OpenShift Container Platform 4.6

Serverless

OpenShift Serverless 的安装、使用与发行注记

Last Updated: 2023-02-27

OpenShift Container Platform 4.6 Serverless

OpenShift Serverless 的安装、使用与发行注记

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

法律通告

摘要

本文档提供有关如何在 OpenShift Container Platform 中使用 OpenShift Serverless 的信息.

目录

第 1 章 发行注记

发行注记包含有关新的和已弃用的功能、破坏更改以及已知问题的信息。以下发行注记适用于 OpenShift Container Platform 的最新 OpenShift Serverless 版本。

如需了解 OpenShift Serverless 功能概述，请参阅 关于 OpenShift Serverless。

注意

OpenShift Serverless 基于开源的 Knative 项目。

有关最新 Knative 组件发行版本的详情，请参阅 Knative 博客。

1.1. 关于 API 版本

API 版本是 OpenShift Serverless 中特定函数和自定义资源的开发状态的重要因素。在没有使用正确 API 版本的集群中创建资源可能会导致部署出现问题。

OpenShift Serverless Operator 会自动升级使用已弃用 API 版本的旧资源以使用最新版本。例如，如果您在集群中创建了使用旧版本的 **ApiServerSource** API（如 **v1beta1** ）的资源，OpenShift Serverless Operator 会在可用时自动更新这些资源以使用 API 的 **v1** 版本，并弃用 **v1beta1** 版本。

弃用后，可能会在任何即将发布的发行版本中删除旧版本的 API。使用已弃用的 API 版本不会导致资源失败。但是，如果您尝试使用已删除的 API 版本，则会导致资源失败。确保您的清单已更新为使用最新版本以避免出现问题。

1.2. 正式发布（GA）和技术预览（TP）功能

正式发布（GA）的功能被完全支持，并适用于生产环境。技术预览功能为实验性功能，不适用于生产环境。有关 TP 功能的更多信息，请参阅红帽客户门户网站中的技术支持范围。

下表提供了有关哪些 OpenShift Serverless 功能是 GA 以及 TP 的信息：

表 1.1. 正式发布的功能和技术预览功能

功能	1.23	1.24	1.25
kn func	TP	TP	TP
Service Mesh mTLS	GA	GA	GA
emptyDir 卷	GA	GA	GA
HTTPS 重定向	GA	GA	GA
Kafka 代理	TP	TP	GA
Kafka 接收器	TP	TP	GA
init 容器支持 Knative 服务	TP	GA	GA

功能	1.23	1.24	1.25
Knative 服务的 PVC 支持	TP	TP	TP
TLS 用于内部流量	-	-	TP

1.3. 弃用和删除的功能

一些在以前发行本中正式发布 (GA) 或技术预览 (TP) 的功能已被弃用或删除。弃用的功能仍然包含在 OpenShift Serverless 中，并且仍然被支持。但是，弃用的功能可能会在以后的发行版本中被删除，且不建议在新的部署中使用。

有关 OpenShift Serverless 中已弃用并删除的主要功能的最新列表，请参考下表：

表 1.2. 弃用和删除的功能

功能	1.20	1.21	1.22 到 1.25
KafkaBinding API	已弃用	已弃用	删除
kn func emit (**kn func invoke** 在 1.21+ 中)	已弃用	删除	删除

1.4. RELEASE NOTES FOR RED HAT OPENSHIFT SERVERLESS 1.25.0

OpenShift Serverless 1.25.0 现已正式发布。OpenShift Container Platform 上的 OpenShift Serverless 的新功能、改变以及已知的问题包括在此文档中。

1.4.1. 新功能

- OpenShift Serverless 现在使用 Knative Serving 1.4。
- OpenShift Serverless 现在使用 Knative Eventing 1.4。
- OpenShift Serverless 现在使用 Kourier 1.4。
- OpenShift Serverless 现在使用 Knative (**kn**) CLI 1.4。
- OpenShift Serverless 现在使用 Knative Kafka 1.4。
- **kn func** CLI 插件现在使用 **func** 1.7.0。
- 用于创建和部署功能的集成开发环境(IDE)插件现在可用于 Visual Studio Code 和 IntelliJ。
- Knative Kafka 代理现在是 GA。Knative Kafka 代理是 Knative 代理 API 的高性能实现，直接以 Apache Kafka 为目标。
  建议您不要使用 MT-Channel-Broker，而是使用 Knative Kafka 代理。
- Knative Kafka sink 现在为 GA。**KafkaSink** 使用 **CloudEvent**，并将其发送到 Apache Kafka 主题。事件可以在结构化或二进制内容模式中指定。
- 为内部流量启用 TLS 现在作为技术预览提供。

1.4.2. 修复的问题

- 在以前的版本中，如果在存活度探测失败后重启容器，Knative Serving 会出现一个就绪度探测失败。这个问题已被解决。

1.4.3. 已知问题

- Kafka 代理、Kafka 源和 Kafka sink 禁用 Federal Information Processing Standards (FIPS) 模式。
- **SinkBinding** 对象不支持服务的自定义修订名称。

其他资源

- 配置 TLS 身份验证

1.5. RED HAT OPENSHIFT SERVERLESS 1.24.0 发行注记

OpenShift Serverless 1.24.0 现已正式发布。OpenShift Container Platform 上的 OpenShift Serverless 的新功能、改变以及已知的问题包括在此文档中。

1.5.1. 新功能

- OpenShift Serverless 现在使用 Knative Serving 1.3。
- OpenShift Serverless 现在使用 Knative Eventing 1.3。
- OpenShift Serverless 现在使用 Kourier 1.3。
- OpenShift Serverless 现在使用 Knative **kn** CLI 1.3。
- OpenShift Serverless 现在使用 Knative Kafka 1.3。
- **kn func** CLI 插件现在使用 **func** 0.24。
- 现在，提供了对 Knative 服务的 init 容器支持 (GA) 。
- OpenShift Serverless 逻辑现在作为技术预览提供。它启用了定义声明工作流模型来管理无服务器应用程序。
- 现在，您可以在 OpenShift Serverless 中使用成本管理服务。

1.5.2. 修复的问题

- 当集群中存在太多 secret 时，将 OpenShift Serverless 与 Red Hat OpenShift Service Mesh 集成会导致 **net-istio-controller** pod 在启动时耗尽内存。
  现在，可以启用 secret 过滤，这会导致 **net-istio-controller** 只考虑带有 **networking.internal.knative.dev/certificate-uid** 标签的 secret，从而减少所需的内存量。
- OpenShift Serverless 功能技术预览现在默认使用 Cloud Native Buildpacks 构建容器镜像。

1.5.3. 已知问题

- Kafka 代理、Kafka 源和 Kafka sink 禁用 Federal Information Processing Standards (FIPS) 模式。

- 在 OpenShift Serverless 1.23 中，删除了 KafkaBindings 和 **kafka-binding** Webhook 的支持。但是，现有 **kafkabindings.webhook.sources.knative.dev MutatingWebhookConfiguration** 可能保留，指向 **kafka-source-webhook** 服务，该服务不再存在。
  对于集群上 KafkaBindings 的某些规格，**kafkabindings.webhook.kafka.sources.knative.dev MutatingWebhookConfiguration** 可能会被配置，将任何创建和更新事件传递给各种资源，如 Deployment、Knative Services 或 Jobs，然后 Webhook 会失败。

  要临时解决这个问题，请在升级到 OpenShift Serverless 1.23 后从集群中删除 **kafkabindings.webhook.sources.knative.dev MutatingWebhookConfiguration**：

  ```
  $ oc delete mutatingwebhookconfiguration kafkabindings.webhook.kafka.sources.knative.dev
  ```

1.6. RED HAT OPENSHIFT SERVERLESS 1.23.0 发行注记

OpenShift Serverless 1.23.0 现已发布。OpenShift Container Platform 上的 OpenShift Serverless 的新功能、改变以及已知的问题包括在此文档中。

1.6.1. 新功能

- OpenShift Serverless 现在使用 Knative Serving 1.2。
- OpenShift Serverless 现在使用 Knative Eventing 1.2。
- OpenShift Serverless 现在使用 Kourier 1.2。
- OpenShift Serverless 现在使用 Knative (**kn**) CLI 1.2。
- OpenShift Serverless 现在使用 Knative Kafka 1.2。
- **kn func** CLI 插件现在使用 **func** 0.24。
- 现在，可以在 Kafka 代理中使用 **kafka.eventing.knative.dev/external.topic** 注解。此注解可以使用现有的外部管理主题，而不是代理自行创建内部主题。
- **kafka-ch-controller** 和 **kafka-webhook** Kafka 组件不再存在。这些组件已被 **kafka-webhook-eventing** 组件替代。
- OpenShift Serverless 功能技术预览现在默认使用 Source-to-Image (S2I) 来构建容器镜像。

1.6.2. 已知问题

- Kafka 代理、Kafka 源和 Kafka sink 禁用 Federal Information Processing Standards (FIPS) 模式。
- 如果您要删除包括 Kafka 代理的命名空间，当 **auth.secret.ref.name** secret 在代理被删除之前被删除，则命名空间终结器（finalizer）可能无法删除。
- 使用大量 Knative 服务运行 OpenShift Serverless 时，可能会导致运行的 Knativeivator pod 接近其默认的内存限值 600MB。如果使用的内存达到这个限值，则这些 pod 可能会重启。通过修改 **KnativeServing** 自定义资源，可以配置激活器部署的请求和限值：

  ```
  apiVersion: operator.knative.dev/v1alpha1
  kind: KnativeServing
  metadata:
    name: knative-serving
  ```

```
    namespace: knative-serving
spec:
  deployments:
  - name: activator
    resources:
    - container: activator
      requests:
        cpu: 300m
        memory: 60Mi
      limits:
        cpu: 1000m
        memory: 1000Mi
```

- 如果您使用 Cloud Native Buildpacks 作为一个函数的本地构建策略，**kn func** 将无法自动启动 podman，或使用 SSH 隧道到远程守护进程。这些问题的解决方法是，在部署函数前，在本地开发计算机上已在运行 Docker 或 podman 守护进程。
- On-cluster 函数构建当前针对 Quarkus 和 Golang 运行时会失败。它们适用于 Node、Typescript、Python 和 Springboot 运行时。

其他资源

- Source-to-Image

1.7. RED HAT OPENSHIFT SERVERLESS 1.22.0 发行注记

OpenShift Serverless 1.22.0 现已正式发布。OpenShift Container Platform 上的 OpenShift Serverless 的新功能、改变以及已知的问题包括在此文档中。

1.7.1. 新功能

- OpenShift Serverless 现在使用 Knative Serving 1.1。
- OpenShift Serverless 现在使用 Knative Eventing 1.1。
- OpenShift Serverless 现在使用 Kourier 1.1。
- OpenShift Serverless 现在使用 Knative (**kn**) CLI 1.1。
- OpenShift Serverless 现在使用 Knative Kafka 1.1。
- **kn func** CLI 插件现在使用 **func** 0.23。
- init 容器支持 Knative 服务现在作为技术预览提供。
- 持久性卷声明 (PVC) 对 Knative 服务的支持现在作为技术预览提供。
- **knative-serving**、**knative-serving-ingress**、**knative-eventing** 和 **knative-kafka** 系统命名空间现在默认具有 **knative.openshift.io/part-of: "openshift-serverless"** 标签。
- 添加了 **Knative Eventing - Kafka Broker/Trigger** 仪表板，它允许在 web 控制台中视觉化 Kafka 代理并触发指标。
- 添加了 **Knative Eventing - KafkaSink** 仪表板，它允许在 web 控制台中视觉化 KafkaSink 指标。

- **Knative Eventing - Broker/Trigger** 仪表板现在被称为 **Knative Eventing - 基于频道的代理/触发器**。
- **knative.openshift.io/part-of: "openshift-serverless"** 标签已替换 **knative.openshift.io/system-namespace** 标签。
- Knative Serving YAML 配置文件中的命名样式从 camel 格式（**ExampleName**）改为连字符格式 (**example-name**) 。从这个版本开始，在创建或编辑 Knative Serving YAML 配置文件时使用连字符格式表示法。

1.7.2. 已知问题

- Kafka 代理、Kafka 源和 Kafka sink 禁用 Federal Information Processing Standards (FIPS) 模式。

1.8. RED HAT OPENSHIFT SERVERLESS 1.21.0 发行注记

OpenShift Serverless 1.21.0 现已正式发布。OpenShift Container Platform 上的 OpenShift Serverless 的新功能、改变以及已知的问题包括在此文档中。

1.8.1. 新功能

- OpenShift Serverless 现在使用 Knative Serving 1.0
- OpenShift Serverless 现在使用 Knative Eventing 1.0。
- OpenShift Serverless 现在使用 Kourier 1.0.
- OpenShift Serverless 现在使用 Knative (**kn**) CLI 1.0。
- OpenShift Serverless 现在使用 Knative Kafka 1.0。
- **kn func** CLI 插件现在使用 **func** 0.21。
- Kafka sink 现在作为技术预览提供。
- Knative 开源项目已开始弃用发现配置密钥，而是统一使用 kebab-cased 键。因此，OpenShift Serverless 1.18.0 发行注记中提到的 **defaultExternalScheme** 键现已弃用，并被 **default-external-scheme** 键替代。键的使用说明保持不变。

1.8.2. 修复的问题

- 在 OpenShift Serverless 1.20.0 中，存在一个与发送事件相关的问题，会影响使用 **kn event send** 向服务发送事件。这个问题现已解决。
- 在 OpenShift Serverless 1.20.0 (**func** 0.20) 中，使用 **http** 模板创建的 TypeScript 功能无法在集群中部署。这个问题现已解决。
- 在 OpenShift Serverless 1.20.0 (**func** 0.20) 中，使用 **gcr.io** registry 部署功能会失败，并出现错误。这个问题现已解决。
- 在 OpenShift Serverless 1.20.0 (**func** 0.20) 中，使用 **kn func create** 命令创建 Springboot 功能项目目录，然后运行 **kn func build** 命令失败并显示错误消息。这个问题现已解决。
- 在 OpenShift Serverless 1.19.0 (**func** 0.19) 中，一些运行时无法使用 podman 来构建功能。这个问题现已解决。

1.8.3. 已知问题

- 目前，域映射控制器无法处理包含当前不支持的路径的代理 URI。
这意味着，如果要使用 **DomainMapping** 自定义资源 (CR) 将自定义域映射到代理，则必须使用代理的 ingress 服务配置 **DomainMapping** CR，并将代理的确切路径附加到自定义域：

  DomainMapping CR 示例

```
apiVersion: serving.knative.dev/v1alpha1
kind: DomainMapping
metadata:
  name: <domain-name>
  namespace: knative-eventing
spec:
  ref:
    name: broker-ingress
    kind: Service
    apiVersion: v1
```

  代理的 URI 为 **<domain-name>/<broker-namespace>/<broker-name>**。

1.9. RED HAT OPENSHIFT SERVERLESS 1.20.0 发行注记

OpenShift Serverless 1.20.0 现已正式发布。OpenShift Container Platform 上的 OpenShift Serverless 的新功能、改变以及已知的问题包括在此文档中。

1.9.1. 新功能

- OpenShift Serverless 现在使用 Knative Serving 0.26。
- OpenShift Serverless 现在使用 Knative Eventing 0.26。
- OpenShift Serverless 现在使用 Kourier 0.26。
- OpenShift Serverless 现在使用 Knative (**kn**) CLI 0.26。
- OpenShift Serverless 现在使用 Knative Kafka 0.26。
- **kn func** CLI 插件现在使用 **func** 0.20。
- Kafka 代理现在作为技术预览提供。

  重要

  FIPS 不支持 Kafka 代理（当前为技术预览）。

- **kn event** 插件现在作为技术预览提供。
- **kn service create** 命令的 **--min-scale** 和 **--max-scale** 标志已弃用。使用 **--scale-min** 和 **--scale-max** 标志。

1.9.2. 已知问题

- OpenShift Serverless 使用 HTTPS 的默认地址部署 Knative 服务。将事件发送到集群中的资源时，发件人不会配置集群证书颁发机构 (CA) 。这会导致事件交付失败，除非集群使用全局接受的证书。
  例如，向公开访问的地址发送事件可正常工作：

  ```
  $ kn event send --to-url https://ce-api.foo.example.com/
  ```

  另一方面，如果服务使用由自定义 CA 发布的 HTTPS 证书的公共地址，则此交付会失败：

  ```
  $ kn event send --to Service:serving.knative.dev/v1:event-display
  ```

  将事件发送到其他可寻址的对象（如代理或频道）不受此问题的影响，并可以正常工作。

- Kafka 代理目前无法在启用了联邦信息处理标准 (FIPS) 模式的集群中工作。

- 如果您使用 **kn func create** 命令创建 Springboot 功能项目目录，后续的 **kn func build** 命令运行会失败并显示以下错误消息：

  ```
  [analyzer] no stack metadata found at path ''
  [analyzer] ERROR: failed to : set API for buildpack 'paketo-buildpacks/ca-certificates@3.0.2': buildpack API version '0.7' is incompatible with the lifecycle
  ```

  作为临时解决方案，您可以在函数配置文件 **func.yaml** 中将 **builder** 属性更改为 **gcr.io/paketo-buildpacks/builder:base**。

- 使用 **gcr.io** registry 部署函数会失败并显示以下错误消息：

  ```
  Error: failed to get credentials: failed to verify credentials: status code: 404
  ```

  作为临时解决方案，请使用与 **gcr.io** 不同的 registry，如 **quay.io** 或 **docker.io**。

- 使用 **http** 模板创建的 TypeScript 函数无法在集群中部署。
  作为临时解决方案，在 **func.yaml** 文件中替换以下部分：

  ```
  buildEnvs: []
  ```

  使用这个：

  ```
  buildEnvs:
  - name: BP_NODE_RUN_SCRIPTS
    value: build
  ```

- 在 **func** 版本 0.20 中，一些运行时可能无法使用 podman 构建函数。您可能会看到类似如下的错误消息：

  ```
  ERROR: failed to image: error during connect: Get "http://%2Fvar%2Frun%2Fdocker.sock/v1.40/info": EOF
  ```

