OpenShift Container Platform 4.2

Operator

在 OpenShift Container Platform 4.2 中使用 Operator

Last Updated: 2020-08-21

OpenShift Container Platform 4.2 Operator

在 OpenShift Container Platform 4.2 中使用 Operator

法律通告

摘要

本文档提供有关在 OpenShift Container Platform 4.2 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 了解 OPERATOR

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

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

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

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

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 相比怎么样？

Service Broker 朝着实现应用程序的编程式发现和部署的目标前进了一步。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。

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) 控制集群中 Operator 的安装、升级和基于角色的访问控制 (RBAC)。在 OpenShift Container Platform 4.2 中默认部署。

Operator Registry

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

OperatorHub

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

Operator Metering

Operator Metering 收集集群上有关 Operator 的操作指标（用于第 2 天的管理）和聚集使用指标。

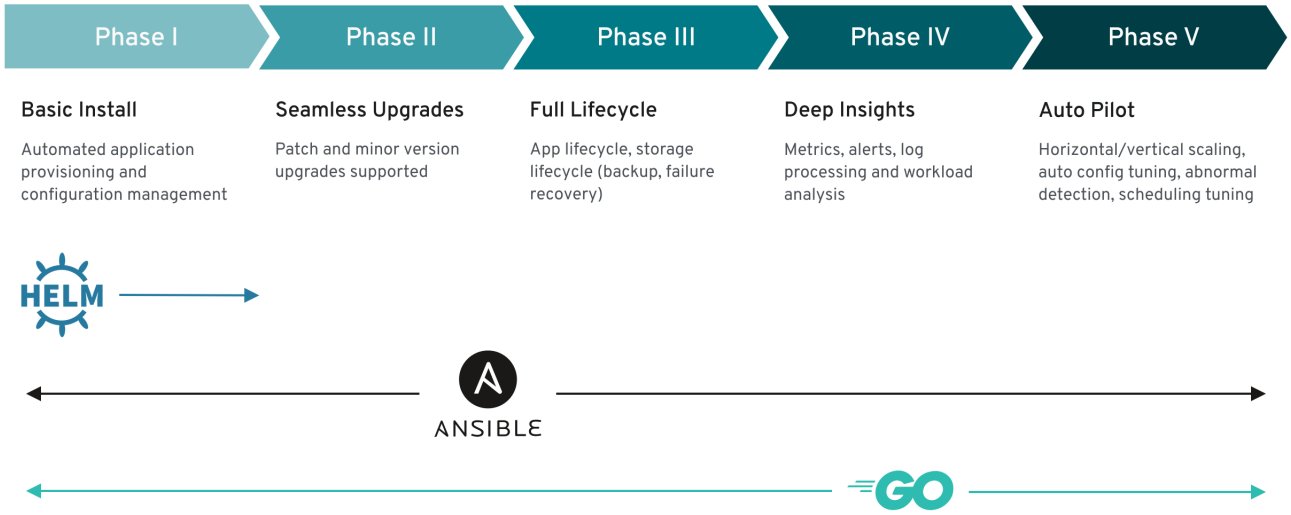
这些工具可组合使用，因此您可自由选择对您有用的工具。

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。

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

图 1.1. Operator 成熟度模型



以上模型还展示了如何通过 Operator SDK 的 Helm、Go 和 Ansible 功能来更好地开发这些功能。

第 2 章 了解 OPERATOR LIFECYCLE MANAGER (OLM)

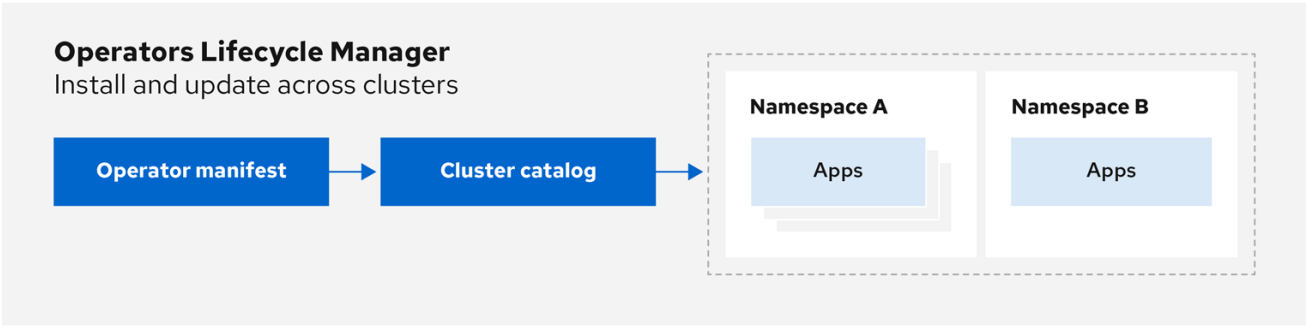
2.1. OPERATOR LIFECYCLE MANAGER 工作流和架构

该指南概述了 OpenShift Container Platform 中 Operator Lifecycle Manager (OLM) 的概念和架构。

2.1.1. Operator Lifecycle Manager 概述

在 OpenShift Container Platform 4.2 中，*Operator Lifecycle Manager* (OLM) 可帮助用户安装、更新和管理所有 Operator 以及在用户集群中运行的关联服务的生命周期。Operator Lifecycle Manager 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 Kubernetes 原生应用程序 (Operator)。

图 2.1. Operator Lifecycle Manager 工作流



OLM 默认在 OpenShift Container Platform 4.2 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。

2.1.2. ClusterServiceVersions (CSV)

ClusterServiceVersion (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。

CSV 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)。

CSV 包含以下内容：

元数据

- 应用程序元数据：
  - 名称、描述、版本（符合 semver）、链接、标签、图标等。

安装策略

- 类型：部署
  - 服务账户和所需权限集

- 部署集。

CRD

- 类型
- 自有：由该服务管理
- 必需：集群中必须存在，该服务才可运行
- 资源：Operator 与之交互的资源列表
- 描述符：注解 CRD 规格和状态字段以提供语义信息

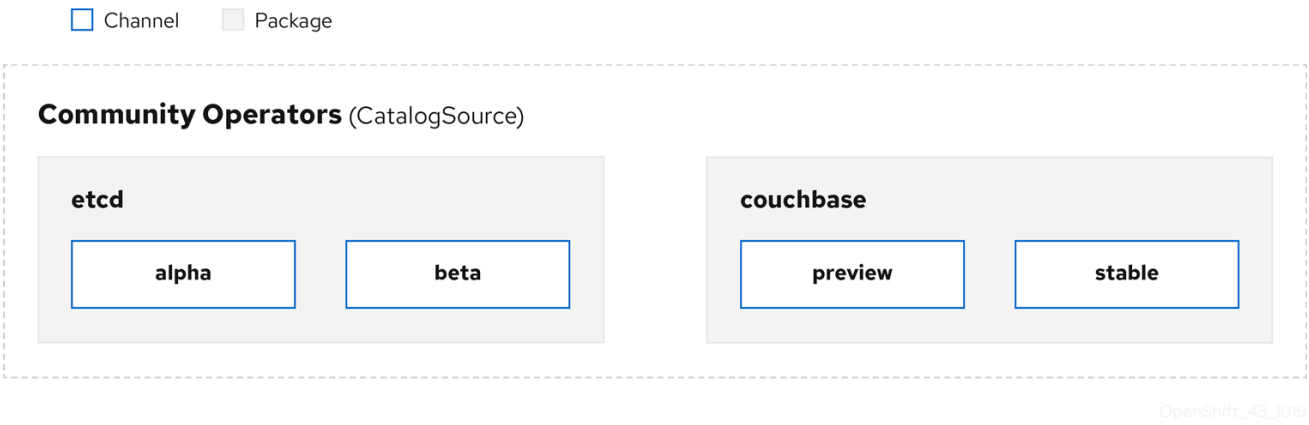
2.1.3. OLM 中的 Operator 安装和升级工作流

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

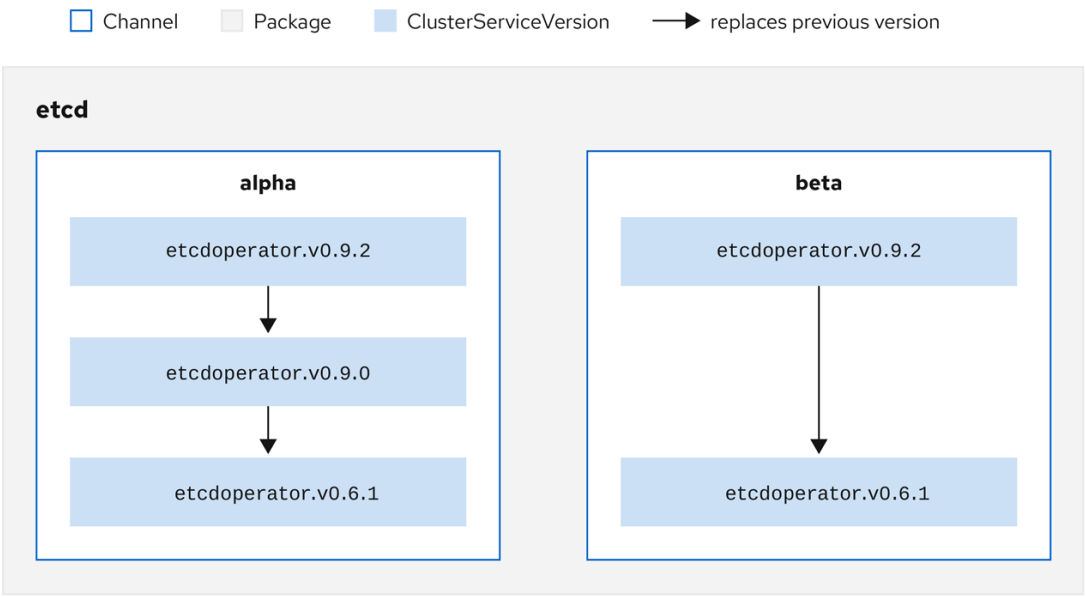
CSV 中定义的 Operator 元数据可保存在一个名为 CatalogSource 的集合中。CatalogSource 使用 Operator Registry API，OLM 又使用 CatalogSource 来查询是否有可用的 Operator 及已安装 Operator 是否有升级版本。

图 2.2. CatalogSource 概述

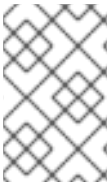


在 CatalogSource 中，Operator 被整合为更新*软件包*和更新流，我们称为*频道*，这应是 OpenShift Container Platform 或其他软件（如 Web 浏览器）在持续发行周期中的常见更新模式。

图 2.3. CatalogSource 中的软件包和频道



用户在 *Subscription* 中的特定 CatalogSource 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。

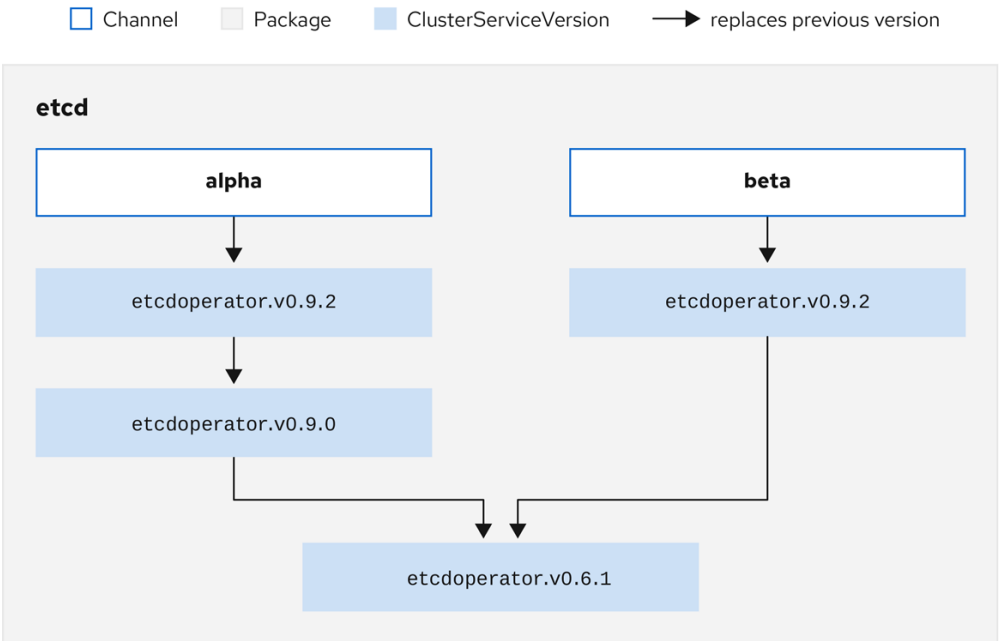


注意

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 2.4. OLM 的可用频道更新图表



例如：

软件包中的频道

```
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 CatalogSource、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。

2.1.3.1. 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 CatalogSource，并在订阅的频道中检测升级包，新的 **0.1.3** 版 CSV 取代了旧的但未安装的 **0.1.2** 版 CSV，后者又取代了更旧的已安装的 **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。

2.1.3.2. 跳过升级

OLM 的基本升级路径如下：

- 通过对 Operator 的一个或多个更新来更新 CatalogSource。
- OLM 会遍历 Operator 的所有版本，直到到达 CatalogSource 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。

例如：

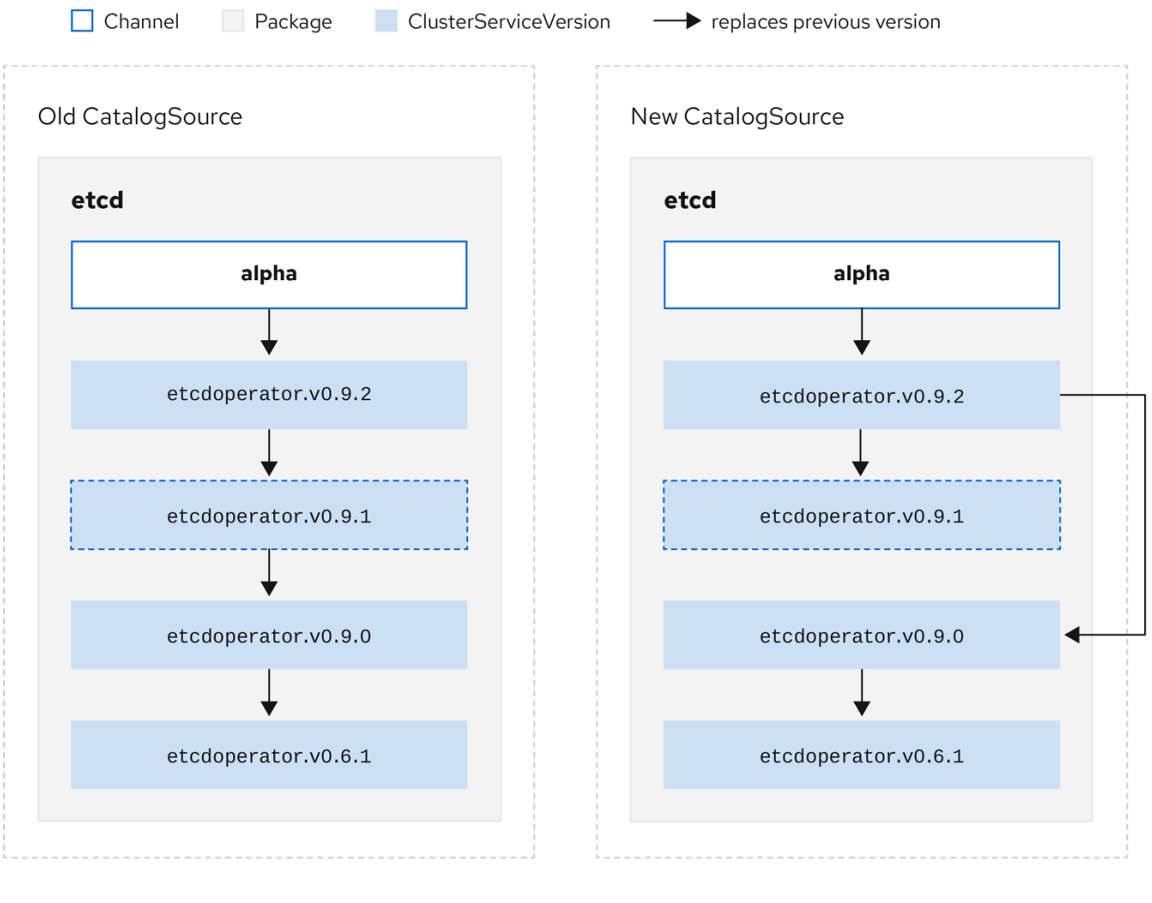
带有跳过发行版本的 CSV

```
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 2.5. 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。

2.1.3.3. 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。

例如：

附带 skipRange 的 CSV

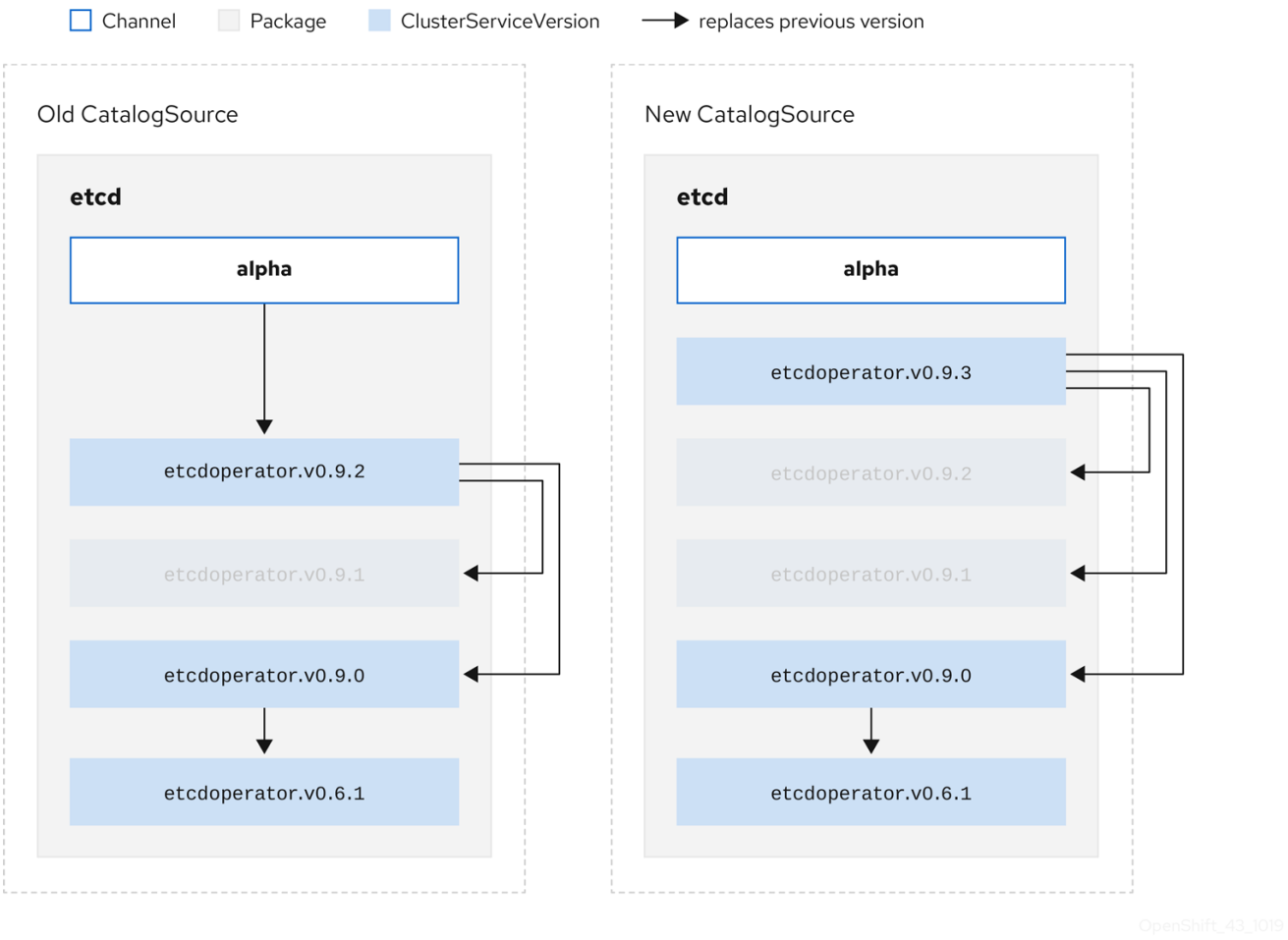
```
apiVersion: operators.coreos.com/v1alpha1
kind: ClusterServiceVersion
metadata:
  name: elasticsearch-operator.v4.1.2
  namespace: <namespace>
  annotations:
    olm.skipRange: '>=4.1.0 <4.1.2'
```

2.1.3.4. 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 不关注主版本、从版本或补丁版本，只需要在目录中构建正确图表。

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

图 2.6. 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。

2.1.4. Operator Lifecycle Manager 架构

Operator Lifecycle Manager 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 2.1. 由 OLM 和 Catalog Operator 管理的 CRD

资源	短名称	所有者	描述
ClusterServiceVersion	**csv**	OLM	应用程序元数据：名称、版本、图标、所需资源、安装等。

资源	短名称	所有者	描述
InstallPlan	**ip**	Catalog	为自动安装或升级 CSV 而需创建的资源的计算列表。
CatalogSource	**catsrc**	Catalog	定义应用程序的 CSV、CRD 和软件包存储库。
Subscription	**sub**	Catalog	用于通过跟踪软件包中的频道来保持 CSV 最新。
OperatorGroup	**og**	OLM	用于对多个命名空间进行分组，并准备好供 Operator 使用。

每个 Operator 还要负责创建资源：

表 2.2. 由 OLM 和 Catalog Operator 创建的资源

资源	所有者
部署	OLM
ServiceAccounts	
(Cluster)Roles	
(Cluster)RoleBindings	
自定义资源定义 (CRD)	Catalog
ClusterServiceVersions (CSV)	

2.1.4.1. 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。

虽然 OLM Operator 通常被配置为监视所有命名空间，但也可与其他 OLM Operator 一同使用，只要所管理的命名空间不同即可。

OLM Operator 工作流

- 监视命名空间中的 ClusterServiceVersion (CSV)，并检查是否满足要求。如果满足，则运行 CSV 的安装策略。

注意

CSV 必须为 OperatorGroup 的活跃成员才可运行该安装策略。

2.1.4.2. Catalog Operator

Catalog Operator 负责解析和安装 CSV 及其指定的所需资源。另外还负责监视频道中的 CatalogSource 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。

希望跟踪频道中软件包的用户可创建 Subscription 资源，该资源将配置所需软件包、频道和 CatalogSource，以便从中拉取更新。找到更新后，便会代表用户将适当 InstallPlan 写入命名空间。

用户也可直接创建 InstallPlan 资源，其中包含所需 CSV 和批准策略的名称，而 Catalog Operator 会为创建所有所需资源创建一个执行计划。批准后，Catalog Operator 将在 InstallPlan 中创建所有资源；然后单独满足 OLM Operator 的要求，从而继续安装 CSV。

