置备程序会在 **Bundle** 和 **BundleDeployment** 资源上明确观察到引用置备程序的资源。对于给定捆绑包，置备程序会将 **Bundle** 资源的内容解压缩到集群中。然后，如果一个指向该捆绑包的 **BundleDeployment** 资源，置备程序会安装捆绑包内容，并负责管理这些资源的生命周期。

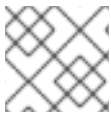
7.2.3.3. 捆绑包 (Bundle)

RukPak **Bundle** 对象代表可供集群中的其他使用者使用的内容。就像需要拉取和解包容器镜像的内容一样，pod 才能开始使用它们，使用 **Bundle** 对象来引用可能需要拉取和解包的内容。因此，捆绑包是镜像概念的规范化，可用于表示任何类型的内容。

捆绑包无法自行执行；它们需要置备程序来解包并在集群中提供其内容。它们可以解包到任何任意存储介质，如挂载到 provisioner pod 目录中的 **tar.gz** 文件。每个 **Bundle** 对象都有一个关联的 **spec.provisionerClassName** 字段，它指示 **Provisioner** 对象监视和解包该特定捆绑包类型。

配置为与普通置备程序一起使用的 Bundle 对象示例

```
apiVersion: core.rukpak.io/v1alpha1
kind: Bundle
metadata:
  name: my-bundle
spec:
  source:
    type: image
    image:
      ref: my-bundle@sha256:xyz123
  provisionerClassName: core-rukpak-io-plain
```



注意

捆绑包在创建后被视为不可变。

7.2.3.3.1. 捆绑包不可变

在 API 服务器接受 **Bundle** 对象后，捆绑包被视为 RukPak 系统的其余部分的不可变工件。这个行为强制捆绑包代表集群中要 **source** 的一些唯一静态内容。用户可以放心，特定捆绑包指向特定的清单集合，在没有创建新捆绑包的情况下无法更新。对于由嵌入式 **BundleTemplate** 对象创建的单机捆绑包和动态捆绑包，此属性为 **true**。

捆绑包的不可变性由核心 RukPak webhook 强制执行。此 webhook 监视 **Bundle** 对象事件，并对捆绑包的任何更新，检查现有捆绑包的 **spec** 字段是否与建议的捆绑包中的 **spec** 字段相同。如果它们不相等，则 Webhook 将拒绝更新。在捆绑包的生命周期中，其他 **Bundle** 对象字段（如 **metadata** 或 **status**）会更新，它只是被视为不可变的 **spec** 字段。

应用 **Bundle** 对象，然后尝试更新其 **spec** 会失败。例如，以下示例会创建一个捆绑包：

```
$ oc apply -f -<<EOF
apiVersion: core.rukpak.io/v1alpha1
kind: Bundle
metadata:
  name: combo-tag-ref
spec:
  source:
    type: git
    git:
      ref:
        tag: v0.0.2
      repository: https://github.com/operator-framework/combo
  provisionerClassName: core-rukpak-io-plain
EOF
```


输出示例

```
bundle.core.rukpak.io/combo-tag-ref created
```

然后，修补捆绑包以指向较新的标签会返回错误：

```
$ oc patch bundle combo-tag-ref --type='merge' -p '{"spec":{"source":{"git":{"ref":{"tag":"v0.0.3"}}}}}'
```

输出示例

```
Error from server (bundle.spec is immutable): admission webhook "vbundles.core.rukpak.io" denied the request: bundle.spec is immutable
```

核心 RukPak 准入 Webhook 会拒绝补丁，因为捆绑包的 spec 是不可变的。推荐的修改捆绑包内容的方法是通过创建一个新的 **Bundle** 对象而不是原位更新。

进一步的不可变注意事项

虽然 **Bundle** 对象的 **spec** 字段不可变，但 **BundleDeployment** 对象仍有可能变为较新的捆绑包内容版本，而无需更改底层 **spec** 字段。这个非预期的行为可能会在以下情况中出现：

1. 用户在 **Bundle** 对象的 **spec.source** 字段中设置镜像标签、Git 分支或 Git 标签。
2. 镜像标签移到新的摘要、用户将更改推送到 Git 分支，或者用户删除并在不同的提交上重新推送 Git 标签。
3. 用户执行一些操作，以便重新创建捆绑包解包 pod，如删除解包 pod。

如果发生这种情况，步骤 2 中的新内容会因为第 3 步而解包。捆绑包部署会检测更改，并探测到新版本的内容。

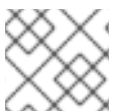
这与 pod 的行为类似，其中一个 pod 的容器镜像使用标签，标签将移到不同的摘要中，然后在将来将现有 pod 重新调度到其他节点上。此时，节点会将新镜像拉取到新摘要中，并在没有用户明确要求的情况下运行不同的镜像。

为了确定底层 **Bundle** spec 内容没有改变，请在创建捆绑包时使用基于摘要的镜像或 Git 提交引用。

7.2.3.3.2. 普通捆绑包规格

RukPak 中的普通捆绑包是给定目录中静态、任意的 Kubernetes YAML 清单的集合。

当前实施的普通捆绑包格式是 **plain+v0** 格式。捆绑包格式的名称 (**plain+v0**) 组合了捆绑包类型 (**plain**) 和当前模式版本 (**v0**)。



注意

plain+v0 捆绑包格式是 schema 版本 **v0**，这意味着它可能会在以后有所变化。

例如，下面显示了 **plain+v0** 捆绑包中的文件树。它必须具有一个 **manifests/** 目录，其中包含部署应用程序所需的 Kubernetes 资源。

plain+v0 捆绑包文件树示例

```
$ tree manifests
```

```
manifests
├── namespace.yaml
├── service_account.yaml
├── cluster_role.yaml
├── cluster_role_binding.yaml
└── deployment.yaml
```

静态清单必须位于 **manifests/** 目录中，其中至少包含一个资源，以便捆绑包是置备程序可以解包的有效 **plain+v0** 捆绑包。**manifests/** 目录还必须是扁平的；所有清单都必须位于顶层，且无子目录。



重要

不要在不是静态清单的普通捆绑包的 **manifests/** 目录中包含任何内容。否则，从该捆绑包创建内容时会出现错误。任何不能使用 **oc apply** 命令应用的文件将导致错误。多对象 YAML 或 JSON 文件也有效。

7.2.3.3.3. registry 捆绑包规格

registry 捆绑包或 **registry+v1** 捆绑包包含一组以旧 Operator Lifecycle Manager (OLM) 捆绑包格式组织的静态 Kubernetes YAML 清单。

其他资源

- [旧的 OLM 捆绑包格式](#)

7.2.3.4. BundleDeployment



警告

BundleDeployment 对象通过安装和删除对象来更改 Kubernetes 集群的状态。务必要验证并信任正在安装和限制访问权限的内容，方法是使用 RBAC 到 **BundleDeployment** API 到需要这些权限的用户。

RukPak **BundleDeployment** API 指向 **Bundle** 对象，并表明它应当处于活动状态。这包括从活跃捆绑包的旧版本获取。**BundleDeployment** 对象可能还包括所需捆绑包的嵌入式 spec。

与 pod 生成容器镜像实例一样，捆绑包部署会生成捆绑包部署的版本。捆绑包部署可被视为 pod 概念的一般化。

捆绑包部署如何根据引用的捆绑包对集群进行更改，具体由配置为监视该捆绑包部署的置备程序定义。

配置为与普通置备程序一起工作的 BundleDeployment 对象示例

```
apiVersion: core.rukpak.io/v1alpha1
kind: BundleDeployment
metadata:
  name: my-bundle-deployment
spec:
```

```

provisionerClassName: core-rukpak-io-plain
template:
  metadata:
    labels:
      app: my-bundle
  spec:
    source:
      type: image
      image:
        ref: my-bundle@sha256:xyz123
    provisionerClassName: core-rukpak-io-plain

```

7.2.4. OLM 1.0 中的依赖项解析 (技术预览)

Operator Lifecycle Manager (OLM) 1.0 使用依赖项管理器来解析 RukPak 捆绑包目录的限制。



重要

OLM 1.0 只是一个技术预览功能。技术预览功能不受红帽产品服务等级协议 (SLA) 支持，且功能可能并不完整。红帽不推荐在生产环境中使用它们。这些技术预览功能可以使用户提早试用新的功能，并有机会在开发阶段提供反馈意见。