  - 这个问题存在以下临时解决方案：

    a. 通过在 service **ExecStart** 定义中添加 **--time=0** 来更新 podman 服务：

    服务配置示例

```
ExecStart=/usr/bin/podman $LOGGING system service --time=0
```

b. 运行以下命令来重启 podman 服务：

```
$ systemctl --user daemon-reload
```

```
$ systemctl restart --user podman.socket
```

- 或者，您可以使用 TCP 公开 podman API：

```
$ podman system service --time=0 tcp:127.0.0.1:5534 &
export DOCKER_HOST=tcp://127.0.0.1:5534
```

1.10. RED HAT OPENSHIFT SERVERLESS 1.19.0 发行注记

OpenShift Serverless 1.19.0 现已正式发布。OpenShift Container Platform 上的 OpenShift Serverless 的新功能、改变以及已知的问题包括在此文档中。

1.10.1. 新功能

- OpenShift Serverless 现在使用 Knative Serving 0.25。
- OpenShift Serverless 现在使用 Knative Eventing 0.25。
- OpenShift Serverless 现在使用 Kourier 0.25。
- OpenShift Serverless 现在使用 Knative (**kn**) CLI 0.25。
- OpenShift Serverless 现在使用 Knative Kafka 0.25。
- **kn func** CLI 插件现在使用 **purc** 0.19。
- **KafkaBinding** API 在 OpenShift Serverless 1.19.0 中已弃用，并将在以后的发行版本中删除。
- HTTPS 重定向现在被支持，并可以为集群或每个 Knative 服务配置。

1.10.2. 修复的问题

- 在以前的版本中，Kafka 频道分配程序仅在响应前等待本地提交成功，这可能会在 Apache Kafka 节点失败时导致事件丢失。Kafka 频道分配程序现在会在响应前等待所有同步副本提交。

1.10.3. 已知问题

- 在 **func** 版本 0.19 中，一些运行时可能无法使用 podman 构建函数。您可能会看到类似如下的错误消息：

```
ERROR: failed to image: error during connect: Get
"http://%2Fvar%2Frun%2Fdocker.sock/v1.40/info": EOF
```

- 这个问题存在以下临时解决方案：

a. 通过在 service **ExecStart** 定义中添加 **--time=0** 来更新 podman 服务：

服务配置示例

```
ExecStart=/usr/bin/podman $LOGGING system service --time=0
```

b. 运行以下命令来重启 podman 服务：

```
$ systemctl --user daemon-reload
```

```
$ systemctl restart --user podman.socket
```

- 或者，您可以使用 TCP 公开 podman API：

```
$ podman system service --time=0 tcp:127.0.0.1:5534 &
export DOCKER_HOST=tcp://127.0.0.1:5534
```

1.11. RED HAT OPENSHIFT SERVERLESS 1.18.0 发行注记

OpenShift Serverless 1.18.0 现已正式发布。OpenShift Container Platform 上的 OpenShift Serverless 的新功能、改变以及已知的问题包括在此文档中。

1.11.1. 新功能

- OpenShift Serverless 现在使用 Knative Serving 0.24.0。
- OpenShift Serverless 现在使用 Knative Eventing 0.24.0。
- OpenShift Serverless 现在使用 Kourier 0.24.0。
- OpenShift Serverless 现在使用 Knative (**kn**) CLI 0.24.0。
- OpenShift Serverless 现在使用 Knative Kafka 0.24.7。
- **kn func** CLI 插件现在使用 **func** 0.18.0。
- 在即将发布的 OpenShift Serverless 1.19.0 发行版本中，外部路由的 URL 方案将默认为 HTTPS 以增强安全性。
  如果您不希望此更改应用到工作负载，您可以在升级到 1.19.0 前覆盖默认设置，方法是将以下 YAML 添加到 **KnativeServing** 自定义资源 (CR) ：

```
...
spec:
  config:
    network:
      defaultExternalScheme: "http"
...
```

如果您想在 1.18.0 中应用更改，请添加以下 YAML：

```
...
spec:
  config:
```

```
    network:
      defaultExternalScheme: "https"
...
```

- 在接下来的 OpenShift Serverless 1.19.0 发行版本中，公开 Kourier 网关的默认服务类型将是 **ClusterIP**，而不是 **LoadBalancer**。
  如果您不希望此更改应用到工作负载，您可以在升级到 1.19.0 前覆盖默认设置，方法是将以下 YAML 添加到 **KnativeServing** 自定义资源 (CR)：

```
...
spec:
  ingress:
    kourier:
      service-type: LoadBalancer
...
```

- 现在，您可以在 OpenShift Serverless 中使用 **emptyDir** 卷。详情请参阅 OpenShift Serverless 文档中的 Knative Serving 文档。

- 现在，当您使用 **kn func** 创建函数时，可以使用 Rust 模板。

1.11.2. 修复的问题

- 1.4 之前的 Camel-K 版本与 OpenShift Serverless 1.17.0 不兼容。Camel-K 中的问题已被解决，Camel-K 版本 1.4.1 可以用于 OpenShift Serverless 1.17.0。

- 在以前的版本中，如果您为 Kafka 频道或新 Kafka 源创建新订阅，则 Kafka 数据平面可能会在新创建的订阅或 sink 报告就绪状态后准备好发送信息。
  因此，数据平面没有报告就绪状态时发送的信息可能没有传送到订阅者或 sink。

  在 OpenShift Serverless 1.18.0 中，这个问题已被解决，初始消息不再丢失。有关此问题的更多信息，请参阅 知识库文章 #6343981。

1.11.3. 已知问题

- 较旧版本的 Knative **kn** CLI 可能会使用较旧版本的 Knative Serving 和 Knative Eventing API。例如，**kn** CLI 版本 0.23.2 使用 **v1alpha1** API 版本。
  另一方面，较新的 OpenShift Serverless 发行版本可能不再支持旧的 API 版本。例如，OpenShift Serverless 1.18.0 不再支持 **kafkasources.sources.knative.dev API** 的版本 **v1alpha1**。

  因此，使用带有较新的 OpenShift Serverless 的 Knative **kn** CLI 的旧版本可能会产生错误，因为 **kn** 无法找到过时的 API。例如，kn **CLI** 的 0.23.2 版本无法用于 OpenShift Serverless 1.18.0。

  为避免出现问题，请使用适用于 OpenShift Serverless 发行版本的最新 **kn** CLI 版本。对于 OpenShift Serverless 1.18.0，使用 Knative **kn** CLI 0.24.0。

1.12. RED HAT OPENSHIFT SERVERLESS 1.17.0 发行注记

OpenShift Serverless 1.17.0 现已正式发布。OpenShift Container Platform 上的 OpenShift Serverless 的新功能、改变以及已知的问题包括在此文档中。

1.12.1. 新功能

- OpenShift Serverless 现在使用 Knative Serving 0.23.0。
- OpenShift Serverless 现在使用 Knative Eventing 0.23.0。
- OpenShift Serverless 现在使用 Kourier 0.23.0。
- OpenShift Serverless 现在使用 Knative **kn** CLI 0.23.0。
- OpenShift Serverless 现在使用 Knative Kafka 0.23.0。
- **kn func** CLI 插件现在使用 **func** 0.17.0.
- 在即将发布的 OpenShift Serverless 1.19.0 发行版本中，外部路由的 URL 方案将默认为 HTTPS 以增强安全性。
  如果您不希望此更改应用到工作负载，您可以在升级到 1.19.0 前覆盖默认设置，方法是将以下 YAML 添加到 **KnativeServing** 自定义资源(CR)：

```
...
spec:
  config:
    network:
      defaultExternalScheme: "http"
...
```

- mTLS 功能现在正式发布 (GA)。
- 现在，在使用 **kn func** 创建函数时，typeScript 模板可用。
- Knative Eventing 0.23.0 中的 API 版本更改：
  - **KafkaChannel** API 的 **v1alpha1** 版本已在 OpenShift Serverless 版本 1.14.0 中弃用，它已被删除。如果配置映射的 **ChannelTemplateSpec** 参数包含对此旧版本的引用，您必须更新 spec 的这一部分以使用正确的 API 版本。

1.12.2. 已知问题

- 如果您试图将较旧版本的 Knative **kn** CLI 与较新的 OpenShift Serverless 发行版本搭配使用，则不会找到 API，并出现错误。
  例如，如果您使用 **kn** CLI 的 1.16.0 发行版本，它使用版本 0.22.0，其 1.17.0 OpenShift Serverless 发行版本使用 Knative Serving 和 Knative Eventing API 的 0.23.0 版本，则 CLI 无法正常工作，因为它仍然会查找过时的 0.22.0 API 版本。

  确保您使用 OpenShift Serverless 发行版本的最新 **kn** CLI 版本来避免问题。
- 此发行版本中的相应 web 控制台仪表板中不会监控或显示 Kafka 频道指标。这是因为 Kafka 分配程序协调过程中的中断更改。
- 如果您为 Kafka 频道或新 Kafka 源创建新订阅，在新创建的订阅或 sink 报告就绪状态后 Kafka 数据平面可能会延迟分配信息。
  因此，在 data plane 没有报告就绪状态时发送的信息可能无法传送到订阅者或 sink。

  有关此问题和可能的解决方案的更多信息，请参阅知识库文章 #6343981。
- Camel-K 1.4 发行版本与 OpenShift Serverless 版本 1.17.0 不兼容。这是因为 Camel-K 1.4 使用 Knative 版本 0.23.0 中删除的 API。目前，这个问题还没有可用的临时解决方案。如果您需要在 OpenShift Serverless 中使用 Camel-K 1.4，请不要升级到 OpenShift Serverless 版本 1.17.0。

注意

这个问题已被解决，Camel-K 版本 1.4.1 与 OpenShift Serverless 1.17.0 兼容。

1.13. RED HAT OPENSHIFT SERVERLESS 1.16.0 发行注记

OpenShift Serverless 1.16.0 现已正式发布。OpenShift Container Platform 上的 OpenShift Serverless 的新功能、改变以及已知的问题包括在此文档中。

1.13.1. 新功能

- OpenShift Serverless 现在使用 Knative Serving 0.22.0。
- OpenShift Serverless 现在使用 Knative Eventing 0.22.0。
- OpenShift Serverless 现在使用 Kourier 0.22.0。
- OpenShift Serverless 现在使用 Knative **kn** CLI 0.22.0。
- OpenShift Serverless 现在使用 Knative Kafka 0.22.0。
- **kn func** CLI 插件现在使用 **func** 0.16.0。
- **kn func emit** 命令已添加到功能 **kn** 插件中。您可以使用此命令发送事件来测试本地部署的功能。

1.13.2. 已知问题

- 在升级到 OpenShift Serverless 1.16.0 前，您必须将 OpenShift Container Platform 升级到 4.6.30、4.7.11 或更高版本。
- AMQ Streams Operator 可能会阻止安装或升级 OpenShift Serverless Operator。如果发生这种情况，Operator Lifecycle Manager (OLM) 会抛出以下错误：

  ```
  WARNING: found multiple channel heads: [amqstreams.v1.7.2 amqstreams.v1.6.2], please check the `replaces`/`skipRange` fields of the operator bundles.
  ```

  您可以在安装或升级 OpenShift Serverless Operator 前卸载 AMQ Streams Operator，从而解决这个问题。然后您可以重新安装 AMQ Streams Operator。

- 如果使用 mTLS 启用 Service Mesh，则 Knative Serving 的指标会被默认禁用，因为 Service Mesh 会防止 Prometheus 提取指标。有关启用 Knative Serving 指标以用于 Service Mesh 和 mTLS 的说明，请参阅 Serverless 文档中的"集成 Service Mesh with OpenShift Serverless"部分。
- 如果您部署启用了 Istio ingress 的 Service Mesh CR，您可能会在 **istio-ingressgateway** pod 中看到以下警告：

  ```
  2021-05-02T12:56:17.700398Z warning envoy config [external/envoy/source/common/config/grpc_subscription_impl.cc:101] gRPC config for type.googleapis.com/envoy.api.v2.Listener rejected: Error adding/updating listener(s) 0.0.0.0_8081: duplicate listener 0.0.0.0_8081 found
  ```

  您的 Knative 服务也可能无法访问。

您可以按照以下的临时解决方案，通过重新创建 **knative-local-gateway** 服务来解决这个问题：

a. 删除 **istio-system** 命名空间中现有的 **knative-local-gateway** 服务：

```
$ oc delete services -n istio-system knative-local-gateway
```

b. 创建并应用包含以下 YAML 的 **knative-local-gateway** 服务：

```
apiVersion: v1
kind: Service
metadata:
 name: knative-local-gateway
 namespace: istio-system
 labels:
   experimental.istio.io/disable-gateway-port-translation: "true"
spec:
 type: ClusterIP
 selector:
   istio: ingressgateway
 ports:
   - name: http2
     port: 80
     targetPort: 8081
```

- 如果您在集群中有 1000 个 Knative 服务，然后执行 Knative Serving 的重新安装或升级，则在 **KnativeServing** 自定义资源(CR)变为 **Ready** 后创建第一个新服务会有一个延迟。**3scale-kourier-control** 服务会在创建新服务前协调所有以前存在的 Knative 服务。因此，新服务在状态被更新为 **Ready** 前，大约会有 800 秒的时间处于 **IngressNotConfigured** 或 **Unknown** 状态。

- 如果您为 Kafka 频道或新 Kafka 源创建新订阅，在新创建的订阅或 sink 报告就绪状态后 Kafka 数据平面可能会延迟分配信息。
因此，在 data plane 没有报告就绪状态时发送的信息可能无法传送到订阅者或 sink。

  有关此问题和可能的解决方案的更多信息，请参阅知识库文章 #6343981。

1.14. RED HAT OPENSHIFT SERVERLESS 1.15.0 发行注记

OpenShift Serverless 1.15.0 现已正式发布。OpenShift Container Platform 上的 OpenShift Serverless 的新功能、改变以及已知的问题包括在此文档中。

1.14.1. 新功能

- OpenShift Serverless 现在使用 Knative Serving 0.21.0。
- OpenShift Serverless 现在使用 Knative Eventing 0.21.0。
- OpenShift Serverless 现在使用 Kourier 0.21.0。
- OpenShift Serverless 现在使用 Knative **kn** CLI 0.21.0。
- OpenShift Serverless 现在使用 Knative Kafka 0.21.1。
- OpenShift Serverless 功能现在作为技术预览提供。

重要

service.knative.dev/visibility 标签（以前用于创建私有服务）现已弃用。您必须更新现有服务来改用 **networking.knative.dev/visibility** 标签。

1.14.2. 已知问题

- 如果您为 Kafka 频道或新 Kafka 源创建新订阅，在新创建的订阅或 sink 报告就绪状态后 Kafka 数据平面可能会延迟分配信息。
  因此，在 data plane 没有报告就绪状态时发送的信息可能无法传送到订阅者或 sink。

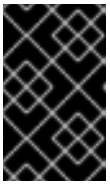
  有关此问题和可能的解决方案的更多信息，请参阅知识库文章 #6343981。

1.15. RED HAT OPENSHIFT SERVERLESS 1.14.0 发行注记

OpenShift Serverless 1.14.0 现已正式发布。OpenShift Container Platform 上的 OpenShift Serverless 的新功能、改变以及已知的问题包括在此文档中。

1.15.1. 新功能

- OpenShift Serverless 现在使用 Knative Serving 0.20.0。
- OpenShift Serverless 使用 Knative Eventing 0.20.0。
- OpenShift Serverless 现在使用 Kourier 0.20.0。
- OpenShift Serverless 现在使用 Knative **kn** CLI 0.20.0。
- OpenShift Serverless 现在使用 Knative Kafka 0.20.0。
- OpenShift Serverless 上的 Knative Kafka 现已正式发布（GA）。



重要

仅支持 OpenShift Serverless 上的 **KafkaChannel** 和 **KafkaSource** 对象的 **v1beta1** API 版本。不要使用这些 API 的 **v1alpha1** 版本，因为这个版本现已弃用。

- 对于 OpenShift Container Platform 4.6 及更新的版本，用于安装和升级 OpenShift Serverless 的 Operator 频道已更新为 **stable**。
- OpenShift Serverless 现在在 IBM Power Systems、IBM Z 和 LinuxONE 上被支持，但以下功能除外，这些功能不被支持：
  - Knative Kafka 功能。
  - OpenShift Serverless 功能开发人员预览。

1.15.2. 已知问题

- Kafka 频道的订阅有时无法标记为 **READY**，并处于 **SubscriptionNotMarkedReadyByChannel** 状态。您可以通过重启 Kafka 频道的分配程序来解决这个问题。
- 如果您为 Kafka 频道或新 Kafka 源创建新订阅，在新创建的订阅或 sink 报告就绪状态后 Kafka 数据平面可能会延迟分配信息。

因此，在 data plane 没有报告就绪状态时发送的信息可能无法传送到订阅者或 sink。

有关此问题和可能的解决方案的更多信息，请参阅知识库文章 #6343981。

第 2 章 发现

2.1. 关于 OPENSHIFT SERVERLESS

OpenShift Serverless 提供 Kubernetes 原生构建块，供开发人员在 OpenShift Container Platform 中创建和部署无服务器、事件驱动的应用程序。OpenShift Serverless 基于开源 Knative 项目，通过启用企业级无服务器平台为混合和多云环境提供可移植性和一致性。

2.1.1. Knative Serving

Knative Serving 可以帮助需要创建、部署和管理云原生应用程序的开发人员。它以 Kubernetes 自定义资源定义 (CRD) 的形式提供一组对象，用于定义和控制 OpenShift Container Platform 集群上无服务器工作负载的行为。

开发人员使用这些 CRD 创建自定义资源（CR）实例，这些实例可作为构建块用于处理复杂用例。例如：

- 快速部署无服务器容器。
- 自动缩放 pod。

2.1.1.1. Knative Serving 资源

服务

service.serving.knative.dev CRD 会自动管理工作负载的生命周期，以确保应用程序通过网络部署并可访问。每次用户创建的服务或自定义资源发生变化时，它都会创建一个路由、配置和新修订。Knative 中进行的大多数开发人员交互都是通过修改服务进行的。