Catalog Operator 工作流

- 拥有 CRD 和 CSV 缓存，按名称索引。
- 监视是否有用户创建的未解析 InstallPlan：
  - 查找与请求名称相匹配的 CSV，并将其添加为已解析的资源。
  - 对于每个受管或所需 CRD，将其添加为已解析的资源。
  - 对于每个所需 CRD，找到管理相应 CRD 的 CSV。
- 监视是否有已解析的 InstallPlan 并为其创建已发现的所有资源（用户批准或自动）。
- 监视 CatalogSource 和 Subscription，并根据它们创建 InstallPlan。

2.1.4.3. 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。

2.1.5. 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 2.3. OLM 公开的指标

名称	描述
csv_count	成功注册的 CSV 数量。

名称	描述
install_plan_count	InstallPlan 的数量。
subscription_count	Subscription 数。
csv_upgrade_count	CatalogSource 的单调计数。

2.2. OPERATOR LIFECYCLE MANAGER 依赖项解析

该指南概述了 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。

2.2.1. 关于依赖项解析

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

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

这意味着，OLM 绝不会：

- 安装一组系统无法提供其所需 API 的 Operator，或
- 更新某个 Operator 之时导致依赖该 Operator 的另一 Operator 中断。

2.2.2. 升级自定义资源定义 (CRD)

如果自定义资源定义 (CRD) 属于单一 ClusterServiceVersion (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。
- 在对照新 CRD 验证模式进行验证时，与 CRD 的服务版本相关联的所有现有实例或自定义资源 (CR) 均有效。

2.2.2.1. 添加新版 CRD

流程

要添加新版 CRD：

1. 在 **versions** 部分的 CRD 资源中新增一个条目。
   例如：如果当前 CRD 有一个 **v1alpha1** 版本，而您想新增 **v1beta1** 版本，并将其标记为新存储版本：

   ```
   versions:
   ```

```
  - name: v1alpha1
    served: true
    storage: false
  - name: v1beta1 (1)
    served: true
    storage: true
```

(1) 新增 **v1beta1** 条目。

2. 如果 CSV 打算使用新版本，请确保已更新您的 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。

2.2.2.2. 弃用或删除 CRD 版本

OLM 不允许立即删除 CRD 的服务版本。弃用的 CRD 版本应首先通过将 CRD 的 **served** 字段设置为 **false** 来禁用。随后在升级 CRD 时便可将非服务版本删除。

流程

要弃用和删除特定 CRD 版本：

1. 将弃用版本标记为非服务版本，表明该版本已不再使用且后续升级时可删除。例如：

```
versions:
  - name: v1alpha1
    served: false (1)
    storage: true
```

(1) 设置为 **false**。

2. 如果要弃用的版本目前为 **storage** 版本，则将该 **storage** 版本切换至服务版本。例如：

```
versions:
  - name: v1alpha1
    served: false
    storage: false (1)
  - name: v1beta1
    served: true
    storage: true (2)
```

(1) (2) 对应更新 **storage** 字段。

注意

要从 CRD 中删除曾是或现在是 **storage** 的特定版本，该版本必须从 CRD 状态下的 **storedVersion** 中删除。OLM 一旦检测到某个已存储版本在新 CRD 中不再存在，OLM 将尝试执行这一操作。

3. 使用以上更改来升级 CRD。
4. 在后续升级周期中，非服务版本可从 CRD 中完全删除。例如：

```
versions:
  - name: v1beta1
    served: true
    storage: true
```

5. 如果该版本已从 CSV 中删除，请确保您的 CSV **owned** 部分中的 CRD 引用版本会相应更新。

2.2.3. 依赖项解析方案示例

在以下示例中，*provider* 是指"拥有"某个 CRD 或 APIService 的 Operator。

示例：弃用从属 API
A 和 B 均为 API（如 CRD）：

- A 的供应商依赖 B。
- B 的供应商有 Subscription。
- B 的供应商会更新以提供 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 的供应商更新至（提供 A2，需要 B2）并弃用 A。
- B 的供应商更新至（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。

2.3. OPERATORGROUP

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

2.3.1. 关于 OperatorGroup

OperatorGroup 是一个 OLM 资源，为 OLM 安装的 Operator 提供多租户配置。OperatorGroup 选择一组目标命名空间，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 ClusterServiceVersion (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。

2.3.2. OperatorGroup 成员资格

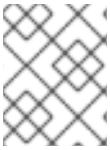
满足以下任一条件，Operator 即可被视为 OperatorGroup 的 *member*：

- Operator 的 CSV 与 OperatorGroup 存在于同一命名空间中。
- Operator 的 CSV InstallMode 支持 OperatorGroup 所针对的命名空间集。

InstallMode 由 **InstallModeType** 字段和 **Supported** 布尔字段组成。CSV 的 spec 可包含由四个不同 **InstallModeType** 组成的 InstallMode 集：

表 2.4. InstallMode 和受支持的 OperatorGroup

InstallModeType	描述
OwnNamespace	Operator 可以是选择其自有命名空间的 OperatorGroup 的成员。
SingleNamespace	Operator 可以是选择某一个命名空间的 OperatorGroup 的成员。
MultiNamespace	Operator 可以是选择多个命名空间的 OperatorGroup 的成员。
AllNamespaces	Operator 可以是选择所有命名空间的 OperatorGroup 的成员（目标命名空间集为空字符串 **""**）。



注意

如果 CSV 的 spec 省略了 **InstallModeType** 条目，则该类型将被认为不受支持，除非可通过暗示支持的现有条目推算出其受支持。

2.3.3. 目标命名空间选择

使用带有 **spec.selector** 字段的标签选择器为 OperatorGroup 指定命名空间集：

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

```
  selector:
    matchLabels:
      cool.io/prod: "true"
```

您还可使用 **spec.targetNamespaces** 字段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
  - my-other-namespace
  - my-other-other-namespace
```

注意

如果 **spec.targetNamespaces** 和 **spec.selector** 均已定义，则会忽略 **spec.selector**。

另外，您还可省略 **spec.selector** 和 **spec.targetNamespaces**，以指定选择所有命名空间的*全局* OperatorGroup：

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

选定命名空间的解析集会显示在 OperatorGroup 的 **status.namespaces** 字段中。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。

2.3.4. Operatorgroup CSV 注解

OperatorGroup 的成员 CSV 附有以下注解：

注解	描述
olm.operatorGroup=<group_name>	包含 OperatorGroup 的名称。
olm.operatorGroupNamespace=<group_namespace>	包含 OperatorGroup 的命名空间。
olm.targetNamespaces=<target_namespaces>	包含以逗号分隔的字符串，该字符串列出了 OperatorGroup 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。

2.3.5. 所提供的 API 注解

olm.providedAPIs 注解中会显示有关 OperatorGroup 提供了哪些 **GroupVersionKinds** (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 OperatorGroup 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
```

2.3.6. 基于角色的访问控制

创建 OperatorGroup 时，会生成三个 ClusterRole。每个 ClusterRole 均包含一个 AggregationRule，后者设置了 ClusterRoleSelector 以匹配标签，如下所示：

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

当 CSV 成为 OperatorGroup 的活跃成员时，只要该 CSV 正在使用 **AllNamespaces** InstallMode 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 ClusterRole
- 来自 APIService 中每个 API 资源的 ClusterRole
- 额外的 Role 和 RoleBinding

表 2.5. 来自 CRD，针对每个 API 资源生成的 ClusterRole

ClusterRole	设置
<kind>.<group>-<version>-admin	**<kind>** 上的操作动词： • ***** 聚合标签： • **rbac.authorization.k8s.io/aggregate-to-admin: true** • **olm.opgroup.permissions/aggregate-to-admin: <operatorgroup_name>**
<kind>.<group>-<version>-edit	**<kind>** 上的操作动词： • **create** • **update** • **patch** • **delete** 聚合标签： • **rbac.authorization.k8s.io/aggregate-to-edit: true** • **olm.opgroup.permissions/aggregate-to-edit: <operatorgroup_name>**

ClusterRole	设置
<kind>.<group>-<version>-view	**<kind>** 上的操作动词： • **get** • **list** • **watch** 聚合标签： • **rbac.authorization.k8s.io/aggregate-to-view: true** • **olm.opgroup.permissions/aggregate-to-view: <operatorgroup_name>**
<kind>.<group>-<version>-view-crdview	**apiextensions.k8s.io customresourcedefinitions <crd-name>** 上的操作动词： • **get** 聚合标签： • **rbac.authorization.k8s.io/aggregate-to-view: true** • **olm.opgroup.permissions/aggregate-to-view: <operatorgroup_name>**

表 2.6. 来自 APIService，针对每个 API 资源生成的 ClusterRole

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

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

额外的 Role 和 RoleBinding

- 如果 CSV 精确定义了一个包含 * 的目标命名空间，则会针对 CSV 权限字段中定义的每个权限生成一个 ClusterRole 和相应的 ClusterRoleBinding。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*精确定义一个包含 * 的目标命名空间，则 Operator 命名空间中所有带有 **olm.owner: <csv_name>** 和 **olm.owner.namespace: <csv_namespace>** 标签的 Role 和 RoleBinding 均会被复制到目标命名空间中。

2.3.7. 复制的 CSV

OLM 会在每个 OperatorGroup 的目标命名空间中复制 OperatorGroup 的所有活跃成员 CSV。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 OperatorGroup 不再指向复制的 CSV 命名空间时，将会删除复制的 CSV。

2.3.8. 静态 OperatorGroup

如果 OperatorGroup 的 **spec.staticProvidedAPIs** 字段设置为 **true**，则该 OperatorGroup 为 *静态*。这样 OLM 就不会修改 OperatorGroup 的 **olm.providedAPIs** 注解，因此可以提前设置该注解。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 OperatorGroup 来防止命名空间集中发生资源争用，则这一操作十分有用。

下面提供一个使用 **something.cool.io/cluster-monitoring: "true"** 注解来保护所有命名空间中 Prometheus 资源的 OperatorGroup 的示例：

```
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"
```

2.3.9. Operatorgroup 交集

如果两个 OperatorGroup 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 OperatorGroup 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，OperatorGroup 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 的 OperatorGroup 和其他所有 OperatorGroup 之间在所提供的 API 方面的交集。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 OperatorGroup 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* OperatorGroup 提供的 API 的子集：
  - 如果 OperatorGroup 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 OperatorGroup *不是* 静态的：

        - 将 OperatorGroup 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* OperatorGroup 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 OperatorGroup 提供的 API 的子集：
    - 如果 OperatorGroup 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 OperatorGroup *不是*静态的：
        - 将 OperatorGroup 的 **olm.providedAPIs** 注解替换为其本身与 CSV 提供的 API 的差集。

注意

由 OperatorGroup 造成的故障状态并非终端状态。

每次 OperatorGroup 同步时均会执行以下操作：

- 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。

2.3.10. OperatorGroup 故障排除

成员资格

- 如果一个命名空间中存在多个 OperatorGroup，则在该命名空间中创建的所有 CSV 均会变成故障状态，故障原因为：**TooManyOperatorGroups**。一旦命名空间中 OperatorGroup 的数量变成 1，因该原因处于故障状态的 CSV 将转变为等待处理状态。
- 如果 CSV 的 InstallMode 不支持其命名空间中 OperatorGroup 的目标命名空间选择，CSV 将变为故障状态，故障原因为 **UnsupportedOperatorGroup**。一旦 OperatorGroup 的目标命名空间选择变为受支持的配置，或 CSV 的 InstallMode 经修改后支持 OperatorGroup 的目标命名空间选择，因该原因处于故障状态的 CSV 将转变为等待处理状态。

第 3 章 了解 OPERATORHUB

本指南主要概述 OperatorHub 的构架。

3.1. OPERATORHUB 概述

OperatorHub 可通过 OpenShift Container Platform Web 控制台获得，是集群管理员用于发现和安装 Operator 的界面。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成以下类别的 OperatorSource 中选择：

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

3.2. 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。

Marketplace Operator 负责管理 OperatorHub 和 OperatorSource 自定义资源定义 (CRD)。

注意

尽管某些 OperatorSource 信息会通过 OperatorHub 用户界面公开，但只有创建自有 Operator 的用户可直接使用这些信息。

注意

虽然 OperatorHub 不再使用 CatalogSourceConfig 资源，但在 OpenShift Container Platform 中仍支持这些资源。

3.2.1. OperatorHub CRD

您可使用 OperatorHub CRD，将集群上 OperatorHub 提供的默认 OperatorSource 的状态更改为启用或禁用。该功能对于在有限网络环境中配置 OpenShift Container Platform 来说非常实用。

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** 是一种覆盖，用于控制在 OpenShift Container Platform 安装期间默认配置的所有默认 OperatorSource 的可用性。
2. 通过更改每个源的 **disabled** 参数值，即可分别禁用各个默认 OperatorSource。

3.2.2. OperatorSource CRD

对于各个 Operator，OperatorSource CRD 用于定义用于存储 Operator 包的外部数据存储。

OperatorSource 自定义资源示例

```
apiVersion: operators.coreos.com/v1
kind: OperatorSource
metadata:
  name: community-operators
  namespace: marketplace
spec:
  type: appregistry (1)
  endpoint: https://quay.io/cnr (2)
  registryNamespace: community-operators (3)
  displayName: "Community Operators" (4)
  publisher: "Red Hat" (5)
```

1. 要将数据存储标识为应用程序 registry，请将 **type** 设置为 **appregistry**。
2. 目前，Quay 是 OperatorHub 使用的外部数据存储，因此 Quay.io **appregistry** 的端点被设置为 **https://quay.io/cnr**。
3. 对于社区 Operator，**registryNamespace** 会被设置为 **community-operator**。
4. 可选择将 **displayName** 设置为 OperatorHub UI 中显示的 Operator 的名称。
5. 也可选择将 **publisher** 设置为 OperatorHub UI 中显示的发布该 Operator 的个人或机构。

第 4 章 在集群中添加 OPERATOR

本指南指导集群管理员将 Operator 安装至 OpenShift Container Platform 集群并将 Operator 订阅到命名空间。

4.1. 安装来自 OPERATORHUB 的 OPERATOR

作为集群管理员，您可使用 OpenShift Container Platform Web 控制台或 CLI 安装来自 OperatorHub 的 Operator。然后，您可将 Operator 订阅至一个或多个命名空间，供集群上的开发人员使用。

安装过程中，您必须为 Operator 确定以下初始设置：

安装模式

选择 **All namespaces on the cluster (default)**将 Operator 安装至所有命名空间；或选择单个命名空间（如果可用），仅在选定命名空间中安装 Operator。本例选择 **All namespaces...**，以便 Operator 可用于所有用户和项目。

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

您可选择自动或手动更新。如果选择自动更新某个已安装的 Operator，则当相应 Operator 有可用的新版本时，Operator Lifecycle Manager (OLM) 将自动升级该 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。

4.1.1. 使用 Web 控制台从 OperatorHub 安装

此流程以 Cuchbase Operator 为例，使用 OpenShift Container Platform Web 控制台从 OperatorHub 安装并订阅 Operator。

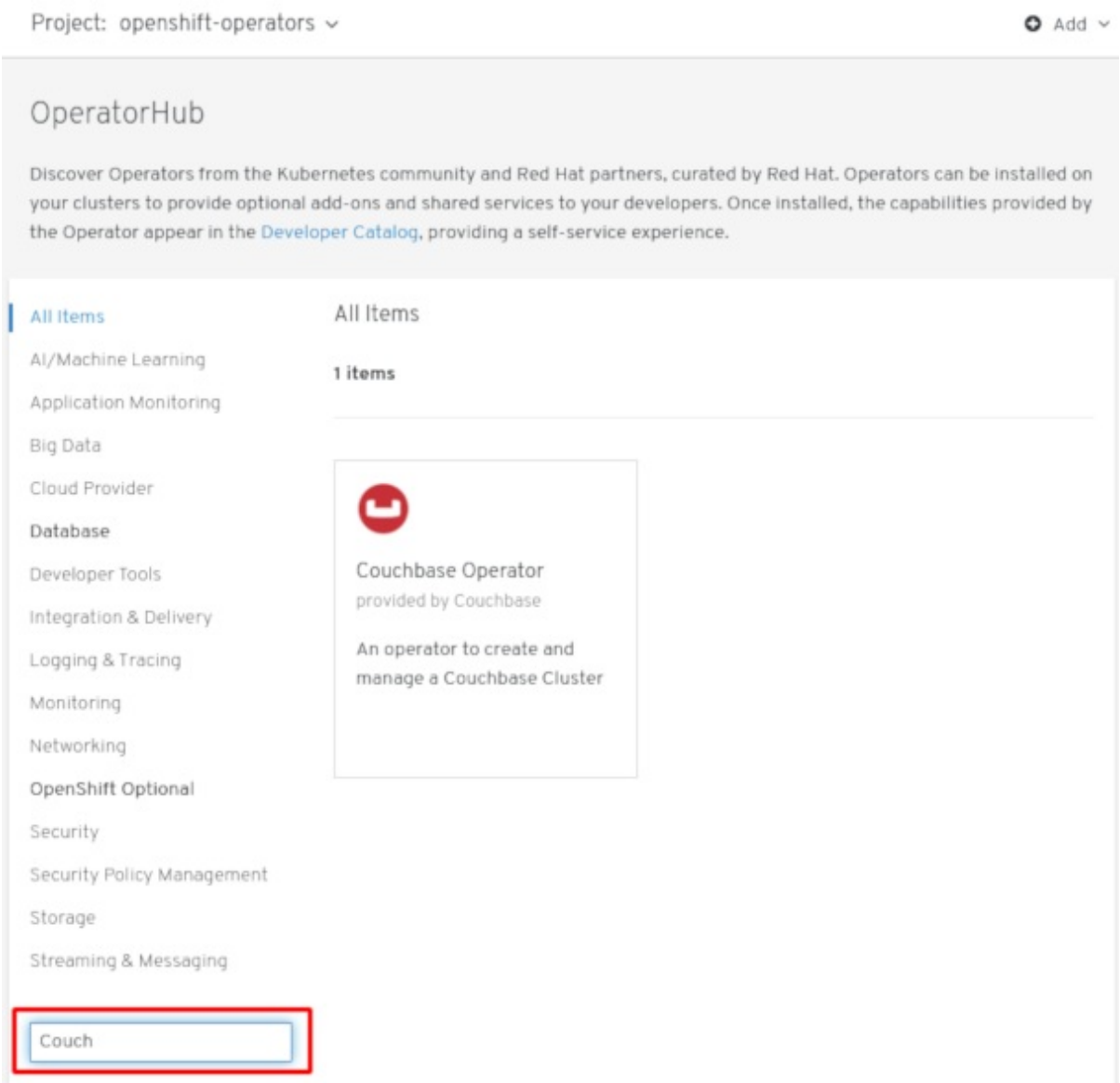
先决条件

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

流程

1. 在 Web 控制台中导航至 **Operators → OperatorHub** 页面。
2. 在 **Filter by keyword** 方框中滚动或键入关键词（本例中为 **Couchbase**），以查找所需 Operator。

图 4.1. 通过关键词筛选 Operator



3. 选定 Operator。对于社区 Operator 来说，系统会提醒您红帽对这类 Operator 不予认证。您必须确认收到该提醒才可继续。这时会显示 Operator 信息。