有关红帽技术预览功能支持范围的更多信息，请参阅[技术预览功能支持范围](#)。

7.2.4.1. 概念

用户有一组预期，软件包管理器不应执行以下操作：

- 安装无法实现依赖关系或与另一个软件包依赖项冲突的软件包
- 安装其约束无法被当前可安装软件包满足的软件包
- 更新软件包会导致破坏依赖它的另一个软件包

7.2.4.1.1. 示例：成功解析

用户想要安装具有以下依赖关系的软件包 A 和 B：

软件包 A v0.1.0	软件包 B latest
↓ (依赖于)	↓ (依赖于)
软件包 C v0.1.0	软件包 D latest

此外，用户希望将 A 的版本固定到 **v0.1.0**。

传递给 OLM 1.0 的软件包和限制

软件包

- A
- B

约束 (constraint)

- **v0.1.0** 依赖于 C **v0.1.0**
- A 固定到 **v0.1.0**
- B 依赖于 D

输出

- 解决方案集 :
 - A **v0.1.0**
 - B **latest**
 - C **v0.1.0**
 - D **latest**

7.2.4.1.2. 示例：不成功的解析

用户想要安装具有以下依赖关系的软件包 A 和 B：

软件包 A v0.1.0	软件包 B latest
↓ (依赖于)	↓ (依赖于)
软件包 C v0.1.0	软件包 C v0.1.0

此外，用户希望将 A 的版本固定到 **v0.1.0**。

传递给 OLM 1.0 的软件包和限制

软件包

- A
- B

约束 (constraint)

- **v0.1.0** 依赖于 C **v0.1.0**
- A 固定到 **v0.1.0**
- **v0.1.0** 依赖于 C **v0.1.0**

输出

- 解决方案集 :
 - 无法解决，因为 A **v0.1.0** 需要 C **v0.1.0**，这与 B **latest** 需要 C **v0.2.0** 相冲突

7.2.5. Catalogd (技术预览)

Operator Lifecycle Manager (OLM) 1.0 使用 catalogd 组件及其资源来管理 Operator 和扩展目录。



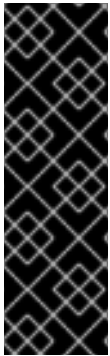
重要

OLM 1.0 只是一个技术预览功能。技术预览功能不受红帽产品服务等级协议 (SLA) 支持，且功能可能并不完整。红帽不推荐在生产环境中使用它们。这些技术预览功能可以使用户提早试用新的功能，并有机会在开发阶段提供反馈意见。

有关红帽技术预览功能支持范围的更多信息，请参阅[技术预览功能支持范围](#)。

7.2.5.1. 关于 OLM 1.0 中的目录

您可以使用 catalogd 组件查询 Kubernetes 扩展的目录，如 Operator 和控制器，从而发现可安装的内容。Catalogd 是一个 Kubernetes 扩展，为集群客户端解包目录内容，并是微服务的 Operator Lifecycle Manager (OLM) 1.0 套件的一部分。目前，catalogd 解包要打包并分发为容器镜像的目录内容。



重要

如果您尝试安装没有唯一名称的 Operator 或扩展，则安装可能会失败，或者导致无法预计的结果。这是因为以下原因：

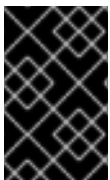
- 如果在集群中安装了 multiple 目录，OLM 1.0 不包括在安装 Operator 或扩展时指定目录的机制。
- Operator Lifecycle Manager (OLM) 1.0 中的依赖项解析需要集群中用于安装的所有 Operator 和扩展都使用其捆绑包和软件包的唯一名称。

其他资源

- [基于文件的目录](#)

7.2.5.1.1. OLM 1.0 中红帽提供的 Operator 目录

Operator Lifecycle Manager (OLM) 1.0 默认不包括红帽提供的 Operator 目录。如果要在集群中添加红帽提供的目录，请为目录创建一个自定义资源 (CR)，并将其应用到集群。以下自定义资源 (CR) 示例演示了如何为 OLM 1.0 创建目录资源。



重要

如果要使用托管在安全 registry 上的目录，如来自 [registry.redhat.io](#) 的红帽提供的 Operator 目录，则必须有一个范围到 **openshift-catalogd** 命名空间的 pull secret。如需更多信息，请参阅["为在安全 registry 上托管的目录创建 pull secret"](#)。

Red Hat Operator 目录示例

```
apiVersion: catalogd.operatorframework.io/v1alpha1
kind: Catalog
metadata:
  name: redhat-operators
spec:
  source:
    type: image
```

```

image:
  ref: registry.redhat.io/redhat/redhat-operator-index:v4.15
  pullSecret: <pull_secret_name>
  pollInterval: <poll_interval_duration> 1

```

- 1** 指定轮询远程 registry 以获取较新的镜像摘要的时间间隔。默认值为 **24h**。有效单位包括秒 (**s**)、分钟 (**m**) 和小时 (**h**)。要禁用轮询，请设置零值，如 **0s**。

认证的 Operator 目录示例

```

apiVersion: catalogd.operatorframework.io/v1alpha1
kind: Catalog
metadata:
  name: certified-operators
spec:
  source:
    type: image
    image:
      ref: registry.redhat.io/redhat/certified-operator-index:v4.15
      pullSecret: <pull_secret_name>
      pollInterval: 24h

```

Community Operators 目录示例

```

apiVersion: catalogd.operatorframework.io/v1alpha1
kind: Catalog
metadata:
  name: community-operators
spec:
  source:
    type: image
    image:
      ref: registry.redhat.io/redhat/community-operator-index:v4.15
      pullSecret: <pull_secret_name>
      pollInterval: 24h

```

以下命令在集群中添加目录：

命令语法

```
$ oc apply -f <catalog_name>.yaml 1
```

- 1** 指定目录 CR，如 **redhat-operators.yaml**。

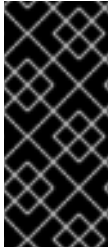
其他资源

- [在集群中添加目录](#)
- [关于红帽提供的 Operator 目录](#)

7.3. 从 OLM 1.0 中的目录安装 OPERATOR（技术预览）

集群管理员可向其集群添加 [目录](#)、或策展 Operator 和 Kubernetes 扩展集合。Operator 作者将其产品发布到这些目录中。当您向集群添加目录时，您可以访问发布到目录中的 Operator 和扩展的版本、补丁和无线更新。

在 Operator Lifecycle Manager (OLM) 1.0 的当前技术预览版本中，您可以使用自定义资源(CR)从 CLI 以声明性方式管理目录和 Operator。



重要

OLM 1.0 只是一个技术预览功能。技术预览功能不受红帽产品服务等级协议 (SLA) 支持，且功能可能并不完整。红帽不推荐在生产环境中使用它们。这些技术预览功能可以使用户提早试用新的功能，并有机会在开发阶段提供反馈意见。

有关红帽技术预览功能支持范围的更多信息，请参阅[技术预览功能支持范围](#)。

7.3.1. 先决条件

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



注意

对于 OpenShift Container Platform 4.15，适用于 OLM 1.0 的流程都是基于 CLI 的。另外，管理员也可以使用普通方法（如 [Import YAML](#) 和 [Search](#) 页面）在 web 控制台中创建和查看相关对象。但是，现有的 **OperatorHub** 和 **Installed Operators** 页面还不会显示 OLM 1.0 组件。

- 在集群中启用 **TechPreviewNoUpgrade** 功能集



警告

启用 **TechPreviewNoUpgrade** 功能集将无法撤消，并防止次版本更新。不建议在生产环境集群中使用这些功能集。

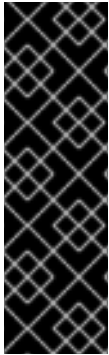
- 在您的工作站上安装的 OpenShift CLI (**oc**)

其他资源

- [使用功能门启用功能](#)

7.3.2. 关于 OLM 1.0 中的目录

您可以使用 `catalogd` 组件查询 Kubernetes 扩展的目录，如 Operator 和控制器，从而发现可安装的内容。`Catalogd` 是一个 Kubernetes 扩展，为集群客户端解包目录内容，并是微服务的 Operator Lifecycle Manager (OLM) 1.0 套件的一部分。目前，`catalogd` 解包要打包并分发为容器镜像的目录内容。



重要

如果您尝试安装没有唯一名称的 Operator 或扩展，则安装可能会失败，或者导致无法预计的结果。这是因为以下原因：

- 如果在集群中安装了 multiple 目录，OLM 1.0 不包括在安装 Operator 或扩展时指定目录的机制。
- Operator Lifecycle Manager (OLM) 1.0 中的依赖项解析需要集群中用于安装的所有 Operator 和扩展都使用其捆绑包和软件包的唯一名称。

其他资源

- [基于文件的目录](#)