Revision（修订）

revision.serving.knative.dev CRD 是每次对工作负载进行修改所涉及代码和配置的时间点快照。所有修订均为不可变对象，可以根据需要保留。

Route（路由）

route.serving.knative.dev CRD 可将网络端点映射到一个或多个修订。您可通过多种方式管理流量，包括部分流量和指定路由。

Configuration（配置）

configuration.serving.knative.dev CRD 可保持部署所需状态。它可在使编程过程和配置配置过程相互分离。修改配置则会创建一个新修订。

2.1.2. Knative Eventing

OpenShift Container Platform 上的 Knative Eventing 可让开发人员使用 事件驱动的架构和无服务器应用程序。事件驱动的体系结构是基于事件和事件用户间分离关系的概念。

事件生成者创建事件，事件 *sink*、或消费者接收事件。Knative Eventing 使用标准 HTTP POST 请求来发送和接收事件制作者和 sink 之间的事件。这些事件符合 CloudEvents 规范，它允许在任何编程语言中创建、解析、发送和接收事件。

Knative Eventing 支持以下用例：

在不创建消费者的情况下发布事件

您可以将事件作为 HTTP POST 发送到代理，并使用绑定分离生成事件的应用程序的目标配置。

在不创建发布程序的情况下消费事件

您可以使用 Trigger 来根据事件属性消费来自代理的事件。应用程序以 HTTP POST 的形式接收事件。

要启用多种 sink 类型的交付，Knative Eventing 会定义以下通用接口，这些接口可由多个 Kubernetes 资源实现：

可寻址的资源

能够接收和确认通过 HTTP 发送的事件到 Event 的 **status.address.url** 字段中定义的地址。Kubernetes **Service** 资源也满足可寻址的接口。

可调用的资源

能够通过 HTTP 接收事件并转换它，并在 HTTP 响应有效负载中返回 **0** 或 **1** 新事件。这些返回的事件可能会象处理外部事件源中的事件一样进一步处理。

您可以使用以下方法将事件从事件源传播到多个事件接收器：

- 频道和订阅，或者
- 代理和触发器。

2.1.3. 支持的配置

OpenShift Serverless 支持的功能、配置和集成（当前和过去的版本）包括在 支持的配置页面中。

2.1.4. 可伸缩性和性能

OpenShift Serverless 已使用配置为 3 个主要节点和 3 个 worker 节点进行测试，每个节点都有 64 个 CPU、457 GB 内存和 394 GB 存储。

使用此配置创建的最大 Knative 服务数为 3,000。这与 OpenShift Container Platform Kubernetes 服务限制 10,000 对应，因为 1 个 Knative 服务创建 3 个 Kubernetes 服务。

零响应时间的平均缩放约为 3.4 秒，最大响应时间为 8 秒，而一个简单 Quarkus 应用程序使用 99.9thile 的 4.5 秒。这些时间可能因应用程序和应用程序的运行时的不同而有所不同。

2.1.5. 其他资源

- 使用自定义资源定义来扩展 Kubernetes API
- 管理自定义资源定义中的资源
- 什么是无服务器？

2.2. 关于 OPENSHIFT SERVERLESS FUNCTIONS

OpenShift Serverless Functions 帮助开发人员在 OpenShift Container Platform 上创建和部署无状态、事件驱动的函数，作为 Knative 服务。**kn func** CLI 作为 Knative **kn** CLI 的插件提供。您可以使用 **kn func** CLI 在集群中创建、构建和部署容器镜像作为 Knative 服务。

重要

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

有关红帽技术预览功能支持范围的详情，请参考 https://access.redhat.com/support/offerings/techpreview/。

2.2.1. 包括的运行时

OpenShift Serverless Functions 提供了一组模板，可用于为以下运行时创建基本函数：

- Node.js
- Python
- Go
- Quarkus
- TypeScript

2.2.2. 后续步骤

- 函数入门。

2.3. 事件源

Knative *事件源*可以是生成或导入云事件的任何 Kubernetes 对象，并将这些事件转发到另一个端点，称为*接收器（sink）*。事件源对于开发对事件做出反应的分布式系统至关重要。

您可以使用 OpenShift Container Platform Web 控制台、Knative (**kn**) CLI 或应用 YAML 文件的 **Developer** 视角创建和管理 Knative 事件源。

目前，OpenShift Serverless 支持以下事件源类型：

API 服务器源

将 Kubernetes API 服务器事件引入 Knative。每次创建、更新或删除 Kubernetes 资源时，API 服务器源会发送一个新事件。

Ping 源

根据指定的 cron 计划生成带有固定有效负载的事件。

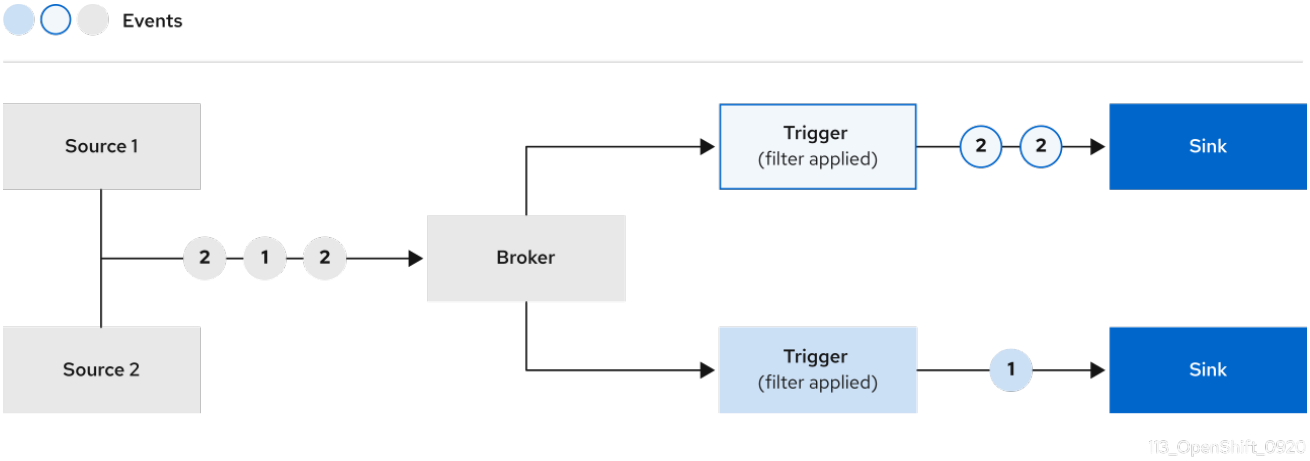
Kafka 事件源

将 Kafka 集群连接到接收器作为事件源。

您还可以创建自定义事件源。

2.4. 代理（BROKER）

代理可与触发器结合使用，用于将事件源发送到事件 sink。事件从事件源发送到代理，作为 HTTP **POST** 请求。事件进入代理后，可使用触发器根据 CloudEvent 属性 进行过滤，并作为 HTTP **POST** 请求发送到事件 sink。



2.4.1. 代理类型

集群管理员可为集群设置默认代理实施。创建代理时，会使用默认代理实现，除非在 **Broker** 对象中提供配置。

2.4.1.1. 用于开发的默认代理实现

Knative 提供基于频道的默认代理实现。这个基于频道的代理可用于开发和测试目的，但不为生产环境提供适当的事件交付保证。默认代理由 **InMemoryChannel** 频道实现支持。

如果要使用 Kafka 降低网络跃点，请使用 Kafka 代理实现。不要将基于频道的代理配置为由 **KafkaChannel** 频道实现支持。

2.4.1.2. 生产环境就绪的 Kafka 代理实现

对于生产环境就绪的 Knative Eventing 部署，红帽建议您使用 Knative Kafka 代理实现。Kafka 代理是 Knative 代理的 Apache Kafka 原生实现，它将 CloudEvents 直接发送到 Kafka 实例。

重要

Kafka 代理禁用联邦信息处理标准 (FIPS) 模式。

Kafka 代理具有与 Kafka 的原生集成，用于存储和路由事件。它可以更好地与 Kafka 集成用于代理，并在其他代理类型中触发模型，并减少网络跃点。Kafka 代理实现的其他优点包括：

- 最少一次的交付保证
- 根据 CloudEvents 分区扩展排序事件交付
- 数据平面的高可用性
- 水平扩展数据平面

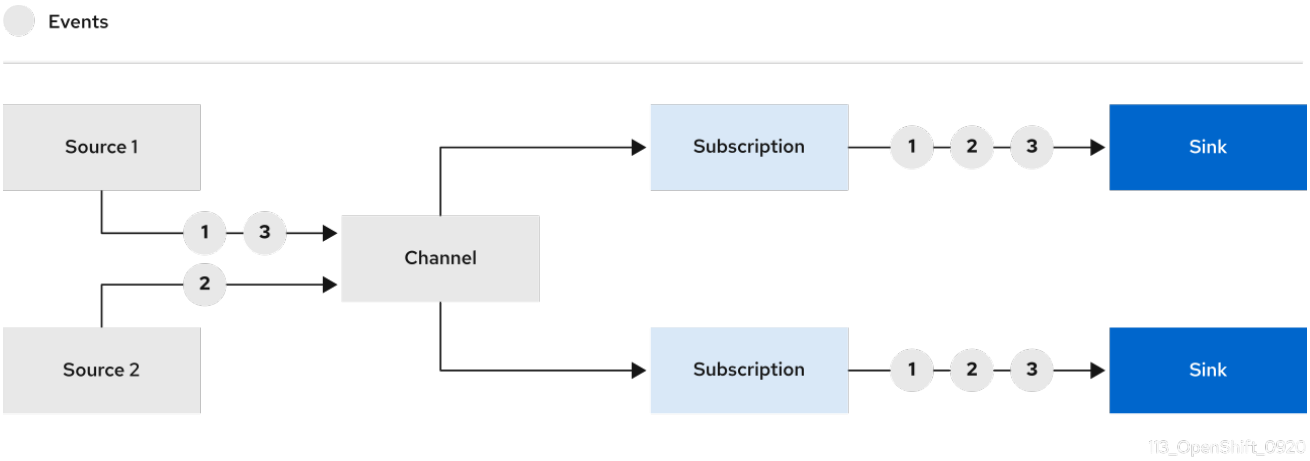
Knative Kafka 代理使用二进制内容模式将传入的 CloudEvents 存储为 Kafka 记录。这意味着，所有 CloudEvent 属性和扩展都会在 Kafka 记录上映射，而 CloudEvent 的 **data** 规格与 Kafka 记录的值对应。

2.4.2. 后续步骤

- 创建代理

2.5. 频道和订阅

频道是定义单一事件转发和持久层的自定义资源。事件源或生成程序在将事件发送到频道后，可使用订阅将这些事件发送到多个 Knative 服务或其他 sink。

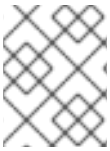


您可以通过实例化受支持的 **Channel** 对象来创建频道，并通过修改 **Subscription** 对象中的 **delivery** 规格来配置重新发送尝试。

创建 **Channel** 对象后，根据默认频道实现，一个经过更改的准入 Webhook 会为 **Channel** 对象添加一组 **spec.channelTemplate** 属性。例如，对于 **InMemoryChannel** 默认实现，**Channel** 对象如下所示：

```
apiVersion: messaging.knative.dev/v1
kind: Channel
metadata:
  name: example-channel
  namespace: default
spec:
  channelTemplate:
    apiVersion: messaging.knative.dev/v1
    kind: InMemoryChannel
```

然后，频道控制器将根据这个 **spec.channelTemplate** 配置创建后备频道实例。



注意

创建后，**spec.channelTemplate** 属性将无法更改，因为它们由默认频道机制设置，而不是由用户设置。

当此机制与上例搭配使用时，会创建两个对象：一个通用的后备频道和一个 **InMemoryChannel** 频道。如果您使用不同的默认频道实现，使用特定于您的实现的频道替换 **InMemoryChannel**。例如，在 Knative Kafka 中，创建 **KafkaChannel** 频道。

后备频道充当将订阅复制到用户创建的频道对象的代理，并设置用户创建的频道对象状态来反映后备频道的状态。

2.5.1. 频道实现类型

InMemoryChannel 和 **KafkaChannel** 频道实现可用于 OpenShift Serverless 进行开发。

以下是 **InMemoryChannel** 类型频道的限制：

- 事件没有持久性。如果 Pod 停机，则 Pod 上的事件将会丢失。
- **InMemoryChannel** 频道没有实现事件排序，因此同时接收到的两个事件可能会以任何顺序传送给订阅者。
- 如果订阅者拒绝某个事件，则不会默认重新发送尝试。您可以通过修改 **Subscription** 对象中的 **delivery** 规格来配置重新发送尝试。

有关 Kafka 频道的更多信息，请参阅 Knative Kafka 文档。

2.5.2. 后续步骤

- 创建一个频道。
- 如果您是集群管理员，可以配置频道的默认设置。请参阅配置频道默认设置。

第 3 章 安装

3.1. 安装 OPENSHIFT SERVERLESS OPERATOR

安装 OpenShift Serverless Operator 后，您就可以在 OpenShift Container Platform 集群中安装和使用 Knative Serving、Knative Eventing 和 Knative Kafka。OpenShift Serverless Operator 管理集群的 Knative 自定义资源定义 (CRD)，并可让您在不直接为每个组件修改单个配置映射的情况下配置它们。

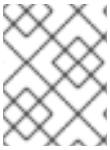
3.1.1. 开始前

在安装 OpenShift Serverless 前，请阅读以下有关支持的配置和先决条件的信息。

- OpenShift Serverless 支持在受限网络环境中安装。
- OpenShift Serverless 目前无法在单个集群的多租户配置中使用。

3.1.1.1. 定义集群大小要求

要安装和使用 OpenShift Serverless，OpenShift Container Platform 集群必须正确定义大小。运行 OpenShift Serverless 的总大小要求取决于安装的组件以及部署的应用程序，并因部署的不同而有所不同。



注意

以下要求仅与 OpenShift Container Platform 集群的 worker 机器池相关。control plane 节点不用于常规调度，它不在要求中。

默认情况下，每个 pod 请求大约休 400m CPU，因此最低要求会基于此值。降低应用程序的实际 CPU 请求可增加可能的副本数。

如果您在集群中启用了高可用性 (HA)，需要 0.5 - 1.5 个内核，每个 Knative Serving control plane 副本需要 200MB 到 2GB 内存。

3.1.1.2. 使用机器集扩展集群

您可以使用 OpenShift Container Platform **MachineSet** API 手动将集群扩展至所需大小。最低要求通常意味着，您需要将一个默认机器集进行扩展，增加两个额外的机器。请参阅手动扩展机器集。

3.1.2. 安装 OpenShift Serverless Operator

您可以使用 OpenShift Container Platform Web 控制台从 OperatorHub 安装 OpenShift Serverless Operator。安装此 Operator 可让您安装和使用 Knative 组件。

先决条件

- 您可以访问具有集群管理员权限的 OpenShift Container Platform 帐户。
- 已登陆到 OpenShift Container Platform Web 控制台。

流程

1. 在 OpenShift Container Platform web 控制台中进入到 **Operators → OperatorHub** 页。

2. 滚动页面，或在 **Filter by keyword** 框中输入关键字 **Serverless** 来查找 OpenShift Serverless Operator。

3. 查看 Operator 信息并单击 **Install**。

4. 在 **Install Operator** 页面中：

   a. **Installation Mode** 是 **All namespaces on the cluster (default)**。此模式将 Operator 安装至默认 **openshift-serverless** 命名空间，以便供集群中的所有命名空间监视和使用。

   b. **安装的命名空间** 是 **openshift-serverless**。

   c. 选择 **stable** 频道作为 **更新频道**。**stable** 频道将启用 OpenShift Serverless Operator 最新稳定版本的安装。

   d. 选择 **Automatic** 或 **Manual** 批准策略。

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

6. 在 **Catalog → Operator Management** 页面中，您可以监控 OpenShift Serverless Operator 订阅的安装和升级进度。

   a. 如果选择了 **Manual** 批准策略，订阅的升级状态将会一直保持在 **Upgrading**，直到您审阅并批准了它的安装计划。在 **Install Plan** 页面批准后，订阅的升级状态将变为 **Up to date**。

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

验证

当订阅的升级状态变为**Up to date** 后，选择 **Catalog → Installed Operators** 来验证 OpenShift Serverless Operator 最终出现，它的 **Status** 在相关的命名空间中最终会变为 **InstallSucceeded**。

如果没有：

1. 切换到 **Catalog → Operator Management** 页，检查 **Operator Subscriptions** 和 **Install Plans** 页中的 **Status** 是否有错误。

2. 检查 **Workloads → Pods** 页中提供的关于**openshift-serverless** 项目中的 pod 的日志信息，以便进一步排除故障。

重要

如果要在 OpenShift Serverless 中使用 Red Hat OpenShift distributed tracing，则必须在安装 Knative Serving 或 Knative Eventing 前安装和配置 Red Hat OpenShift distributed tracing。

3.1.3. 其他资源

- 在受限网络中使用 Operator Lifecycle Manager
- 在 OpenShift Serverless 中配置高可用性副本

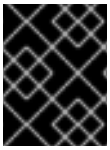
3.1.4. 后续步骤

- 安装 OpenShift Serverless Operator 后，您可以安装 Knative Serving 或 安装 Knative Eventing。

3.2. 安装 KNATIVE SERVING

安装 Knative Serving 可让您在集群中创建 Knative 服务和功能。它还允许您为应用程序使用自动扩展和网络选项等其他功能。

安装 OpenShift Serverless Operator 后，您可以使用默认设置安装 Knative Serving，或者在 **KnativeServing** 自定义资源 (CR) 中配置更高级的设置。如需有关 **KnativeServing** CR 的配置选项的更多信息，请参阅全局配置。



重要

如果要在 OpenShift Serverless 中使用 Red Hat OpenShift distributed tracing，则必须在安装 Knative Serving 前安装和配置 Red Hat OpenShift distributed tracing。