4. 阅读 Operator 信息并单击 **Install**。

5. 在 **Create Operator Subscription** 页面：

   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。

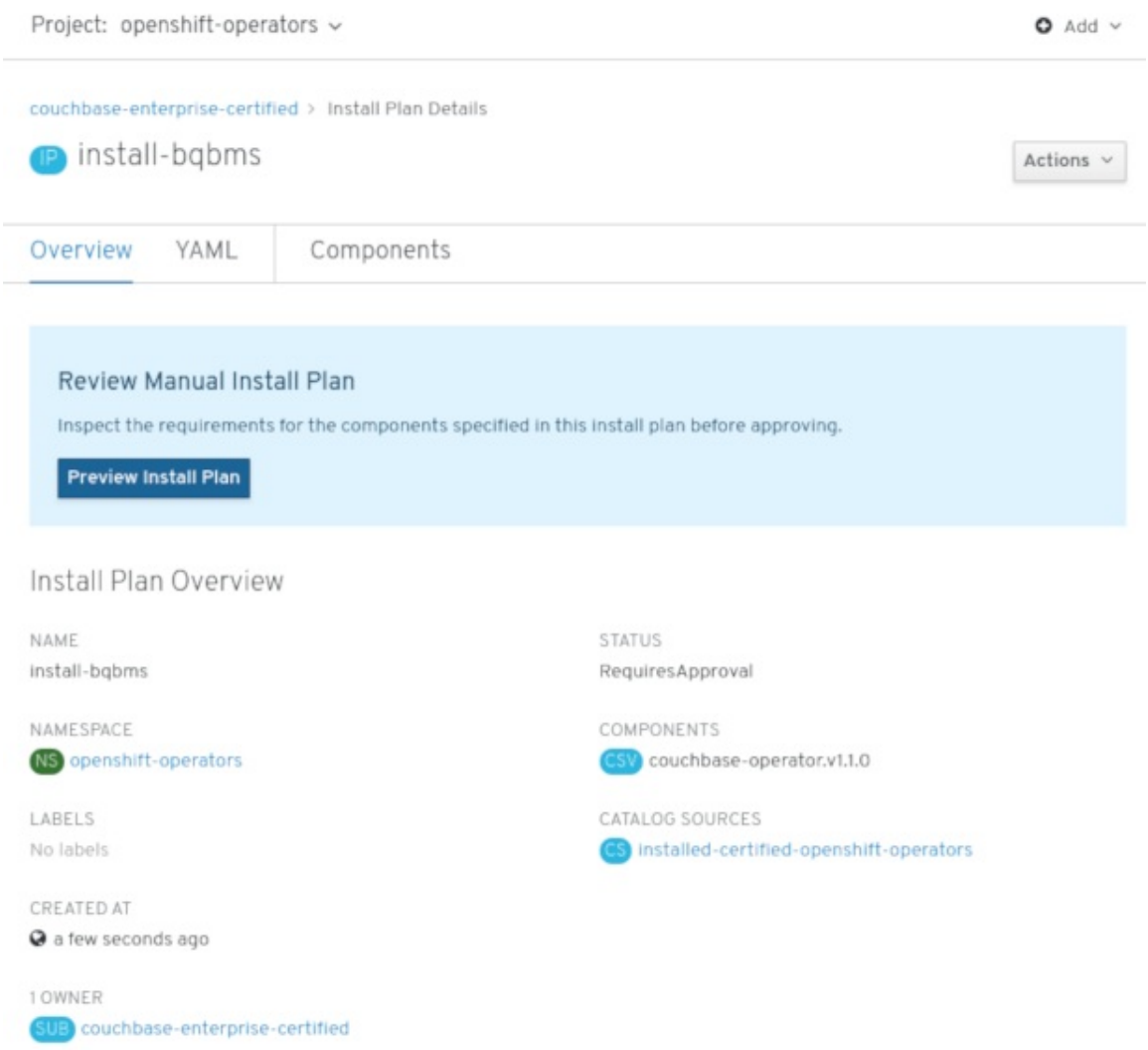
   b. 选择一个**更新频道**（如有多个可用）。

   c. 如前面所述，选择**自动或手动**批准策略。

6. 单击 **Subscribe**，以便该 Operator 可供 OpenShift Container Platform 集群上的所选命名空间使用。

a. 如果选择手动批准策略，则订阅的升级状态将保持**正在升级**，直至您审核并批准该 Install Plan。

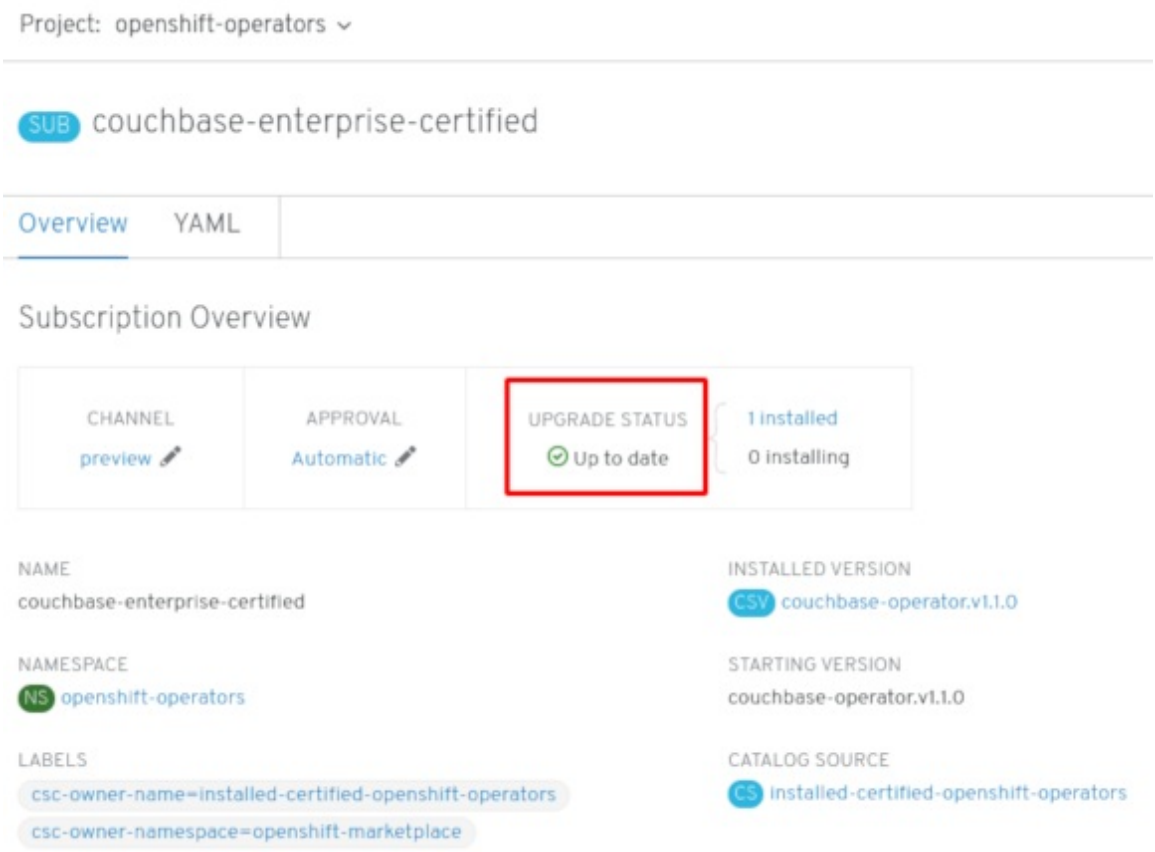
图 4.2. 在 Install Plan 页面手动批准



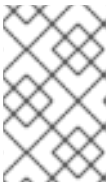
在 **Install Plan** 页面批准后，订阅的升级状态将变为**最新**。

b. 如果选择自动批准策略，则升级状态应自动变为**最新**，而无需人为干预。

图 4.3. 订阅升级状态：最新



7. 当订阅的升级状态变为**最新**后，选择 **Operators → Installed Operators** 来验证最终是否出现了 **Couchbase** ClusterServiceVersion (CSV) 且其 **Status** 在相关命名空间中最终能否解析为 **InstallSucceeded**。



注意

对于 **All namespaces...** 安装模式，该状态会在 **openshift-operators** 命名空间中被解析为 **InstallSucceeded**，但您如果检查其他命名空间，其状态则会变为 **Copied**。

如果没有：

a. 检查"**Workloads → Pods**"页面上的 **openshift-operators** 项目（或其他相关命名空间，如果选中 **A specific namespace...** Installation Mode）中报告问题的 Pod 中的日志，以便进一步排除故障。

4.1.2. 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** 命令来创建或更新订阅对象。

先决条件

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

流程

1. 查看 OperatorHub 中集群可用的 Operator 列表。

```
$ oc get packagemanifests -n openshift-marketplace
NAME                              CATALOG               AGE
3scale-operator                   Red Hat Operators     91m
amq-online                        Red Hat Operators     91m
amq-streams                       Red Hat Operators     91m
...
couchbase-enterprise-certified    Certified Operators   91m
mariadb                           Certified Operators   91m
mongodb-enterprise                Certified Operators   91m
...
etcd                              Community Operators   91m
jaeger                            Community Operators   91m
kubefed                           Community Operators   91m
...
```

注意适用于您所需 Operator 的 CatalogSource。

2. 检查所需 Operator，以验证其支持的 InstallMode 和可用频道：

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

3. 订阅 Operator 的命名空间必须具有与 Operator 的 InstallMode 相匹配的 OperatorGroup，可采用 **AllNamespaces** 模式，也可采用 **SingleNamespace** 模式。如果要安装的 Operator 采用 **AllNamespaces** 模式，则表明 **openshift-operators** 命名空间中已有适当的 OperatorGroup。如果要安装的 Operator 采用 **SingleNamespace** 模式，而您没有适当的 OperatorGroup，则必须创建一个。

注意

如果选择 **SingleNamespace** 模式，该流程的 Web 控制台版本会在后台自动为您处理 OperatorGroup 和订阅对象的创建。

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

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

订阅示例

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: <operator_name>
  namespace: openshift-operators 1
spec:
  channel: alpha
  name: <operator_name> 2
  source: redhat-operators 3
  sourceNamespace: openshift-marketplace 4
```

1. 如果使用 **AllNamespaces** InstallMode，请指定 **openshift-operators** 命名空间。如果使用 **SingleNamespace** InstallMode，则指定相关单一命名空间。
2. 要订阅的 Operator 的名称。
3. 提供 Operator 的 CatalogSource 的名称。
4. CatalogSource 的命名空间。将 **openshift-marketplace** 用于默认的 OperatorHub CatalogSource。

5. 创建订阅对象：

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

此时，OLM 已了解所选的 Operator。Operator 的 ClusterServiceVersion (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

- 要使用 OperatorHub 在集群中安装自定义 Operator，您必须首先将 Operator 工件上传至 Quay.io，然后再将您自己的 **OperatorSource** 添加至集群中。另外，您还可在 Operator 中添加 Secret 以便进行身份验证。之后，您便可像其他 Operator 一样在集群中管理这个 Operator 了。如需了解具体步骤，请参阅 Testing Operators。

其他资源

- 关于 OperatorGroup

4.2. 覆盖 OPERATOR 的代理设置

如果配置了集群范围的出口代理，则使用 Operator Lifecycle Manager (OLM) 创建的应用程序会继承其 Deployment 和 Pod 上的集群范围代理设置。集群管理员还可通过配置 Operator 的订阅来覆盖这些代理设置。

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 **Create Operator Subscription** 页面中，修改 Subscription 对象的 YAML，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 :

   带有代理设置的订阅对象覆盖

```
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
```

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

4. 点击 **Subscribe** 使 Operator 可供选择的命名空间使用。

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

        - 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
...
```

其他资源

- 配置集群范围代理
- 如需了解在覆盖 Operator 的代理设置时取消设置环境变量的已知问题 BZ#1751903 的详细信息，请参阅 OpenShift Container Platform 4.2 发行注记。

第 5 章 从集群中删除 OPERATOR

下面介绍如何使用 Web 控制台或 CLI 从集群中删除 Operator。

5.1. 使用 WEB 控制台从集群中删除 OPERATOR

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

先决条件

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

流程

1. 进入 **Operators → Installed Operators** 页面，在 **Filter by name** 字段滚动鼠标或键入关键词，以查找您想要的 Operator。然后点击。
2. 在 **Operator Details** 页面右侧，从 **Actions** 下拉菜单中选择 **Uninstall Operator**。
3. 在看到 **Remove Operator Subscription** 窗口提示时，可以勾选 **Also completely remove the Operator from the selected namespace** 复选框以删除与 Operator 相关的所有组件。该操作将删除 CSV，继而删除与 Operator 相关的 Pod、Deployment、CRD 和 CR。如果不选中此项，则只会删除订阅。
4. 选择 **Remove**。该 Operator 将停止运行，并不再接受更新。

5.2. 使用 CLI 从集群中删除 OPERATOR

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

先决条件

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

流程

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

```
$ oc get subscription jaeger -n openshift-operators -o yaml | grep currentCSV
  currentCSV: jaeger-operator.v1.8.2
```

2. 删除 Operator 的订阅（如 **Jaeger**）：

```
$ oc delete subscription jaeger -n openshift-operators
subscription.operators.coreos.com "jaeger" deleted
```

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

```
$ oc delete clusterserviceversion jaeger-operator.v1.8.2 -n openshift-operators
clusterserviceversion.operators.coreos.com "jaeger-operator.v1.8.2" deleted
```

第 6 章 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。

6.1. 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 4.2 集群
- 管理员已在全集群安装 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。本示例所用项目名为 **my-etcd**。
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 ClusterServiceVersions (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。

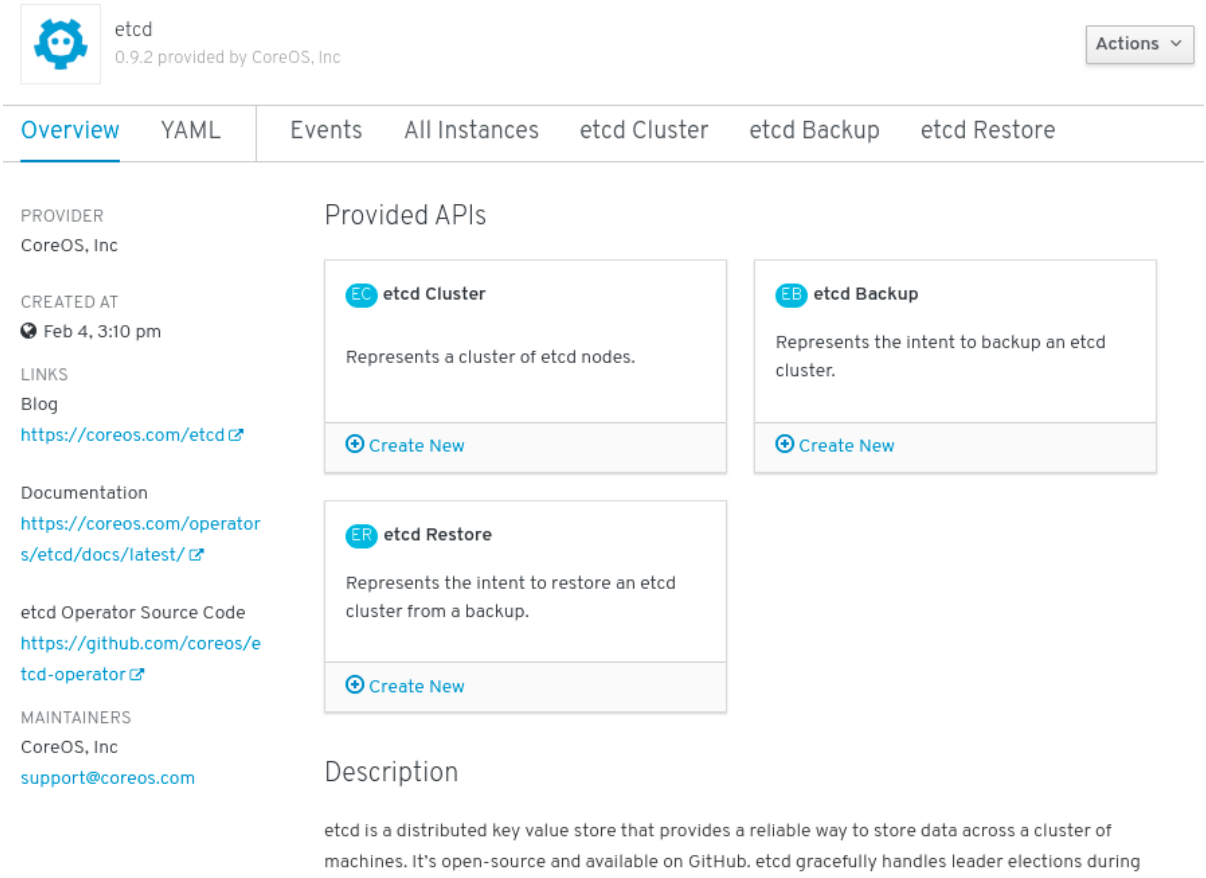
   提示

   使用以下命令从 CLI 获得该列表：

   ```
   $ oc get csv
   ```

3. 进入 **Installed Operators** 页面，点击 **Copied**，然后点击 etcd Operator 查看更多详情和可用操作：

图 6.1. etcd Operator 概述



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类似于内置的原生 Kubernetes 对象（如 **Deployments** 或 **ReplicaSets**），但包含特定于管理 etcd 的逻辑。

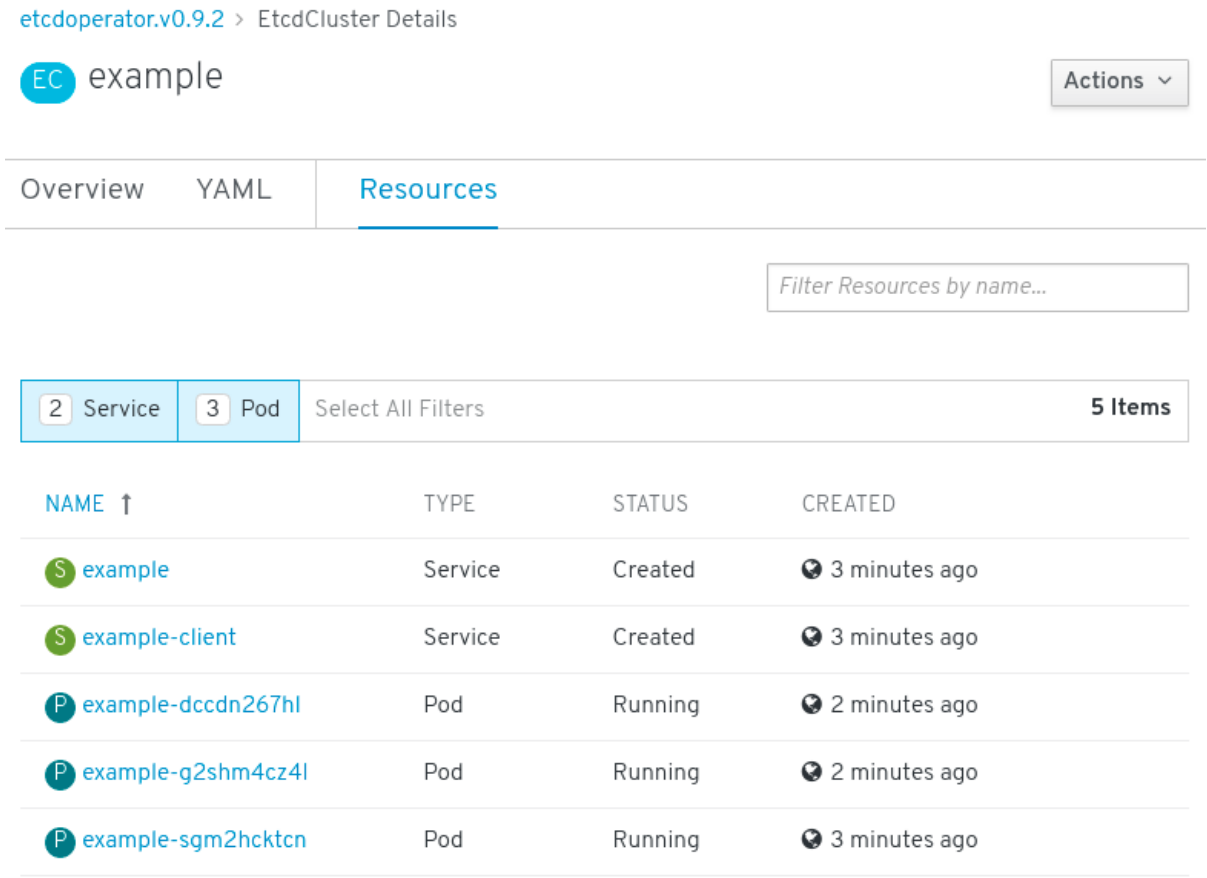
4. 新建 etcd 集群：

   a. 在 **etcd Cluster** API 方框中，点击 **Create New**。

   b. 在下一页上，您可对 **EtcdCluster** 对象的最小起始模板进行任何修改，比如集群大小。现在，点击 **Create** 即可完成。点击后即可触发 Operator 启动 Pod、Services 和新 etcd 集群的其他组件。

5. 单击 **Resources** 选项卡，可以看到您的项目现在包含很多由 Operator 自动创建和配置的资源。

图 6.2. etcd Operator 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。

第 7 章 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 会提供一些有关订阅和相关 CatalogSource 资源的见解，了解它们的状态和所执行的操作。这样有助于用户更好地理解 Operator 的运行状况。

7.1. 状况类型

订阅可报告以下状况类型：

表 7.1. 订阅状况类型

状况	描述
CatalogSourcesUnhealthy	用于解析的全部或部分 CatalogSource 运行不畅。
InstallPlanMissing	订阅的 InstallPlan 缺失。
InstallPlanPending	订阅的 InstallPlan 等待安装。
InstallPlanFailed	订阅的 InstallPlan 失败。

7.2. 使用 CLI 查看 OPERATOR 状态

可以使用 CLI 来查看 Operator 状态。

流程

1. 使用 **oc describe** 命令检查订阅资源：

```
$ oc describe sub <subscription_name>
```

2. 在命令输出中找到 **Conditions** 部分：

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

第 8 章 创建 OPERATOR 的安装与升级策略。

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 ClusterServiceVersion (CSV) 中指定任意权限集，OLM 之后会将其授予 Operator。

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

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

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

8.1. 了解 OPERATOR 安装策略

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

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

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

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

8.1.1. 安装场景

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

- 集群管理员新建了一个 OperatorGroup 并指定了服务账户。已安装与该 OperatorGroup 相关联的所有 Operator，且这些 Operator 将根据相应服务账户获得的权限运行。
- 集群管理员新建了一个 OperatorGroup 但未指定服务账户。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。
- 对于没有指定服务账户的现有 OperatorGroup，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 OperatorGroup 并指定了服务账户。OLM 支持现有 Operator 继续根据当前权限运行。现有 Operator 升级后，它会重新安装并根据相应服务账户获得的权限运行，与新 Operator 一样。
- 由 OperatorGroup 指定的服务账户因添加或删除了某些权限而有所变动，或者现有服务账户被换成了新服务账户。现有 Operator 升级后，它会重新安装并根据更新后的服务账户获得的权限运行，与新 Operator 一样。
- 集群管理员从 OperatorGroup 中删除了服务账户。默认行为保留，并允许安装和升级 Operator。

8.1.2. 安装工作流

当某个 OperatorGroup 关联至某个服务账户，且安装或升级了 Operator 后，OLM 便会使用以下工作流：

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

8.2. 限定 OPERATOR 安装范围

要为在 OLM 上安装和升级 Operator 提供范围规则，请将服务账户与 OperatorGroup 相关联。

集群管理员可借鉴本例，将一组 Operator 限制到指定命名空间中。

流程

1. 新建命名空间：

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

2. 分配 Operator 的权限范围。这一步涉及新建服务账户、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一个 OperatorGroup。该 OperatorGroup 以指定的命名空间为目标，以确保其租期仅限于相应命名空间。另外，OperatorGroup 还允许用户指定服务账户。指定上一步创建的 ServiceAccount：

```
$ 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 均会关联至该 OperatorGroup，因此也会关联至指定的服务账户。

4. 在指定命名空间中创建订阅以安装 Operator：

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

1 指定一个已存在于指定命名空间中或位于全局目录命名空间中的 CatalogSource。

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

8.2.1. 细粒度权限

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

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

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

注意

以下角色只是一个通用示例，具体 Operator 可能需要额外规则。

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

1 2 增加创建其他资源的权限，如此处显示的 Deployment 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。

8.3. 故障排除权限失败

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

流程

1. 查看订阅对象。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 创建必要的 [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
```

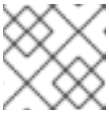
2. 检查 InstallPlan 对象的状态，了解出现的任何错误：

```
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 OperatorGroup 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

OLM 目前未提供首次尝试的完整错误列表，但可能会在后续版本中增加。

第 9 章 在受限网络中使用 OPERATOR LIFECYCLE MANAGER

如果在受限网络中安装 OpenShift Container Platform，Operator Lifecycle Manager (OLM) 将不再使用默认的 OperatorHub 源，因为这类源需要足够的网络连接。集群管理员可禁用这些默认源并创建本地镜像，以便 OLM 从本地源安装和管理 Operator。

9.1. 针对受限网络配置 OPERATORHUB

集群管理员可对 OLM 和 OperatorHub 进行配置，以便在受限网络环境中使用本地内容。

先决条件

- 集群管理员访问 OpenShift Container Platform 集群及其内部 registry。
- 无网络限制的独立工作站。
- 如果向 OpenShift Container Platform 集群的内部 registry 推送镜像，则必须通过路由公开 registry。
- **podman** 1.4.4+ 版

流程

1. **禁用默认 OperatorSource。**
   将 **disableAllDefaultSources: true** 添加至 spec：

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

   该操作将禁用在 OpenShift Container Platform 安装期间默认配置的默认 OperatorSource。

2. **获取软件包列表。**
   要获取默认 OperatorSource 可用的软件包列表，请在不受网络限制的情况下从工作站运行以下 **curl** 命令：

   ```
   $ curl https://quay.io/cnr/api/v1/packages?namespace=redhat-operators > packages.txt
   $ curl https://quay.io/cnr/api/v1/packages?namespace=community-operators >> packages.txt
   $ curl https://quay.io/cnr/api/v1/packages?namespace=certified-operators >> packages.txt
   ```

   新 **packages.txt** 中的每个软件包均为一个 Operator，您可将其添加至您的受限网络目录中。在该列表中，您可拉取每个 Operator 或您想要向用户公开的子集。

3. **拉取 Operator 内容。**
   对于软件包列表中的给定 Operator，您必须拉取最新的内容：

   ```
   $ curl https://quay.io/cnr/api/v1/packages/<namespace>/<operator_name>/<release>
   ```

   本例使用 etcd Operator：

   a. 检索摘要：

   ```
   $ curl https://quay.io/cnr/api/v1/packages/community-operators/etcd/0.0.12
   ```

b. 从该 JSON 中获取摘要并用它拉取压缩存档：