7.3.3. OLM 1.0 中红帽提供的 Operator 目录

Operator Lifecycle Manager (OLM) 1.0 默认不包括红帽提供的 Operator 目录。如果要在集群中添加红帽提供的目录，请为目录创建一个自定义资源 (CR)，并将其应用到集群。以下自定义资源 (CR) 示例演示了如何为 OLM 1.0 创建目录资源。



重要

如果要使用托管在安全 registry 上的目录，如来自 registry.redhat.io 的红帽提供的 Operator 目录，则必须有一个范围到 `openshift-catalogd` 命名空间的 pull secret。如需更多信息，请参阅“为在安全 registry 上托管的目录创建 pull secret”。

Red Hat Operator 目录示例

```

apiVersion: catalogd.operatorframework.io/v1alpha1
kind: Catalog
metadata:
  name: redhat-operators
spec:
  source:
    type: image
    image:
      ref: registry.redhat.io/redhat/redhat-operator-index:v4.15
      pullSecret: <pull_secret_name>
      pollInterval: <poll_interval_duration> 1

```

- 1** 指定轮询远程 registry 以获取较新的镜像摘要的时间间隔。默认值为 **24h**。有效单位包括秒 (**s**)、分钟 (**m**) 和小时 (**h**)。要禁用轮询，请设置零值，如 **0s**。

认证的 Operator 目录示例

```

apiVersion: catalogd.operatorframework.io/v1alpha1
kind: Catalog
metadata:
  name: certified-operators
spec:
  source:
    type: image

```



```

image:
  ref: registry.redhat.io/redhat/certified-operator-index:v4.15
  pullSecret: <pull_secret_name>
  pollInterval: 24h

```

Community Operators 目录示例

```

apiVersion: catalogd.operatorframework.io/v1alpha1
kind: Catalog
metadata:
  name: community-operators
spec:
  source:
    type: image
    image:
      ref: registry.redhat.io/redhat/community-operator-index:v4.15
      pullSecret: <pull_secret_name>
      pollInterval: 24h

```

以下命令在集群中添加目录：

命令语法

```
$ oc apply -f <catalog_name>.yaml 1
```

1 指定目录 CR，如 **redhat-operators.yaml**。

其他资源

- [为托管在安全 registry 上的目录创建 pull secret](#)
- [在集群中添加目录](#)
- [关于红帽提供的 Operator 目录](#)

7.3.4. 为托管在安全 registry 上的目录创建 pull secret

如果要使用托管在安全 registry 上的目录，如来自 **registry.redhat.io** 的红帽提供的 Operator 目录，则必须有一个范围到 **openshift-catalogd** 命名空间的 pull secret。



注意

目前，catalogd 无法从 OpenShift Container Platform 集群中读取全局 pull secret。Catalogd 只能读取对部署它的命名空间中的 secret 的引用。

先决条件

- 安全 registry 的登录凭证
- 工作站上安装的 Docker 或 Podman

流程

- 如果您已经有一个带有安全 registry 的登录凭证的 **.dockercfg** 文件，请运行以下命令创建一个 pull secret：

```
$ oc create secret generic <pull_secret_name> \  
  --from-file=.dockercfg=<file_path>/.dockercfg \  
  --type=kubernetes.io/dockercfg \  
  --namespace=openshift-catalogd
```

例 7.1. 示例命令

```
$ oc create secret generic redhat-cred \  
  --from-file=.dockercfg=/home/<username>/.dockercfg \  
  --type=kubernetes.io/dockercfg \  
  --namespace=openshift-catalogd
```

- 如果您已经有一个带有安全 registry 的登录凭证的 **\$HOME/.docker/config.json** 文件，请运行以下命令创建一个 pull secret：

```
$ oc create secret generic <pull_secret_name> \  
  --from-file=.dockerconfigjson=<file_path>/.docker/config.json \  
  --type=kubernetes.io/dockerconfigjson \  
  --namespace=openshift-catalogd
```

例 7.2. 示例命令

```
$ oc create secret generic redhat-cred \  
  --from-file=.dockerconfigjson=/home/<username>/.docker/config.json \  
  --type=kubernetes.io/dockerconfigjson \  
  --namespace=openshift-catalogd
```

- 如果您没有安全 registry 的登录凭证的 Docker 配置文件，请运行以下命令来创建 pull secret：

```
$ oc create secret docker-registry <pull_secret_name> \  
  --docker-server=<registry_server> \  
  --docker-username=<username> \  
  --docker-password=<password> \  
  --docker-email=<email> \  
  --namespace=openshift-catalogd
```

例 7.3. 示例命令

```
$ oc create secret docker-registry redhat-cred \  
  --docker-server=registry.redhat.io \  
  --docker-username=username \  
  --docker-password=password \  
  --docker-email=user@example.com \  
  --namespace=openshift-catalogd
```

7.3.5. 在集群中添加目录

要在集群中添加目录，请创建一个目录自定义资源(CR)并将其应用到集群。

先决条件

- 如果要使用托管在安全 registry 上的目录，如来自 **registry.redhat.io** 的红帽提供的 Operator 目录，则必须有一个范围到 **openshift-catalogd** 命名空间的 pull secret。如需更多信息，请参阅“为在安全 registry 上托管的目录创建 pull secret”。

流程

1. 创建目录自定义资源(CR)，如下例所示：

redhat-operators.yaml 示例

```
apiVersion: catalogd.operatorframework.io/v1alpha1
kind: Catalog
metadata:
  name: redhat-operators
spec:
  source:
    type: image
    image:
      ref: registry.redhat.io/redhat/redhat-operator-index:v4.15 1
      pullSecret: <pull_secret_name> 2
      pollInterval: <poll_interval_duration> 3
```

- 1 在 **spec.source.image** 字段中指定目录的镜像。
- 2 如果您的目录托管在安全 registry 上，如 **registry.redhat.io**，您必须创建一个范围到 **openshift-catalogd** 命名空间的 pull secret。
- 3 指定轮询远程 registry 以获取较新的镜像摘要的时间间隔。默认值为 **24h**。有效单位包括秒 (**s**)、分钟 (**m**) 和小时 (**h**)。要禁用轮询，请设置零值，如 **0s**。

2. 运行以下命令在集群中添加目录：

```
$ oc apply -f redhat-operators.yaml
```

输出示例

```
catalog.catalogd.operatorframework.io/redhat-operators created
```

验证

- 运行以下命令以验证目录的状态：
 - a. 运行以下命令检查目录是否可用：

```
$ oc get catalog
```

输出示例

```
NAME          AGE
redhat-operators  20s
```

- b. 运行以下命令，检查目录的状态：

```
$ oc describe catalog
```

输出示例

```
Name:      redhat-operators
Namespace:
Labels:    <none>
Annotations: <none>
API Version: catalog.operatorframework.io/v1alpha1
Kind:      Catalog
Metadata:
  Creation Timestamp: 2024-01-10T16:18:38Z
  Finalizers:
    catalogd.operatorframework.io/delete-server-cache
  Generation:      1
  Resource Version: 57057
  UID:             128db204-49b3-45ee-bfea-a2e6fc8e34ea
Spec:
  Source:
    Image:
      Pull Secret: redhat-cred
      Ref:         registry.redhat.io/redhat/redhat-operator-index:v4.15
      Type:        image
  Status: 1
  Conditions:
    Last Transition Time: 2024-01-10T16:18:55Z
    Message:
      Reason:          UnpackSuccessful 2
      Status:          True
      Type:            Unpacked
    Content URL:      http://catalogd-catalogserver.openshift-
catalogd.svc/catalogs/redhat-operators/all.json
    Observed Generation: 1
    Phase:            Unpacked 3
  Resolved Source:
    Image:
      Last Poll Attempt: 2024-01-10T16:18:51Z
      Ref:               registry.redhat.io/redhat/redhat-operator-index:v4.15
      Resolved Ref:     registry.redhat.io/redhat/redhat-operator-
index@sha256:7b536ae19b8e9f74bb521c4a61e5818e036ac1865a932f2157c6c9a766b2e
ea5 4
      Type:            image
  Events:    <none>
```

1 描述目录的状态。

2 显示目录处于当前状态的原因。

- 3 显示安装过程的阶段。
- 4 显示目录的镜像引用。

其他资源

- [为托管在安全 registry 上的目录创建 pull secret](#)

7.3.6. 从目录查找 Operator

在集群中添加目录后，您可以查询目录以查找要安装的 Operator 和扩展。在查询目录前，您必须端口转发目录服务器服务。

前提条件

- 您已在集群中添加目录。
- 已安装 **jq** CLI 工具。

流程

1. 运行以下命令，在 **openshift-catalogd** 命名空间中转发目录服务器服务：

```
$ oc -n openshift-catalogd port-forward svc/catalogd-catalogserver 8080:80
```