3.2.1. 使用 Web 控制台安装 Knative Serving

安装 OpenShift Serverless Operator 后，使用 OpenShift Container Platform Web 控制台安装 Knative Serving。您可以使用默认设置安装 Knative Serving，或者在 **KnativeServing** 自定义资源 (CR) 中配置更高级的设置。

先决条件

- 您可以访问具有集群管理员权限的 OpenShift Container Platform 帐户。
- 已登陆到 OpenShift Container Platform Web 控制台。
- 已安装 OpenShift Serverless Operator。

流程

1. 在 OpenShift Container Platform web 控制台的 **Administrator** 视角中，进入 **Operators → Installed Operators**。
2. 检查页面顶部的 **Project** 下拉菜单是否已设置为 **Project: knative-serving**。
3. 点 OpenShift Serverless Operator 的 **Provided APIs** 列表中的 **Knative Serving** 来进入 **Knative Serving** 选项卡。
4. 点 **Create Knative Serving**。
5. 在 **Create Knative Serving** 页中，您可以使用默认设置安装 Knative Serving。点 **Create**。
   您还可以使用提供的表单或编辑 YAML 来修改 **KnativeServing** 对象来修改 Knative Serving 安装的设置。
   - 建议您在不需要完全控制 **KnativeServing** 对象创建的简单配置中使用该表单。
   - 对于更复杂的配置，建议编辑 YAML，这可以完全控制 **KnativeServing** 对象的创建。您可以通过点 **Create Knative Serving** 页右上角的 **edit YAML** 链接来访问 YAML。
     完成表单后，或者完成对 YAML 的修改后，点 **Create**。

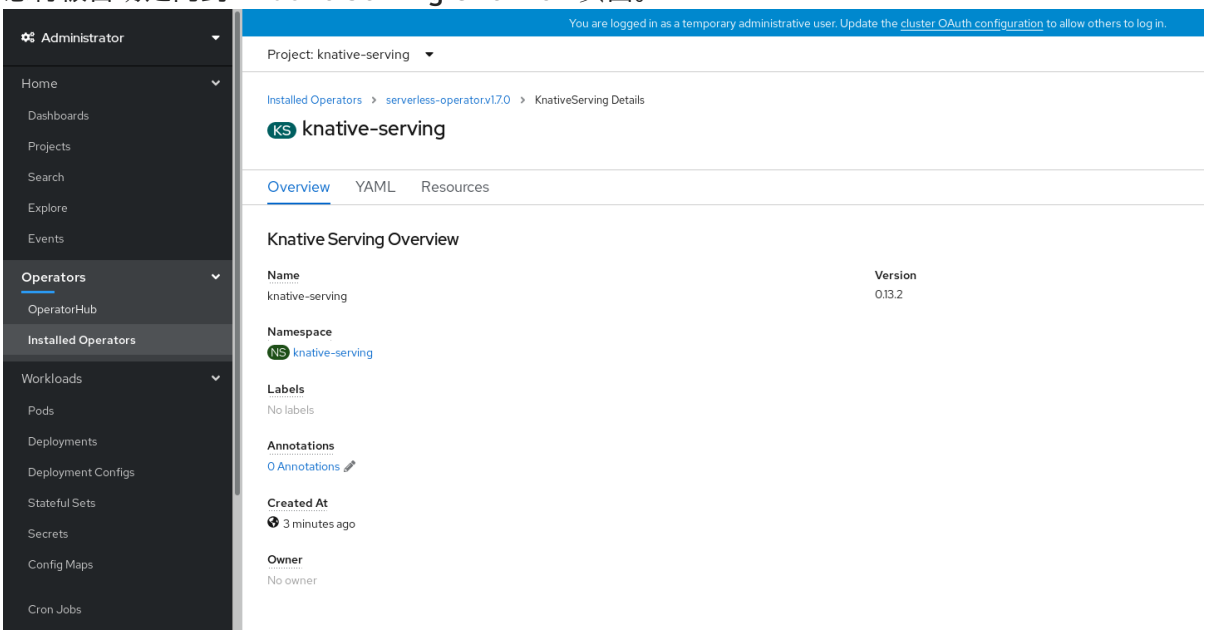
注意

如需有关 KnativeServing 自定义资源定义的配置选项的更多信息，请参阅*高级安装配置选项*。

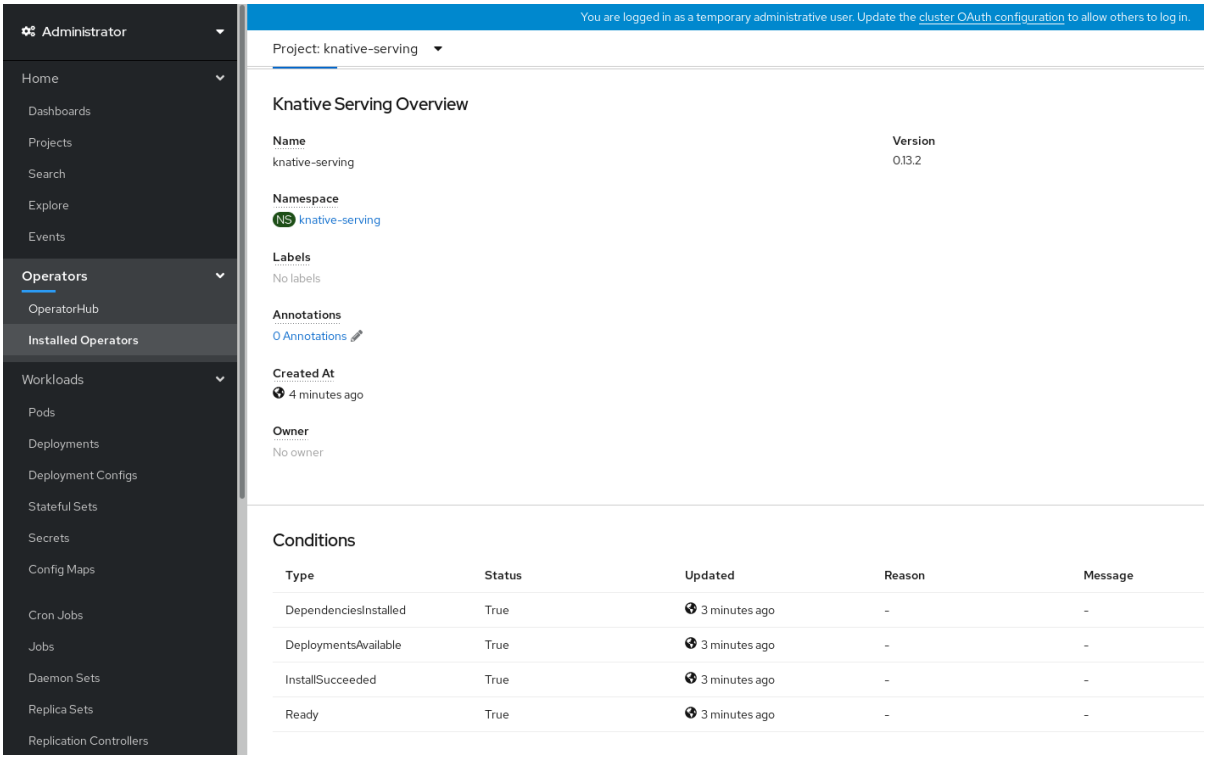
6. 安装 Knative Serving 后，会创建 **KnativeServing** 对象，并自动定向到 **Knative Serving** 选项卡。您可以在资源列表中看到 **knative-serving** 自定义资源。

验证

1. 在 **Knative Serving** 选项卡中点 **knative-serving** 自定义资源。
2. 您将被自动定向到 **Knative Serving Overview** 页面。



3. 向下滚动查看**条件**列表。
4. 您应该看到一个状况为 **True** 的条件列表，如示例镜像所示。



注意

创建 Knative Serving 资源可能需要几秒钟时间。您可以在 **Resources** 选项卡中查看其状态。

5. 如果条件状态为 **Unknown** 或 **False**，请等待几分钟，然后在确认已创建资源后再重新检查。

3.2.2. 使用 YAML 安装 Knative Serving

安装 OpenShift Serverless Operator 后，您可以使用默认设置安装 Knative Serving，或者在 **KnativeServing** 自定义资源 (CR) 中配置更高级的设置。您可以使用 YAML 文件和 **oc** CLI 安装 Knative Serving。

先决条件

- 您可以访问具有集群管理员权限的 OpenShift Container Platform 帐户。
- 已安装 OpenShift Serverless Operator。
- 安装 OpenShift CLI (**oc**) 。

流程

1. 创建名为 **serving.yaml** 的文件并将以下示例 YAML 复制到其中：

```
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeServing
metadata:
  name: knative-serving
  namespace: knative-serving
```

2. 应用 **service.yaml** 文件：

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

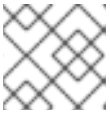
验证

1. 使用以下命令校验安装是否完成：

```
$ oc get knativeserving.operator.knative.dev/knative-serving -n knative-serving --
template='{{range .status.conditions}}{{printf "%s=%s\n" .type .status}}{{end}}'
```

输**出示例**

```
DependenciesInstalled=True
DeploymentsAvailable=True
InstallSucceeded=True
Ready=True
```



注意

创建 Knative Serving 资源可能需要几秒钟时间。

如果条件状态为 **Unknown** 或 **False**，请等待几分钟，然后在确认已创建资源后再重新检查。

2. 检查是否已创建 Knative Serving 资源：

```
$ oc get pods -n knative-serving
```

输**出示例**

```
NAME                                                  READY   STATUS      RESTARTS   AGE
activator-67ddf8c9d7-p7rm5                            2/2     Running     0          4m
activator-67ddf8c9d7-q84fz                            2/2     Running     0          4m
autoscaler-5d87bc6dbf-6nqc6                           2/2     Running     0          3m59s
autoscaler-5d87bc6dbf-h64rl                           2/2     Running     0          3m59s
autoscaler-hpa-77f85f5cc4-lrts7                       2/2     Running     0          3m57s
autoscaler-hpa-77f85f5cc4-zx7hl                       2/2     Running     0          3m56s
controller-5cfc7cb8db-nlccl                           2/2     Running     0          3m50s
controller-5cfc7cb8db-rmv7r                           2/2     Running     0          3m18s
domain-mapping-86d84bb6b4-r746m                       2/2     Running     0          3m58s
domain-mapping-86d84bb6b4-v7nh8                       2/2     Running     0          3m58s
domainmapping-webhook-769d679d45-bkcnj                2/2     Running     0          3m58s
domainmapping-webhook-769d679d45-fff68                2/2     Running     0          3m58s
storage-version-migration-serving-serving-0.26.0--1-6qlkb   0/1     Completed   0          3m56s
webhook-5fb774f8d8-6bqrt                              2/2     Running     0          3m57s
webhook-5fb774f8d8-b8lt5                              2/2     Running     0          3m57s
```

3. 检查所需的网络组件是否已安装到自动创建的 **knative-serving-ingress** 命名空间：

```
$ oc get pods -n knative-serving-ingress
```

输**出示例**

```
NAME                                    READY   STATUS    RESTARTS   AGE
net-kourier-controller-7d4b6c5d95-62mkf   1/1     Running   0          76s
net-kourier-controller-7d4b6c5d95-qmgm2   1/1     Running   0          76s
3scale-kourier-gateway-6688b49568-987qz   1/1     Running   0          75s
3scale-kourier-gateway-6688b49568-b5tnp   1/1     Running   0          75s
```

3.2.3. 后续步骤

- 如果要使用 Knative 事件驱动的架构，您可以安装 Knative Eventing。

3.3. 安装 KNATIVE EVENTING

要在集群上使用事件驱动的构架，请安装 Knative Eventing。您可以创建 Knative 组件，如事件源、代理和频道，然后使用它们向应用程序或外部系统发送事件。

安装 OpenShift Serverless Operator 后，您可以使用默认设置安装 Knative Eventing，或者在 **KnativeEventing** 自定义资源 (CR) 中配置更高级的设置。有关 **KnativeEventing** CR 的配置选项的更多信息，请参阅全局配置。

重要

如果要在 OpenShift Serverless 中使用 Red Hat OpenShift distributed tracing，则必须在安装 Knative Eventing 前安装和配置 Red Hat OpenShift distributed tracing。

3.3.1. 使用 Web 控制台安装 Knative Eventing

安装 OpenShift Serverless Operator 后，使用 OpenShift Container Platform Web 控制台安装 Knative Eventing。您可以使用默认设置安装 Knative Eventing，或者在 **KnativeEventing** 自定义资源 (CR) 中配置更高级的设置。

先决条件

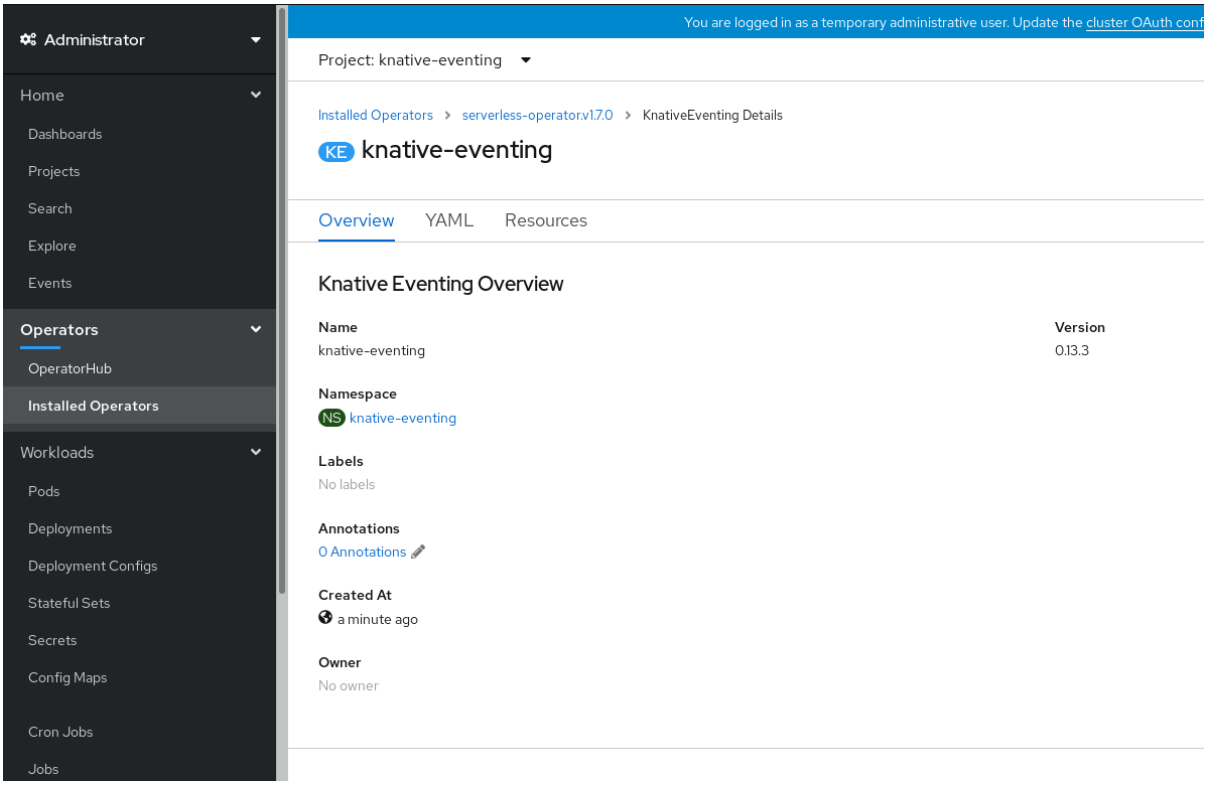
- 您可以访问具有集群管理员权限的 OpenShift Container Platform 帐户。
- 已登陆到 OpenShift Container Platform Web 控制台。
- 已安装 OpenShift Serverless Operator。

流程

1. 在 OpenShift Container Platform web 控制台的 **Administrator** 视角中，进入 **Operators → Installed Operators**。
2. 检查页面顶部的 **Project** 下拉菜单是否已设置为 **Project: knative-eventing**。
3. 点 OpenShift Serverless Operator 的 **Provided APIs** 列表中的 **Knative Eventing** 来进入 **Knative Eventing** 选项卡。
4. 点 **Create Knative Eventing**。
5. 在 **Create Knative Eventing** 页面中，您可以选择使用提供的默认表单或编辑 YAML 来配置 **KnativeEventing** 对象。
   - 建议您在不需要完全控制 **KnativeEventing** 对象创建的简单配置中使用该表单。可选。如果您要使用表单配置 **KnativeEventing** 对象，请为您的 Knative Eventing 部署进行任何要实现的更改。
6. 点 **Create**。
   - 对于更复杂的配置，建议编辑 YAML，这可以完全控制 **KnativeEventing** 对象的创建。您可以通过点 **Create Knative Eventing** 页右上角的 **edit YAML** 链接来访问 YAML。可选。如果您要通过编辑 YAML 配置 **KnativeEventing** 对象，请对您希望用于 Knative Eventing 部署的 YAML 进行更改。
7. 点 **Create**。
8. 安装 Knative Eventing 后，会创建 **KnativeEventing** 对象，并自动定向到 **Knative Eventing** 选项卡。您可以在资源列表中看到 **knative-eventing** 自定义资源。