```
$ curl -XGET https://quay.io/cnr/api/v1/packages/community-
operators/etcd/blobs/sha256/8108475ee5e83a0187d6d0a729451ef1ce6d34c44a868a2001
51c36f3232822b \
    -o etcd.tar.gz
```

c. 要拉取信息，必须将存档文件与您需要的所有其他 Operator 一起解压缩至 **manifests/<operator_name>/** 目录中。例如，解压缩至名为 **manifests/etcd/** 的现有目录：

```
$ mkdir -p manifests/etcd/ (1)
$ tar -xf etcd.tar.gz -C manifests/etcd/
```

(1) 为每个解压的归档创建不同的子目录，这样文件就不会被其他 Operator 的后续解压所覆盖了。

4. **如有必要，请拆分 bundle.yaml 内容。**
在您的新 **manifests/<operator_name>** 目录中，目的是让捆绑包采用以下目录结构：

```
manifests/
└── etcd
    ├── 0.0.12
    │   ├── clusterserviceversion.yaml
    │   └── customresourcedefinition.yaml
    └── package.yaml
```

如果您的文件已采用该结构，则跳过这一步。而如果您只看到了一个名为 **bundle.yaml** 的文件，则必须先拆分该文件以确保与要求的结构一致。

您必须将 **data.clusterServiceVersion** 下的 CSV 内容（列表中的每个文件）、**data.customResourceDefinition** 下的 CRD 内容（列表中的每个文件）和 **data.Package** 下的软件包内容分开，放入各自文件中。

a. 要创建 CSV 文件，在 **bundle.yaml** 文件中找到以下行：

```
data:
  clusterServiceVersions: |
```

省略这些行，但保存由完整 CSV 资源内容组成的新文件，用以下行开头，并删除前置 **-** 字符：

clusterserviceversion.yaml 文件片断示例

```
apiVersion: operators.coreos.com/v1alpha1
kind: ClusterServiceVersion
[...]
```

b. 要创建 CRD 文件，在 **bundle.yaml** 文件中找到以下行：

```
  customResourceDefinitions: |
```

省略该行，但保存由每个完整的 CRD 资源内容组成的新文件，用以下行开头，并删除前置 **-** 字符：

customresourcedefinition.yaml 文件片断示例

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
[...]
```

c. 要创建软件包文件，在 **bundle.yaml** 文件中找到以下行：

```
  packages: |
```

省略这一行，但保存由软件包内容组成的新文件，用以下行开头，删除前置 **-** 字符，并以 **packageName** 条目结尾：

package.yaml 文件示例

```
channels:
- currentCSV: etcdoperator.v0.9.4
  name: singlenamespace-alpha
- currentCSV: etcdoperator.v0.9.4-clusterwide
  name: clusterwide-alpha
defaultChannel: singlenamespace-alpha
packageName: etcd
```

5. **识别您要使用的 Operator 所需的镜像。**
检查每个 Operator 的 CSV 文件中的 **image:** 字段，以明确 Operator 所需镜像的 pull spec，并做好记录以供后续步骤使用。

例如，在 etcd Operator CSV 的以下 **deployments** spec 中：

```
 spec:
  serviceAccountName: etcd-operator
  containers:
  - name: etcd-operator
    command:
    - etcd-operator
    - --create-crd=false
    image: quay.io/coreos/etcd-
operator@sha256:bd944a211eaf8f31da5e6d69e8541e7cada8f16a9f7a5a570b224789978199
43 (1)
    env:
    - name: MY_POD_NAMESPACE
      valueFrom:
        fieldRef:
          fieldPath: metadata.namespace
    - name: MY_POD_NAME
      valueFrom:
        fieldRef:
          fieldPath: metadata.name
```

(1) Operator 所需镜像。

6. **创建 Operator 目录镜像。**

a. 将以下内容保存到某个 Dockerfile 中，如 **custom-registry.Dockerfile**：

```
FROM registry.redhat.io/openshift4/ose-operator-registry:v4.2.24 AS builder

COPY manifests manifests

RUN /bin/initializer -o ./bundles.db

FROM registry.access.redhat.com/ubi7/ubi

COPY --from=builder /registry/bundles.db /bundles.db
COPY --from=builder /usr/bin/registry-server /registry-server
COPY --from=builder /bin/grpc_health_probe /bin/grpc_health_probe

EXPOSE 50051

ENTRYPOINT ["/registry-server"]

CMD ["--database", "bundles.db"]
```

b. 使用 **podman** 命令从 Dockerfile 中创建和标记容器镜像：

```
$ podman build -f custom-registry.Dockerfile \
    -t <local_registry_host_name>:<local_registry_host_port>/<namespace>/custom-
registry (1)
```

(1) 为受限网络 OpenShift Container Platform 集群和任何命名空间的内部 registry 标记镜像。

7. **将 Operator 目录镜像推送至 registry。**
您的新 Operator 目录镜像必须推送至受限网络 OpenShift Container Platform 集群可访问的 registry 中。该 registry 可以是集群本身的内部 registry，也可以是集群可通过网络访问的另一 registry，如内部部署的 Quay Enterprise registry。

   在本例中，需登录并推送该镜像至内部 registry OpenShift Container Platform 集群：

```
$ podman push <local_registry_host_name>:
<local_registry_host_port>/<namespace>/custom-registry
```

8. **创建指向新 Operator 目录镜像的 CatalogSource。**

   a. 将以下内容保存到文件中，如 **my-operator-catalog.yaml** 中：

```
apiVersion: operators.coreos.com/v1alpha1
kind: CatalogSource
metadata:
  name: my-operator-catalog
  namespace: openshift-marketplace
spec:
  displayName: My Operator Catalog
  sourceType: grpc
  image: <local_registry_host_name>:<local_registry_host_port>/<namespace>/custom-
registry:latest
```

b. 创建 CatalogSource 资源：

```
$ oc create -f my-operator-catalog.yaml
```

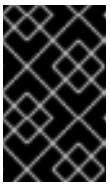
c. 验证 CatalogSource 和软件包清单是否已创建成功：

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

# oc get catalogsource -n openshift-marketplace
NAME DISPLAY TYPE PUBLISHER AGE
my-operator-catalog My Operator Catalog grpc 5s

# oc get packagemanifest -n openshift-marketplace
NAME CATALOG AGE
etcd My Operator Catalog 34s
```

此外，您还可在 Web 控制台的 **OperatorHub** 页面中查看这些内容。

9. 对**您要使用的 Operator 所需的镜像制作镜像。**

a. 确定您所期望的 Operator 定义的镜像。本例使用 etcd Operator，需要 **quay.io/coreos/etcd-operator** 镜像。



重要

该流程仅显示被镜像（mirror）的 Operator 镜像本身，而非 Operand 镜像（受 Operator 管理的组件）。Operand 镜像必须被镜像（mirror）；查看各个 Operator 的文档以识别所需的 Operand 镜像。

b. 要使用被镜像的镜像，您必须首先为每个镜像创建一个 ImageContentSourcePolicy 来更改 Operator 目录镜像的源位置。例如：

```
apiVersion: operator.openshift.io/v1alpha1
kind: ImageContentSourcePolicy
metadata:
  name: etcd-operator
spec:
  repositoryDigestMirrors:
  - mirrors:
    - <local_registry_host_name>:<local_registry_host_port>/coreos/etcd-operator
    source: quay.io/coreos/etcd-operator
```

c. 在不受网络限制的情况下，从工作站使用 **oc image mirror** 命令，将镜像从源 registry 中拉取出来，并推送至内部 registry，而不存储在本地：

```
$ oc image mirror quay.io/coreos/etcd-operator \
    <local_registry_host_name>:<local_registry_host_port>/coreos/etcd-operator
```

现在，您可在受限网络 OpenShift Container Platform 集群上从 OperatorHub 中安装 Operator。

其他资源

- 有关将 OpenShift Container Platform 集群的内部 registry 公开至集群外访问权限的详细信息请参阅公开 registry。
- 有关访问内部 registry 的详细信息，请参阅访问 registry。

第 10 章 CRD

10.1. 使用自定义资源定义来扩展 KUBERNETES API

本指南描述了集群管理员如何通过创建和管理自定义资源定义 (CRD) 来扩展其 OpenShift Container Platform 集群。

10.1.1. 自定义资源定义

在 Kubernetes API 中，资源是存储某一类 API 对象集的端点。例如：内置 Pod 资源包含 Pod 对象集。

自定义资源定义 (CRD) 对象在集群中定义了一个新的、唯一的对象 **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。这一行为会将新资源整合至集群的 RBAC 策略中，就像内置资源一样。

Operator 会通过将 CRD 与任何所需 RBAC 策略和其他软件特定逻辑打包到一起来利用 CRD。集群管理员还可手动将 CRD 添加至 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。

10.1.2. 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：

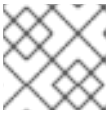
1. 先创建一个包含以下字段类型的 YAML 文件：

CRD 的 YAML 文件示例

```
apiVersion: apiextensions.k8s.io/v1beta1 (1)
kind: CustomResourceDefinition
metadata:
  name: crontabs.stable.example.com (2)
spec:
  group: stable.example.com (3)
```

```
version: v1 4
scope: Namespaced 5
names:
  plural: crontabs 6
  singular: crontab 7
  kind: CronTab 8
  shortNames:
  - ct 9
```

1. 使用 **apiextensions.k8s.io/v1beta1** API。
2. 为定义指定名称。名称必须采用 <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。最好使用组织的完全限定域名。
4. 指定 URL 中要用的版本名称。每个 API 组均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 **Kind** 基于您所创建的 CRD 对象的 **spec.kind** 字段。

10.1.3. 为自定义资源定义创建集群角色

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

重要

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

先决条件

- 创建 CRD。

流程

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

集群角色定义的 YAML 文件示例

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1 (1)
metadata:
  name: aggregate-cron-tabs-admin-edit (2)
  labels:
    rbac.authorization.k8s.io/aggregate-to-admin: "true" (3)
    rbac.authorization.k8s.io/aggregate-to-edit: "true" (4)
rules:
- apiGroups: ["stable.example.com"] (5)
  resources: ["crontabs"] (6)
  verbs: ["get", "list", "watch", "create", "update", "patch", "delete", "deletecollection"] (7)
---
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: aggregate-cron-tabs-view (8)
  labels:
    # Add these permissions to the "view" default role.
    rbac.authorization.k8s.io/aggregate-to-view: "true" (9)
    rbac.authorization.k8s.io/aggregate-to-cluster-reader: "true" (10)
rules:
- apiGroups: ["stable.example.com"] (11)
  resources: ["crontabs"] (12)
  verbs: ["get", "list", "watch"] (13)
```

(1) 使用 **rbac.authorization.k8s.io/v1** API。

(2)(8) 为定义指定名称。

(3) 指定该标签向 admin 默认角色授予权限。

(4) 指定该标签向 edit 默认角色授予权限。

5 11 指定 CRD 的组名。

6 12 指定适用于这些规则的 CRD 的复数名称。

7 13 指定代表角色所获得的权限的操作动词。例如，对 **admin** 和 **edit** 角色应用读写权限，对 **view** 角色应用只读权限。

9 指定该标签向 **view** 默认角色授予权限。

10 指定该标签向 **cluster-reader** 默认角色授予权限。

2. 创建集群角色：

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

10.1.4. 通过文件创建自定义资源

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

先决条件

- 集群管理员已将 CRD 添加至集群中。

流程

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

   CR 的 YAML 文件示例

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

   1 指定自定义资源定义中的组名和 API 版本（名称/版本）。

   2 指定 CRD 中的类型。

   3 指定对象的名称。

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

   5 指定特定于对象类型的条件。

2. 创建完文件后，再创建对象：

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

10.1.5. 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. 要获得特定 **Kind** 的 CR 的信息，请运行：

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

例如：

```
$ oc get crontab

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
```

```
$ oc get ct -o yaml

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。

10.2. 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)。

10.2.1. 自定义资源定义

在 Kubernetes API 中，资源是存储某一类 API 对象集的端点。例如：内置 Pod 资源包含 Pod 对象集。

自定义资源定义 (CRD) 对象在集群中定义了一个新的、唯一的对象 **Kind**，并允许 Kubernetes API 服务器处理其整个生命周期。

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

Operator 会通过将 CRD 与任何所需 RBAC 策略和其他软件特定逻辑打包到一起来利用 CRD。集群管理员还可手动将 CRD 添加至 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。

10.2.2. 通过文件创建自定义资源

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

先决条件

- 集群管理员已将 CRD 添加至集群中。

流程

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

   CR 的 YAML 文件示例

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

1 指定自定义资源定义中的组名和 API 版本（名称/版本）。

2 指定 CRD 中的类型。

3. 指定对象的名称。
4. 指定对象的结束程序（如有）。结束程序可让控制器实现在删除对象之前必须完成的条件。
5. 指定特定于对象类型的条件。

2. 创建完文件后，再创建对象：

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

10.2.3. 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. 要获得特定 **Kind** 的 CR 的信息，请运行：

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

例如：

```
$ oc get crontab

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
```

```
$ oc get ct -o yaml

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 11 章 OPERATOR SDK

11.1. OPERATOR SDK 入门

本指南将介绍 Operator SDK 的基础知识，指导可作为集群管理员访问基于 Kubernetes 的集群（如 OpenShift Container Platform）的 Operator 作者构建简单 Go-based Memcached Operator 的示例，并管理从安装到升级的整个生命周期。

通过以下两个 Operator Framework 核心组件便可实现这一目的：Operator SDK（**operator-sdk** CLI 工具和 **controller-runtime** 库 API）以及 Operator Lifecycle Manager (OLM)。

注意

OpenShift Container Platform 4 支持 Operator SDK v0.7.0 或更高版本。

11.1.1. Operator SDK 构架

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 是一个框架，通过提供以下内容来降低 Operator 的编写难度：

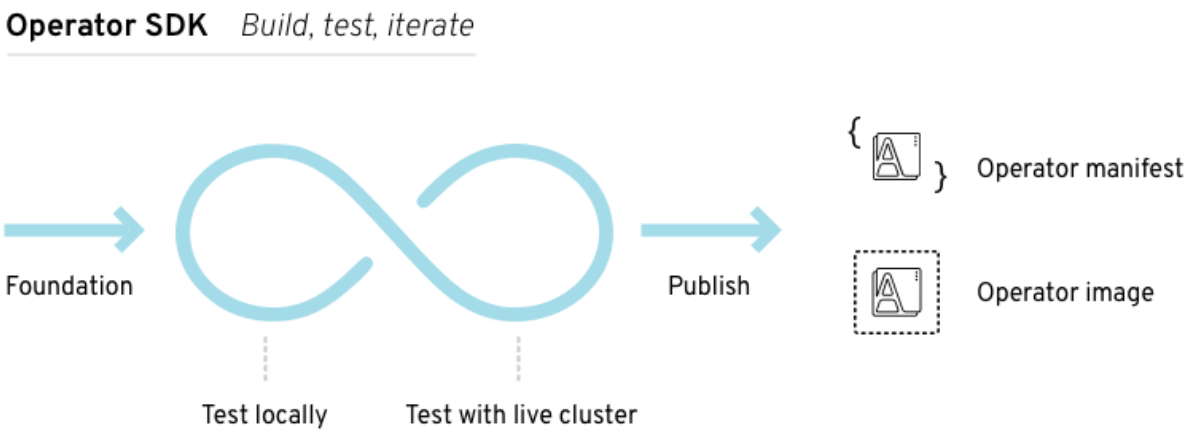
- 高级 API 和抽象，用于更直观地编写操作逻辑
- 支架和代码生成工具，用于快速引导新项目
- 扩展项，覆盖常见的 Operator 用例

11.1.1.1. 工作流

Operator SDK 提供以下工作流来开发新的 Operator：

1. 使用 Operator SDK 命令行界面 (CLI) 新建一个 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 来指定要监视的资源。
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 11.1. Operator SDK 工作流



在高级别上，使用了 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。

11.1.1.2. 管理器文件

Operator 的主要程序为 **cmd/manager/main.go** 中的管理器文件。管理器会自动注册 **pkg/apis/** 下定义的所有自定义资源 (CR) 的方案，并运行 **pkg/controller/** 下的所有控制器。

管理器可限制所有控制器监视资源的命名空间：

```
mgr, err := manager.New(cfg, manager.Options{Namespace: namespace})
```

默认情况下，这是 Operator 运行时所处的命名空间。要监视所有命名空间，把命名空间选项设为空：

```
mgr, err := manager.New(cfg, manager.Options{Namespace: ""})
```

11.1.1.3. 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。

11.1.2. 安装 Operator SDK CLI

Operator SDK 配有一个 CLI 工具，可协助开发人员创建、构建和部署新的 Operator 项目。您可在工作站上安装 SDK CLI，为编写您自己的 Operator 做准备。

注意

本指南中，minikube v0.25.0+ 用作本地 Kubernetes 集群， Quay.io 用于公共 registry。

11.1.2.1. 从 GitHub 版本安装

您可从 GitHub 上的项目下载并安装 SDK CLI 的预构建发行版文件。

先决条件

- **docker** v17.03+
- 已安装 OpenShift CLI (**oc**) v4.1+
- 访问基于 Kubernetes v1.11.3+ 的集群
- 访问容器 registry

流程

1. 设置发行版本变量：

```
RELEASE_VERSION=v0.8.0
```

2. 下载发行版二进制文件。
   - Linux：

```
$ curl -OJL https://github.com/operator-framework/operator-sdk/releases/download/${RELEASE_VERSION}/operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu
```

   - macOS：

```
$ curl -OJL https://github.com/operator-framework/operator-sdk/releases/download/${RELEASE_VERSION}/operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin
```

3. 验证所下载的发行版本二进制文件。
   a. 下载所提供的 ASC 文件。
      - Linux：

```
$ curl -OJL https://github.com/operator-framework/operator-sdk/releases/download/${RELEASE_VERSION}/operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu.asc
```

      - macOS：

```
$ curl -OJL https://github.com/operator-framework/operator-sdk/releases/download/${RELEASE_VERSION}/operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin.asc
```

   b. 将二进制文件与对应的 ASC 文件放在同一个目录中，并运行以下命令来验证该二进制文件：
      - Linux：

```
$ gpg --verify operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu.asc
```

      - macOS：

```
$ gpg --verify operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin.asc
```

如果您的工作站上没有维护人员公钥，则会出现以下错误：

```
$ gpg --verify operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin.asc
$ gpg: assuming signed data in 'operator-sdk-${RELEASE_VERSION}-x86_64-apple-
darwin'
$ gpg: Signature made Fri Apr  5 20:03:22 2019 CEST
$ gpg:                using RSA key <key_id> (1)
$ gpg: Can't check signature: No public key
```

(1) RSA 密钥字符串。

要下载密钥，请运行以下命令，用上一条命令输出的 RSA 密钥字符串来替换 **<key_id>**：

```
$ gpg [--keyserver keys.gnupg.net] --recv-key "<key_id>" (1)
```

(1) 如果您尚未配置密钥服务器，请使用 **--keyserver** 选项指定一个密钥服务器。

4. 在您的 **PATH** 中安装发行版本二进制文件：

- Linux：

```
$ chmod +x operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu
$ sudo cp operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu
/usr/local/bin/operator-sdk
$ rm operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu
```

- macOS：

```
$ chmod +x operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin
$ sudo cp operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin
/usr/local/bin/operator-sdk
$ rm operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin
```

5. 验证是否已正确安装 CLI 工具：

```
$ operator-sdk version
```

11.1.2.2. 通过 Homebrew 安装

可使用 Homebrew 来安装 SDK CLI。

先决条件

- Homebrew
- **docker** v17.03+
- 已安装 OpenShift CLI (**oc**) v4.1+

- 访问基于 Kubernetes v1.11.3+ 的集群
- 访问容器 registry

流程

1. 使用 **brew** 命令安装 SDK CLI：

```
$ brew install operator-sdk
```

2. 验证是否已正确安装 CLI 工具：

```
$ operator-sdk version
```

11.1.2.3. 通过源代码编译并安装

可获取 Operator SDK 源代码来编译和安装 SDK CLI。

先决条件

- dep v0.5.0+
- Git
- Go v1.10+
- **docker** v17.03+
- 已安装 OpenShift CLI (**oc**) v4.1+
- 访问基于 Kubernetes v1.11.3+ 的集群
- 访问容器 registry

流程

1. 克隆 **operator-sdk** 存储库：

```
$ mkdir -p $GOPATH/src/github.com/operator-framework
$ cd $GOPATH/src/github.com/operator-framework
$ git clone https://github.com/operator-framework/operator-sdk
$ cd operator-sdk
```

2. 检查所需的发行版本分支：

```
$ git checkout master
```

3. 编译并安装 SDK CLI：

```
$ make dep
$ make install
```

该操作会在 ***$GOPATH/bin*** 中安装 CLI 二进制 **operator-sdk**。

4. 验证是否已正确安装 CLI 工具：

```
$ operator-sdk version
```

11.1.3. 使用 Operator SDK 来构建 Go-based Memcached Operator

Operator SDK 可简化 Kubernetes 原生应用程序的构建，构建该应用程序原本需要深入掌握特定于应用程序的操作知识。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（如计量或监控）所需的样板代码量。

本流程介绍了一个使用 SDK 提供的工具和库构建简单 Memcached Operator 的示例。

先决条件

- 开发工作站已安装 Operator SDK CLI
- 基于 Kubernetes 的集群（v1.8 或更高版本，支持 **apps/v1beta2** API 组，如 OpenShift Container Platform 4.2）上已安装 Operator Lifecycle Manager (OLM)
- 使用具有 **cluster-admin** 权限的账户访问该集群
- 已安装 OpenShift CLI (**oc**) v4.1+

流程

1. **创建一个新项目。**
   使用 CLI 来新建 **memcached-operator** 项目：

```
$ mkdir -p $GOPATH/src/github.com/example-inc/
$ cd $GOPATH/src/github.com/example-inc/
$ operator-sdk new memcached-operator --dep-manager dep
$ cd memcached-operator
```

2. **添加新的自定义资源定义 (CRD)。**

   a. 使用 CLI 来添加名为 **Memcached** 的新 CRD API，将 **APIVersion** 设置为 **cache.example.com/v1apha1**，将 **Kind** 设置为 **Memcached**：

```
$ operator-sdk add api \
    --api-version=cache.example.com/v1alpha1 \
    --kind=Memcached
```

   这样会将 Memcached 资源 API 构建至 **pkg/apis/cache/v1alpha1/** 下。

   b. 修改 **pkg/apis/cache/v1alpha1/memcached_types.go** 文件中 **Memcached** 自定义资源 (CR) 的 spec 和状态：

```
type MemcachedSpec struct {
 // Size is the size of the memcached deployment
 Size int32 `json:"size"`
}
type MemcachedStatus struct {
```

```
    // Nodes are the names of the memcached pods
    Nodes []string `json:"nodes"`
}
```

c. 修改好 ***_types.go** 文件后，务必要运行以下命令来更新该资源类型的生成代码：

```
$ operator-sdk generate k8s
```