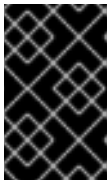
2. 运行以下命令，在本地下载目录的 JSON 文件：

```
$ curl -L http://localhost:8080/catalogs/<catalog_name>/all.json \
-C - -o /<path>/<catalog_name>.json
```

例 7.4. 示例命令

```
$ curl -L http://localhost:8080/catalogs/redhat-operators/all.json \
-C - -o /home/username/catalogs/rhoc.json
```

3. 运行以下命令之一返回目录中的 Operator 和扩展列表。



重要

目前，Operator Lifecycle Manager (OLM) 1.0 支持不使用 Webhook 的扩展，并配置为使用 **AllNamespaces** 安装模式。使用 webhook 或针对单个或指定命名空间集的扩展无法安装。

- 运行以下命令，从本地目录文件获取 Operator 和扩展列表：

```
$ jq -s '[] | select(.schema == "olm.package") | .name' \
/<path>/<filename>.json
```

例 7.5. 示例命令

```
$ jq -s '[] | select(.schema == "olm.package") | .name' \
/home/username/catalogs/rhoc.json
```

例 7.6. 输出示例

```
NAME                                     AGE
"3scale-operator"
"advanced-cluster-management"
"amq-broker-rhel8"
"amq-online"
"amq-streams"
"amq7-interconnect-operator"
"ansible-automation-platform-operator"
"ansible-cloud-addons-operator"
"apicast-operator"
"aws-efs-csi-driver-operator"
"aws-load-balancer-operator"
"bamoe-businessautomation-operator"
"bamoe-kogito-operator"
"bare-metal-event-relay"
"businessautomation-operator"
...
```

- 运行以下命令，获取支持 **AllNamespaces** 安装模式的软件包列表，且不从本地目录文件中
使用 webhook：

```
$ jq -c 'select(.schema == "olm.bundle") | \
{"package":.package, "version":.properties[] | \
select(.type == "olm.bundle.object").value.data | @base64d | fromjson | \
select(.kind == "ClusterServiceVersion" and (.spec.installModes[] | \
select(.type == "AllNamespaces" and .supported == true) != null) \
and .spec.webhookdefinitions == null).spec.version}' \
/<path>/<catalog_name>.json
```

例 7.7. 输出示例

```
{"package":"3scale-operator","version":"0.10.0-mas"}
{"package":"3scale-operator","version":"0.10.5"}
{"package":"3scale-operator","version":"0.11.0-mas"}
{"package":"3scale-operator","version":"0.11.1-mas"}
{"package":"3scale-operator","version":"0.11.2-mas"}
{"package":"3scale-operator","version":"0.11.3-mas"}
{"package":"3scale-operator","version":"0.11.5-mas"}
{"package":"3scale-operator","version":"0.11.6-mas"}
{"package":"3scale-operator","version":"0.11.7-mas"}
{"package":"3scale-operator","version":"0.11.8-mas"}
{"package":"amq-broker-rhel8","version":"7.10.0-opr-1"}
{"package":"amq-broker-rhel8","version":"7.10.0-opr-2"}
{"package":"amq-broker-rhel8","version":"7.10.0-opr-3"}
{"package":"amq-broker-rhel8","version":"7.10.0-opr-4"}
{"package":"amq-broker-rhel8","version":"7.10.1-opr-1"}
```

```

{"package":"amq-broker-rhel8","version":"7.10.1-opr-2"}
{"package":"amq-broker-rhel8","version":"7.10.2-opr-1"}
{"package":"amq-broker-rhel8","version":"7.10.2-opr-2"}
...

```

4. 运行以下命令，检查 Operator 或扩展的元数据的内容：

```

$ jq -s '[] | select( .schema == "olm.package") | \
  select( .name == "<package_name>")' /<path>/<catalog_name>.json

```

例 7.8. 示例命令

```

$ jq -s '[] | select( .schema == "olm.package") | \
  select( .name == "openshift-pipelines-operator-rh")' \
  /home/username/rhoc.json

```

例 7.9. 输出示例

```

{
  "defaultChannel": "stable",
  "icon": {
    "base64data": "PHN2ZyB4bWxu..."
    "mediatype": "image/png"
  },
  "name": "openshift-pipelines-operator-rh",
  "schema": "olm.package"
}

```

7.3.6.1. 常见目录查询

您可以使用 **jq** CLI 工具查询目录。

表 7.1. 常见软件包查询

查询	Request (请求)
目录中的可用软件包	<pre> \$ jq -s '[] select(.schema == "olm.package") \ .name' <catalog_name>.json </pre>

查询	Request (请求)
支持 AllNamespaces 安装模式且不使用 Webhook 的软件包	<pre>\$ jq -c 'select(.schema == "olm.bundle") \ {"package":.package, "version":.properties[] \ select(.type == "olm.bundle.object").value.data \ @base64d fromjson \ select(.kind == "ClusterServiceVersion" and (.spec.installModes[] \ select(.type == "AllNamespaces" and .supported == true) != null) \ and .spec.webhookdefinitions == null).spec.version}' \ <catalog_name>.json</pre>
软件包元数据	<pre>\$ jq -s '.[[] select(.schema == "olm.package") \ select(.name == "<package_name>")' <catalog_name>.json</pre>
软件包中的目录 Blob	<pre>\$ jq -s '.[[] select(.package == "<package_name>")' \ <catalog_name>.json</pre>

表 7.2. 常见频道查询

查询	Request (请求)
软件包中的频道	<pre>\$ jq -s '.[[] select(.schema == "olm.channel") \ select(.package == "<package_name>") .name' \ <catalog_name>.json</pre>
频道中的版本	<pre>\$ jq -s '.[[] select(.package == "<package_name>") \ select(.schema == "olm.channel") \ select(.name == "<channel_name>") \ .entries .[] .name' <catalog_name>.json</pre>
<ul style="list-style-type: none"> ● 频道中的最新版本 ● 升级路径 	<pre>\$ jq -s '.[[] select(.schema == "olm.channel") \ select (.name == "<channel_name>") \ select(.package == "<package_name>")' \ <catalog_name>.json</pre>

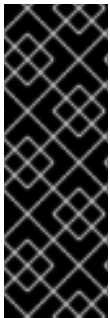
表 7.3. 常见捆绑包查询

查询	Request (请求)
----	--------------

查询	Request (请求)
软件包中的捆绑包	<pre>\$ jq -s '[] select(.schema == "olm.bundle") \ select(.package == "<package_name>") .name' \ <catalog_name>.json</pre>
<ul style="list-style-type: none"> 捆绑包依赖项 可用的 API 	<pre>\$ jq -s '[] select(.schema == "olm.bundle") \ select (.name == "<bundle_name>") \ select(.package == "<package_name>") \ <catalog_name>.json</pre>

7.3.7. 从目录安装 Operator

Operator Lifecycle Manager (OLM) 1.0 支持将 Operator 和扩展范围安装到集群中。您可以通过创建自定义资源(CR)并将其应用到集群来从目录中安装 Operator 或扩展。



重要

目前，OLM 1.0 支持符合以下条件的安装 Operator 和扩展：

- Operator 或扩展必须使用 **AllNamespaces** 安装模式。
- Operator 或扩展不能使用 Webhook。

使用 Webhook 或指定命名空间集的 Operator 和扩展无法安装。

前提条件

- 您已在集群中添加目录。
- 您已下载了目录文件的本地副本。
- 已安装 **jq** CLI 工具。

流程

- 通过完成以下步骤，检查目录文件本地副本中的频道和版本信息：
 - 运行以下命令，从所选软件包中获取频道列表：

```
$ jq -s '[] | select( .schema == "olm.channel" ) | \
select( .package == "<package_name>" ) | \
.name' /<path>/<catalog_name>.json
```

例 7.10. 示例命令

```
$ jq -s '[] | select( .schema == "olm.channel" ) | \
  select( .package == "openshift-pipelines-operator-rh" ) | \
  .name' /home/username/rhoc.json
```

例 7.11. 输出示例

```
"latest"
"pipelines-1.11"
"pipelines-1.12"
"pipelines-1.13"
```

- b. 运行以下命令，获取频道中发布的版本列表：

```
$ jq -s '[] | select( .package == "<package_name>" ) | \
  select( .schema == "olm.channel" ) | \
  select( .name == "<channel_name>" ) | .entries | \
  .[] | .name' /<path>/<catalog_name>.json
```

例 7.12. 示例命令

```
$ jq -s '[] | select( .package == "openshift-pipelines-operator-rh" ) | \
  select( .schema == "olm.channel" ) | select( .name == "latest" ) | \
  .entries | .[] | .name' /home/username/rhoc.json
```