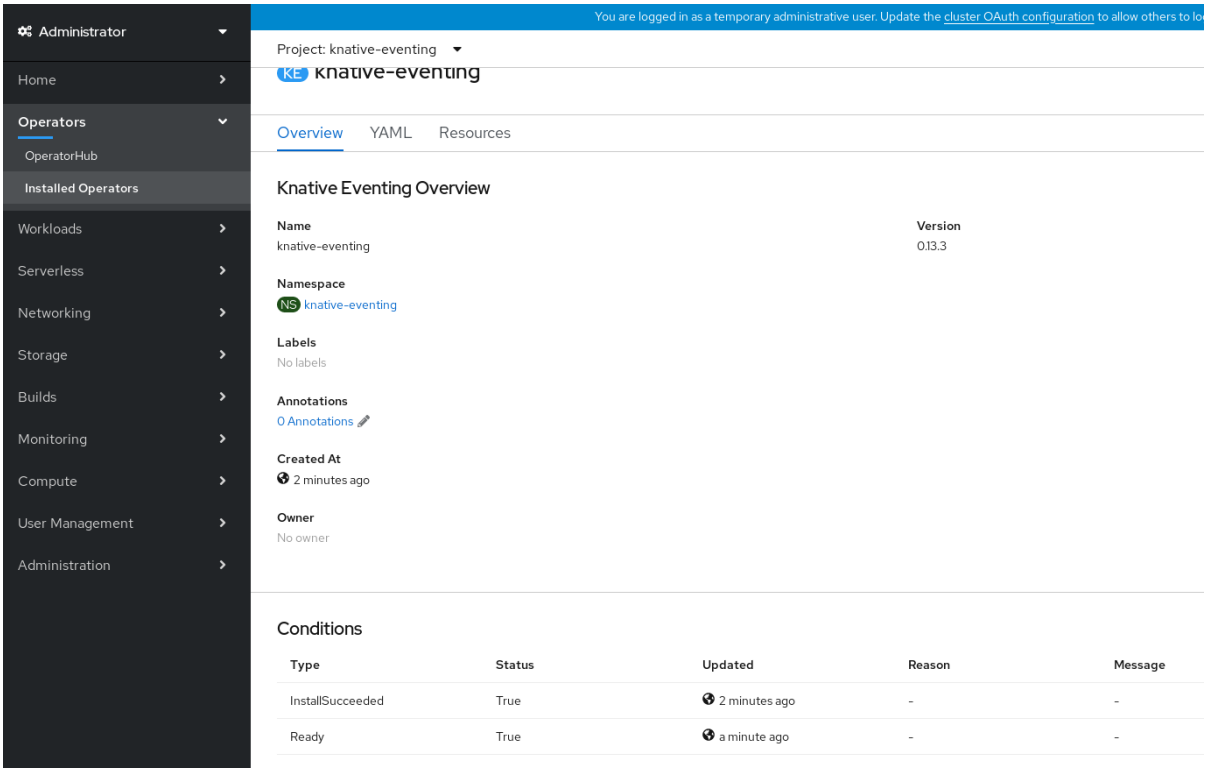
验证

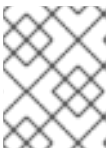
1. 点 **Knative Eventing** 选项卡中的 **knative-eventing** 自定义资源。
2. 您会自动定向到 **Knative Eventing Overview** 页面。



3. 向下滚动查看**条件**列表。

4. 您应该看到一个状况为 **True** 的条件列表，如示例镜像所示。





注意

创建 Knative Eventing 资源可能需要几秒钟时间。您可以在 **Resources** 选项卡中查看其状态。

5. 如果条件状态为 **Unknown** 或 **False**，请等待几分钟，然后在确认已创建资源后再重新检查。

3.3.2. 使用 YAML 安装 Knative Eventing

安装 OpenShift Serverless Operator 后，您可以使用默认设置安装 Knative Eventing，或者在 **KnativeEventing** 自定义资源 (CR) 中配置更高级的设置。您可以使用 YAML 文件和 **oc** CLI 安装 Knative Eventing。

先决条件

- 您可以访问具有集群管理员权限的 OpenShift Container Platform 帐户。
- 已安装 OpenShift Serverless Operator。
- 安装 OpenShift CLI (**oc**) 。

流程

1. 创建名为 **eventing.yaml** 的文件。
2. 将以下示例 YAML 复制到 **eventing.yaml**中：

```
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeEventing
metadata:
  name: knative-eventing
  namespace: knative-eventing
```

3. 可选。根据您的 Knative Eventing 部署，对 YAML 进行相应的更改。
4. 输入以下内容来应用 **eventing.yaml** 文件：

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

验证

1. 输入以下命令验证安装是否完成，并观察输出结果：

```
$ oc get knativeeventing.operator.knative.dev/knative-eventing \
  -n knative-eventing \
  --template='{{range .status.conditions}}{{printf "%s=%s\n" .type .status}}{{end}}'
```

输**出示例**

```
InstallSucceeded=True
Ready=True
```

注意

创建 Knative Eventing 资源可能需要几秒钟时间。

2. 如果条件状态为 **Unknown** 或 **False**，请等待几分钟，然后在确认已创建资源后再重新检查。
3. 使用以下命令检查是否已创建 Knative Eventing 资源：

```
$ oc get pods -n knative-eventing
```

输**出示例**

```
NAME                                  READY   STATUS    RESTARTS   AGE
broker-controller-58765d9d49-g9zp6    1/1     Running   0          7m21s
eventing-controller-65fdd66b54-jw7bh  1/1     Running   0          7m31s
eventing-webhook-57fd74b5bd-kvhlz     1/1     Running   0          7m31s
imc-controller-5b75d458fc-ptvm2       1/1     Running   0          7m19s
imc-dispatcher-64f6d5fccb-kkc4c       1/1     Running   0          7m18s
```

3.3.3. 后续步骤

- 如果要使用 Knative 服务，可以安装 Knative Serving。

3.4. 删除 OPENSHIFT SERVERLESS

如果需要从集群中删除 OpenShift Serverless，您可以手动删除 OpenShift Serverless Operator 和其他 OpenShift Serverless 组件。在删除 OpenShift Serverless Operator 之前，您必须删除 Knative Serving 和 Knative Eventing。

3.4.1. 卸载 Knative Serving

在删除 OpenShift Serverless Operator 之前，您必须删除 Knative Serving。要卸载 Knative Serving，您必须删除 **KnativeServing** 自定义资源 (CR) 并删除 **knative-serving** 命名空间。

先决条件

- 您可以访问具有集群管理员权限的 OpenShift Container Platform 帐户。
- 安装 OpenShift CLI (**oc**) 。

流程

1. 删除 **KnativeServing** CR:

```
$ oc delete knativeservings.operator.knative.dev knative-serving -n knative-serving
```

2. 在该命令运行完成且已从 **knative-serving** 命名空间中移除所有 Pod 后，删除命名空间：

```
$ oc delete namespace knative-serving
```

3.4.2. 卸载 Knative Eventing

在删除 OpenShift Serverless Operator 之前，您必须删除 Knative Eventing。要卸载 Knative Eventing，您必须删除 **KnativeEventing** 自定义资源 (CR) 并删除 **knative-eventing** 命名空间。

先决条件

- 您可以访问具有集群管理员权限的 OpenShift Container Platform 帐户。
- 安装 OpenShift CLI (**oc**) 。

流程

1. 删除 **KnativeEventing** CR：

```
$ oc delete knativeeventings.operator.knative.dev knative-eventing -n knative-eventing
```

2. 在该命令运行完成且已从 **knative-eventing** 命名空间中移除所有 Pod 后，删除命名空间：

```
$ oc delete namespace knative-eventing
```

3.4.3. 删除 OpenShift Serverless Operator

删除 Knative Serving 和 Knative Eventing 后，您可以删除 OpenShift Serverless Operator。您可以使用 OpenShift Container Platform Web 控制台或 **oc** CLI 完成此操作。

3.4.3.1. 使用 Web 控制台从集群中删除 Operator

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

先决条件

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

流程

1. 进入 **Operators → Installed Operators** 页面，在 **Filter by name** 字段滚动鼠标或键入关键词，以查找您想要的 Operator。然后点它。

2. 在 **Operator Details** 页面右侧，从 **Actions** 列表中选择 **Uninstall Operator**。此时会显示 **Uninstall Operator?** 对话框，提醒您：

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

   此操作将删除 Operator 以及 Operator 部署和 pod（若有）。任何 Operands 和由 Operator 管理的资源（包括 CRD 和 CR）都不会被删除。Web 控制台为一些 Operator 启用仪表板和导航项。要在卸载 Operator 后删除这些，您可能需要手动删除 Operator CRD。

3. 选择 **Uninstall**。此 Operator 将停止运行，并且不再接收更新。

3.4.3.2. 使用 CLI 从集群中删除 Operator

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

先决条件

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

流程

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

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

输**出示例**

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

2. 删除订阅（如 **jaeger**）：

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

输**出示例**

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

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

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

输**出示例**

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

3.4.3.3. 刷新失败的订阅

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

输**出示例**

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

输**出示例**

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

因此，订阅会处于这个失败状态，Operator 无法安装或升级。

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

先决条件

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

流程

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

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

输**出示例**

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

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

2. 删除订阅：

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

3. 删除集群服务版本：

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

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

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

输**出示例**

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

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

5. 删除作业：

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

这样可确保尝试拉取无法访问的镜像的 Pod 不会被重新创建。

6. 删除配置映射：

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

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

验证

- 检查是否已成功重新安装 Operator:

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

3.4.4. 删除 OpenShift Serverless 自定义资源定义

卸载 OpenShift Serverless 后，Operator 和 API 自定义资源定义（CRD）会保留在集群中。您可以使用以下步骤删除剩余的 CRD。

重要

移除 Operator 和 API CRD 也会移除所有使用它们定义的资源，包括 Knative 服务。

先决条件

- 您可以访问具有集群管理员权限的 OpenShift Container Platform 帐户。
- 您已卸载了 Knative Serving 并移除了 OpenShift Serverless Operator。
- 安装 OpenShift CLI (**oc**) 。

流程

- 运行以下命令删除 OpenShift Serverless CRD：

```
$ oc get crd -oname | grep 'knative.dev' | xargs oc delete
```

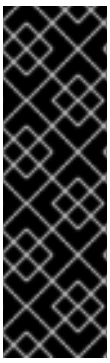
第 4 章 KNATIVE CLI

4.1. 安装 KNATIVE CLI

Knative (**kn**) CLI 本身没有登录机制。要登录到集群，您必须安装 OpenShift CLI (**oc**)，并使用 **oc login** 命令。CLI 的安装选项可能会因您的操作系统而异。

有关为您的操作系统安装 **oc** CLI 并使用 **oc** 登录的更多信息，请参阅 OpenShift CLI 启动文档。

OpenShift Serverless 不能使用 Knative(**kn**)CLI 安装。集群管理员必须安装 OpenShift Serverless Operator 并设置 Knative 组件，如 安装 OpenShift Serverless Operator 文档所述。



重要

如果您试图将较旧版本的 Knative **kn** CLI 与较新的 OpenShift Serverless 发行版本搭配使用，则不会找到 API，并出现错误。

例如，您使用 1.23.0 版本的 Knative (**kn**) CLI（使用版本 1.2），以及 1.24.0 版本的 OpenShift Serverless（使用版本 1.3 的 Knative Serving 和 Knative Eventing API），则 CLI 将无法正常工作，因为它会一直寻找已过时的 1.2 版本的 API。

确保为 OpenShift Serverless 版本使用最新的 Knative (**kn**) CLI 版本以避免出现问题。

4.1.1. 使用 OpenShift Container Platform Web 控制台安装 Knative CLI

使用 OpenShift Container Platform Web 控制台提供了一个简洁、直观的用户界面来安装 Knative (**kn**) CLI。安装 OpenShift Serverless Operator 后，您会看到从 OpenShift Container Platform Web 控制台的 **Command Line Tools** 页面中下载适用于 Linux 的 Knative (**kn**) CLI 的链接（amd64、s390x、ppc64le）、macOS 或 Windows。

先决条件

- 已登陆到 OpenShift Container Platform Web 控制台。
- OpenShift Serverless Operator 和 Knative Serving 已安装在 OpenShift Container Platform 集群中。

重要

如果 **libc** 不可用，您在运行 CLI 命令时可能会看到以下错误：

```
$ kn: No such file or directory
```

- 如果要使用验证步骤，您必须安装 OpenShift (**oc**) CLI。

流程

1. 从 **Command Line Tools** 页面下载 Knative (**kn**) CLI。您可以点 web 控制台右上角的 图标进入 **Command Line Tools** 页，并在列表中选择 **Command Line Tools**。
2. 解包存档：

```
$ tar -xf <file>
```

3. 将 **kn** 二进制文件移到 **PATH** 中的目录中。

4. 运行以下命令可以查看 **PATH** 的值：

```
$ echo $PATH
```

验证

- 运行以下命令检查是否已创建了正确的 Knative CLI 资源和路由：

```
$ oc get ConsoleCLIDownload
```

输**出示例**

```
NAME                 DISPLAY NAME                                     AGE
kn                   kn - OpenShift Serverless Command Line Interface (CLI)   2022-09-
20T08:41:18Z
oc-cli-downloads     oc - OpenShift Command Line Interface (CLI)              2022-09-
20T08:00:20Z
```

```
$ oc get route -n openshift-serverless
```

输**出示例**

```
NAME   HOST/PORT                          PATH   SERVICES                  PORT
TERMINATION     WILDCARD
kn     kn-openshift-serverless.apps.example.com          knative-openshift-metrics-3   http-cli
edge/Redirect   None
```

4.1.2. 使用 RPM 软件包管理器为 Linux 安装 Knative CLI

对于 Red Hat Enterprise Linux (RHEL)，您可以使用软件包管理器（如 **yum** 或 **dnf**）将 Knative (**kn**) CLI 作为 RPM 安装。这允许系统自动管理 Knative CLI 版本。例如，如果有新版本可用，使用 **dnf upgrade** 一样的命令升级所有软件包，包括 **kn**。

先决条件

- 您的红帽帐户必须具有有效的 OpenShift Container Platform 订阅。

流程

1. 使用 Red Hat Subscription Manager 注册：

```
# subscription-manager register
```

2. 获取最新的订阅数据：

```
# subscription-manager refresh
```

3. 将订阅附加到注册的系统：

```
# subscription-manager attach --pool=<pool_id> 1
```

1 活跃的 OpenShift Container Platform 订阅的池 ID

4. 启用 Knative (**kn**) CLI 所需的仓库：

- Linux (x86_64, amd64)

```
# subscription-manager repos --enable="openshift-serverless-1-for-rhel-8-x86_64-rpms"
```

- Linux on IBM Z and LinuxONE (s390x)

```
# subscription-manager repos --enable="openshift-serverless-1-for-rhel-8-s390x-rpms"
```

- Linux on IBM Power (ppc64le)

```
# subscription-manager repos --enable="openshift-serverless-1-for-rhel-8-ppc64le-rpms"
```

5. 使用软件包管理器将 Knative (**kn**) CLI 作为 RPM 安装：

yum 命令示例

```
# yum install openshift-serverless-clients
```

4.1.3. 为 Linux 安装 Knative CLI

如果您使用没有 RPM 或者另一个软件包管理器的 Linux 发行版本，您可以将 Knative (**kn**) CLI 安装为二进制文件。要做到这一点，您必须下载并解包一个 **tar.gz** 存档，并将二进制文件添加到 **PATH** 的目录中。

先决条件

- 如果您不使用 RHEL 或 Fedora，请确保将 **libc** 安装在库路径的目录中。

重要

如果 **libc** 不可用，您在运行 CLI 命令时可能会看到以下错误：

```
$ kn: No such file or directory
```

流程

1. 下载相关的 Knative (**kn**) CLI **tar.gz** 存档：

- Linux (x86_64, amd64)
- Linux on IBM Z and LinuxONE (s390x)
- Linux on IBM Power (ppc64le)

2. 解包存档：

```
$ tar -xf <filename>
```

3. 将 **kn** 二进制文件移到 **PATH** 中的目录中。

4. 运行以下命令可以查看 **PATH** 的值：

```
$ echo $PATH
```

4.1.4. 为 macOS 安装 Knative CLI

如果使用 macOS，您可以将 Knative (**kn**) CLI 安装为二进制文件。要做到这一点，您必须下载并解包一个 **tar.gz** 存档，并将二进制文件添加到 **PATH** 的目录中。

流程

1. 下载 Knative (**kn**) CLI **tar.gz** 存档。

2. 解包并提取存档。

3. 将 **kn** 二进制文件移到 **PATH** 中的目录中。

4. 要查看 **PATH**，打开终端窗口并运行：

```
$ echo $PATH
```

4.1.5. 为 Windows 安装 Knative CLI

如果使用 Windows，您可以将 Knative (**kn**) CLI 安装为二进制文件。要做到这一点，您必须下载并解包 ZIP 存档，并将二进制文件添加到 **PATH** 的目录中。

流程

1. 下载 Knative (**kn**) CLI ZIP 存档。

2. 使用 ZIP 程序解压存档。

3. 将 **kn** 二进制文件移到 **PATH** 中的目录中。

4. 要查看您的 **PATH**，请打开命令窗口并运行以下命令：

```
C:\> path
```

4.2. 配置 KNATIVE CLI

您可以通过创建 **config.yaml** 配置文件来自定义 Knative (**kn**) CLI 设置。您可以使用 **--config** 标志来提供此配置，否则会从默认位置提取配置。默认配置位置符合 XDG Base Directory 规格，对于 UNIX 系统和 Windows 系统有所不同。

对于 UNIX 系统：

- 如果设置了 **XDG_CONFIG_HOME** 环境变量，Knative (**kn**) CLI 查找的默认配置位置为 **$XDG_CONFIG_HOME/kn**。
- 如果没有设置 **XDG_CONFIG_HOME** 环境变量，Knative (**kn**) CLI 会在 **$HOME/.config/kn/config.yaml** 的用户主目录中查找配置。

对于 Windows 系统，默认的 Knative (**kn**) CLI 配置位置为 **%APPDATA%\kn**。

配置文件示例

```
plugins:
  path-lookup: true (1)
  directory: ~/.config/kn/plugins (2)
eventing:
  sink-mappings: (3)
  - prefix: svc (4)
    group: core (5)
    version: v1 (6)
    resource: services (7)
```

1. 指定 Knative (**kn**) CLI 是否应该在 **PATH** 环境变量中查找插件。这是一个布尔值配置选项。默认值为 **false**。
2. 指定 Knative (**kn**) CLI 查找插件的目录。默认路径取决于操作系统，如前面所述。这可以是用户可见的任何目录。
3. **sink-mappings** spec 定义了在 Knative (**kn**) CLI 命令中使用 **--sink** 标志时使用的 Kubernetes 可寻址资源。
4. 您用来描述接收器（sink）的前缀。**svc**（用于服务）、**channel** 和 **broker** 是 Knative (**kn**) CLI 中预定义的前缀。
5. Kubernetes 资源的 API 组。
6. Kubernetes 资源的版本。
7. Kubernetes 资源类型的复数名称。例如，**services** 或 **brokers**。