3. **添加新控制器。**

a. 在项目中添加新控制器以观察和协调 Memcached 资源：

```
$ operator-sdk add controller \
    --api-version=cache.example.com/v1alpha1 \
    --kind=Memcached
```

这样会在 **pkg/controller/memcached/** 下构建新控制器实现。

b. 在本示例中，将所生成的控制器文件 **pkg/controller/memcached/memcached_controller.go** 替换成示例实现。
示例控制器会对每个 **Memcached** CR 执行以下协调逻辑：

- 如果尚无 Memcached Deployment，请创建一个。
- 确保 Deployment 大小与 **Memcached** CR spec 中指定的大小相同。
- 使用 Memcached Pod 的名称来更新 **Memcached** CR 状态。

接下来的两个子步骤用于检查控制器如何监视资源以及协调循环会如何被触发。您可跳过这些步骤，直接构建和运行 Operator。

c. 检查 **pkg/controller/memcached/memcached_controller.go** 文件中的控制器实现，了解控制器如何监视资源。
首先要监视的是作为主要资源的 Memcached 类型。对于每个添加、更新或删除事件，协调循环均会针对该 Memcached 对象发送一个协调 **Request**（**<namespace>:<name>** 键）：

```
err := c.Watch(
  &source.Kind{Type: &cachev1alpha1.Memcached{}},
&handler.EnqueueRequestForObject{})
```

接下来监视 Deployment，但事件处理程序会将每个事件映射到 Deployment 所有者的协调 **Request** 中。即本例中为其创建 Deployment 的 Memcached 对象。这样可让控制器将 Deployment 视为辅助资源：

```
err := c.Watch(&source.Kind{Type: &appsv1.Deployment{}},
&handler.EnqueueRequestForOwner{
  IsController: true,
  OwnerType:    &cachev1alpha1.Memcached{},
 })
```

d. 每个控制器均有一个协调器对象，且该对象带有实现协调循环的 **Reconcile()** 方法。系统会将 **Request** 参数传递到该协调循环，该参数是一个用于从缓存中查找主资源对象 Memcached 的 **<namespace>:<name>** 键：

```
func (r *ReconcileMemcached) Reconcile(request reconcile.Request) (reconcile.Result,
```

```
error) {
  // Lookup the Memcached instance for this reconcile request
  memcached := &cachev1alpha1.Memcached{}
  err := r.client.Get(context.TODO(), request.NamespacedName, memcached)
  ...
}
```

根据 **Reconcile()** 的返回值，协调 **Request** 可能会重新排队，且可能再次触发循环：

```
// Reconcile successful - don't requeue
return reconcile.Result{}, nil
// Reconcile failed due to error - requeue
return reconcile.Result{}, err
// Requeue for any reason other than error
return reconcile.Result{Requeue: true}, nil
```

4. **构建并运行 Operator。**

   a. 在运行 Operator 之前，必须使用 Kubernetes API 服务器来注册 CRD：

   ```
   $ oc create \
       -f deploy/crds/cache_v1alpha1_memcached_crd.yaml
   ```

   b. 注册 CRD 之后，运行 Operator 时有两个选项可供选择：

   - 作为 Kubernetes 集群内的一个 Deployment 来运行
   - 作为集群外的 Go 程序运行

   选择以下任一方法。

      i. *选项 A*：作为集群内的一个 Deployment 来运行。

         A. 构建 **memcached-operator** 镜像并将其推送至 registry：

         ```
         $ operator-sdk build quay.io/example/memcached-operator:v0.0.1
         ```

         B. Deployment 清单在 **deploy/operator.yaml** 中生成。按如下方式更新 Deployment 镜像，因为默认值只是一个占位符：

         ```
         $ sed -i 's|REPLACE_IMAGE|quay.io/example/memcached-operator:v0.0.1|g'
         deploy/operator.yaml
         ```

         C. 确保 Quay.io 上有一个可供下一步使用的账户，或者替换您的首选容器 registry。在 registry 中，创建一个名为 **memcached-operator** 的新公共镜像存储库。

         D. 将镜像推送至 registry：

         ```
         $ docker push quay.io/example/memcached-operator:v0.0.1
         ```

         E. 设置 RBAC 并部署 **memcached-operator**：

         ```
         $ oc create -f deploy/role.yaml
         $ oc create -f deploy/role_binding.yaml
         $ oc create -f deploy/service_account.yaml
         ```

```
$ oc create -f deploy/operator.yaml
```

F. 验证 **memcached-operator** 是否正在运行：

```
$ oc get deployment
NAME                 DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
memcached-operator   1         1         1            1           1m
```

ii. *选项B：*在集群外本地运行。
这是开发循环中的首选方法，可加快部署和测试的速度。

使用 **$HOME/.kube/config** 中的默认 Kubernetes 配置文件在本地运行 Operator：

```
$ operator-sdk up local --namespace=default
```

您可借助标记 **--kubeconfig=<path/to/kubeconfig>** 来使用特定的 **kubeconfig**。

5. 通过创建 Memcached CR 来验证该 **Operator 可否部署 Memcached 应用程序**。

a. 创建 **deploy/crds/cache_v1alpha1_memcached_cr.yaml** 中生成的 **Memcached** CR 示例：

```
$ cat deploy/crds/cache_v1alpha1_memcached_cr.yaml
apiVersion: "cache.example.com/v1alpha1"
kind: "Memcached"
metadata:
  name: "example-memcached"
spec:
  size: 3

$ oc apply -f deploy/crds/cache_v1alpha1_memcached_cr.yaml
```

b. 确保 **memcached-operator** 为 CR 创建 Deployment：

```
$ oc get deployment
NAME                 DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
memcached-operator   1         1         1            1           2m
example-memcached    3         3         3            3           1m
```

c. 检查 Pod 和 CR 状态，以确认其状态中是否更新了 **memcached** Pod 名称：

```
$ oc get pods
NAME                                  READY     STATUS    RESTARTS   AGE
example-memcached-6fd7c98d8-7dqdr     1/1       Running   0          1m
example-memcached-6fd7c98d8-g5k7v     1/1       Running   0          1m
example-memcached-6fd7c98d8-m7vn7     1/1       Running   0          1m
memcached-operator-7cc7cfdf86-vvjqk   1/1       Running   0          2m

$ oc get memcached/example-memcached -o yaml
apiVersion: cache.example.com/v1alpha1
kind: Memcached
metadata:
  clusterName: ""
  creationTimestamp: 2018-03-31T22:51:08Z
```

```
  generation: 0
  name: example-memcached
  namespace: default
  resourceVersion: "245453"
  selfLink:
/apis/cache.example.com/v1alpha1/namespaces/default/memcacheds/example-
memcached
  uid: 0026cc97-3536-11e8-bd83-0800274106a1
spec:
  size: 3
status:
  nodes:
  - example-memcached-6fd7c98d8-7dqdr
  - example-memcached-6fd7c98d8-g5k7v
  - example-memcached-6fd7c98d8-m7vn7
```

6. 通过更新部署的大小来验证 Operator 可否管理所部署的 Memcached 应用程序。

   a. 将 **memcached** CR 中的 **spec.size** 字段从 **3** 改为 **4**：

```
$ cat deploy/crds/cache_v1alpha1_memcached_cr.yaml
apiVersion: "cache.example.com/v1alpha1"
kind: "Memcached"
metadata:
  name: "example-memcached"
spec:
  size: 4
```

   b. 应用更改：

```
$ oc apply -f deploy/crds/cache_v1alpha1_memcached_cr.yaml
```

   c. 确认 Operator 已更改 Deployment 大小：

```
$ oc get deployment
NAME                DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
example-memcached   4         4         4            4           5m
```

7. **清理资源：**

```
$ oc delete -f deploy/crds/cache_v1alpha1_memcached_cr.yaml
$ oc delete -f deploy/crds/cache_v1alpha1_memcached_crd.yaml
$ oc delete -f deploy/operator.yaml
$ oc delete -f deploy/role.yaml
$ oc delete -f deploy/role_binding.yaml
$ oc delete -f deploy/service_account.yaml
```

11.1.4. 使用 Operator Lifecycle Manager 来管理 Memcached Operator

上一节介绍了如何手动运行 Operator。下面我们将探索如何使用 Operator Lifecycle Manager (OLM)，OLM 将为生产环境中运行的 Operator 启用更强大的部署模型。

OLM 可帮助您在 Kubernetes 集群中安装、更新所有 Operator（及其相关服务）并对其整个生命周期实施一般性管理。OLM 将作为 Kubernetes 的扩展程序运行，支持您使用 **oc** 实现所有生命周期管理功能，而无需额外工具。

先决条件

- 基于 Kubernetes 的集群（v1.8 或更高版本，支持 **apps/v1beta2** API 组，如已启用预览 OLM 的 OpenShift Container Platform 4.2）上已安装 OLM
- 已构建 Memcached Operator

流程

1. **生成 Operator 清单。**
   Operator 清单描述了如何整体显示、创建和管理应用程序，即本例中的 Memcached。该清单由 **ClusterServiceVersion** (CSV) 对象定义，是运行 OLM 的必要条件。

   您可使用以下命令来生成 CSV 清单：

   ```
   $ operator-sdk olm-catalog gen-csv --csv-version 0.0.1
   ```

   注意

   该命令通过您在构建 Memcached Operator 时创建的 **memcached-operator/** 目录运行。

   在本指南中，我们将继续使用该预定义清单文件来执行后续步骤。您可更改清单中的镜像字段，以反映通过前面的步骤构建的镜像，但这不是硬性要求。

   注意

   更多有关手动定义清单文件的信息，请参阅构建用于 Operator Framework 的 CSV。

2. **部署 Operator。**

   a. 创建一个 OperatorGroup，指定 Operator 的目标命名空间。在要创建 CSV 的命名空间中创建以下 OperatorGroup。本例中使用了 **default** 命名空间：

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

   b. 将 Operator 的 CSV 清单应用于集群中指定的命名空间：

   ```
   $ curl -Lo memcachedoperator.0.0.1.csv.yaml
   https://raw.githubusercontent.com/operator-framework/getting-
   started/master/memcachedoperator.0.0.1.csv.yaml
   ```

```
$ oc apply -f memcachedoperator.0.0.1.csv.yaml
$ oc get csv memcachedoperator.v0.0.1 -n default -o json | jq '.status'
```

应用该清单时，集群不会立即更新，因为尚未满足清单中指定的要求。

c. 创建角色、角色绑定和服务账户以便向 Operator 授予资源权限，创建自定义资源定义 (CRD) 以便创建受 Operator 管理的 Memcached 类型:

```
$ oc create -f deploy/crds/cache_v1alpha1_memcached_crd.yaml
$ oc create -f deploy/service_account.yaml
$ oc create -f deploy/role.yaml
$ oc create -f deploy/role_binding.yaml
```

注意

这些文件在您构建 Memcached Operator 时由 Operator SDK 生成并存储至 **deploy/** 目录中。

因为在应用清单时，OLM 会在特定命名空间中创建 Operator，所以管理员可利用原生 Kubernetes RBAC 权限模型来限制哪些用户可以安装 Operator。

3. **创建应用程序实例。**
Memcached Operator 正在 **default** 命名空间中运行。用户通过 **CustomResource** 实例与 Operator 交互；这种情况下，资源拥有类型 **Memcached**。原生 Kubernetes RBAC 也适用于 **CustomResource**，让管理员能够控制与每个 Operator 的交互。

在该命名空间中创建 Memcached 实例将触发 Memcached Operator，以针对运行由 Operator 管理的 Memcached 服务器的 Pod 进行实例化。您创建的 **CustomResource** 越多，在此命名空间中运行的 Memcached Operator 管理的 Memcached 实例就越独特。

```
$ cat <<EOF | oc apply -f -
apiVersion: "cache.example.com/v1alpha1"
kind: "Memcached"
metadata:
  name: "memcached-for-wordpress"
spec:
  size: 1
EOF

$ cat <<EOF | oc apply -f -
apiVersion: "cache.example.com/v1alpha1"
kind: "Memcached"
metadata:
  name: "memcached-for-drupal"
spec:
  size: 1
EOF

$ oc get Memcached
NAME                    AGE
memcached-for-drupal    22s
memcached-for-wordpress   27s

$ oc get pods
NAME                                  READY     STATUS    RESTARTS   AGE
```

```
memcached-app-operator-66b5777b79-pnsfj    1/1     Running   0          14m
memcached-for-drupal-5476487c46-qbd66      1/1     Running   0          3s
memcached-for-wordpress-65b75fd8c9-7b9x7   1/1     Running   0          8s
```

4. **更新应用程序。**
通过引用旧 Operator 清单的 **replaces** 字段来创建新 Operator 清单，以手动更新 Operator。OLM 可确保由旧 Operator 管理的所有资源的所有权均转移到至新 Operator，而不用担心任何程序停止执行。升级资源以便在新版 Operator 下运行所需的所有数据迁移均将由 Operator 直接执行。

   以下命令演示了如何使用新版 Operator 来应用新的 Operator 清单文件，并显示了 Pod 仍在执行：

```
$ curl -Lo memcachedoperator.0.0.2.csv.yaml https://raw.githubusercontent.com/operator-framework/getting-started/master/memcachedoperator.0.0.2.csv.yaml
$ oc apply -f memcachedoperator.0.0.2.csv.yaml
$ oc get pods
NAME                                       READY     STATUS    RESTARTS   AGE
memcached-app-operator-66b5777b79-pnsfj    1/1       Running   0          3s
memcached-for-drupal-5476487c46-qbd66      1/1       Running   0          14m
memcached-for-wordpress-65b75fd8c9-7b9x7   1/1       Running   0          14m
```

11.1.5. 其他资源

- 有关 Operator SDK 所创建的项目目录结构的信息，请参阅附录。
- 面向红帽合作伙伴的 Operator 开发指南

11.2. 创建基于 ANSIBLE 的 OPERATOR

本指南概述了 Operator SDK 中的 Ansible 支持，通过一些示例指导 Operator 作者通过使用 Ansible playbook 和模块的 **operator-sdk** CLI 工具来构建和运行基于 Ansible 的 Operator。

11.2.1. Operator SDK 中的 Ansible 支持

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*。该框架中包含 Operator SDK，可协助开发人员利用自己的专业知识来引导和构建 Operator，而无需了解 Kubernetes API 复杂性。

通过 Operator SDK 生成 Operator 项目的其中一种方案是利用现有 Ansible playbook 和模块来部署 Kubernetes 资源作为统一应用程序，而无需编写任何 Go 代码。

11.2.1.1. 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 11.1. 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 **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 11.2. 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: "foo.example.com/v1alpha1"
kind: "Foo"
metadata:
  name: "example"
annotations:
  ansible.operator-sdk/reconcile-period: "30s"
```

11.2.1.2. Watches 文件

Watches 文件中包含从自定义资源 (CR)（通过 **Group**、**Version** 和 **Kind** 标识）到 Ansible 角色或 playbook 的映射列表。Operator 期望该映射文件位于预定义位置，**/opt/ansible/watches.yaml**。

表 11.3. Watches 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

字段	描述
role（默认）	添加至容器中的 Ansible 角色的路径。例如：如果您的 **roles** 目录位于 **/opt/ansible/roles/** 中，角色名为 **busybox**，则该值应为 **/opt/ansible/roles/busybox**。该字段与 **playbook** 字段相互排斥。
playbook	添加至容器中的 Ansible playbook 的路径。该 playbook 需要以简单方式调用角色。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 文件示例

```
- version: v1alpha1 (1)
  group: foo.example.com
  kind: Foo
  role: /opt/ansible/roles/Foo

- version: v1alpha1 (2)
  group: bar.example.com
  kind: Bar
  playbook: /opt/ansible/playbook.yml

- version: v1alpha1 (3)
  group: baz.example.com
  kind: Baz
  playbook: /opt/ansible/baz.yml
  reconcilePeriod: 0
  manageStatus: false
```

1. 将 **Foo** 映射至 **Foo** 角色的简单示例。
2. 将 **Bar** 映射至 playbook 的简单示例。
3. 针对 **Baz** 类型的更复杂示例。在 playbook 中禁止对 CR 状态重新排队和管理。

11.2.1.2.1. 高级选项

高级功能可通过添加至每个 GVK（group、version 和 kind）的 Watches 文件中进行启用。它们可放在 **group**、**version**、**kind** 和 **playbook** 或 **role** 字段下方。

可使用自定义资源 (CR) 上的注解覆盖每个资源的某些功能。可覆盖的选项会指定以下注解。

表 11.4. 高级 Watches 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。	**ansbile.operator-sdk/reconcile-period**	**1m**
管理状态	**manageStatus**	支持 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 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
```

11.2.1.3. 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: {{ meta.name }}, {{ meta.namespace }}"
```

11.2.1.4. Ansible Runner 目录

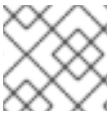
Ansible Runner 会将与 Ansible 运行相关的信息保存至容器中。具体位于：**/tmp/ansible-operator/runner/<group>/<version>/<kind>/<namespace>/<name>**。

其他资源

- 要了解有关 **runner** 目录的更多信息，请参阅 Ansible Runner 文档。

11.2.2. 安装 Operator SDK CLI

Operator SDK 配有一个 CLI 工具，可协助开发人员创建、构建和部署新的 Operator 项目。您可在工作站上安装 SDK CLI，为编写您自己的 Operator 做准备。



注意

本指南中，minikube v0.25.0+ 用作本地 Kubernetes 集群， Quay.io 用于公共 registry。

11.2.2.1. 从 GitHub 版本安装

您可从 GitHub 上的项目下载并安装 SDK CLI 的预构建发行版文件。

先决条件

- **docker** v17.03+
- 已安装 OpenShift CLI (**oc**) v4.1+
- 访问基于 Kubernetes v1.11.3+ 的集群

- 访问容器 registry

流程

1. 设置发行版本变量：

```
RELEASE_VERSION=v0.8.0
```

2. 下载发行版二进制文件。
   - Linux：

```
$ curl -OJL https://github.com/operator-framework/operator-sdk/releases/download/${RELEASE_VERSION}/operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu
```

   - macOS：

```
$ curl -OJL https://github.com/operator-framework/operator-sdk/releases/download/${RELEASE_VERSION}/operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin
```

3. 验证所下载的发行版本二进制文件。
   a. 下载所提供的 ASC 文件。
      - Linux：

```
$ curl -OJL https://github.com/operator-framework/operator-sdk/releases/download/${RELEASE_VERSION}/operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu.asc
```

      - macOS：

```
$ curl -OJL https://github.com/operator-framework/operator-sdk/releases/download/${RELEASE_VERSION}/operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin.asc
```

   b. 将二进制文件与对应的 ASC 文件放在同一个目录中，并运行以下命令来验证该二进制文件：
      - Linux：

```
$ gpg --verify operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu.asc
```

      - macOS：

```
$ gpg --verify operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin.asc
```

      如果您的工作站上没有维护人员公钥，则会出现以下错误：

```
$ gpg --verify operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin.asc
$ gpg: assuming signed data in 'operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin'
```

```
$ gpg: Signature made Fri Apr  5 20:03:22 2019 CEST
$ gpg:                using RSA key <key_id> (1)
$ gpg: Can't check signature: No public key
```

(1) RSA 密钥字符串。

要下载密钥，请运行以下命令，用上一条命令输出的 RSA 密钥字符串来替换 **<key_id>**：

```
$ gpg [--keyserver keys.gnupg.net] --recv-key "<key_id>" (1)
```

(1) 如果您尚未配置密钥服务器，请使用 **--keyserver** 选项指定一个密钥服务器。

4. 在您的 **PATH** 中安装发行版本二进制文件：

- Linux：

```
$ chmod +x operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu
$ sudo cp operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu
/usr/local/bin/operator-sdk
$ rm operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu
```

- macOS：

```
$ chmod +x operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin
$ sudo cp operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin
/usr/local/bin/operator-sdk
$ rm operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin
```

5. 验证是否已正确安装 CLI 工具：

```
$ operator-sdk version
```

11.2.2.2. 通过 Homebrew 安装

可使用 Homebrew 来安装 SDK CLI。

先决条件

- Homebrew
- **docker** v17.03+
- 已安装 OpenShift CLI (**oc**) v4.1+
- 访问基于 Kubernetes v1.11.3+ 的集群
- 访问容器 registry

流程

1. 使用 **brew** 命令安装 SDK CLI：

```
$ brew install operator-sdk
```

2. 验证是否已正确安装 CLI 工具：

```
$ operator-sdk version
```

11.2.2.3. 通过源代码编译并安装

可获取 Operator SDK 源代码来编译和安装 SDK CLI。

先决条件

- dep v0.5.0+
- Git
- Go v1.10+
- **docker** v17.03+
- 已安装 OpenShift CLI (**oc**) v4.1+
- 访问基于 Kubernetes v1.11.3+ 的集群
- 访问容器 registry

流程

1. 克隆 **operator-sdk** 存储库：

```
$ mkdir -p $GOPATH/src/github.com/operator-framework
$ cd $GOPATH/src/github.com/operator-framework
$ git clone https://github.com/operator-framework/operator-sdk
$ cd operator-sdk
```

2. 检查所需的发行版本分支：

```
$ git checkout master
```

3. 编译并安装 SDK CLI：

```
$ make dep
$ make install
```

该操作会在 ***$GOPATH/bin*** 中安装 CLI 二进制 **operator-sdk**。

4. 验证是否已正确安装 CLI 工具：