例 7.13. 输出示例

```
"openshift-pipelines-operator-rh.v1.11.1"
"openshift-pipelines-operator-rh.v1.12.0"
"openshift-pipelines-operator-rh.v1.12.1"
"openshift-pipelines-operator-rh.v1.12.2"
"openshift-pipelines-operator-rh.v1.13.0"
"openshift-pipelines-operator-rh.v1.13.1"
```

2. 创建一个类似以下示例的 CR：

pipelines-operator.yaml CR 示例

```
apiVersion: operators.operatorframework.io/v1alpha1
kind: Operator
metadata:
  name: pipelines-operator
spec:
  packageName: openshift-pipelines-operator-rh
  channel: <channel>
  version: "<version>"
```

其中：

<channel>

可选：为您要安装或更新的软件包指定频道，如 **pipelines-1.11** 或 **latest**。

<version>

可选：指定您要安装或更新的软件包的版本或版本范围，如 **1.11.1**、**1.12.x** 或 **>=1.12.1**。如需更多信息，请参阅"示例自定义资源(CR)指定目标版本"和"支持版本范围"。

**重要**

如果您尝试安装没有唯一名称的 Operator 或扩展，则安装可能会失败，或者导致无法预计的结果。这是因为以下原因：

- 如果在集群中安装了 multiple 目录，OLM 1.0 不包括在安装 Operator 或扩展时指定目录的机制。
- Operator Lifecycle Manager (OLM) 1.0 中的依赖项解析需要集群中用于安装的所有 Operator 和扩展都使用其捆绑包和软件包的唯一名称。

3. 运行以下命令，将 CR 应用到集群：

```
$ oc apply -f pipeline-operator.yaml
```

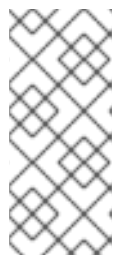
输出示例

```
operator.operators.operatorframework.io/pipelines-operator created
```

验证

1. 运行以下命令，以 YAML 格式查看 Operator 或扩展 CR：

```
$ oc get operator.operators.operatorframework.io pipelines-operator -o yaml
```

**注意**

如果您在 Operator 或扩展的 CR 中指定频道或定义版本范围，OLM 1.0 不会显示集群中安装的解析版本。仅会显示 CR 中指定的版本和频道信息。

如果要查找安装的特定版本，您必须将 **spec.source.image.ref** 字段镜像的 SHA 与目录中的镜像引用进行比较。

例 7.14. 输出示例

```
apiVersion: operators.operatorframework.io/v1alpha1
kind: Operator
metadata:
  annotations:
    kubectrl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"operators.operatorframework.io/v1alpha1","kind":"Operator","metadata":
      {"annotations":{},"name":"pipelines-operator"},"spec":
      {"channel":"latest","packageName":"openshift-pipelines-operator-rh","version":"1.11.x"}}

```

```

creationTimestamp: "2024-01-30T20:06:09Z"
generation: 1
name: pipelines-operator
resourceVersion: "44362"
uid: 4272d228-22e1-419e-b9a7-986f982ee588
spec:
  channel: latest
  packageName: openshift-pipelines-operator-rh
  upgradeConstraintPolicy: Enforce
  version: 1.11.x
status:
  conditions:
  - lastTransitionTime: "2024-01-30T20:06:15Z"
    message: resolved to "registry.redhat.io/openshift-pipelines/pipelines-operator-
bundle@sha256:e09d37bb1e754db42324fd18c1cb3e7ce77e7b7fcbf4932d0535391579938
280"
    observedGeneration: 1
    reason: Success
    status: "True"
    type: Resolved
  - lastTransitionTime: "2024-01-30T20:06:31Z"
    message: installed from "registry.redhat.io/openshift-pipelines/pipelines-operator-
bundle@sha256:e09d37bb1e754db42324fd18c1cb3e7ce77e7b7fcbf4932d0535391579938
280"
    observedGeneration: 1
    reason: Success
    status: "True"
    type: Installed
  installedBundleResource: registry.redhat.io/openshift-pipelines/pipelines-operator-
bundle@sha256:e09d37bb1e754db42324fd18c1cb3e7ce77e7b7fcbf4932d0535391579938
280
  resolvedBundleResource: registry.redhat.io/openshift-pipelines/pipelines-operator-
bundle@sha256:e09d37bb1e754db42324fd18c1cb3e7ce77e7b7fcbf4932d0535391579938
280

```

2. 运行以下命令，获取有关捆绑包部署的信息：

```
$ oc get bundleDeployment pipelines-operator -o yaml
```

例 7.15. 输出示例

```

apiVersion: core.rukpak.io/v1alpha1
kind: BundleDeployment
metadata:
  creationTimestamp: "2024-01-30T20:06:15Z"
  generation: 2
  name: pipelines-operator
  ownerReferences:
  - apiVersion: operators.operatorframework.io/v1alpha1
    blockOwnerDeletion: true
    controller: true
    kind: Operator
    name: pipelines-operator
    uid: 4272d228-22e1-419e-b9a7-986f982ee588

```

```

resourceVersion: "44464"
uid: 0a0c3525-27e2-4c93-bf57-55920a7707c0
spec:
  provisionerClassName: core-rukpak-io-plain
  template:
    metadata: {}
    spec:
      provisionerClassName: core-rukpak-io-registry
      source:
        image:
          ref: registry.redhat.io/openshift-pipelines/pipelines-operator-
bundle@sha256:e09d37bb1e754db42324fd18c1cb3e7ce77e7b7fcbf4932d0535391579938
280
          type: image
status:
  activeBundle: pipelines-operator-29x720cjzx8yiowf13a3j75fil2zs3mfw
  conditions:
  - lastTransitionTime: "2024-01-30T20:06:15Z"
    message: Successfully unpacked the pipelines-operator-
29x720cjzx8yiowf13a3j75fil2zs3mfw
    Bundle
    reason: UnpackSuccessful
    status: "True"
    type: HasValidBundle
  - lastTransitionTime: "2024-01-30T20:06:28Z"
    message: Instantiated bundle pipelines-operator-29x720cjzx8yiowf13a3j75fil2zs3mfw
    successfully
    reason: InstallationSucceeded
    status: "True"
    type: Installed
  - lastTransitionTime: "2024-01-30T20:06:40Z"
    message: BundleDeployment is healthy
    reason: Healthy
    status: "True"
    type: Healthy
  observedGeneration: 2

```

其他资源

- [指定目标版本的自定义资源\(CR\)示例](#)
- [支持版本范围](#)

7.3.8. 更新 Operator

您可以通过手动编辑自定义资源(CR)并应用更改来更新 Operator 或扩展。

先决条件

- 已安装目录。
- 您已下载了目录文件的本地副本。
- 已安装 Operator 或扩展。

- 已安装 **jq** CLI 工具。

流程

1. 通过完成以下步骤，检查目录文件本地副本中的频道和版本信息：

- a. 运行以下命令，从所选软件包中获取频道列表：

```
$ jq -s '[] | select( .schema == "olm.channel" ) | \
  select( .package == "<package_name>" ) | \
  .name' /<path>/<catalog_name>.json
```

例 7.16. 示例命令

```
$ jq -s '[] | select( .schema == "olm.channel" ) | \
  select( .package == "openshift-pipelines-operator-rh" ) | \
  .name' /home/username/rhoc.json
```

例 7.17. 输出示例

```
"latest"
"pipelines-1.11"
"pipelines-1.12"
"pipelines-1.13"
```

- b. 运行以下命令，获取频道中发布的版本列表：

```
$ jq -s '[] | select( .package == "<package_name>" ) | \
  select( .schema == "olm.channel" ) | \
  select( .name == "<channel_name>" ) | .entries | \
  .[] | .name' /<path>/<catalog_name>.json
```

例 7.18. 示例命令

```
$ jq -s '[] | select( .package == "openshift-pipelines-operator-rh" ) | \
  select( .schema == "olm.channel" ) | select( .name == "latest" ) | \
  .entries | .[] | .name' /home/username/rhoc.json
```

例 7.19. 输出示例

```
"openshift-pipelines-operator-rh.v1.11.1"
"openshift-pipelines-operator-rh.v1.12.0"
"openshift-pipelines-operator-rh.v1.12.1"
"openshift-pipelines-operator-rh.v1.12.2"
"openshift-pipelines-operator-rh.v1.13.0"
"openshift-pipelines-operator-rh.v1.13.1"
```

2. 运行以下命令，查找在 Operator 或扩展 CR 中指定哪个版本或频道：

```
$ oc get operator.operators.operatorframework.io <operator_name> -o yaml
```