4.3. KNATIVE CLI 插件

Knative (**kn**) CLI 支持使用插件，这允许您通过添加不是核心发行版本一部分的自定义命令和其他共享命令来扩展 **kn** 安装的功能。Knative (**kn**) CLI 插件的使用方式与主 **kn** 功能相同。

目前，红帽支持 **kn-source-kafka** 插件和 **kn-event** 插件。

重要

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

有关红帽技术预览功能支持范围的详情，请参考 https://access.redhat.com/support/offerings/techpreview/。

4.3.1. 使用 kn-event 插件构建事件

您可以使用 **kn event build** 命令的 builder 接口来构建事件。然后，您可以稍后发送该事件或在另一个上下文中使用它。

先决条件

- 已安装 Knative (**kn**) CLI。

流程

- 构建事件：

```
$ kn event build --field <field-name>=<value> --type <type-name> --id <id> --output <format>
```

其中：

  - **--field** 标志将数据作为字段值对添加到事件中。您可以多次使用它。
  - **--type** 标志允许您指定指定事件类型的字符串。
  - **--id** 标志指定事件的 ID。
  - 您可以将 **json** 或 **yaml** 参数与 **--output** 标志一起使用，以更改事件的输出格式。所有这些标记都是可选的。

    构建简单的事件

```
$ kn event build -o yaml
```

    结果为 YAML 格式

```
data: {}
datacontenttype: application/json
id: 81a402a2-9c29-4c27-b8ed-246a253c9e58
source: kn-event/v0.4.0
specversion: "1.0"
time: "2021-10-15T10:42:57.713226203Z"
type: dev.knative.cli.plugin.event.generic
```

    构建示例事务事件

```
$ kn event build \
    --field operation.type=local-wire-transfer \
    --field operation.amount=2345.40 \
    --field operation.from=87656231 \
    --field operation.to=2344121 \
    --field automated=true \
    --field signature='FGzCPLvYWdEgsdpb3qXkaVp7Da0=' \
    --type org.example.bank.bar \
    --id $(head -c 10 < /dev/urandom | base64 -w 0) \
    --output json
```

    JSON 格式的结果事件

```
{
  "specversion": "1.0",
  "id": "RjtL8UH66X+UJg==",
  "source": "kn-event/v0.4.0",
  "type": "org.example.bank.bar",
  "datacontenttype": "application/json",
  "time": "2021-10-15T10:43:23.113187943Z",
  "data": {
    "automated": true,
    "operation": {
      "amount": "2345.40",
      "from": 87656231,
      "to": 2344121,
      "type": "local-wire-transfer"
    },
    "signature": "FGzCPLvYWdEgsdpb3qXkaVp7Da0="
  }
}
```


4.3.2. 使用 kn-event 插件发送事件

您可以使用 **kn event send** 命令来发送事件。事件可以发送到公开的地址，或发送到集群中的可寻址资源，如 Kubernetes 服务，以及 Knative 服务、代理和频道。命令使用与 **kn event build** 命令相同的 builder 接口。

先决条件

- 已安装 Knative (**kn**) CLI。

流程

- 发送事件：

```
$ kn event send --field <field-name>=<value> --type <type-name> --id <id> --to-url <url> --to <cluster-resource> --namespace <namespace>
```

其中：

  - **--field** 标志将数据作为字段值对添加到事件中。您可以多次使用它。
  - **--type** 标志允许您指定指定事件类型的字符串。
  - **--id** 标志指定事件的 ID。
  - 如果您要将事件发送到公开的目的地，请使用 **--to-url** 标志指定 URL。
  - 如果要将事件发送到集群内 Kubernetes 资源，请使用 **--to** 标志指定目的地。
    - 使用 **<Kind>:<ApiVersion>:<name>** 格式指定 Kubernetes 资源。
  - **--namespace** 标志指定命名空间。如果省略，则会从当前上下文中获取命名空间。
    所有这些标志都是可选的，除了目的地规格外，您需要使用 **--to-url** 或 **--to**。

    以下示例显示向 URL 发送事件：

示例命令

```
$ kn event send \
    --field player.id=6354aa60-ddb1-452e-8c13-24893667de20 \
    --field player.game=2345 \
    --field points=456 \
    --type org.example.gaming.foo \
    --to-url http://ce-api.foo.example.com/
```

以下示例显示了将事件发送到 in-cluster 资源：

示例命令

```
$ kn event send \
    --type org.example.kn.ping \
    --id $(uuidgen) \
    --field event.type=test \
    --field event.data=98765 \
    --to Service:serving.knative.dev/v1:event-display
```

4.4. KNATIVE SERVING CLI 命令

您可以使用以下 Knative (**kn**) CLI 命令，在集群中完成 Knative Serving 任务。

4.4.1. kn service 命令

您可以使用以下命令创建和管理 Knative 服务。

4.4.1.1. 使用 Knative CLI 创建无服务器应用程序

通过使用 Knative (**kn**) CLI 创建无服务器应用程序，通过直接修改 YAML 文件来提供更精简且直观的用户界面。您可以使用 **kn service create** 命令创建基本无服务器应用程序。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 已安装 Knative (**kn**) CLI。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。

流程

- 创建 Knative 服务：

  ```
  $ kn service create <service_name> --image <image> --tag <tag-value>
  ```

  其中：

  - **--image** 是应用的镜像的 URI。
  - **--tag** 是一个可选标志，可用于向利用服务创建的初始修订版本添加标签。

示例命令

```
$ kn service create event-display \
    --image quay.io/openshift-knative/knative-eventing-sources-event-display:latest
```

输**出示例**

```
Creating service 'event-display' in namespace 'default':

  0.271s The Route is still working to reflect the latest desired specification.
  0.580s Configuration "event-display" is waiting for a Revision to become ready.
  3.857s ...
  3.861s Ingress has not yet been reconciled.
  4.270s Ready to serve.

Service 'event-display' created with latest revision 'event-display-bxshg-1' and URL:
http://event-display-default.apps-crc.testing
```

4.4.1.2. 使用 Knative CLI 更新无服务器应用程序

在以递增方式构建服务时，您可以使用 **kn service update** 命令进行命令行上的互动会话。与 **kn service apply** 命令不同，在使用 **kn service update** 命令时，只需要指定您要更新的更改，而不是指定 Knative 服务的完整配置。

示例命令

- 通过添加新环境变量来更新服务：

```
$ kn service update <service_name> --env <key>=<value>
```

- 通过添加新端口来更新服务：

```
$ kn service update <service_name> --port 80
```

- 通过添加新的请求和限制参数来更新服务：

```
$ kn service update <service_name> --request cpu=500m --limit memory=1024Mi --limit
cpu=1000m
```

- 为修订分配 **latest** 标签：

```
$ kn service update <service_name> --tag <revision_name>=latest
```

- 为服务的最新 **READY** 修订将标签从 **testing** 更新为 **staging**：

```
$ kn service update <service_name> --untag testing --tag @latest=staging
```

- 将 **test** 标签添加到接收 10% 流量的修订，并将其它流量发送到服务的最新 **READY** 修订：

```
$ kn service update <service_name> --tag <revision_name>=test --traffic test=10,@latest=90
```

4.4.1.3. 应用服务声明

您可以使用 **kn service apply** 命令声明性配置 Knative 服务。如果服务不存在，则使用已更改的选项更新现有服务。

kn service apply 命令对 shell 脚本或持续集成管道特别有用，因为用户通常希望在单个命令中完全指定服务的状态来声明目标状态。

使用 **kn service apply** 时，必须为 Knative 服务提供完整的配置。这与 **kn service update** 命令不同，它只在命令中指定您要更新的选项。

示例命令

- 创建服务：

```
$ kn service apply <service_name> --image <image>
```

- 将环境变量添加到服务：

```
$ kn service apply <service_name> --image <image> --env <key>=<value>
```

- 从 JSON 或 YAML 文件中读取服务声明：

```
$ kn service apply <service_name> -f <filename>
```

4.4.1.4. 使用 Knative CLI 描述无服务器应用程序

您可以使用 **kn service describe** 命令来描述 Knative 服务。

示例命令

- 描述服务：

```
$ kn service describe --verbose <service_name>
```

--verbose 标志是可选的，但可以包含它以提供更详细的描述。常规输出和详细输出之间的区别在以下示例中显示：

没有 --verbose 标记的输出示例

```
Name:       hello
Namespace:  default
Age:        2m
URL:        http://hello-default.apps.ocp.example.com

Revisions:
  100%  @latest (hello-00001) [1] (2m)
        Image:  docker.io/openshift/hello-openshift (pinned to aaea76)

Conditions:
  OK TYPE                 AGE REASON
  ++ Ready                1m
  ++ ConfigurationsReady     1m
  ++ RoutesReady             1m
```

-

带有 **--verbose** 标记的输出示例

```
Name:        hello
Namespace:   default
Annotations: serving.knative.dev/creator=system:admin
             serving.knative.dev/lastModifier=system:admin
Age:         3m
URL:         http://hello-default.apps.ocp.example.com
Cluster:     http://hello.default.svc.cluster.local

Revisions:
  100%  @latest (hello-00001) [1] (3m)
        Image:  docker.io/openshift/hello-openshift (pinned to aaea76)
        Env:    RESPONSE=Hello Serverless!

Conditions:
  OK TYPE                   AGE REASON
  ++ Ready                  3m
  ++ ConfigurationsReady     3m
  ++ RoutesReady            3m
```

- 以 YAML 格式描述服务：

```
$ kn service describe <service_name> -o yaml
```

- 以 JSON 格式描述服务：

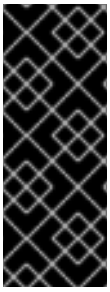
```
$ kn service describe <service_name> -o json
```

- 仅输出服务 URL：

```
$ kn service describe <service_name> -o url
```

4.4.2. 关于 Knative CLI 离线模式

执行 **kn service** 命令时，更改会立即传播到集群。但是，作为替代方案，您可以在离线模式下执行 **kn service** 命令。当您以离线模式创建服务时，集群不会发生任何更改，而是在本地计算机上创建服务描述符文件。



重要

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

有关红帽技术预览功能支持范围的详情，请参考 https://access.redhat.com/support/offerings/techpreview/。

创建描述符文件后，您可以手动修改并在版本控制系统中跟踪该文件。您还可以使用 **kn service create -f**、**kn service apply -f** 或 **oc apply -f** 命令将更改传播到集群。

离线模式有几种用途：

- 在使用描述符文件对集群进行更改之前，您可以手动修改该文件。
- 您可以在本地跟踪版本控制系统中服务的描述符文件。这可让您在目标集群以外的位置重复使用描述符文件，例如在持续集成（CI）管道、开发环境或演示中。
- 您可以检查创建的描述符文件，以了解 Knative 服务的信息。特别是，您可以看到生成的服务如何受到传递给 **kn** 命令的不同参数的影响。

离线模式有其优点：速度非常快，不需要连接到集群。但是，离线模式缺少服务器端验证。因此，您无法验证服务名称是否唯一，或者是否可以拉取指定镜像。

4.4.2.1. 使用离线模式创建服务

您可以在离线模式下执行 **kn service** 命令，以便集群中不会发生任何更改，而是在本地机器上创建服务描述符文件。创建描述符文件后，您可以在向集群传播更改前修改该文件。

重要

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

有关红帽技术预览功能支持范围的详情，请参考 https://access.redhat.com/support/offerings/techpreview/。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 已安装 Knative (**kn**) CLI。

流程

1. 在离线模式下，创建一个本地 Knative 服务描述符文件：

```
$ kn service create event-display \
  --image quay.io/openshift-knative/knative-eventing-sources-event-display:latest \
  --target ./ \
  --namespace test
```

输出示例

```
Service 'event-display' created in namespace 'test'.
```

- **--target ./** 标志启用脱机模式，并将 **./** 指定为用于存储新目录树的目录。
  如果您没有指定现有目录，但使用文件名，如 **--target my-service.yaml**，则不会创建目录树。相反，当前目录中只创建服务描述符 **my-service.yaml** 文件。

  文件名可以具有 **.yaml**、**.yml** 或 **.json** 扩展名。选择 **.json** 以 JSON 格式创建服务描述符文件。

- **namespace test** 选项将新服务放在 **test** 命名空间中。
  如果不使用 **--namespace**，并且您登录了 OpenShift 集群，则会在当前命名空间中创建描述符文件。否则，描述符文件会在 **default** 命名空间中创建。

2. 检查创建的目录结构：

```
$ tree ./
```

输出示例

```
./
└── test
    └── ksvc
        └── event-display.yaml

2 directories, 1 file
```

- 使用 **--target** 指定的当前 **./** 目录包含新的 **test/** 目录，它在指定的命名空间后命名。
- **test/** 目录包含 **ksvc**，它在资源类型后命名。
- **ksvc** 目录包含描述符文件 **event-display.yaml**，它根据指定的服务名称命名。

3. 检查生成的服务描述符文件：

```
$ cat test/ksvc/event-display.yaml
```

输出示例

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  creationTimestamp: null
  name: event-display
  namespace: test
spec:
  template:
    metadata:
      annotations:
        client.knative.dev/user-image: quay.io/openshift-knative/knative-eventing-sources-event-
display:latest
      creationTimestamp: null
    spec:
      containers:
      - image: quay.io/openshift-knative/knative-eventing-sources-event-display:latest
        name: ""
        resources: {}
status: {}
```

4. 列出新服务的信息：

```
$ kn service describe event-display --target ./ --namespace test
```

输出示例

```
Name:       event-display
Namespace:  test
```

```
Age:
URL:

Revisions:

Conditions:
  OK TYPE    AGE REASON
```

- **--target ./** 选项指定包含命名空间子目录的目录结构的根目录。
  另外，您可以使用 **--target** 选项直接指定 YAML 或 JSON 文件名。可接受的文件扩展包括 **.yaml**、**.yml** 和 **.json**。

- **--namespace** 选项指定命名空间，与 **kn** 通信包含所需服务描述符文件的子目录。
  如果您不使用 **--namespace**，且您已登录到 OpenShift 集群，**kn** 在以当前命名空间命名的子目录中搜索该服务。否则，**kn** 在 **default/** 子目录中搜索。

5. 使用服务描述符文件在集群中创建服务：

```
$ kn service create -f test/ksvc/event-display.yaml
```

输出示例

```
Creating service 'event-display' in namespace 'test':

  0.058s The Route is still working to reflect the latest desired specification.
  0.098s ...
  0.168s Configuration "event-display" is waiting for a Revision to become ready.
 23.377s ...
 23.419s Ingress has not yet been reconciled.
 23.534s Waiting for load balancer to be ready
 23.723s Ready to serve.

Service 'event-display' created to latest revision 'event-display-00001' is available at URL:
http://event-display-test.apps.example.com
```

4.4.3. kn 容器命令

您可以使用以下命令在 Knative 服务规格中创建和管理多个容器。

4.4.3.1. Knative 客户端多容器支持

您可以使用 **kn container add** 命令将 YAML 容器 spec 打印到标准输出。此命令对多容器用例很有用，因为它可以与其他标准 **kn** 标志一起使用来创建定义。

kn container add 命令接受与容器相关的所有标志，它们都支持与 **kn service create** 命令搭配使用。**kn container add** 命令也可以使用 UNIX 管道 (|) 一次创建多个容器定义来串联。

示例命令

- 从镜像添加容器并将其打印到标准输出中：

```
$ kn container add <container_name> --image <image_uri>
```

示例命令

```
$ kn container add sidecar --image docker.io/example/sidecar
```

输**出示例**

```
containers:
- image: docker.io/example/sidecar
  name: sidecar
  resources: {}
```

- 将两个 **kn container add** 命令链接在一起，然后将它们传递给 **kn service create** 命令创建带有两个容器的 Knative 服务：

```
$ kn container add <first_container_name> --image <image_uri> | \
kn container add <second_container_name> --image <image_uri> | \
kn service create <service_name> --image <image_uri> --extra-containers -
```

--extra-containers - 指定一个特殊情况，**kn** 读取管道输入，而不是 YAML 文件。

示例命令

```
$ kn container add sidecar --image docker.io/example/sidecar:first | \
kn container add second --image docker.io/example/sidecar:second | \
kn service create my-service --image docker.io/example/my-app:latest --extra-containers -
```

--extra-containers 标志也可以接受到 YAML 文件的路径：

```
$ kn service create <service_name> --image <image_uri> --extra-containers <filename>
```

示例命令

```
$ kn service create my-service --image docker.io/example/my-app:latest --extra-containers
my-extra-containers.yaml
```

4.4.4. kn 域命令

您可以使用下列命令创建和管理域映射。

4.4.4.1. 使用 Knative CLI 创建自定义域映射

您可以通过将您自己的自定义域名映射到 Knative 服务来自定义 Knative 服务域。您可以使用 Knative (**kn**) CLI 创建映射到可寻址目标 CR 的 **DomainMapping** 自定义资源 (CR)，如 Knative 服务或 Knative 路由。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 您已创建了 Knative 服务或路由，并控制要映射到该 CR 的自定义域。

注意

您的自定义域必须指向 OpenShift Container Platform 集群的 DNS。

- 已安装 Knative (**kn**) CLI。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。

流程

- 将域映射到当前命名空间中的 CR：

```
$ kn domain create <domain_mapping_name> --ref <target_name>
```

示例命令

```
$ kn domain create example.com --ref example-service
```

--ref 标志为域映射指定一个可寻址的目标 CR。

如果使用 **--ref** 标志时没有提供前缀，则会假定目标为当前命名空间中的 Knative 服务。

- 将域映射到指定命名空间中的 Knative 服务：

```
$ kn domain create <domain_mapping_name> --ref
<ksvc:service_name:service_namespace>
```

示例命令