```
$ operator-sdk version
```

11.2.3. 使用 Operator SDK 来构建基于 Ansible 的 Operator

本流程介绍了使用 Operator SDK 中的工具和库来构建由 Ansible playbook 和模块提供技术支持的简单 Memcached Operator 示例。

先决条件

- 开发工作站已安装 Operator SDK CLI
- 使用具有 **cluster-admin** 权限的账户访问基于 Kubernetes 的集群 v1.11.3+（如 OpenShift Container Platform 4.2）
- 已安装 OpenShift CLI (**oc**) v4.1+
- **ansible** v2.6.0+
- **ansible-runner** v1.1.0+
- **ansible-runner-http** v1.0.0+

流程

1. 使用 **operator-sdk new** 命令来创建新 Operator 项目，可以是全命名空间范围，也可以是全集群范围。选择以下任意一项：

   a. *全命名空间范围的 Operator*（默认）会监视和管理单一命名空间中的资源。全命名空间范围的 Operator 因灵活性高而常被视为首选。这类 Operator 支持解耦升级、针对故障和监控隔离命名空间以及区别化 API 定义。
   要创建基于 Ansible 的、全命名空间范围的新 **memcached-operator** 项目并改为其目录，请使用以下命令：

   ```
   $ operator-sdk new memcached-operator \
     --api-version=cache.example.com/v1alpha1 \
     --kind=Memcached \
     --type=ansible
   $ cd memcached-operator
   ```

   这样会创建一个 **memcached-operator** 项目，专门用于监视 APIVersion 为 **example.com/v1apha1**、Kind 为 **Memcached** 的 Memcached 资源。

   b. *全集群范围的 Operator* 会监视并管理整个集群的资源，在某些情况下非常实用。例如，**cert-manager** Operator 通常会以全集群范围权限和监视功能进行部署，以便管理整个集群的证书颁发。
   要创建全集群范围的 **memcached-operator** 项目并改为其目录，请使用以下命令：

   ```
   $ operator-sdk new memcached-operator \
     --cluster-scoped \
     --api-version=cache.example.com/v1alpha1 \
     --kind=Memcached \
     --type=ansible
   $ cd memcached-operator
   ```

   使用 **--cluster-scoped** 标记来构建进行了以下修改的新 Operator：

   - **deploy/operator.yaml**：设置 **WATCH_NAMESPACE=""** 而不要将其设置为 Pod 的命名空间。
   - **deploy/role.yaml**：使用 **ClusterRole** 来代替 **Role**。

- **deploy/role_binding.yaml**：
  - 使用 **ClusterRoleBinding** 来代替 **rolebinding**。
  - 将主题命名空间设置为 **REPLACE_NAMESPACE**。该命名空间必须改为部署 Operator 的命名空间。

2. **自定义 Operator 逻辑。**
在本例中，**memcached-operator** 会针对每个 **Memcached** 自定义资源 (CR) 执行以下协调逻辑：

   - 如果尚无 **memcached** Deployment，请创建一个。
   - 确保 Deployment 大小与 **Memcached** CR 指定的大小相同。

   默认情况下，**memcached-operator** 会监视 **Memcached** 资源事件，如 **watches.yaml** 文件中所示，并执行 Ansible 角色 **Memcached**：

   ```
   - version: v1alpha1
     group: cache.example.com
     kind: Memcached
   ```

   您可选择在 **watches.yaml** 文件中自定义以下逻辑：

   a. 指定 **role** 选项会将 Operator 配置为在通过 Ansible 角色启动 **ansible-runner** 时使用该指定路径。默认情况下，新命令会填写一个绝对路径，指向您的角色应前往的位置：

   ```
   - version: v1alpha1
     group: cache.example.com
     kind: Memcached
     role: /opt/ansible/roles/memcached
   ```

   b. 指定 **watches.yaml** 文件中的 **playbook** 选项会将 Operator 配置为在通过 Ansible playbook 启动 **ansible-runner** 时使用该指定路径。

   ```
   - version: v1alpha1
     group: cache.example.com
     kind: Memcached
     playbook: /opt/ansible/playbook.yaml
   ```

3. **构建 Memcached Ansible 角色。**
修改 **roles/memcached/** 目录下生成的 Ansible 角色。该 Ansible 角色会控制修改资源时所执行的逻辑。

   a. **定义 Memcached spec。**
   可完全在 Ansible 中定义基于 Ansible 的 Operator spec。Ansible Operator 会将 CR spec 字段中列出的所有键值对作为变量传递给 Ansible。在运行 Ansible 之前，Operator 会将 spec 字段中所有变量的名称转换为 snake 格式（小写并附带下划线）。例如，spec 中的 **serviceAccount** 在 Ansible 中会变成 **service_account**。

   提示

   您应在 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。

如果用户未设置 **spec** 字段，则请修改 **roles/memcached/defaults/main.yml** 文件设置默认值：

```
size: 1
```

b. **定义 Memcached Deployment。**
定义了 **Memcached** spec 后，即可定义资源发生更改时实际应执行的 Ansible。因为这是一个 Ansible 角色，所以默认行为是执行 **roles/memcached/tasks/main.yml** 文件中的任务。

目的是在不存在 Deployment 的情况下让 Ansible 创建 Deployment，该 Deployment 将运行 **memcached:1.4.36-alpine** 镜像。Ansible 2.7+ 支持 k8s Ansible 模块，本示例利用该模块来控制 Deployment 定义。

修改 **roles/memcached/tasks/main.yml** 以匹配以下内容：

```
- name: start memcached
  k8s:
    definition:
      kind: Deployment
      apiVersion: apps/v1
      metadata:
        name: '{{ meta.name }}-memcached'
        namespace: '{{ 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
```

注意

本示例使用 **size** 变量来控制 **Memcached** Deployment 的副本数。本示例将默认值设定为 **1**，但任何用户都可以创建一个 CR 覆盖该默认值。

4. **部署 CRD。**
在运行 Operator 之前，Kubernetes 需要了解 Operator 将会监视的新自定义资源定义 (CRD)。部署 **Memcached** CRD：

```
$ oc create -f deploy/crds/cache_v1alpha1_memcached_crd.yaml
```

5. **构建并运行 Operator。**
有两种方式来构建和运行 Operator：

- 作为 Kubernetes 集群内的一个 Pod。
- 作为集群外的 Go 程序，使用 **operator-sdk up** 命令。

选择以下任一方法：

a. 作为 Kubernetes 集群内的**一个 Pod 来运行**。这是生产环境的首先方法。

i. 构建 **memcached-operator** 镜像并将其推送至 registry：

```
$ operator-sdk build quay.io/example/memcached-operator:v0.0.1
$ podman push quay.io/example/memcached-operator:v0.0.1
```

ii. **deploy/operator.yaml** 文件中会生成 Deployment 清单。本文件中的部署镜像需要从占位符 **REPLACE_IMAGE** 修改为之前构建的镜像。为此，请运行：

```
$ sed -i 's|REPLACE_IMAGE|quay.io/example/memcached-operator:v0.0.1|g'
deploy/operator.yaml
```

iii. 如果使用 **--cluster-scoped=true** 标记来创建 Operator，请更新所生成的 **ClusterRoleBinding** 中的服务账户命名空间，以匹配 Operator 的部署位置：

```
$ export OPERATOR_NAMESPACE=$(oc config view --minify -o
jsonpath='{.contexts[0].context.namespace}')
$ sed -i "s|REPLACE_NAMESPACE|$OPERATOR_NAMESPACE|g"
deploy/role_binding.yaml
```

如果要在 OSX 中执行这些步骤，请使用以下命令：

```
$ sed -i "" 's|REPLACE_IMAGE|quay.io/example/memcached-operator:v0.0.1|g'
deploy/operator.yaml
$ sed -i "" "s|REPLACE_NAMESPACE|$OPERATOR_NAMESPACE|g"
deploy/role_binding.yaml
```

iv. 部署 **memcached-operator**：

```
$ oc create -f deploy/service_account.yaml
$ oc create -f deploy/role.yaml
$ oc create -f deploy/role_binding.yaml
$ oc create -f deploy/operator.yaml
```

v. 验证 **memcached-operator** 是否正在运行：

```
$ oc get deployment
NAME                 DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
memcached-operator   1         1         1            1           1m
```

b. **在集群外运行。**这是开发阶段的首选方法，可加快部署和测试的速度。

请确保已安装 Ansible Runner 和 Ansible Runner HTTP Plug-in，否则在创建 CR 时，系统会显示 Ansible Runner 出现意外错误。

此外，还务必要确保您的计算机上存在 **watches.yaml** 文件中引用的角色路径。因为通常会在磁盘上角色所处的位置使用容器，所以必须手动将角色复制到已配置的 Ansible 角色路径中（如 **/etc/ansible/roles**）。

i. 要使用 **$HOME/.kube/config** 中的默认 Kubernetes 配置文件在本地运行 Operator：

```
$ operator-sdk up local
```

要使用所提供的 Kubernetes 配置文件在本地运行 Operator：

```
$ operator-sdk up local --kubeconfig=config
```

6. **创建一个 Memcached CR。**

a. 按所示方式修改 **deploy/crds/cache_v1alpha1_memcached_cr.yaml** 文件并创建一个 **Memcached** CR：

```
$ cat deploy/crds/cache_v1alpha1_memcached_cr.yaml
apiVersion: "cache.example.com/v1alpha1"
kind: "Memcached"
metadata:
  name: "example-memcached"
spec:
  size: 3

$ oc apply -f deploy/crds/cache_v1alpha1_memcached_cr.yaml
```

b. 确保 **memcached-operator** 为 CR 创建 Deployment：

```
$ oc get deployment
NAME                 DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
memcached-operator       1         1         1            1          2m
example-memcached        3         3         3            3          1m
```

c. 检查 Pod，确认已创建三个副本：

```
$ oc get pods
NAME                                  READY     STATUS    RESTARTS   AGE
example-memcached-6fd7c98d8-7dqdr     1/1       Running   0          1m
example-memcached-6fd7c98d8-g5k7v     1/1       Running   0          1m
example-memcached-6fd7c98d8-m7vn7     1/1       Running   0          1m
memcached-operator-7cc7cfdf86-vvjqk   1/1       Running   0          2m
```

7. **更新大小。**

a. 将 **memcached** CR 中的 **spec.size** 字段从 **3** 改为 **4**，并应用以下更改：

```
$ cat deploy/crds/cache_v1alpha1_memcached_cr.yaml
apiVersion: "cache.example.com/v1alpha1"
kind: "Memcached"
metadata:
```

```
  name: "example-memcached"
spec:
  size: 4

$ oc apply -f deploy/crds/cache_v1alpha1_memcached_cr.yaml
```

b. 确认 Operator 已更改 Deployment 大小：

```
$ oc get deployment
NAME                DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
example-memcached   4         4         4            4           5m
```

8. **清理资源：**

```
$ oc delete -f deploy/crds/cache_v1alpha1_memcached_cr.yaml
$ oc delete -f deploy/operator.yaml
$ oc delete -f deploy/role_binding.yaml
$ oc delete -f deploy/role.yaml
$ oc delete -f deploy/service_account.yaml
$ oc delete -f deploy/crds/cache_v1alpha1_memcached_crd.yaml
```

11.2.4. 使用 k8s Ansible 模块来管理应用程序生命周期

要使用 Ansible 管理 Kubernetes 上的应用程序生命周期，您可使用 **k8s** Ansible 模块。该 Ansible 模块支持开发人员利用其现有 Kubernetes 资源文件（用 YAML 编写），或用原生 Ansible 来表达生命周期管理。

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。

本部分将详细介绍 **k8s** Ansible 模块的用途。开始之前，请先在本地工作站安装该模块，然后使用 playbook 进行测试，最后移至 Operator 中继续使用。

11.2.4.1. 安装 k8s Ansible 模块

要在本地工作站中安装 **k8s** Ansible 模块：

流程

1. 安装 Ansible 2.6+：

```
$ sudo yum install ansible
```

2. 使用 **pip** 来安装 OpenShift python 客户端软件包：

```
$ pip install openshift
```

11.2.4.2. 在本地测试 k8s Ansible 模块

有时，开发人员最好在其本地机器上运行 Ansible 代码，而不必每次都运行和重构 Operator。

流程

1. 初始化基于 Ansible 的新 Operator 项目：

```
$ operator-sdk new --type ansible --kind Foo --api-version foo.example.com/v1alpha1 foo-operator
Create foo-operator/tmp/init/galaxy-init.sh
Create foo-operator/tmp/build/Dockerfile
Create foo-operator/tmp/build/test-framework/Dockerfile
Create foo-operator/tmp/build/go-test.sh
Rendering Ansible Galaxy role [foo-operator/roles/Foo]...
Cleaning up foo-operator/tmp/init
Create foo-operator/watches.yaml
Create foo-operator/deploy/rbac.yaml
Create foo-operator/deploy/crd.yaml
Create foo-operator/deploy/cr.yaml
Create foo-operator/deploy/operator.yaml
Run git init ...
Initialized empty Git repository in /home/dymurray/go/src/github.com/dymurray/opsdk/foo-operator/.git/
Run git init done
```

```
$ cd foo-operator
```

2. 使用所需 Ansible 逻辑来修改 **roles/Foo/tasks/main.yml** 文件。本示例通过变量切换来创建和删除命名空间。

```
- name: set test namespace to {{ state }}
  k8s:
    api_version: v1
    kind: Namespace
    state: "{{ state }}"
  ignore_errors: true (1)
```

(1) 设置 **ignore_errors: true** 可确保删除不存在的项目不会失败。

3. 修改 **roles/Foo/defaults/main.yml** 文件，将默认 **state** 设置为 **present**。

```
state: present
```

4. 在顶层目录中创建一个 Ansible playbook **playbook.yml** 文件，其中包含 **Foo** 角色：

```
- hosts: localhost
  roles:
    - Foo
```

5. 运行 playbook：

```
$ ansible-playbook playbook.yml
 [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 [Foo : set test namespace to present]
changed: [localhost]

PLAY RECAP *********************************************************************************
localhost                  : ok=2    changed=1    unreachable=0    failed=0
```

6. 检查是否已创建命名空间：

```
$ oc get namespace
NAME          STATUS    AGE
default       Active    28d
kube-public   Active    28d
kube-system   Active    28d
test          Active    3s
```

7. 重新运行 playbook，设置 **state** 为 **absent**：

```
$ ansible-playbook playbook.yml --extra-vars state=absent
 [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 [Foo : set test namespace to absent]
changed: [localhost]

PLAY RECAP *********************************************************************************
localhost                  : ok=2    changed=1    unreachable=0    failed=0
```

8. 检查是否已删除命名空间：

```
$ oc get namespace
NAME          STATUS    AGE
default       Active    28d
kube-public   Active    28d
kube-system   Active    28d
```

11.2.4.3. 在 Operator 内测试 k8s Ansible 模块

熟悉在本地使用 **k8s** Ansible 模块后，您可在 Operator 内触发自定义资源 (CR) 发生变化时的相同 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 文件中完成。

11.2.4.3.1. 本地测试基于 Ansible 的 Operator

熟悉在本地测试 Ansible 工作流后，您可以在本地运行的基于 Ansible 的 Operator 内测试逻辑。

为此，请使用 Operator 项目顶层目录中的 **operator-sdk up local** 命令。该命令读取 **./watches.yaml** 文件，并使用 **~/.kube/config** 文件与 Kubernetes 集群通信，就像 **k8s** Ansible 一样。

流程

1. 因为 **up local** 命令是从 **./watches.yaml** 文件中读取的，所以有一些选项可供 Operator 作者使用。如果 **role** 独立存在（默认为 **/opt/ansible/roles/<name>**），则必须直接从 Operator 中复制该角色至 **/opt/ansible/roles/** 目录。
这很麻烦，因为更改不会反映在当前目录中。相反，可更改 **role** 字段以指向当前目录并注释掉现有行:

```
- version: v1alpha1
  group: foo.example.com
  kind: Foo
  #  role: /opt/ansible/roles/Foo
  role: /home/user/foo-operator/Foo
```

2. 为自定义资源 (CR) **Foo** 创建自定义资源定义 (CRD) 和适当的基于角色的访问控制 (RBAC) 定义。**operator-sdk** 命令会在 **deploy/** 目录中自动生成这些文件：

```
$ oc create -f deploy/crds/foo_v1alpha1_foo_crd.yaml
$ oc create -f deploy/service_account.yaml
$ oc create -f deploy/role.yaml
$ oc create -f deploy/role_binding.yaml
```

3. 运行 **up local** 命令：

```
$ operator-sdk up local
[...]
INFO[0000] Starting to serve on 127.0.0.1:8888
INFO[0000] Watching foo.example.com/v1alpha1, Foo, default
```

4. 现在 Operator 正在监视资源 **Foo** 中的事件，创建 CR 会触发您的 Ansible 角色运行。查看 **deploy/cr.yaml** 文件：

```
apiVersion: "foo.example.com/v1alpha1"
kind: "Foo"
metadata:
  name: "example"
```

因为未设置 **spec** 字段，所以调用 Ansible 时无额外变量。下一部分将介绍额外变量如何从 CR 传递至 Ansible。这也是务必要为 Operator 设置合理默认值的原因。

5. 创建 **Foo** 的 CR 实例，并将默认变量 **state** 设置为 **present**：

```
$ oc create -f deploy/cr.yaml
```

6. 检查是否已创建命名空间 **test**：

```
$ oc get namespace
NAME          STATUS    AGE
default       Active    28d
kube-public   Active    28d
kube-system   Active    28d
test          Active    3s
```

7. 修改 **deploy/cr.yaml** 文件，将 **state** 字段设置为 **absent**：

```
apiVersion: "foo.example.com/v1alpha1"
kind: "Foo"
metadata:
  name: "example"
spec:
  state: "absent"
```

8. 应用这些更改，并确认已删除命名空间：

```
$ oc apply -f deploy/cr.yaml

$ oc get namespace
NAME          STATUS    AGE
default       Active    28d
kube-public   Active    28d
kube-system   Active    28d
```

11.2.4.3.2. 在集群上测试基于 Ansible 的 Operator

熟悉了在基于 Ansible 的 Operator 内部本地运行 Ansible 逻辑之后，您可在 Kubernetes 集群（如 OpenShift Container Platform）上的 Pod 内部测试 Operator。作为 Pod 在集群中运行是生产环境的首选方法。

流程

1. 构建 **foo-operator** 镜像并将其推送至 registry：

```
$ operator-sdk build quay.io/example/foo-operator:v0.0.1
$ podman push quay.io/example/foo-operator:v0.0.1
```

2. **deploy/operator.yaml** 文件中会生成 Deployment 清单。本文件中的 Deployment 镜像必须从占位符 **REPLACE_IMAGE** 修改为之前构建的镜像。为此，请运行以下命令：

```
$ sed -i 's|REPLACE_IMAGE|quay.io/example/foo-operator:v0.0.1|g' deploy/operator.yaml
```

如果要在 OSX 中执行这些步骤，使用以下命令：

```
$ sed -i "" 's|REPLACE_IMAGE|quay.io/example/foo-operator:v0.0.1|g' deploy/operator.yaml
```

3. 部署 **foo-operator**：

```
$ oc create -f deploy/crds/foo_v1alpha1_foo_crd.yaml # if CRD doesn't exist already
$ oc create -f deploy/service_account.yaml
$ oc create -f deploy/role.yaml
$ oc create -f deploy/role_binding.yaml
$ oc create -f deploy/operator.yaml
```

4. 验证 **foo-operator** 是否正在运行：

```
$ oc get deployment
NAME                 DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
foo-operator         1         1         1            1           1m
```

11.2.5. 使用 k8s_status Ansible 模块来管理自定义资源状态

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。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。

流程

1. 要通过应用程序手动跟踪 CR 状态，请将 **manageStatus** 字段设置为 **false** 来更新 Watches 文件：

```
- version: v1
  group: api.example.com
  kind: Foo
  role: /opt/ansible/roles/Foo
  manageStatus: false
```

2. 然后，使用 **k8s_status** Ansible 模块来更新子资源。例如：要使用键 **foo** 和值 **bar** 来更新，可使用 **k8s_status**，如下所示：

```
- k8s_status:
  api_version: app.example.com/v1
  kind: Foo
  name: "{{ meta.name }}"
  namespace: "{{ meta.namespace }}"
  status:
    foo: bar
```

其他资源

- 有关基于 Ansible 的 Operator 中用户驱动状态管理的更多详情，请参阅 Ansible Operator 状态建议。

11.2.5.1. 本地测试时使用 k8s_status Ansible 模块

如果 Operator 使用 **k8s_status** Ansible 模块，而您希望使用 **operator-sdk up local** 命令在本地测试 Operator，则必须将模块安装至 Ansible 期望的位置。该操作可通过 Ansible 的 **library** 配置选项完成。

在本示例中，假设用户将第三方 Ansible 模块放至 **/usr/share/ansible/library/** 目录中。

流程

1. 要安装 **k8s_status** 模块，请将 **ansible.cfg** 文件设置为在 **/usr/share/ansible/library/** 目录中搜索已安装的 Ansible 模块：

```
$ echo "library=/usr/share/ansible/library/" >> /etc/ansible/ansible.cfg
```

2. 将 **k8s_status.py** 文件添加到 **/usr/share/ansible/library/** 目录中：

```
$ wget https://raw.githubusercontent.com/openshift/ocp-release-operator-
sdk/master/library/k8s_status.py -O /usr/share/ansible/library/k8s_status.py
```

11.2.6. 其他资源

- 有关 Operator SDK 所创建的项目目录结构的信息，请参阅附录。
- Reaching for the Stars with Ansible Operator - 红帽 OpenShift 博客
- 面向红帽合作伙伴的 Operator 开发指南

11.3. 创建基于 HELM 的 OPERATOR

本指南概述了 Operator SDK 中的 Helm Chart 支持，借助示例引导 Operator 作者通过使用现有 Helm Chart 的 **operator-sdk** CLI 工具来构建和运行 Nginx Operator。

11.3.1. Operator SDK 中的 Helm Chart 支持

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*。该框架中包含 Operator SDK，可协助开发人员利用自己的专业知识来引导和构建 Operator，而无需了解 Kubernetes API 复杂性。

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 的 **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**）会通过以下命令传播到 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。

11.3.2. 安装 Operator SDK CLI

Operator SDK 配有一个 CLI 工具，可协助开发人员创建、构建和部署新的 Operator 项目。您可在工作站上安装 SDK CLI，为编写您自己的 Operator 做准备。

注意

本指南中，minikube v0.25.0+ 用作本地 Kubernetes 集群， Quay.io 用于公共 registry。

11.3.2.1. 从 GitHub 版本安装

您可从 GitHub 上的项目下载并安装 SDK CLI 的预构建发行版文件。

先决条件

- **docker** v17.03+
- 已安装 OpenShift CLI (**oc**) v4.1+
- 访问基于 Kubernetes v1.11.3+ 的集群
- 访问容器 registry

流程

1. 设置发行版本变量：

```
RELEASE_VERSION=v0.8.0
```

2. 下载发行版二进制文件。
   - Linux：

```
$ curl -OJL https://github.com/operator-framework/operator-sdk/releases/download/${RELEASE_VERSION}/operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu
```

- macOS：

```
$ curl -OJL https://github.com/operator-framework/operator-sdk/releases/download/${RELEASE_VERSION}/operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin
```

3. 验证所下载的发行版本二进制文件。

a. 下载所提供的 ASC 文件。

- Linux：

```
$ curl -OJL https://github.com/operator-framework/operator-sdk/releases/download/${RELEASE_VERSION}/operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu.asc
```

- macOS：

```
$ curl -OJL https://github.com/operator-framework/operator-sdk/releases/download/${RELEASE_VERSION}/operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin.asc
```

b. 将二进制文件与对应的 ASC 文件放在同一个目录中，并运行以下命令来验证该二进制文件：

- Linux：

```
$ gpg --verify operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu.asc
```

- macOS：

```
$ gpg --verify operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin.asc
```

如果您的工作站上没有维护人员公钥，则会出现以下错误：

```
$ gpg --verify operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin.asc
$ gpg: assuming signed data in 'operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin'
$ gpg: Signature made Fri Apr  5 20:03:22 2019 CEST
$ gpg:                using RSA key <key_id> (1)
$ gpg: Can't check signature: No public key
```

(1) RSA 密钥字符串。

要下载密钥，请运行以下命令，用上一条命令输出的 RSA 密钥字符串来替换 **<key_id>**：