示例命令

```
$ oc get operator.operators.operatorframework.io pipelines-operator -o yaml
```

例 7.20. 输出示例

```
apiVersion: operators.operatorframework.io/v1alpha1
kind: Operator
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
{"apiVersion":"operators.operatorframework.io/v1alpha1","kind":"Operator","metadata":
{"annotations":{},"name":"pipelines-operator"},"spec":
{"channel":"latest","packageName":"openshift-pipelines-operator-rh","version":"1.11.1"}}
  creationTimestamp: "2024-02-06T17:47:15Z"
  generation: 2
  name: pipelines-operator
  resourceVersion: "84528"
  uid: dffe2c89-b9c4-427e-b694-ada0b37fc0a9
spec:
  channel: latest 1
  packageName: openshift-pipelines-operator-rh
  upgradeConstraintPolicy: Enforce
  version: 1.11.1 2
status:
  conditions:
  - lastTransitionTime: "2024-02-06T17:47:21Z"
    message: bundledeployment status is unknown
    observedGeneration: 2
    reason: InstallationStatusUnknown
    status: Unknown
    type: Installed
  - lastTransitionTime: "2024-02-06T17:50:58Z"
    message: resolved to "registry.redhat.io/openshift-pipelines/pipelines-operator-
bundle@sha256:e09d37bb1e754db42324fd18c1cb3e7ce77e7b7fcbf4932d0535391579938
280"
    observedGeneration: 2
    reason: Success
    status: "True"
    type: Resolved
  resolvedBundleResource: registry.redhat.io/openshift-pipelines/pipelines-operator-
bundle@sha256:e09d37bb1e754db42324fd18c1cb3e7ce77e7b7fcbf4932d0535391579938
280
```

- 1** 指定 Operator 或扩展的频道。
- 2** 指定 Operator 或扩展名的版本或版本范围。



注意

如果您在 Operator 或扩展的 CR 中指定频道或定义版本范围，OLM 1.0 不会显示集群中安装的解析版本。仅会显示 CR 中指定的版本和频道信息。

如果要查找安装的特定版本，您必须将 `spec.source.image.ref` 字段镜像的 SHA 与目录中的镜像引用进行比较。

3. 使用以下方法之一编辑 CR：

- 如果要将在 Operator 或扩展固定到特定版本，如 **1.12.1**，请编辑类似以下示例的 CR：

pipelines-operator.yaml CR 示例

```
apiVersion: operators.operatorframework.io/v1alpha1
kind: Operator
metadata:
  name: pipelines-operator
spec:
  packageName: openshift-pipelines-operator-rh
  version: 1.12.1 1
```

- 1** 将版本从 **1.11.1** 更新至 **1.12.1**

- 如果要定义可接受的更新版本范围，请编辑类似以下示例的 CR：

指定了版本范围的 CR 示例

```
apiVersion: operators.operatorframework.io/v1alpha1
kind: Operator
metadata:
  name: pipelines-operator
spec:
  packageName: openshift-pipelines-operator-rh
  version: ">1.11.1, <1.13" 1
```

- 1** 指定所需的版本范围大于 **1.11.1** 版本，且小于 **1.13**。如需更多信息，请参阅“支持版本范围”和“Version 比较字符串”。

- 如果要更新到可以从频道解析的最新版本，请编辑类似以下示例的 CR：

带有指定频道的 CR 示例

```
apiVersion: operators.operatorframework.io/v1alpha1
kind: Operator
metadata:
  name: pipelines-operator
spec:
  packageName: openshift-pipelines-operator-rh
  channel: pipelines-1.13 1
```

- 1** 安装可从指定频道解析的最新版本。对频道的更新会自动安装。

- 如果要指定频道和版本范围，请编辑类似以下示例的 CR：

带有指定频道和版本范围的 CR 示例

```
apiVersion: operators.operatorframework.io/v1alpha1
kind: Operator
metadata:
  name: pipelines-operator
spec:
  packageName: openshift-pipelines-operator-rh
  channel: latest
  version: "<1.13"
```

如需更多信息，请参阅“指定目标版本的示例自定义资源(CR)”。

4. 运行以下命令，将更新应用到集群：

```
$ oc apply -f pipelines-operator.yaml
```

输出示例

```
operator.operators.operatorframework.io/pipelines-operator configured
```

提示

您可以运行以下命令来通过 CLI 对 CR 进行补丁并应用更改：

```
$ oc patch operator.operators.operatorframework.io/pipelines-operator -p \
 '{"spec":{"version":"1.12.1"}}' \
 --type=merge
```

输出示例

```
operator.operators.operatorframework.io/pipelines-operator patched
```

验证

- 运行以下命令验证频道和版本更新是否已应用：

```
$ oc get operator.operators.operatorframework.io pipelines-operator -o yaml
```

例 7.21. 输出示例

```
apiVersion: operators.operatorframework.io/v1alpha1
kind: Operator
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |

{"apiVersion":"operators.operatorframework.io/v1alpha1","kind":"Operator","metadata":
{"annotations":{},"name":"pipelines-operator"},"spec":
{"channel":"latest","packageName":"openshift-pipelines-operator-rh","version":"1.12.1"}}
```

```

creationTimestamp: "2024-02-06T19:16:12Z"
generation: 4
name: pipelines-operator
resourceVersion: "58122"
uid: 886bbf73-604f-4484-9f87-af6ce0f86914
spec:
  channel: latest
  packageName: openshift-pipelines-operator-rh
  upgradeConstraintPolicy: Enforce
  version: 1.12.1 ❶
status:
  conditions:
  - lastTransitionTime: "2024-02-06T19:30:57Z"
    message: installed from "registry.redhat.io/openshift-pipelines/pipelines-operator-
bundle@sha256:2f1b8ef0fd741d1d686489475423dabc07c55633a4dfebc45e1d533183179f
6a"
    observedGeneration: 3
    reason: Success
    status: "True"
    type: Installed
  - lastTransitionTime: "2024-02-06T19:30:57Z"
    message: resolved to "registry.redhat.io/openshift-pipelines/pipelines-operator-
bundle@sha256:2f1b8ef0fd741d1d686489475423dabc07c55633a4dfebc45e1d533183179f
6a"
    observedGeneration: 3
    reason: Success
    status: "True"
    type: Resolved
  installedBundleResource: registry.redhat.io/openshift-pipelines/pipelines-operator-
bundle@sha256:2f1b8ef0fd741d1d686489475423dabc07c55633a4dfebc45e1d533183179f
6a
  resolvedBundleResource: registry.redhat.io/openshift-pipelines/pipelines-operator-
bundle@sha256:2f1b8ef0fd741d1d686489475423dabc07c55633a4dfebc45e1d533183179f
6a

```

❶ 验证版本是否已更新至 **1.12.1**。

故障排除

- 如果指定了不存在的目标版本或频道，您可以运行以下命令来检查 Operator 或扩展的状态：

```
$ oc get operator.operators.operatorframework.io <operator_name> -o yaml
```

例 7.22. 输出示例

```

oc get operator.operators.operatorframework.io pipelines-operator -o yaml
apiVersion: operators.operatorframework.io/v1alpha1
kind: Operator
metadata:
  annotations:
    kubectrl.kubernetes.io/last-applied-configuration: |
{"apiVersion":"operators.operatorframework.io/v1alpha1","kind":"Operator","metadata":

```

```

{"annotations": {}, "name": "pipelines-operator", "spec":
{"channel": "latest", "packageName": "openshift-pipelines-operator-rh", "version": "2.0.0"}}
creationTimestamp: "2024-02-06T17:47:15Z"
generation: 1
name: pipelines-operator
resourceVersion: "82667"
uid: dffe2c89-b9c4-427e-b694-ada0b37fc0a9
spec:
channel: latest
packageName: openshift-pipelines-operator-rh
upgradeConstraintPolicy: Enforce
version: 2.0.0
status:
conditions:
- lastTransitionTime: "2024-02-06T17:47:21Z"
message: installation has not been attempted due to failure to gather data for
resolution
observedGeneration: 1
reason: InstallationStatusUnknown
status: Unknown
type: Installed
- lastTransitionTime: "2024-02-06T17:47:21Z"
message: no package "openshift-pipelines-operator-rh" matching version "2.0.0"
found in channel "latest"
observedGeneration: 1
reason: ResolutionFailed
status: "False"
type: Resolved

```

其他资源

- [指定目标版本的自定义资源\(CR\)示例](#)
- [版本比较字符串](#)