```
$ kn domain create example.com --ref ksvc:example-service:example-namespace
```

- 将域映射到 Knative 路由：

```
$ kn domain create <domain_mapping_name> --ref <kroute:route_name>
```

示例命令

```
$ kn domain create example.com --ref kroute:example-route
```

4.4.4.2. 使用 Knative CLI 管理自定义域映射

创建 **DomainMapping** 自定义资源 (CR) 后，您可以使用 Knative (**kn**) CLI 列出现有 CR、查看现有 CR 的信息、更新 CR 或删除 CR。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 您至少已创建了一个 **DomainMapping** CR。
- 已安装 Knative (**kn**) CLI 工具。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。

流程

- 列出现有的 **DomainMapping** CR：

```
$ kn domain list -n <domain_mapping_namespace>
```

- 查看现有 **DomainMapping** CR 的详情：

```
$ kn domain describe <domain_mapping_name>
```

- 更新 **DomainMapping** CR 以指向新目标：

```
$ kn domain update --ref <target>
```

- 删除 **DomainMapping** CR：

```
$ kn domain delete <domain_mapping_name>
```

4.5. KNATIVE EVENTING CLI 命令

您可以使用以下 Knative (**kn**) CLI 命令在集群中完成 Knative Eventing 任务。

4.5.1. kn source 命令

您可以使用以下命令列出、创建和管理 Knative 事件源。

4.5.1.1. 使用 Knative CLI 列出可用事件源类型

使用 Knative (**kn**) CLI 提供了简化和直观的用户界面，用来在集群中查看可用事件源类型。您可以使用 **kn source list-types** CLI 命令列出集群中创建和使用的事件源类型。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Eventing。
- 已安装 Knative (**kn**) CLI。

流程

1. 列出终端中的可用事件源类型：

```
$ kn source list-types
```

输**出示例**

```
TYPE              NAME                                DESCRIPTION
ApiServerSource   apiserversources.sources.knative.dev   Watch and send Kubernetes API events to a sink
PingSource        pingsources.sources.knative.dev       Periodically send ping events to a sink
SinkBinding       sinkbindings.sources.knative.dev      Binding for connecting a PodSpecable to a sink
```

2. 可选：您也可以以 YAML 格式列出可用事件源类型：

```
$ kn source list-types -o yaml
```

4.5.1.2. Knative CLI sink 标记

当使用 Knative (**kn**) CLI 创建事件源时，您可以使用 **--sink** 标志指定事件从该资源发送到的接收器。sink 可以是任何可寻址或可调用的资源，可以从其他资源接收传入的事件。

以下示例创建使用服务 **http://event-display.svc.cluster.local** 的接收器绑定作为接收器：

使用 sink 标记的命令示例

```
$ kn source binding create bind-heartbeat \
  --namespace sinkbinding-example \
  --subject "Job:batch/v1:app=heartbeat-cron" \
  --sink http://event-display.svc.cluster.local \ 1
  --ce-override "sink=bound"
```

1 **http://event-display.svc.cluster.local** 中的 **svc** 确定接收器是一个 Knative 服务。其他默认的接收器前缀包括 **channel** 和 **broker**。

4.5.1.3. 使用 Knative CLI 创建和管理容器源

您可以使用 **kn source container** 命令来使用 Knative (**kn**) 创建和管理容器源。使用 Knative CLI 创建事件源提供了比直接修改 YAML 文件更精简且直观的用户界面。

创建容器源

```
$ kn source container create <container_source_name> --image <image_uri> --sink <sink>
```

删除容器源

```
$ kn source container delete <container_source_name>
```

描述容器源

```
$ kn source container describe <container_source_name>
```

列出现有容器源

```
$ kn source container list
```

以 YAML 格式列出现有容器源

```
$ kn source container list -o yaml
```

更新容器源

此命令为现有容器源更新镜像 URI：

-

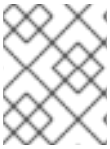
```
$ kn source container update <container_source_name> --image <image_uri>
```

4.5.1.4. 使用 Knative CLI 创建 API 服务器源

您可以使用 **kn source apiserver create** 命令，使用 **kn** CLI 创建 API 服务器源。使用 **kn** CLI 创建 API 服务器源可提供比直接修改 YAML 文件更精简且直观的用户界面。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Eventing。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。
- 已安装 OpenShift CLI(**oc**)。
- 已安装 Knative (**kn**) CLI。



流程

如果要重新使用现有服务帐户，您可以修改现有的 **ServiceAccount** 资源，使其包含所需的权限，而不是创建新资源。

1. 以 YAML 文件形式,为事件源创建服务帐户、角色和角色绑定：

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: events-sa
  namespace: default 1

---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: event-watcher
  namespace: default 2
rules:
  - apiGroups:
      - ""
    resources:
      - events
    verbs:
      - get
      - list
      - watch

---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: k8s-ra-event-watcher
  namespace: default 3
roleRef:
```

```
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: event-watcher
subjects:
  - kind: ServiceAccount
    name: events-sa
    namespace: default 4
```

1 2 3 4 将这个命名空间更改为已选择安装事件源的命名空间。

2. 应用 YAML 文件：

```
$ oc apply -f <filename>
```

3. 创建具有事件 sink 的 API 服务器源。在以下示例中，sink 是一个代理：

```
$ kn source apiserver create <event_source_name> --sink broker:<broker_name> --
resource "event:v1" --service-account <service_account_name> --mode Resource
```

4. 要检查 API 服务器源是否已正确设置，请创建一个 Knative 服务，在日志中转储传入的信息：

```
$ kn service create <service_name> --image quay.io/openshift-knative/knative-eventing-
sources-event-display:latest
```

5. 如果您使用代理作为事件 sink，请创建一个触发器将事件从 **default** 代理过滤到服务：

```
$ kn trigger create <trigger_name> --sink ksvc:<service_name>
```

6. 通过在 default 命名空间中启动 pod 来创建事件：

```
$ oc create deployment hello-node --image quay.io/openshift-knative/knative-eventing-
sources-event-display:latest
```

7. 通过检查以下命令生成的输出来检查是否正确映射了控制器：

```
$ kn source apiserver describe <source_name>
```

输出示例

```
Name:              mysource
Namespace:         default
Annotations:       sources.knative.dev/creator=developer,
sources.knative.dev/lastModifier=developer
Age:               3m
ServiceAccountName:  events-sa
Mode:              Resource
Sink:
  Name:       default
  Namespace:  default
  Kind:       Broker (eventing.knative.dev/v1)
Resources:
  Kind:       event (v1)
```

```
  Controller:  false
Conditions:
  OK TYPE                  AGE REASON
  ++ Ready                 3m
  ++ Deployed               3m
  ++ SinkProvided            3m
  ++ SufficientPermissions     3m
  ++ EventTypesProvided        3m
```

验证

您可以通过查看消息转储程序功能日志来验证 Kubernetes 事件是否已发送到 Knative。

1. 获取 pod:

```
$ oc get pods
```

2. 查看 pod 的消息转储程序功能日志：

```
$ oc logs $(oc get pod -o name | grep event-display) -c user-container
```

输出示例

```
☁  cloudevents.Event
Validation: valid
Context Attributes,
  specversion: 1.0
  type: dev.knative.apiserver.resource.update
  datacontenttype: application/json
  ...
Data,
  {
    "apiVersion": "v1",
    "involvedObject": {
      "apiVersion": "v1",
      "fieldPath": "spec.containers{hello-node}",
      "kind": "Pod",
      "name": "hello-node",
      "namespace": "default",
       .....
    },
    "kind": "Event",
    "message": "Started container",
    "metadata": {
      "name": "hello-node.159d7608e3a3572c",
      "namespace": "default",
      ....
    },
    "reason": "Started",
    ...
  }
```

删除 API 服务器源

1. 删除触发器：

```
$ kn trigger delete <trigger_name>
```

2. 删除事件源：

```
$ kn source apiserver delete <source_name>
```

3. 删除服务帐户、集群角色和集群绑定：

```
$ oc delete -f authentication.yaml
```

4.5.1.5. 使用 Knative CLI 创建 ping 源

您可以使用 **kn source ping create** 命令，通过 Knative (**kn**) CLI 创建 ping 源。使用 Knative CLI 创建事件源提供了比直接修改 YAML 文件更精简且直观的用户界面。

先决条件

- 在集群中安装了 OpenShift Serverless Operator、Knative Serving 和 Knative Eventing。
- 已安装 Knative (**kn**) CLI。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。
- 可选：如果要使用此流程验证步骤，请安装 OpenShift CLI (**oc**) 。

流程

1. 要验证 ping 源是否可以工作，请创建一个简单的 Knative 服务，在服务日志中转储传入的信息：

```
$ kn service create event-display \
    --image quay.io/openshift-knative/knative-eventing-sources-event-display:latest
```

2. 对于您要请求的每一组 ping 事件，请在与事件消费者相同的命名空间中创建一个 ping 源：

```
$ kn source ping create test-ping-source \
    --schedule "*/2 * * * *" \
    --data '{"message": "Hello world!"}' \
    --sink ksvc:event-display
```

3. 输入以下命令并检查输出，检查是否正确映射了控制器：

```
$ kn source ping describe test-ping-source
```

输出示例

```
Name:         test-ping-source
Namespace:    default
Annotations:  sources.knative.dev/creator=developer,
sources.knative.dev/lastModifier=developer
Age:          15s
Schedule:     */2 * * * *
```

```
Data:       {"message": "Hello world!"}

Sink:
  Name:       event-display
  Namespace:  default
  Resource:   Service (serving.knative.dev/v1)

Conditions:
  OK TYPE                 AGE REASON
  ++ Ready                8s
  ++ Deployed             8s
  ++ SinkProvided         15s
  ++ ValidSchedule        15s
  ++ EventTypeProvided    15s
  ++ ResourcesCorrect     15s
```

验证

您可以通过查看 sink pod 的日志来验证 Kubernetes 事件是否已发送到 Knative 事件。

默认情况下，如果在 60 秒内都没有流量，Knative 服务会终止其 Pod。本指南中演示的示例创建了一个 ping 源，每 2 分钟发送一条消息，因此每个消息都应该在新创建的 pod 中观察到。

1. 查看新创建的 pod：

```
$ watch oc get pods
```

2. 使用 Ctrl+C 取消查看 pod，然后查看所创建 pod 的日志：

```
$ oc logs $(oc get pod -o name | grep event-display) -c user-container
```

输**出示例**

```
☁  cloudevents.Event
Validation: valid
Context Attributes,
  specversion: 1.0
  type: dev.knative.sources.ping
  source: /apis/v1/namespaces/default/pingsources/test-ping-source
  id: 99e4f4f6-08ff-4bff-acf1-47f61ded68c9
  time: 2020-04-07T16:16:00.000601161Z
  datacontenttype: application/json
Data,
  {
    "message": "Hello world!"
  }
```

删除 ping 源

- 删除 ping 源：

```
$ kn delete pingsources.sources.knative.dev <ping_source_name>
```

4.5.1.6. 使用 Knative CLI 创建 Kafka 事件源

您可以使用 **kn source kafka create** 命令，使用 Knative (**kn**) CLI 创建 Kafka 源。使用 Knative CLI 创建事件源提供了比直接修改 YAML 文件更精简且直观的用户界面。

先决条件

- OpenShift Serverless Operator、Knative Eventing、Knative Serving 和 **KnativeKafka** 自定义资源（CR）已安装在集群中。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。
- 您可以访问 Red Hat AMQ Streams（Kafka）集群，该集群会生成您要导入的 Kafka 信息。
- 已安装 Knative (**kn**) CLI。
- 可选：如果您想要使用此流程中的验证步骤，已安装 OpenShift CLI (**oc**)。

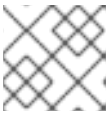
流程

1. 要验证 Kafka 事件源是否可以工作，请创建一个 Knative 服务，在服务日志中转储传入的事件：

```
$ kn service create event-display \
  --image quay.io/openshift-knative/knative-eventing-sources-event-display
```

2. 创建 **KafkaSource** CR：

```
$ kn source kafka create <kafka_source_name> \
  --servers <cluster_kafka_bootstrap>.kafka.svc:9092 \
  --topics <topic_name> --consumergroup my-consumer-group \
  --sink event-display
```



注意

将此命令中的占位符值替换为源名称、引导服务器和主题的值。

--servers、**--topics** 和 **--consumergroup** 选项指定到 Kafka 集群的连接参数。**--consumergroup** 选项是可选的。

3. 可选：查看您创建的 **KafkaSource** CR 的详情：

```
$ kn source kafka describe <kafka_source_name>
```

输**出示例**

```
Name:              example-kafka-source
Namespace:         kafka
Age:               1h
BootstrapServers:  example-cluster-kafka-bootstrap.kafka.svc:9092
Topics:            example-topic
ConsumerGroup:     example-consumer-group

Sink:
  Name:       event-display
  Namespace:  default
```

```
  Resource:   Service (serving.knative.dev/v1)

Conditions:
  OK TYPE           AGE REASON
  ++ Ready          1h
  ++ Deployed        1h
  ++ SinkProvided    1h
```

验证步骤

1. 触发 Kafka 实例将信息发送到主题：

```
$ oc -n kafka run kafka-producer \
    -ti --image=quay.io/strimzi/kafka:latest-kafka-2.7.0 --rm=true \
    --restart=Never -- bin/kafka-console-producer.sh \
    --broker-list <cluster_kafka_bootstrap>:9092 --topic my-topic
```

在提示符后输入信息。这个命令假设：

- Kafka 集群安装在 **kafka** 命名空间中。
- **KafkaSource** 对象已被配置为使用 **my-topic** 主题。

2. 通过查看日志来验证消息是否显示：

```
$ oc logs $(oc get pod -o name | grep event-display) -c user-container
```

输**出示例**

```
☁  cloudevents.Event
Validation: valid
Context Attributes,
  specversion: 1.0
  type: dev.knative.kafka.event
  source: /apis/v1/namespaces/default/kafkasources/example-kafka-source#example-topic
  subject: partition:46#0
  id: partition:46/offset:0
  time: 2021-03-10T11:21:49.4Z
Extensions,
  traceparent: 00-161ff3815727d8755848ec01c866d1cd-7ff3916c44334678-00
Data,
  Hello!
```

4.6. 功能命令

4.6.1. 创建功能

在构建和部署功能之前，您必须使用 Knative (**kn**) CLI 创建功能。您可以在命令行中指定路径、运行时、模板和镜像 registry，也可以使用 **-c** 标志在终端中启动交互式体验。

重要

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

有关红帽技术预览功能支持范围的详情，请参考 https://access.redhat.com/support/offerings/techpreview/。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 已安装 Knative (**kn**) CLI。

流程

- 创建功能项目：

```
$ kn func create -r <repository> -l <runtime> -t <template> <path>
```

  - 可接受的运行时值包括 **quarkus**、**node**、**typescript**、**go**、**python**、**springboot** 和 **rust**。
  - 可接受的模板值包括 **http** 和 **cloudevents**。

    示例命令

```
$ kn func create -l typescript -t cloudevents examplefunc
```

    输出示例

```
Created typescript function in /home/user/demo/examplefunc
```

  - 或者，您可以指定包含自定义模板的存储库。

    示例命令

```
$ kn func create -r https://github.com/boson-project/templates/ -l node -t hello-world
examplefunc
```

    输出示例

```
Created node function in /home/user/demo/examplefunc
```

4.6.2. 在本地运行一个函数

您可以使用 **kn func run** 命令在当前目录中本地运行函数，或者在 **--path** 标志指定的目录中运行。如果您运行的函数之前没有被构建，或者项目文件自上次构建以来已修改过，**kn func run** 命令将在运行它前构建该函数。

在当前目录中运行函数的命令示例

```
$ kn func run
```

-

在指定为路径的目录中运行函数的示例

```
$ kn func run --path=<directory_path>
```

您也可以在运行该函数前强制重建现有镜像，即使项目文件没有更改项目文件，则使用 **--build** 标志：

使用 build 标记的 run 命令示例

```
$ kn func run --build
```

如果将 **build** 标志设置为 false，这将禁用构建镜像，并使用之前构建的镜像运行该功能：

使用 build 标记的 run 命令示例

```
$ kn func run --build=false
```

您可以使用 help 命令了解更多有关 **kn func run** 命令选项的信息：

构建 help 命令

```
$ kn func help run
```

4.6.3. 构建函数

在运行功能前，您必须构建 function 项目。如果使用 **kn func run** 命令，则该函数会自动构建。但是，您可以使用 **kn func build** 命令在不运行的情况下构建函数，这对于高级用户或调试场景非常有用。

kn func build 命令创建可在您的计算机或 OpenShift Container Platform 集群中运行的 OCI 容器镜像。此命令使用功能项目名称和镜像 registry 名称为您的功能构建完全限定镜像名称。

4.6.3.1. 镜像容器类型

默认情况下，**kn func build** 使用 Red Hat Source-to-Image (S2I) 技术创建一个容器镜像。

使用 Red Hat Source-to-Image (S2I) 的 build 命令示例.

```
$ kn func build
```

您可以通过在命令中添加 **--builder** 标志并指定 **pack** 策略，来使用 CNCF Cloud Native Buildpacks 技术：

使用 CNCF Cloud Native Buildpacks 的 build 命令示例

```
$ kn func build --builder pack
```

4.6.3.2. 镜像 registry 类型

OpenShift Container Registry 默认用作存储功能镜像的镜像 registry。

使用 OpenShift Container Registry 的 build 命令示例

-

```
$ kn func build
```

输出示例

```
Building function image
Function image has been built, image: registry.redhat.io/example/example-function:latest
```

您可以使用 **--registry** 标志覆盖使用 OpenShift Container Registry 作为默认镜像 registry：

build 命令覆盖 OpenShift Container Registry 以使用 quay.io

```
$ kn func build --registry quay.io/username
```

输出示例

```
Building function image
Function image has been built, image: quay.io/username/example-function:latest
```