```
$ gpg [--keyserver keys.gnupg.net] --recv-key "<key_id>" (1)
```

(1) 如果您尚未配置密钥服务器，请使用 **--keyserver** 选项指定一个密钥服务器。

4. 在您的 **PATH** 中安装发行版本二进制文件：

   - Linux：

     ```
     $ chmod +x operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu
     $ sudo cp operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu
     /usr/local/bin/operator-sdk
     $ rm operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu
     ```

   - macOS：

     ```
     $ chmod +x operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin
     $ sudo cp operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin
     /usr/local/bin/operator-sdk
     $ rm operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin
     ```

5. 验证是否已正确安装 CLI 工具：

   ```
   $ operator-sdk version
   ```

11.3.2.2. 通过 Homebrew 安装

可使用 Homebrew 来安装 SDK CLI。

先决条件

- Homebrew
- **docker** v17.03+
- 已安装 OpenShift CLI (**oc**) v4.1+
- 访问基于 Kubernetes v1.11.3+ 的集群
- 访问容器 registry

流程

1. 使用 **brew** 命令安装 SDK CLI：

   ```
   $ brew install operator-sdk
   ```

2. 验证是否已正确安装 CLI 工具：

   ```
   $ operator-sdk version
   ```

11.3.2.3. 通过源代码编译并安装

可获取 Operator SDK 源代码来编译和安装 SDK CLI。

先决条件

- dep v0.5.0+

- Git
- Go v1.10+
- **docker** v17.03+
- 已安装 OpenShift CLI (**oc**) v4.1+
- 访问基于 Kubernetes v1.11.3+ 的集群
- 访问容器 registry

流程

1. 克隆 **operator-sdk** 存储库：

```
$ mkdir -p $GOPATH/src/github.com/operator-framework
$ cd $GOPATH/src/github.com/operator-framework
$ git clone https://github.com/operator-framework/operator-sdk
$ cd operator-sdk
```

2. 检查所需的发行版本分支：

```
$ git checkout master
```

3. 编译并安装 SDK CLI：

```
$ make dep
$ make install
```

该操作会在 ***$GOPATH/bin*** 中安装 CLI 二进制 **operator-sdk**。

4. 验证是否已正确安装 CLI 工具：

```
$ operator-sdk version
```

11.3.3. 使用 Operator SDK 来构建基于 Helm 的 Operator

本流程介绍了使用 Operator SDK 中的工具和库来构建由 Helm Chart 提供技术支持的简单 Nginx Operator 的示例。

提示

最好为每个 Chart 新建一个 Operator。如果要在 Go 中编写一个成熟的 Operator，摆脱基于 Helm 的 Operator，这种做法支持更多具有原生行为的 Kubernetes API（如 **oc get Nginx**），也更灵活。

先决条件

- 开发工作站已安装 Operator SDK CLI
- 使用具有 **cluster-admin** 权限的账户访问基于 Kubernetes 的集群 v1.11.3+（如 OpenShift Container Platform 4.2）
- 已安装 OpenShift CLI (**oc**) v4.1+

流程

1. 使用 **operator-sdk new** 命令来创建新 Operator 项目，可以是全命名空间范围，也可以是全集群范围。选择以下任意一项：

   a. *全命名空间范围的 Operator*（默认）会监视和管理单一命名空间中的资源。全命名空间范围的 Operator 因灵活性高而常被视为首选。这类 Operator 支持解耦升级、针对故障和监控隔离命名空间以及区别化 API 定义。
   要创建基于 Helm 的、全命名空间范围的新 **nginx-operator** 项目，请使用以下命令：

   ```
   $ operator-sdk new nginx-operator \
     --api-version=example.com/v1alpha1 \
     --kind=Nginx \
     --type=helm
   $ cd nginx-operator
   ```

   这样会创建一个 **nginx-operator** 项目，专门用于监视 APIVersion 为 **example.com/v1apha1**、Kind 为 **Nginx** 的 Nginx 资源。

   b. *全集群范围的 Operator* 会监视并管理整个集群的资源，在某些情况下非常实用。例如，**cert-manager** Operator 通常会以全集群范围权限和监视功能进行部署，以便管理整个集群的证书颁发。
   要创建全集群范围的 **nginx-operator** 项目，请使用以下命令：

   ```
   $ operator-sdk new nginx-operator \
       --cluster-scoped \
       --api-version=example.com/v1alpha1 \
       --kind=Nginx \
       --type=helm
   ```

   使用 **--cluster-scoped** 标记来构建进行了以下修改的新 Operator：

   - **deploy/operator.yaml**：设置 **WATCH_NAMESPACE=""** 而不要将其设置为 Pod 的命名空间。
   - **deploy/role.yaml**：使用 **ClusterRole** 来代替 **Role**。
   - **deploy/role_binding.yaml**：
     - 使用 **ClusterRoleBinding** 来代替 **rolebinding**。
     - 将主题命名空间设置为 **REPLACE_NAMESPACE**。该命名空间必须改为部署 Operator 的命名空间。

2. **自定义 Operator 逻辑。**
   在本例中，**nginx-operator** 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 Deployment，请创建一个。
   - 如果尚无 Nginx Service，请创建一个。
   - 如果支持 Nginx Ingress 但尚不存在，请创建一个。
   - 确保 Deployment、Service 和可选的 Ingress 符合 Nginx CR 指定的所需配置（如副本数、镜像、服务类型）。

默认情况下，**nginx-operator** 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：

```
- version: v1alpha1
  group: example.com
  kind: Nginx
  chart: /opt/helm/helm-charts/nginx
```

a. **查阅 Nginx Helm Chart。**
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 Deployment、Service 和 Ingress 资源提供了模板，另外还有 **NOTES.txt** 模板，Helm Chart 开发人员可利用该模板传达有关发型版本的实用信息。

如果您对 Helm Chart 还不够熟悉，请花一点时间查阅 Helm Chart开发人员文档。

b. **了解 Nginx CR spec。**
Helm 会使用一个名为 values 的概念来自定义 Helm Chart 的默认值，具体值在 Helm Chart 的 **values.yaml** 文件中定义。

通过在 CR spec 中设置所需值来覆盖这些默认值。以副本数量为例：

i. 首先，检查 **helm-charts/nginx/values.yaml** 文件，会发现 Chart 有一个名为 **replicaCount** 的值，默认设置为 **1**。要在部署中使用 2 个 Nginx 实例，您的 CR spec 中必须包含 **replicaCount: 2**。
更新 **deploy/crds/example_v1alpha1_nginx_cr.yaml** 文件，如下所示：

```
apiVersion: example.com/v1alpha1
kind: Nginx
metadata:
  name: example-nginx
spec:
  replicaCount: 2
```

ii. 同样，服务端口默认设置为 **80**。要改用 **8080**，请通过添加服务端口覆盖来再次更新 **deploy/crds/example_v1alpha1_nginx_cr.yaml** 文件：

```
apiVersion: example.com/v1alpha1
kind: Nginx
metadata:
  name: example-nginx
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。

3. **部署 CRD。**
在运行 Operator 之前，Kubernetes 需要了解 Operator 将会监视的新自定义资源定义 (CRD)。部署以下 CRD：

```
$ oc create -f deploy/crds/example_v1alpha1_nginx_crd.yaml
```

4. **构建并运行 Operator。**
有两种方式来构建和运行 Operator：

   - 作为 Kubernetes 集群内的一个 Pod。
   - 作为集群外的 Go 程序，使用 **operator-sdk up** 命令。

   选择以下任一方法：

   a. 作为 Kubernetes 集群内的**一个 Pod 来运行**。这是生产环境的首先方法。

      i. 构建 **nginx-operator** 镜像并将其推送至 registry：

      ```
      $ operator-sdk build quay.io/example/nginx-operator:v0.0.1
      $ docker push quay.io/example/nginx-operator:v0.0.1
      ```

      ii. **deploy/operator.yaml** 文件中会生成 Deployment 清单。本文件中的部署镜像需要从占位符 **REPLACE_IMAGE** 修改为之前构建的镜像。为此，请运行：

      ```
      $ sed -i 's|REPLACE_IMAGE|quay.io/example/nginx-operator:v0.0.1|g'
      deploy/operator.yaml
      ```

      iii. 如果使用 **--cluster-scoped=true** 标记来创建 Operator，请更新所生成的 **ClusterRoleBinding** 中的服务账户命名空间，以匹配 Operator 的部署位置：

      ```
      $ export OPERATOR_NAMESPACE=$(oc config view --minify -o
      jsonpath='{.contexts[0].context.namespace}')
      $ sed -i "s|REPLACE_NAMESPACE|$OPERATOR_NAMESPACE|g"
      deploy/role_binding.yaml
      ```

      如果要在 OSX 中执行这些步骤，请使用以下命令：

      ```
      $ sed -i "" 's|REPLACE_IMAGE|quay.io/example/nginx-operator:v0.0.1|g'
      deploy/operator.yaml
      $ sed -i "" "s|REPLACE_NAMESPACE|$OPERATOR_NAMESPACE|g"
      deploy/role_binding.yaml
      ```

      iv. 部署 **nginx-operator**：

      ```
      $ oc create -f deploy/service_account.yaml
      $ oc create -f deploy/role.yaml
      $ oc create -f deploy/role_binding.yaml
      $ oc create -f deploy/operator.yaml
      ```

      v. 验证 **nginx-operator** 是否正在运行：

      ```
      $ oc get deployment
      NAME             DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
      nginx-operator   1         1         1            1           1m
      ```

   b. **在集群外运行。**这是开发阶段的首选方法，可加快部署和测试的速度。
   此外，还务必要确保您的计算机上存在 **watches.yaml** 文件中引用的 Chart 路径。**watches.yaml** 文件默认能够与通过 **operator-sdk build** 命令构建的 Operator 镜像一起使用。当使用 **operator-sdk up local** 命令开发和测试您的 Operator 时，SDK 会在您的本地

文件系统中查找该路径。

i. 在该位置创建符号链接，以指向您的 Helm Chart 路径：

```
$ sudo mkdir -p /opt/helm/helm-charts
$ sudo ln -s $PWD/helm-charts/nginx /opt/helm/helm-charts/nginx
```

ii. 要使用 **$HOME/.kube/config** 中的默认 Kubernetes 配置文件在本地运行 Operator：

```
$ operator-sdk up local
```

要使用所提供的 Kubernetes 配置文件在本地运行 Operator：

```
$ operator-sdk up local --kubeconfig=<path_to_config>
```

5. **部署 Nginx CR。**
应用之前修改的 **Nginx** CR：

```
$ oc apply -f deploy/crds/example_v1alpha1_nginx_cr.yaml
```

确保 **nginx-operator** 为 CR 创建部署：

```
$ oc get deployment
NAME                                         DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
example-nginx-b9phnoz9spckcrua7ihrbkrt1      2         2         2            2           1m
```

检查 Pod，确认已创建两个副本：

```
$ oc get pods
NAME                                                READY     STATUS    RESTARTS   AGE
example-nginx-b9phnoz9spckcrua7ihrbkrt1-f8f9c875d-fjcr9   1/1       Running   0          1m
example-nginx-b9phnoz9spckcrua7ihrbkrt1-f8f9c875d-ljbzl   1/1       Running   0          1m
```

检查 Service 端口是否已设置为 **8080**：

```
$ oc get service
NAME                                         TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
example-nginx-b9phnoz9spckcrua7ihrbkrt1      ClusterIP   10.96.26.3   <none>        8080/TCP
1m
```

6. **更新 replicaCount 并删除端口。**
将 **spec.replicaCount** 字段从 **2** 改为 **3**，删除 **spec.service** 字段并应用更改：

```
$ cat deploy/crds/example_v1alpha1_nginx_cr.yaml
apiVersion: "example.com/v1alpha1"
kind: "Nginx"
metadata:
  name: "example-nginx"
spec:
  replicaCount: 3

$ oc apply -f deploy/crds/example_v1alpha1_nginx_cr.yaml
```

确认 Operator 已更改 Deployment 大小：

```
$ oc get deployment
NAME                                     DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
example-nginx-b9phnoz9spckcrua7ihrbkrt1   3         3         3            3           1m
```

检查 Service 端口是否已设置为默认值 **80**。

```
$ oc get service
NAME                                      TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
example-nginx-b9phnoz9spckcrua7ihrbkrt1   ClusterIP   10.96.26.3   <none>        80/TCP
1m
```

7. **清理资源：**

```
$ oc delete -f deploy/crds/example_v1alpha1_nginx_cr.yaml
$ oc delete -f deploy/operator.yaml
$ oc delete -f deploy/role_binding.yaml
$ oc delete -f deploy/role.yaml
$ oc delete -f deploy/service_account.yaml
$ oc delete -f deploy/crds/example_v1alpha1_nginx_crd.yaml
```

11.3.4. 其他资源

- 有关 Operator SDK 所创建的项目目录结构的信息，请参阅附录。
- 面向红帽合作伙伴的 Operator 开发指南

11.4. 生成 CLUSTERSERVICEVERSION (CSV)

ClusterServiceVersion (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 中包含 **olm-catalog gen-csv** 子命令，用于通过手动定义的 YAML 清单和 Operator 源文件中包含的信息为自定义的当前 Operator 项目生成 *ClusterServiceVersion* (CSV)。

借助生成 CSV 的命令，Operator 作者便无需深入掌握为了让其 Operator 与 Operator Lifecycle Manager (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。

CSV 版本与 Operator 的版本相同，在升级 Operator 版本的同时会生成新 CSV。Operator 作者可使用 **--csv-version** 标记将其 Operator 状态与所提供的语义版本一起封装至 CSV 中：

```
$ operator-sdk olm-catalog gen-csv --csv-version <version>
```

此操作采用等幂方式，仅会在提供新版本或更改 YAML 清单或源文件时才会更新 CSV 文件。Operator 作者不必直接修改 CSV 清单中的大部分字段。本指南中已经明确了需要修改的字段。例如：**metadata.name** 中必须包含 CSV 版本信息。

11.4.1. CSV 生成的工作方式

Operator 项目的 **deploy/** 目录是部署 Operator 需要的所有清单的标准位置。Operator SDK 可使用 **deploy/** 中的清单数据来编写 CSV。以下命令：

```
$ operator-sdk olm-catalog gen-csv --csv-version <version>
```

默认将 CSV YAML 文件写入 **deploy/olm-catalog/** 目录中。

生成 CSV 只需三类清单：

- **operator.yaml**
- ***_{crd,cr}.yaml**
- RBAC 角色文件，如 **role.yaml**

Operator 作者对这些文件可能有不同的版本要求，并且可配置 **deploy/olm-catalog/csv-config.yaml** 中要包含哪些特定文件。

工作流
根据是否检测到了现有 CSV，并假设使用所有配置默认值，**olm-catalog gen-csv** 子命令将：

- 使用 YAML 清单和源文件中的可用数据新建一个 CSV，其位置和命名约定与当前存在的信息相同。
    a. 该更新机制会检查 **deploy/** 中是否存在现有 CSV。如果未找到，便会创建一个 ClusterServiceVersion 对象，这里称为 *cache*，并填充容易从 Operator 元数据派生的字段，如 Kubernetes API **ObjectMeta**。
    b. 更新机制在 **deploy/** 中搜索包含 CSV 所用数据（如 Deployment 资源）的清单，并使用该数据设置 cache 中的适当 CSV 字段。
    c. 搜索完成后，每个填充的 cache 字段都会写回到 CSV YAML 文件中。

或：

- 使用 YAML 清单和源文件中的可用数据更新当前预定义位置的现有 CSV。
    a. 该更新机制会检查 **deploy/** 中是否存在现有 CSV。找到后，CSV YAML 文件内容会被封送至 ClusterServiceVersion cache 中。
    b. 更新机制在 **deploy/** 中搜索包含 CSV 所用数据（如 Deployment 资源）的清单，并使用该数据设置 cache 中的适当 CSV 字段。
    c. 搜索完成后，每个填充的 cache 字段都会写回到 CSV YAML 文件中。

注意

单个 YAML 字段（而非整个文件）会被覆盖，因为 CSV 的描述和其他未生成的部分将被保留。

11.4.2. CSV 组成配置

Operator 作者可通过填充 **deploy/olm-catalog/csv-config.yaml** 文件中的几个字段来配置 CSV 组成：

字段	描述
operator-path (string)	Operator 资源清单文件路径。默认为 **deploy/operator.yaml**。
crd-cr-path-list (string(, string)*)	CRD 和 CR 清单文件路径列表。默认为 **[deploy/crds/*_{crd,cr}.yaml]**。
rbac-path-list (string(, string)*)	RBAC 角色清单文件路径列表。默认为 **[deploy/role.yaml]**。

11.4.3. 手动定义的 CSV 字段

很多 CSV 字段无法使用生成的非 SDK 特定清单进行填充。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) 的英文元数据。

Operator 作者必须直接修改其 CSV YAML 文件，将个性化数据添加至以下必填字段。当检测到任何必填字段中缺少数据时，Operator SDK 就会发出一个 CSV 生成警告。

表 11.5. 必填

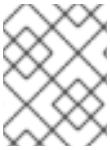
字段	描述
metadata.name	该 CSV 的唯一名称。Operator 版本应包含在名称中，以保证唯一性，如 **app-operator.v0.1.1**。
spec.displayName	用于标识 Operator 的公共名称。
spec.description	简单描述 Operator 的功能。
spec.keywords	描述 Operator 的关键词。
spec.maintainers	维护 Operator 的个人或组织实体，含**名称**和**电子邮件地址**。
spec.provider	Operator 供应商（通常为一个组织），含**名称**。
spec.labels	供 Operator 内部使用的键值对。
spec.version	Operator 的语义版本，如 **0.1.1**。

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： • **description**：CRD 描述。 • **resources**：CRD 利用的任何 Kubernetes 资源，如 Pod 和 StatefulSets。 • **specDescriptors**：用于 Operator 输入和输出的 UI 提示。

表 11.6. 选填

字段	描述
spec.replaces	被该 CSV 替换的 CSV 名称。
spec.links	与被管理的 Operator 或应用程序相关的 URL（如网站和文档），各自含**名称**和 **url**。
spec.selector	Operator 可用于配对群集中资源的选择器。
spec.icon	Operator 独有的 base64 编码图标，通过 **mediatype** 在 **base64data** 字段中设置。
spec.maturity	根据 Operator 成熟度模型划分的 Operator 能力水平，如**无缝升级**。

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

当前需要用户干预的多个 YAML 字段可能会从 Operator 代码中解析出来；这种 Operator SDK 功能将在未来的设计文件中阐述。

其他资源

- Operator 成熟度模型

11.4.4. 生成 CSV

先决条件

- 使用 Operator SDK 生成一个 Operator 项目

流程

1. 在 Operator 项目中，通过修改 **deploy/olm-catalog/csv-config.yaml** 文件来配置您的 CSV 组成（如果需要）。
2. 生成 CSV：

```
$ operator-sdk olm-catalog gen-csv --csv-version <version>
```

-

3. 在 **deploy/olm-catalog/** 目录中生成的新 CSV 中，确保正确设置所有必填的手动定义字段。

11.4.5. 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。

11.4.5.1. 拥有的 CRD

Operator 拥有的 CRD 是 CSV 最重要的组成部分。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 ReplicaSets 表示形式。这在 CSV 文件中应逐一列出。

表 11.7. 拥有的 CRD 字段

字段	描述	必填/选填
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	您的 CRD 可能拥有一类或多类 Kubernetes 对象。它们将在资源部分列出，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 Service 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，存储不应由用户修改的内部状态的 ConfigMaps 不应出现在此处。	选填

字段	描述	必填/选填
SpecDescriptors、StatusDescriptors 和 ActionDescriptors	这些描述符是通过对终端用户来说最重要的 Operator 的某些输入或输出提示 UI 的一种方式。如果您的 CRD 包含用户必须提供的 Secret 或 ConfigMap 的名称，您可在此处指定。这些项目在兼容的 UI 中链接并突出显示。 共有以下三类描述符： • **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。 所有描述符均接受以下字段： • **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。 有关描述符的更多一般信息，请参见 **openshift/console** 项目。	选填

以下示例描述了一个 **MongoDB Standalone** CRD，要求某些用户以 Secret 和 ConfigMap 的形式输入，并编排 Service、StatefulSet、Pod 和 ConfigMap：

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.
```

11.4.5.2. 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 11.8. 必需的 CRD 字段

字段	描述	必填/选填
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.
```

11.4.5.3. CRD 模板

Operator 用户需要了解哪个选项必填，哪个选项选填。您可为您的每个 CRD 提供模板，并以最小配置集作为名为 **alm-examples** 的注解。兼容 UI 会预先填充该模板，供用户进一步自定义。

该注解由一个 **kind** 列表组成，如 CRD 名称和对应的 Kubernetes 对象的 **metadata** 和 **spec**。

以下完整示例提供了 **EtcdCluster**、**EtcdBackup** 和 **EtcdRestore** 模板：

```
metadata:
  annotations:
    alm-examples: >-
      [{"apiVersion":"etcd.database.coreos.com/v1beta2","kind":"EtcdCluster","metadata":
{"name":"example","namespace":"default"},"spec":{"size":3,"version":"3.2.13"}},
{"apiVersion":"etcd.database.coreos.com/v1beta2","kind":"EtcdRestore","metadata":
{"name":"example-etcd-cluster"},"spec":{"etcdCluster":{"name":"example-etcd-
cluster"},"backupStorageType":"S3","s3":{"path":"<full-s3-path>","awsSecret":"<aws-secret>"}}},
{"apiVersion":"etcd.database.coreos.com/v1beta2","kind":"EtcdBackup","metadata":
{"name":"example-etcd-cluster-backup"},"spec":{"etcdEndpoints":["<etcd-cluster-
endpoints>"],"storageType":"S3","s3":{"path":"<full-s3-path>","awsSecret":"<aws-secret>"}}}]
```

11.4.6. 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*。

11.4.6.1. 拥有的 APIService

如果 CSV 拥有 APIService，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 **group-version-kinds**。

APIService 由其提供的 **group-version** 唯一标识，且可多次列出，以表示其期望提供的不同类型。

表 11.9. 拥有的 APIService 字段

字段	描述	必填/选填
Group	由 APIService 提供的组，如 **database.example.com**。	必填
Version	APIService 的版本，如 **v1alpha1**。	必填
Kind	预期由 APIService 提供的类型。	必填
名称	所提供的 APIService 的复数名称	必填

字段	描述	必填/选填
DeploymentName	由您的 CSV 定义的部署名称，对应您的 APIService（对于拥有的 APIService 来说必填）。在 CSV 待定阶段，OLM Operator 会在您的 CSV InstallStrategy 中搜索具有匹配名称的部署 spec，如果未找到，则不会将 CSV 转换至安装就绪阶段。	必填
DisplayName	APIService 名称的人类可读版本，如 **MongoDB Standalone**。	必填
描述	有关 Operator 如何使用该 APIService 的简短描述，或有关 APIService 所提供功能的描述。	必填
Resources	您的 APIService 可能拥有一类或多类 Kubernetes 对象。它们将在资源部分列出，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 Service 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，存储不应由用户修改的内部状态的 ConfigMaps 不应出现在此处。	选填
SpecDescriptors、**StatusDescriptors** 和 **ActionDescriptors**	与拥有的 CRD 基本相同。	选填

11.4.6.1.1. APIService 资源创建

Operator Lifecycle Manager (OLM) 负责为每个唯一拥有的 APIService 创建或替换 Service 及 APIService 资源：