7.3.8.1. 支持语义版本

在 OLM 1.0 中默认启用 [语义版本\(semver\)](#)。Operator 扩展作者可以使用 semver 标准来定义兼容的更新。

Operator Lifecycle Manager (OLM) 1.0 可以使用 Operator 或扩展版本号来确定更新是否可以成功解决。

集群管理员可以定义一系列可接受的版本来安装和自动管理更新。对于遵循 semver 标准的 Operator 和扩展，您可以使用比较字符串来定义指定所需的版本范围。



注意

OLM 1.0 不支持对下一个主要版本的自动更新。如果要执行主版本更新，则必须手动验证并应用更新。如需更多信息，请参阅“[设置更新或回滚](#)”。

7.3.8.1.1. 主发行版本零发行版本

semver 标准指定为初始开发保留主要版本为零版本 (**O.y.z**)。在初始开发阶段，API 不稳定，并可能会在任何发布的版本中引入破坏更改。因此，主发行版本为零版本会应用一组特殊的更新条件。

更新主版本零发行版本的条件

- 当主版本和次版本都为零时，您无法应用自动更新，如 **0.0.***。例如，不允许版本范围为 **>=0.0.1 <0.1.0** 的自动更新。
- 您不能在一个主版本的零版本中应用从一个次版本到另外一个次版本的自动更新。例如，OLM 1.0 不会自动应用从 **0.1.0** 到 **0.2.0** 的更新。
- 您可以从补丁版本应用自动更新，如 **>=0.1.0 <0.2.0** 或 **>=0.2.0 <0.3.0**。

当 OLM 1.0 阻止自动更新时，您必须通过编辑 Operator 或扩展的自定义资源(CR)来手动验证并强制更新。

其他资源

- [强制更新或回滚](#)

7.3.8.2. 支持版本范围

在 Operator Lifecycle Manager (OLM) 1.0 中，您可以使用 Operator 或扩展的自定义资源(CR)中的比较字符串来指定版本范围。如果您在 CR 中指定版本范围，OLM 1.0 会安装或升级到可以在版本范围内解析的 Operator 的最新版本。

解析的版本 workflow

- 解析的版本是 Operator 的最新版本，满足 Operator 和环境的依赖项和限制。
- 如果成功解决，则会自动安装指定范围内的 Operator 更新。
- 当更新在指定的范围之外，或者无法成功解决，则不会安装更新。

有关 OLM 1.0 中依赖项和约束解析的更多信息，请参阅"OLM 1.0 中的依赖解析"。

其他资源

- [OLM 1.0 中的依赖项解析](#)

7.3.8.3. 版本比较字符串

您可以通过在 Operator 或扩展的自定义资源(CR)的 **spec.version** 字段中添加比较字符串来定义版本范围。比较字符串是一个空格或以逗号分隔的值列表，以及以双引号括起的一个或多个比较运算符("**"**)。您可以通过包括 **OR**、或双垂直栏(**||**)来添加另一个比较字符串，比较字符串之间的运算符。

表 7.4. 基本比较

比较运算符	定义
=	等于
!=	不等于
>	大于

比较运算符	定义
<	小于
>=	大于或等于
<=	小于或等于

您可以使用类似以下示例的范围比较在 Operator 或扩展 CR 中指定版本范围：

版本范围比较示例

```
apiVersion: operators.operatorframework.io/v1alpha1
kind: Operator
metadata:
  name: pipelines-operator
spec:
  packageName: openshift-pipelines-operator-rh
  version: ">=1.11, <1.13"
```

您可以在所有类型的比较字符串中使用通配符字符。OLM 1.0 接受 **x**、**X** 和星号(*)作为通配符字符。您可以在等号 (=) 比较运算符中使用通配符，代表在补丁或次版本级别上定义比较。

表 7.5. 比较字符串中的通配符字符示例

通配符比较	匹配字符串
1.11.x	>=1.11.0, <1.12.0
>=1.12.X	>=1.12.0
<=2.x	<3
*	>=0.0.0

您可以使用波形符 (~) 比较运算符进行补丁版本比较。补丁版本比较指定最高为下一个主版本的次版本。

表 7.6. 补丁版本比较示例

补丁版本比较	匹配字符串
~1.11.0	>=1.11.0, <1.12.0
~1	>=1, <2
~1.12	>=1.12, <1.13
~1.12.x	>=1.12.0, <1.13.0

补丁版本比较	匹配字符串
~1.x	>=1, <2

您可以使用 caret (^) 比较运算符来比较主版本。如果您在第一个稳定版本发布前使用主发行版本比较，则次版本会定义 API 的稳定性级别。在语义版本(SemVer)规格中，第一个稳定版本被发布为 **1.0.0** 版本。

表 7.7. 主发行版本比较示例

主发行版本比较	匹配字符串
^0	>=0.0.0, <1.0.0
^0.0	>=0.0.0, <0.1.0
^0.0.3	>=0.0.3, <0.0.4
^0.2	>=0.2.0, <0.3.0
^0.2.3	>=0.2.3, <0.3.0
^1.2.x	>= 1.2.0, < 2.0.0
^1.2.3	>= 1.2.3, < 2.0.0
^2.x	>= 2.0.0, < 3
^2.3	>= 2.3, < 3

7.3.8.4. 指定目标版本的自定义资源(CR)示例

在 Operator Lifecycle Manager (OLM) 1.0 中，集群管理员可以在自定义资源(CR)中声明性地设置 Operator 或扩展的目标版本。

您可以通过指定以下字段来定义目标版本：

- Channel
- 版本号
- 版本范围

如果您在 CR 中指定频道，OLM 1.0 会安装可在指定频道中解析的 Operator 或扩展的最新版本。当向指定的频道发布更新时，OLM 1.0 会自动更新至可以从频道解析的最新发行版本。

带有指定频道的 CR 示例

```
apiVersion: operators.operatorframework.io/v1alpha1
kind: Operator
metadata:
```

```
name: pipelines-operator
spec:
  packageName: openshift-pipelines-operator-rh
  channel: latest ❶
```

- ❶ 安装可从指定频道解析的最新版本。对频道的更新会自动安装。

如果在 CR 中指定 Operator 或扩展的目标版本，OLM 1.0 将安装指定的版本。当在 CR 中指定目标版本时，OLM 1.0 在向目录发布更新时不会更改目标版本。

如果要更新集群中安装的 Operator 版本，您必须手动编辑 Operator 的 CR。指定 Operator 的目标版本将 Operator 的版本固定到指定的发行版本。

指定了目标版本的 CR 示例

```
apiVersion: operators.operatorframework.io/v1alpha1
kind: Operator
metadata:
  name: pipelines-operator
spec:
  packageName: openshift-pipelines-operator-rh
  version: 1.11.1 ❶
```

- ❶ 指定目标版本。如果要更新安装的 Operator 或扩展版本，您必须手动将 CR 更新至所需的目标版本。

如果要为 Operator 或扩展定义可接受的版本范围，您可以使用比较字符串指定版本范围。当您指定版本范围时，OLM 1.0 会安装可由 Operator Controller 解析的 Operator 或扩展的最新版本。

指定了版本范围的 CR 示例

```
apiVersion: operators.operatorframework.io/v1alpha1
kind: Operator
metadata:
  name: pipelines-operator
spec:
  packageName: openshift-pipelines-operator-rh
  version: >1.11.1 ❶
```

- ❶ 指定所需的版本范围大于 1.11.1 版本。如需更多信息，请参阅“支持版本范围”。

创建或更新 CR 后，运行以下命令来应用配置文件：

命令语法

```
$ oc apply -f <extension_name>.yaml
```

7.3.8.5. 强制更新或回滚

OLM 1.0 不支持对下一个主版本的自动更新，或回滚到较早的版本。如果要执行主版本更新或回滚，您必须验证并强制更新。



警告

您必须验证强制进行手动更新或回滚的结果。验证强制更新或回滚失败可能会产生灾难性后果，如数据丢失。

先决条件

- 已安装目录。
- 已安装 Operator 或扩展。

流程

1. 编辑 Operator 或扩展的自定义资源 (CR)，如下例所示：

CR 示例

```

apiVersion: olm.operatorframework.io/v1alpha1
kind: Operator
metadata:
  name: <operator_name> 1
spec:
  packageName: <package_name> 2
  version: <version> 3
  upgradeConstraintPolicy: Ignore 4

```

- 1 指定 Operator 或扩展的名称，如 **pipelines-operator**
- 2 指定软件包名称，如 **openshift-pipelines-operator-rh**。
- 3 指定被阻断的更新或回滚版本。
- 4 可选：指定升级约束策略。要强制更新或回滚，请将字段设置为 **Ignore**。如果未指定，则默认设置为 **Enforce**。

2. 运行以下命令，对 Operator 或 extensions CR 应用更改：

```
$ oc apply -f <extension_name>.yaml
```