- Service Pod 选择器从与 APIServiceDescription 的 **DeploymentName** 相匹配的 CSV 部署中复制。
- 每次安装都会生成一个新的 CA 密钥/证书对，且 base64 编码的 CA 捆绑包会嵌入到相应 APIService 资源中。

11.4.6.1.2. APIService 服务证书

每当安装拥有的 APIService 时，OLM 均会处理服务密钥/证书对的生成。服务证书有一个包含所生成 Service 资源的主机名的 CN，并由嵌入至对应 APIService 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，一个名为 **apiservice-cert** 的卷会自动附加至 CSV 中与 APIServiceDescription 的 **DeploymentName** 字段相匹配的部署的 Volumes 部分中。

如果尚不存在，则具有匹配名称的 VolumeMount 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 VolumeMount，以适应任何自定义路径要求。所生成的 VolumeMount 的默认路径为 **/apiserver.local.config/certificates**，具有相同路径的任何现有 VolumeMount 均会被替换掉。

11.4.6.2. 必需的 APIService

OLM 可保证所有必需的 CSV 均有可用 APIService，且所有预期的 **group-version-kinds** 在试图安装前均可发现。这样 CSV 便可依赖由 APIService 提供的、并非它拥有的特定类别。

表 11.10. 必需的 APIService 字段

字段	描述	必填/选填
Group	由 APIService 提供的组，如 **database.example.com**。	必填
Version	APIService 的版本，如 **v1alpha1**。	必填
Kind	预期由 APIService 提供的类型。	必填
DisplayName	APIService 名称的人类可读版本，如 **MongoDB Standalone**。	必填
描述	有关 Operator 如何使用该 APIService 的简短描述，或有关 APIService 所提供功能的描述。	必填

11.5. 使用 PROMETHEUS 配置内置监控

本指南介绍了 Operator SDK 通过 Prometheus Operator 提供的内置监控支持，及其对 Operator 作者的详细用途。

11.5.1. 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。

11.5.2. 指标帮助函数

在使用 Operator SDK 生成的基于 Go 的 Operator 中，以下函数会公开有关运行中程序的一般指标：

```
func ExposeMetricsPort(ctx context.Context, port int32) (*v1.Service, error)
```

这些指标从 **controller-runtime** 库 API 继承而来。默认在 **0.0.0.0:8383/metrics** 上提供指标。

创建一个 Service 对象并公开指标端口，之后可通过 Prometheus 访问该端口。删除领导 Pod 的 root 所有者时，Service 对象便会被垃圾回收。

以下示例出现在使用 Operator SDK 生成的所有 Operator 的 **cmd/manager/main.go** 文件中：

```
import(
    "github.com/operator-framework/operator-sdk/pkg/metrics"
    "machine.openshift.io/controller-runtime/pkg/manager"
)

var (
    // Change the below variables to serve metrics on a different host or port.
```

```
    metricsHost       = "0.0.0.0" (1)
    metricsPort int32 = 8383 (2)
)
...
func main() {
    ...
    // Pass metrics address to controller-runtime manager
    mgr, err := manager.New(cfg, manager.Options{
        Namespace:          namespace,
        MetricsBindAddress: fmt.Sprintf("%s:%d", metricsHost, metricsPort),
    })

    ...
    // Create Service object to expose the metrics port.
    _, err = metrics.ExposeMetricsPort(ctx, metricsPort)
    if err != nil {
        // handle error
        log.Info(err.Error())
    }
    ...
}
```

(1) 在其上公开指标的主机。

(2) 在其上公开指标的端口。

11.5.2.1. 修改指标端口

Operator 作者可修改在其上公开指标的端口。

先决条件

- 使用 Operator SDK 生成基于 Go 的 Operator
- 基于 Kubernetes 的集群已部署 Prometheus Operator

流程

- 在所生成的 Operator **cmd/manager/main.go** 文件中，修改 **var metricsPort int32 = 8383** 一行中 **metricsPort** 的值。

11.5.3. ServiceMonitor 资源

ServiceMonitor 是由 Prometheus Operator 提供的一个自定义资源定义 (CRD)，可发现 Service 对象中的 **Endpoints**，配置 Prometheus 以监控这些 Pod。

在使用 Operator SDK 生成的基于 Go 的 Operator 中，**GenerateServiceMonitor()** 帮助函数可使用 Service 对象，并基于该对象生成 ServiceMonitor 自定义资源 (CR)。

其他资源

- 有关 ServiceMonitor CRD 的更多详情，请参阅 Prometheus Operator 文档。

11.5.3.1. 创建 ServiceMonitor 资源

Operator 作者可使用 **metrics.CreateServiceMonitor()** 帮助函数来添加已创建监控 Service 的 Service 目标发现，该函数接受新创建的 Service。

先决条件

- 使用 Operator SDK 生成基于 Go 的 Operator
- 基于 Kubernetes 的集群已部署 Prometheus Operator

流程

- 将 **metrics.CreateServiceMonitor()** 帮助函数添加至您的 Operator 代码中：

```
import(
   "k8s.io/api/core/v1"
   "github.com/operator-framework/operator-sdk/pkg/metrics"
   "machine.openshift.io/controller-runtime/pkg/client/config"
)
func main() {

   ...
   // Populate below with the Service(s) for which you want to create ServiceMonitors.
   services := []*v1.Service{}
   // Create one ServiceMonitor per application per namespace.
   // Change the below value to name of the Namespace you want the ServiceMonitor to be
created in.
   ns := "default"
   // restConfig is used for talking to the Kubernetes apiserver
   restConfig := config.GetConfig()

   // Pass the Service(s) to the helper function, which in turn returns the array of
ServiceMonitor objects.
   serviceMonitors, err := metrics.CreateServiceMonitors(restConfig, ns, services)
   if err != nil {
      // Handle errors here.
   }
   ...
}
```

11.6. 配置领导选举机制

在 Operator 的生命周期中，在任意给定时间可能有多个实例在运行，例如，推出 Operator 升级程序。这种情况下，需要使用领导选举机制来避免多个 Operator 实例争用。这样可确保只有一个领导实例处理协调，其他实例均不活跃，但却会做好准备，随时接管领导实例的的工作。

有两种不同的领导选举实现可供选择，每种机制都有各自的利弊权衡问题：

- *Leader-for-life*：领导 Pod 只有在被删除时才会交出领导权（通过垃圾回收程序）。这种实现可避免两个实例同时作为领导运行（脑裂）。但这种方法可能会延迟选举新的领导。例如，当领导 Pod 位于无响应或分区的节点上时，**pod-eviction-timeout** 将决定领导 Pod 从节点上删除和停止运行的方式（默认 **5m**）。详情请参见 Leader-for-life Go 文档。

- *Leader-with-lease*：领导 Pod 定期更新领导租约，并在无法更新租约时放弃领导权。当现有领导 Pod 被隔离时，这种实现方式可更快速地过渡至新领导，但在某些情况下存在脑裂的可能性。详情请参见 Leader-with-lease Go 文档。

Operator SDK 默认启用 Leader-for-life 实现。请查阅相关 Go 文档，了解这两种方法，权衡利弊，决定对您的用例来说更有意义的方法。

以下示例演示了如何使用这两个选项。

11.6.1. 使用 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 ConfigMap 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 的命名空间。

11.6.2. 使用 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 不在集群中运行时，Manager 会在启动时返回一个错误，因其无法检测 Operator 的命名空间以针对领导选举机制创建 ConfigMap。您可通过设置 Manager 的 **LeaderElectionNamespace** 选项来覆盖该命名空间。

11.7. OPERATOR SDK CLI 参考

本指南记录 Operator SDK CLI 命令及其语法：

```
$ operator-sdk <command> [<subcommand>] [<argument>] [<flags>]
```

11.7.1. build

operator-sdk build 命令编译代码并构建可执行文件。**build** 完成后，镜像会在 **docker** 本地构建。之后必须推送至远程 registry。

表 11.11. build 参数

参数	描述
<image>	要构建的容器镜像，如：**quay.io/example/operator:v0.0.1**。

表 11.12. build 标记

标记	描述
--enable-tests (bool)	通过在镜像中添加测试二进制文件来启用集群内测试。
--namespaced-manifest (string)	用于测试的命名空间资源清单的路径。默认值：**deploy/operator.yaml**。
--test-location (string)	测试位置。默认值：**./test/e2e**
-h, --help	使用方法帮助输出。

如果设定了 **--enable-tests**，**build** 命令还会构建测试二进制文件，将其添加到容器镜像中，并生成一个 **deploy/test-pod.yaml** 文件，允许用户在群集上作为 Pod 运行测试。

输出示例

```
$ operator-sdk build quay.io/example/operator:v0.0.1

building example-operator...

building container quay.io/example/operator:v0.0.1...
Sending build context to Docker daemon  163.9MB
Step 1/4 : FROM alpine:3.6
 ---> 77144d8c6bdc
Step 2/4 : ADD tmp/_output/bin/example-operator /usr/local/bin/example-operator
 ---> 2ada0d6ca93c
Step 3/4 : RUN adduser -D example-operator
 ---> Running in 34b4bb507c14
Removing intermediate container 34b4bb507c14
 ---> c671ec1cff03
Step 4/4 : USER example-operator
 ---> Running in bd336926317c
```

```
Removing intermediate container bd336926317c
 ---> d6b58a0fcb8c
Successfully built d6b58a0fcb8c
Successfully tagged quay.io/example/operator:v0.0.1
```

11.7.2. completion

operator-sdk completion 命令生成 shell completion，以便更迅速、更轻松地发出 CLI 命令。

表 11.13. completion 子命令

子命令	描述
bash	生成 bash completion。
zsh	生成 zsh completion。

表 11.14. completion 标记

标记	描述
-h, --help	使用方法帮助输出。

输出示例

```
$ operator-sdk completion bash

# bash completion for operator-sdk                  -*- shell-script -*-
...
# ex: ts=4 sw=4 et filetype=sh
```

11.7.3. print-deps

operator-sdk print-deps 命令打印 Operator 所需的最新 Golang 软件包和版本。默认以列示格式打印。

表 11.15. print-deps 标记

标记	描述
--as-file	以 **Gopkg.toml** 格式打印软件包和版本。

输出示例

```
$ operator-sdk print-deps --as-file
required = [
  "k8s.io/code-generator/cmd/defaulter-gen",
  "k8s.io/code-generator/cmd/deepcopy-gen",
  "k8s.io/code-generator/cmd/conversion-gen",
  "k8s.io/code-generator/cmd/client-gen",
```

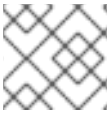
```
  "k8s.io/code-generator/cmd/lister-gen",
  "k8s.io/code-generator/cmd/informer-gen",
  "k8s.io/code-generator/cmd/openapi-gen",
  "k8s.io/gengo/args",
]

[[override]]
  name = "k8s.io/code-generator"
  revision = "6702109cc68eb6fe6350b83e14407c8d7309fd1a"
...
```

11.7.4. generate

operator-sdk generate 命令调用特定生成器以便根据需要生成代码。

表 11.16. generate 子命令

子命令	描述
k8s	在 **pkg/apis/** 下针对所有 CRD API 运行 Kubernetescode-generators。目前，**k8s** 只会运行 **deepcopy-gen** 来为所有自定义资源 (CR) 类型生成所需的 **DeepCopy()** 函数。



注意

每次需要更新自定义资源类型的 API（**spec** 和 **status**）时，便必须运行该命令。

输出示例

```
$ tree pkg/apis/app/v1alpha1/
pkg/apis/app/v1alpha1/
├── appservice_types.go
├── doc.go
├── register.go

$ operator-sdk generate k8s
Running code-generation for Custom Resource (CR) group versions: [app:v1alpha1]
Generating deepcopy funcs

$ tree pkg/apis/app/v1alpha1/
pkg/apis/app/v1alpha1/
├── appservice_types.go
├── doc.go
├── register.go
└── zz_generated.deepcopy.go
```

11.7.5. olm-catalog

operator-sdk olm-catalog 是所有 Operator Lifecycle Manager (OLM) 目录相关命令的父命令。

11.7.5.1. gen-csv

使用 **gen-csv** 子命令可将 ClusterServiceVersion (CSV) 清单和（可选）自定义资源定义 (CRD) 文件写入 **deploy/olm-catalog/<operator_name>/<csv_version>** 中。

表 11.17. olm-catalog gen-csv 标记

标记	描述
--csv-version (string)	CSV 清单的语义版本。必需。
--from-version (string)	CSV 清单的语义版本，用作新版本基础。
--csv-config (string)	CSV 配置文件的路径。默认值：**deploy/olm-catalog/csv-config.yaml**。
--update-crds	使用最新 CRD 清单更新 **deploy/<operator_name>/<csv_version>** 中的 CRD 清单。

输出示例

```
$ operator-sdk olm-catalog gen-csv --csv-version 0.1.0 --update-crds
INFO[0000] Generating CSV manifest version 0.1.0
INFO[0000] Fill in the following required fields in file deploy/olm-catalog/operator-
name/0.1.0/operator-name.v0.1.0.clusterserviceversion.yaml:
 spec.keywords
 spec.maintainers
 spec.provider
 spec.labels
INFO[0000] Created deploy/olm-catalog/operator-name/0.1.0/operator-
name.v0.1.0.clusterserviceversion.yaml
```

11.7.6. new

使用 **operator-sdk new** 命令可创建新的 Operator 应用程序，并根据输入的 **<project_name>** 生成（或*构建*）默认项目目录布局。

表 11.18. new 参数

参数	描述
<project_name>	新项目的名称。

表 11.19. new 标记

标记	描述
--api-version	以 **$GROUP_NAME/$VERSION** 格式呈现的 CRD **APIVersion**，如 **app.example.com/v1alpha1**。与 **ansible** 或 **helm** 类型一同使用。
--dep-manager [dep|modules]	新项目将使用的依赖项管理器。与 **go** 类型一同使用。（默认值：**modules**）

标记	描述
--generate-playbook	生成 Ansible playbook 框架。与 **ansible** 类型一同使用。
--header-file <string>	包含所生成 Go 文件标题的文件路径。复制到 **hack/boilerplate.go.txt**。
--helm-chart <string>	用现有 Helm Chart **<url>**、**<repo>/<name>** 或本地路径来初始化 Helm Operator。
--helm-chart-repo <string>	用于所请求 Helm Chart 的 Chart 存储库 URL。
--helm-chart-version <string>	Helm Chart 的特定版本。（默认值：最新版本）
--help, -h	使用方法和帮助输出。
--kind <string>	CRD **Kind**，如 **AppService**。与 **ansible** 或 **helm** 类型一同使用。
--skip-git-init	不要将目录初始化为 Git 存储库。
--type	要初始化的 Operator 类型：**go**、**ansible** 或 **helm**。（默认值：**go**）

Go 项目使用方法示例

```
$ mkdir $GOPATH/src/github.com/example.com/
$ cd $GOPATH/src/github.com/example.com/
$ operator-sdk new app-operator
```

Ansible 项目使用方法示例

```
$ operator-sdk new app-operator \
    --type=ansible \
    --api-version=app.example.com/v1alpha1 \
    --kind=AppService
```

11.7.7. add

使用 **operator-sdk add** 命令可为项目添加控制器或资源。该命令必须从 Operator 项目根目录运行。

表 11.20. add 子命令

子命令	描述
api	在 **pkg/apis** 下为新的自定义资源 (CR) 添加新 API 定义，并在 **deploy/crds/** 下生成自定义资源定义 (CRD) 和自定义资源 (CR) 文件。如果 **pkg/apis/<group>/<version>** 中已存在 API，则该命令不会覆盖，且会返回错误。

子命令	描述
controller	在 **pkg/controller/<kind>/** 下添加新控制器。控制器希望使用 **pkg/apis/<group>/<version>** 下应该已经通过 **operator-sdk add api --kind=<kind> --api-version=<group/version>** 命令定义的 CR 类型。如果 **pkg/controller/<kind>** 中已存在该 **Kind** 的控制器软件包，则该命令不会进行覆盖，并且会返回错误。
crd	添加 CRD 和 CR 文件。**<project-name>/deploy** 路径必须已存在。需要 **--api-version** 和 **--kind** 标记来生成新的 Operator 应用程序。 ● 所生成的 CRD 文件名：**<project-name>/deploy/crds/<group>_<version>_<kind>_crd.yaml** ● 所生成的 CR 文件名：**<project-name>/deploy/crds/<group>_<version>_<kind>_cr.yaml**

表 11.21. **add api** 标记

标记	描述
--api-version (string)	以 **$GROUP_NAME/$VERSION** 格式呈现的 CRD **APIVersion**，如 **app.example.com/v1alpha1**。
--kind (string)	CRD **Kind**（如 **AppService**）。

add api 输出示例

```
$ operator-sdk add api --api-version app.example.com/v1alpha1 --kind AppService
Create pkg/apis/app/v1alpha1/appservice_types.go
Create pkg/apis/addtoscheme_app_v1alpha1.go
Create pkg/apis/app/v1alpha1/register.go
Create pkg/apis/app/v1alpha1/doc.go
Create deploy/crds/app_v1alpha1_appservice_cr.yaml
Create deploy/crds/app_v1alpha1_appservice_crd.yaml
Running code-generation for Custom Resource (CR) group versions: [app:v1alpha1]
Generating deepcopy funcs

$ tree pkg/apis
pkg/apis/
├── addtoscheme_app_appservice.go
├── apis.go
└── app
    └── v1alpha1
        ├── doc.go
        ├── register.go
        ├── types.go
```

add controller 输出示例

```
$ operator-sdk add controller --api-version app.example.com/v1alpha1 --kind AppService
Create pkg/controller/appservice/appservice_controller.go
Create pkg/controller/add_appservice.go

$ tree pkg/controller
pkg/controller/
├── add_appservice.go
├── appservice
│   └── appservice_controller.go
└── controller.go
```

add crd 输出示例

```
$ operator-sdk add crd --api-version app.example.com/v1alpha1 --kind AppService
Generating Custom Resource Definition (CRD) files
Create deploy/crds/app_v1alpha1_appservice_crd.yaml
Create deploy/crds/app_v1alpha1_appservice_cr.yaml
```

11.7.8. test

operator-sdk test 命令可在本地测试 Operator。

11.7.8.1. local

通过 **local** 子命令，在本地运行使用 Operator SDK 测试框架构建的 Go 测试。

表 11.22. test local 参数

参数	描述
<test_location> (string)	e2e 测试文件的位置（如 **./test/e2e/**）。

表 11.23. test local 标记

标记	描述
--kubeconfig (string)	集群的 **kubeconfig** 位置。默认值：**~/.kube/config**。
--global-manifest (string)	全局资源清单的路径。默认值：**deploy/crd.yaml**。
--namespaced-manifest (string)	按测试的、命名空间资源的清单路径。默认将 **deploy/service_account.yaml**、**deploy/rbac.yaml** 与 **deploy/operator.yaml** 合并到一起。
--namespace (string)	如果非空，则为在其中运行测试的单个命名空间（如 **operator-test**）。默认值：**""**
--go-test-flags (string)	要传递至 **go test** 的额外参数（如 **-f "-v -parallel=2"**）。

标记	描述
--up-local	使用 **go run** 在本地运行 Operator，而非作为集群中的镜像运行。
--no-setup	禁用测试资源创建。
--image (string)	使用与命名空间清单中指定的镜像不同的 Operator 镜像。
-h, --help	使用方法帮助输出。

输出示例

```
$ operator-sdk test local ./test/e2e/

# Output:
ok   github.com/operator-framework/operator-sdk-samples/memcached-operator/test/e2e 20.410s
```

11.7.9. up

operator-sdk up 命令包含可通过多种方式启动 Operator 的子命令。

11.7.9.1. local

local 子命令通过构建 Operator 二进制文件来启动本地机器上的 Operator，该二进制文件可使用 **kubeconfig** 文件来访问 Kubernetes 集群。

表 11.24. up local 参数

参数	描述
--kubeconfig (string)	Kubernetes 配置文件的文件路径。默认值：**$HOME/.kube/config**
--namespace (string)	Operator 监视是否发生变化的命名空间。默认值：**default**
--operator-flags	本地 Operator 可能需要的标记。示例：**--flag1 value1 --flag2=value2**
-h, --help	使用方法帮助输出。

输出示例

```
$ operator-sdk up local \
  --kubeconfig "mycluster.kubecfg" \
  --namespace "default" \
  --operator-flags "--flag1 value1 --flag2=value2"
```

以下示例使用默认 **kubeconfig**，即默认的命名空间环境变量，并为 Operator 传递标记。要使用 Operator 标记，您的 Operator 必须知道如何处理该选项。例如，对于理解 **resync-interval** 标记的 Operator 来说：

```
$ operator-sdk up local --operator-flags "--resync-interval 10"
```

如果您计划使用与默认命名空间不同的命名空间，请使用 **--namespace** 标记来更改 Operator 监视要创建自定义资源 (CR) 的位置：

```
$ operator-sdk up local --namespace "testing"
```

要实现这一目的，您的 Operator 必须负责处理 **WATCH_NAMESPACE** 环境变量。这可通过使用 Operator 中的 utility 函数 **k8sutil.GetWatchNamespace** 来完成。

11.8. 附录

11.8.1. Operator 项目构建布局

Operator-sdk CLI 会为每个 Operator 项目生成大量软件包。以下部分描述每个生成的文件和目录的基本概要。

11.8.1.1. 基于 Go 的项目

使用 **operator-sdk new** 命令生成的基于 Go 的 Operator 项目（默认类型）包含以下目录和文件：

文件/文件夹	用途
cmd/	包含 **manager/main.go** 文件，该文件是 Operator 的主程序。这将实例化一个新管理器，它会在 **pkg/apis/** 下注册所有自定义资源定义，并启动 **pkg/controllers/** 下的所有控制器。
pkg/apis/	包含定义自定义资源定义 (CRD) 的 API 的目录树。用户需要编辑 **pkg/apis/<group>/<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。
pkg/controller	该 **pkg** 包含控制器实现。用户需要编辑 **pkg/controller/<kind>/<kind>_controller.go** 文件，以定义控制器处理指定 **kind** 资源类型的协调逻辑。
build/	包含用于构建 Operator 的 **Dockerfile** 和构建脚本。
deploy/	包含各种 YAML 清单，用于注册 CRD、设置 RBAC 和将 Operator 部署为 Deployment。
Gopkg.toml **Gopkg.lock**	描述该 Operator 外部依赖项的 Go Dep 清单。
vendor/	golang vendor 文件夹，包含满足此项目导入内容的外部依赖项的本地副本。Go Dep 可直接管理 vendor。

11.8.1.2. 基于 Helm 的项目

使用 **operator-sdk new --type helm** 命令生成的基于 Helm 的 Operator 项目（默认类型）包含以下目录和文件：

文件/文件夹	用途
deploy/	包含各种 YAML 清单，用于注册 CRD、设置 RBAC 和将 Operator 部署为 Deployment。
helm-charts/<kind>	包含使用 **helm create** 对等命令初始化的 Helm Chart。
build/	包含用于构建 Operator 的 **Dockerfile** 和构建脚本。
watches.yaml	包含 **Group**、**Version**、**Kind** 和 Helm Chart 位置。