其他资源

- [支持版本范围](#)

7.3.9. 删除 Operator

您可以通过删除 Operator 的自定义资源(CR)来删除 Operator 及其自定义资源定义(CRD)。

先决条件

- 已安装目录。
- 已安装 Operator。

流程

- 运行以下命令来删除 Operator 及其 CRD：

```
$ oc delete operator.operators.operatorframework.io <operator_name>
```

输出示例

```
operator.operators.operatorframework.io "<operator_name>" deleted
```

验证

- 运行以下命令验证您的 Operator 及其资源已被删除：
 - 运行以下命令验证 Operator 已被删除：

```
$ oc get operator.operators.operatorframework.io
```

输出示例

```
No resources found
```

- 运行以下命令验证 Operator 的系统命名空间是否已删除：

```
$ oc get ns <operator_name>-system
```

输出示例

```
Error from server (NotFound): namespaces "<operator_name>-system" not found
```

7.3.10. 删除目录

您可以通过删除其自定义资源 (CR) 来删除目录。

先决条件

- 已安装目录。

流程

- 运行以下命令来删除目录：

```
$ oc delete catalog <catalog_name>
```

输出示例

```
catalog.catalogd.operatorframework.io "my-catalog" deleted
```

验证

- 运行以下命令验证目录是否已删除：

```
$ oc get catalog
```

7.4. 在 OLM 1.0 中管理普通捆绑包（技术预览）

在 Operator Lifecycle Manager (OLM) 1.0 中，**普通捆绑包**是 YAML 格式的任何 Kubernetes 清单的静态集合。在 **olm.bundle** schema 对象的实验性 **olm.bundle.mediatype** 属性中，普通捆绑包(**plain+v0**)与常规(**registry+v1**)捆绑包不同。



重要

OLM 1.0 只是一个技术预览功能。技术预览功能不受红帽产品服务等级协议 (SLA) 支持，且功能可能并不完整。红帽不推荐在生产环境中使用它们。这些技术预览功能可以使用户提早试用新的功能，并有机会在开发阶段提供反馈意见。

有关红帽技术预览功能支持范围的更多信息，请参阅[技术预览功能支持范围](#)。

作为集群管理员，您可以通过完成以下步骤来构建和发布包含普通捆绑包镜像的基于文件的目录：

1. 构建普通捆绑包镜像。
2. 创建基于文件的目录。
3. 将普通捆绑包镜像添加到您的基于文件的目录中。
4. 将目录构建为镜像。
5. 发布您的目录镜像。

其他资源

- [RukPak 组件和打包格式](#)

7.4.1. 先决条件

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



注意

对于 OpenShift Container Platform 4.15，适用于 OLM 1.0 的流程都是基于 CLI 的。另外，管理员也可以使用普通方法（如 **Import YAML** 和 **Search** 页面）在 web 控制台中创建和查看相关对象。但是，现有的 **OperatorHub** 和 **Installed Operators** 页面还不会显示 OLM 1.0 组件。

- 在集群中启用 **TechPreviewNoUpgrade** 功能集



警告

启用 **TechPreviewNoUpgrade** 功能集将无法撤消，并防止次版本更新。不建议在生产环境集群中使用这些功能集。

- 在您的工作站上安装的 OpenShift CLI (**oc**)
- 工作站上安装的 **opm** CLI
- 工作站上安装的 Docker 或 Podman
- 推送到容器 registry，如 [Quay](#)
- 在项目根目录的一个扁平目录中您的捆绑包的 Kubernetes 清单，类似于以下结构：

目录结构示例

```
manifests
├── namespace.yaml
├── service_account.yaml
├── cluster_role.yaml
├── cluster_role_binding.yaml
└── deployment.yaml
```

其他资源

- [使用功能门启用功能](#)

7.4.2. 从镜像源构建普通捆绑包镜像

Operator Controller 目前支持安装仅从 *普通捆绑包镜像* 创建的普通捆绑包。

流程

1. 在项目的根目录中，创建一个可构建捆绑包镜像的 Dockerfile：

plainbundle.Dockerfile 示例

```
FROM scratch 1
ADD manifests /manifests
```

- 1** 使用 **FROM scratch** 指令可以使镜像的大小更小。捆绑包镜像不需要其他文件或目录。

2. 使用您首选的构建工具构建开放容器项目(OCI)兼容镜像，如下例所示：

```
$ podman build -f plainbundle.Dockerfile -t \
  quay.io/<organization_name>/<repository_name>:<image_tag> . 1
```

- 1 使用引用已推送访问权限的存储库的镜像标签。

3. 运行以下命令将镜像推送到远程 registry :

```
$ podman push quay.io/<organization_name>/<repository_name>:<image_tag>
```

7.4.3. 创建基于文件的目录

如果您没有基于文件的目录，则必须执行以下步骤来初始化目录。

流程

1. 运行以下命令，为目录创建一个目录：

```
$ mkdir <catalog_dir>
```

2. 通过在与上一步中相同的目录级别运行 **opm generate dockerfile** 命令生成可构建目录镜像的 Dockerfile：

```
$ opm generate dockerfile <catalog_dir> \
  -i registry.redhat.io/openshift4/ose-operator-registry:v4.15 1
```

- 1 使用 **-i** 标志指定官方红帽基础镜像，否则 Dockerfile 使用默认的上游镜像。



注意

生成的 Dockerfile 必须与您在上一步中创建的目录目录位于同一个父目录中：

目录结构示例

```
.
├── <catalog_dir>
└── <catalog_dir>.Dockerfile
```

3. 运行 **opm init** 命令，使用扩展的软件包定义填充目录：

```
$ opm init <extension_name> \
  --output json \
  > <catalog_dir>/index.json
```

此命令在指定的目录配置文件中生成 **olm.package** 声明性配置 blob。

7.4.4. 在基于文件的目录中添加普通捆绑包

opm render 命令不支持将普通捆绑包添加到目录中。您必须手动将纯文本捆绑包添加到基于文件的目录中，如以下步骤所示。

流程

1. 验证目录的 **index.json** 或 **index.yaml** 文件是否类似以下示例：

<catalog_dir>/index.json 文件示例

```
{
  {
    "schema": "olm.package",
    "name": "<extension_name>",
    "defaultChannel": ""
  }
}
```

2. 要创建 **olm.bundle** blob，请编辑 **index.json** 或 **index.yaml** 文件，如下例所示：

带有 olm.bundle blob 的 <catalog_dir>/index.json 文件示例

```
{
  "schema": "olm.bundle",
  "name": "<extension_name>.v<version>",
  "package": "<extension_name>",
  "image": "quay.io/<organization_name>/<repository_name>:<image_tag>",
  "properties": [
    {
      "type": "olm.package",
      "value": {
        "packageName": "<extension_name>",
        "version": "<bundle_version>"
      }
    },
    {
      "type": "olm.bundle.mediatype",
      "value": "plain+v0"
    }
  ]
}
```

3. 要创建 **olm.channel** blob，请编辑 **index.json** 或 **index.yaml** 文件，如下例所示：

带有 olm.channel blob 的 <catalog_dir>/index.json 文件示例

```
{
  "schema": "olm.channel",
  "name": "<desired_channel_name>",
  "package": "<extension_name>",
  "entries": [
    {
      "name": "<extension_name>.v<version>"
    }
  ]
}
```

验证

1. 打开 **index.json** 或 **index.yaml** 文件，并确保它类似以下示例：

<catalog_dir>/index.json 文件示例

```

{
  "schema": "olm.package",
  "name": "example-extension",
  "defaultChannel": "preview"
}
{
  "schema": "olm.bundle",
  "name": "example-extension.v0.0.1",
  "package": "example-extension",
  "image": "quay.io/example-org/example-extension-bundle:v0.0.1",
  "properties": [
    {
      "type": "olm.package",
      "value": {
        "packageName": "example-extension",
        "version": "0.0.1"
      }
    },
    {
      "type": "olm.bundle.mediatype",
      "value": "plain+v0"
    }
  ]
}
{
  "schema": "olm.channel",
  "name": "preview",
  "package": "example-extension",
  "entries": [
    {
      "name": "example-extension.v0.0.1"
    }
  ]
}

```

2. 运行以下命令验证您的目录：

```
$ opm validate <catalog_dir>
```

7.4.5. 构建和发布基于文件的目录

流程

1. 运行以下命令，将基于文件的目录构建为镜像：

```
$ podman build -f <catalog_dir>.Dockerfile -t \
  quay.io/<organization_name>/<repository_name>:<image_tag> .
```

2. 运行以下命令来推送目录镜像：

```
$ podman push quay.io/<organization_name>/<repository_name>:<image_tag>
```