4.6.3.3. push 标记

您可以将 **--push** 标志添加到 **kn func build** 命令中，以便在成功构建后自动推送功能镜像：

使用 OpenShift Container Registry 的 build 命令示例

```
$ kn func build --push
```

4.6.3.4. help 命令

您可以使用 help 命令了解更多有关 **kn func build** 命令选项的信息：

构建 help 命令

```
$ kn func help build
```

4.6.4. 部署功能

您可以使用 **kn func deploy** 命令将功能部署到集群中，作为 Knative 服务。如果已经部署了目标功能，则会使用推送到容器镜像 registry 的新容器镜像进行更新，并更新 Knative 服务。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 已安装 Knative (**kn**) CLI。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。
- 您必须已创建并初始化要部署的功能。

流程

- 部署功能：

```
$ kn func deploy [-n <namespace> -p <path> -i <image>]
```

输**出示例**

```
Function deployed at: http://func.example.com
```

  - 如果没有指定 **namespace**，则该函数部署到当前命名空间中。
  - 此函数从当前目录中部署，除非指定了 **path**。
  - Knative 服务名称派生自项目名称，无法使用此命令进行更改。

4.6.5. 列出现有功能

您可以使用 **kn func list** 列出现有功能。如果要列出部署为 Knative 服务的功能，也可以使用 **kn service list**。

流程

- 列出现有功能：

```
$ kn func list [-n <namespace> -p <path>]
```

输**出示例**

```
NAME            NAMESPACE  RUNTIME  URL
READY
example-function default    node     http://example-function.default.apps.ci-ln-g9f36hb-
d5d6b.origin-ci-int-aws.dev.rhcloud.com  True
```

- 列出部署为 Knative 服务的功能：

```
$ kn service list -n <namespace>
```

输**出示例**

```
NAME            URL                                                                LATEST
AGE   CONDITIONS   READY   REASON
example-function   http://example-function.default.apps.ci-ln-g9f36hb-d5d6b.origin-ci-int-
aws.dev.rhcloud.com   example-function-gzl4c   16m   3 OK / 3     True
```

4.6.6. 描述函数

kn func info 命令输出有关已部署功能的信息，如功能名称、镜像、命名空间、Knative 服务信息、路由信息和事件订阅。

流程

- 描述函数：

```
$ kn func info [-f <format> -n <namespace> -p <path>]
```

示例命令

```
$ kn func info -p function/example-function
```

输**出示例**

```
Function name:
  example-function
Function is built in image:
  docker.io/user/example-function:latest
Function is deployed as Knative Service:
  example-function
Function is deployed in namespace:
  default
Routes:
  http://example-function.default.apps.ci-ln-g9f36hb-d5d6b.origin-ci-int-aws.dev.rhcloud.com
```

4.6.7. 使用测试事件调用部署的功能

您可以使用 **kn func invoke** CLI 命令发送测试请求，在本地或 OpenShift Container Platform 集群中调用功能。您可以使用此命令测试功能是否正常工作并且能够正确接收事件。本地调用函数可用于在功能开发期间进行快速测试。在测试与生产环境更接近的测试时，在集群中调用函数非常有用。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 已安装 Knative (**kn**) CLI。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。
- 您必须已部署了要调用的功能。

流程

- 调用函数：

```
$ kn func invoke
```

  - **kn func invoke** 命令仅在当前运行本地容器镜像时或在集群中部署功能时才有效。
  - **kn func invoke** 命令默认在本地目录上执行，并假定此目录是一个功能项目。

4.6.7.1. kn func 调用可选参数

您可以使用以下 **kn func invoke** CL 命令标记为请求指定可选参数。

标记	描述
-t,--target	指定调用函数的目标实例，如 **local** 或 **remote** 或 https://staging.example.com/。默认目标是 **local**。
-f,--format	指定消息的格式，如 **cloudevent** 或 **http**。
--id	指定请求的唯一字符串标识符。
-n,--namespace	指定集群上的命名空间。
--source	指定请求的发件人名称。这与 CloudEvent **source** 属性对应。
--type	指定请求类型，例如 **boson.fn**。这与 CloudEvent **type** 属性对应。
--data	指定请求的内容。对于 CloudEvent 请求，这是 CloudEvent **data** 属性。
--file	指定包含要发送数据的本地文件的路径。
--content-type	指定请求的 MIME 内容类型。
-p,--path	指定项目目录的路径。
-c,--confirm	启用系统提示，以交互方式确认所有选项。
-v,--verbose	启用打印详细输出。
-h,--help	输出有关使用 **kn func invoke** 的信息。

4.6.7.1.1. 主要参数

以下参数定义 **kn func invoke** 命令的主要属性：

事件目标 (-t,--target)

调用函数的目标实例。接受 **local** 值用于本地部署的函数、**remote** 值用于远程部署函数，或一个 URL 用于一个任意的端点。如果没有指定目标，则默认为 **local**。

事件消息格式 (-f,--format)

事件的消息格式，如 **http** 或 **cloudevent**。默认为创建函数时使用的模板格式。

事件类型 (--type)

发送的事件类型。您可以查找有关各个事件制作者文档中设置的 **type** 参数的信息。例如，API 服务器源可能会将生成的事件的 **type** 参数设置为 **dev.knative.apiserver.resource.update**。

事件源 (--source)

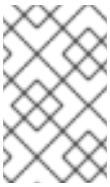
生成该事件的唯一事件源。这可能是事件源的 URI，如 **https://10.96.0.1/** 或事件源的名称。

事件 ID (--id)

由事件制作者创建的随机唯一 ID。

事件数据 (--data)

允许您为 **kn func invoke** 命令发送的事件指定 **data** 值。例如，您可以指定一个 **--data** 值，如 **"Hello World"**，以便事件包含此数据字符串。默认情况下，**kn func invoke** 创建的事件中不包含任何数据。



注意

已部署到集群的功能可以对现有事件源的事件响应，该源提供属性（如 **source** 和 **type**）的值。这些事件通常具有 JSON 格式的 **data** 值，用于捕获事件的特定域上下文。通过使用本文档中介绍的 CLI 标志，开发人员可以模拟这些事件以进行本地测试。

您还可以使用 **--file** 标志发送事件数据，以提供包含事件数据的本地文件。在这种情况下，使用 **--content-type** 指定内容类型。

数据内容类型 (--content-type)

如果您使用 **--data** 标志为事件添加数据，您可以使用 **--content-type** 标志指定事件传输的数据类型。在上例中，数据是纯文本，因此您可以指定 **kn func call --data "Hello world!" --content-type "text/plain** "。

4.6.7.1.2. 示例命令

这是 **kn func invoke** 命令的一般调用：

```
$ kn func invoke --type <event_type> --source <event_source> --data <event_data> --content-type
<content_type> --id <event_ID> --format <format> --namespace <namespace>
```

例如，要发送 "Hello world!" 事件，您可以运行：

```
$ kn func invoke --type ping --source example-ping --data "Hello world!" --content-type "text/plain" --
id example-ID --format http --namespace my-ns
```

4.6.7.1.2.1. 使用数据指定文件

要指定磁盘上包含事件数据的文件，请使用 **--file** 和 **--content-type** 标志：

```
$ kn func invoke --file <path> --content-type <content-type>
```

例如，要发送存储在 **test.json** 文件中的 JSON 数据，请使用以下命令：

```
$ kn func invoke --file ./test.json --content-type application/json
```

4.6.7.1.2.2. 指定功能项目

您可以使用 **--path** 标志指定到功能项目的路径：

```
$ kn func invoke --path <path_to_function>
```

例如，要使用位于 **./example/example-function** 目录中的功能项目，请使用以下命令：

```
$ kn func invoke --path ./example/example-function
```

4.6.7.1.2.3. 指定部署目标功能的位置

默认情况下，**kn func invoke** 作为功能本地部署的目标：

```
$ kn func invoke
```

要使用不同的部署，请使用 **--target** 标志：

```
$ kn func invoke --target <target>
```

例如，要使用在集群中部署的功能，请使用 **--target remote** 标志：

```
$ kn func invoke --target remote
```

要使用在任意 URL 中部署的功能，请使用 **--target <URL>** 标志：

```
$ kn func invoke --target "https://my-event-broker.example.com"
```

您可以明确以本地部署为目标。在这种情况下，如果这个功能没有在本地运行，命令会失败：

```
$ kn func invoke --target local
```

4.6.8. 删除函数

您可以使用 **kn func delete** 命令删除功能。当不再需要某个函数时，这很有用，并有助于在集群中保存资源。

流程

- 删除函数：

```
$ kn func delete [<function_name> -n <namespace> -p <path>]
```

  - 如果没有指定要删除的功能的名称或路径，则会搜索当前目录以查找用于决定要删除的功能的 **func.yaml** 文件。
  - 如果没有指定命名空间，则默认为 **func.yaml** 文件中的 **namespace** 值。

第 5 章 开发

5.1. 无服务器应用程序

无服务器应用程序已创建并部署为 Kubernetes 服务，由路由和配置定义，并包含在 YAML 文件中。要使用 OpenShift Serverless 部署无服务器应用程序，您必须创建一个 Knative **Service** 对象。

Knative Service 对象 YAML 文件示例

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: hello 1
  namespace: default 2
spec:
  template:
    spec:
      containers:
        - image: docker.io/openshift/hello-openshift 3
          env:
            - name: RESPONSE 4
              value: "Hello Serverless!"
```

1. 应用程序的名称。
2. 应用程序使用的命名空间。
3. 应用程序的镜像。
4. 示例应用程序输出的环境变量。

使用以下任一方法创建一个无服务器应用程序：

- 从 OpenShift Container Platform web 控制台创建 Knative 服务。请参阅有关使用 Developer 视角创建应用程序 的文档。
- 使用 Knative (**kn**) CLI 创建 Knative 服务。
- 使用 **oc** CLI 创建并应用 Knative **Service** 对象作为 YAML 文件。

5.1.1. 使用 Knative CLI 创建无服务器应用程序

通过使用 Knative (**kn**) CLI 创建无服务器应用程序，通过直接修改 YAML 文件来提供更精简且直观的用户界面。您可以使用 **kn service create** 命令创建基本无服务器应用程序。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 已安装 Knative (**kn**) CLI。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。

流程

- 创建 Knative 服务：

```
$ kn service create <service_name> --image <image> --tag <tag-value>
```

其中：

  - **--image** 是应用的镜像的 URI。
  - **--tag** 是一个可选标志，可用于向利用服务创建的初始修订版本添加标签。

示例命令

```
$ kn service create event-display \
    --image quay.io/openshift-knative/knative-eventing-sources-event-display:latest
```

输**出示例**

```
Creating service 'event-display' in namespace 'default':

  0.271s The Route is still working to reflect the latest desired specification.
  0.580s Configuration "event-display" is waiting for a Revision to become ready.
  3.857s ...
  3.861s Ingress has not yet been reconciled.
  4.270s Ready to serve.

Service 'event-display' created with latest revision 'event-display-bxshg-1' and URL:
http://event-display-default.apps-crc.testing
```

5.1.2. 使用离线模式创建服务

您可以在离线模式下执行 **kn service** 命令，以便集群中不会发生任何更改，而是在本地机器上创建服务描述符文件。创建描述符文件后，您可以在向集群传播更改前修改该文件。

重要

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

有关红帽技术预览功能支持范围的详情，请参考 https://access.redhat.com/support/offerings/techpreview/。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 已安装 Knative (**kn**) CLI。

流程

1. 在离线模式下，创建一个本地 Knative 服务描述符文件：

```
$ kn service create event-display \
    --image quay.io/openshift-knative/knative-eventing-sources-event-display:latest \
    --target ./ \
    --namespace test
```

输**出示例**

```
Service 'event-display' created in namespace 'test'.
```

- **--target ./** 标志启用脱机模式，并将 **./** 指定为用于存储新目录树的目录。
  如果您没有指定现有目录，但使用文件名，如 **--target my-service.yaml**，则不会创建目录树。相反，当前目录中只创建服务描述符 **my-service.yaml** 文件。

  文件名可以具有 **.yaml**、**.yml** 或 **.json** 扩展名。选择 **.json** 以 JSON 格式创建服务描述符文件。

- **namespace test** 选项将新服务放在 **test** 命名空间中。
  如果不使用 **--namespace**，并且您登录了 OpenShift 集群，则会在当前命名空间中创建描述符文件。否则，描述符文件会在 **default** 命名空间中创建。

2. 检查创建的目录结构：

```
$ tree ./
```

输**出示例**

```
./
└── test
    └── ksvc
        └── event-display.yaml

2 directories, 1 file
```

- 使用 **--target** 指定的当前 **./** 目录包含新的 **test/** 目录，它在指定的命名空间后命名。
- **test/** 目录包含 **ksvc**，它在资源类型后命名。
- **ksvc** 目录包含描述符文件 **event-display.yaml**，它根据指定的服务名称命名。

3. 检查生成的服务描述符文件：

```
$ cat test/ksvc/event-display.yaml
```

输**出示例**

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  creationTimestamp: null
  name: event-display
  namespace: test
spec:
  template:
```

```
    metadata:
      annotations:
        client.knative.dev/user-image: quay.io/openshift-knative/knative-eventing-sources-event-
display:latest
      creationTimestamp: null
    spec:
      containers:
      - image: quay.io/openshift-knative/knative-eventing-sources-event-display:latest
        name: ""
        resources: {}
status: {}
```

4. 列出新服务的信息：

```
$ kn service describe event-display --target ./ --namespace test
```

输**出示例**

```
Name:       event-display
Namespace:  test
Age:
URL:

Revisions:

Conditions:
  OK TYPE    AGE REASON
```

- **--target ./** 选项指定包含命名空间子目录的目录结构的根目录。
  另外，您可以使用 **--target** 选项直接指定 YAML 或 JSON 文件名。可接受的文件扩展包括 **.yaml**、**.yml** 和 **.json**。

- **--namespace** 选项指定命名空间，与 **kn** 通信包含所需服务描述符文件的子目录。
  如果您不使用 **--namespace**，且您已登录到 OpenShift 集群，**kn** 在以当前命名空间命名的子目录中搜索该服务。否则，**kn** 在 **default/** 子目录中搜索。

5. 使用服务描述符文件在集群中创建服务：

```
$ kn service create -f test/ksvc/event-display.yaml
```

输**出示例**

```
Creating service 'event-display' in namespace 'test':

  0.058s The Route is still working to reflect the latest desired specification.
  0.098s ...
  0.168s Configuration "event-display" is waiting for a Revision to become ready.
 23.377s ...
 23.419s Ingress has not yet been reconciled.
 23.534s Waiting for load balancer to be ready
 23.723s Ready to serve.

Service 'event-display' created to latest revision 'event-display-00001' is available at URL:
http://event-display-test.apps.example.com
```

5.1.3. 使用 YAML 创建无服务器应用程序

使用 YAML 文件创建 Knative 资源使用声明性 API，它允许您以声明性的方式描述应用程序，并以可重复的方式描述应用程序。要使用 YAML 创建无服务器应用程序，您必须创建一个 YAML 文件来定义 Knative **Service** 对象，然后使用 **oc apply** 来应用它。

创建服务并部署应用程序后，Knative 会为应用程序的这个版本创建一个不可变的修订版本。Knative 还将执行网络操作，为您的应用程序创建路由、入口、服务和负载平衡器，并根据流量自动扩展或缩减 pod。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。
- 安装 OpenShift CLI (**oc**) 。

流程

1. 创建包含以下示例代码的 YAML 文件：

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: event-delivery
  namespace: default
spec:
  template:
    spec:
      containers:
        - image: quay.io/openshift-knative/knative-eventing-sources-event-display:latest
          env:
            - name: RESPONSE
              value: "Hello Serverless!"
```

2. 导航到包含 YAML 文件的目录，并通过应用 YAML 文件来部署应用程序：

```
$ oc apply -f <filename>
```

5.1.4. 验证无服务器应用程序的部署

要验证您的无服务器应用程序是否已成功部署，您必须获取 Knative 创建的应用程序的 URL，然后向该 URL 发送请求并检查其输出。OpenShift Serverless 支持 HTTP 和 HTTPS URL，但 **oc get ksvc** 的输出始终使用 **http://** 格式打印 URL。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 已安装 **oc** CLI。
- 您已创建了 Knative 服务。

先决条件

- 安装 OpenShift CLI (**oc**) 。

流程

1. 查找应用程序 URL:

```
$ oc get ksvc <service_name>
```

示例命令

```
$ oc get ksvc event-delivery
```

输**出示例**

```
NAME           URL                                        LATESTCREATED        LATESTREADY          READY   REASON
event-delivery   http://event-delivery-default.example.com   event-delivery-4wsd2   event-delivery-4wsd2   True
```

2. 向集群发出请求并观察其输出。

HTTP 请求示例

```
$ curl http://event-delivery-default.example.com
```

HTTPS 请求示例

```
$ curl https://event-delivery-default.example.com
```

输**出示例**

```
Hello Serverless!
```

3. 可选。如果您在证书链中收到与自签名证书相关的错误，可以在 curl 命令中添加 **--insecure** 标志来忽略以下错误：

```
$ curl https://event-delivery-default.example.com --insecure
```

输**出示例**

```
Hello Serverless!
```

重要

在生产部署中不能使用自签名证书。这个方法仅用于测试目的。

4. 可选。如果 OpenShift Container Platform 集群配置有证书颁发机构 (CA) 签名但尚未为您的系统配置全局证书，您可以使用 **curl** 命令指定此证书。证书的路径可使用 **--cacert** 标志传递给 curl 命令：

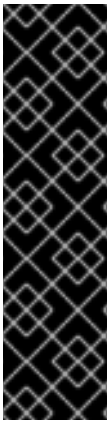
```
$ curl https://event-delivery-default.example.com --cacert <file>
```

输**出示例**

```
Hello Serverless!
```

5.1.5. 使用 HTTP2 和 gRPC 与无服务器应用程序交互

OpenShift Serverless 只支持不安全或边缘终端路由。不安全或边缘终端路由不支持 OpenShift Container Platform 中的 HTTP2。这些路由也不支持 gRPC，因为 gRPC 由 HTTP2 传输。如果您在应用程序中使用这些协议，则必须使用入口（ingress）网关直接调用应用程序。要做到这一点，您必须找到 ingress 网关的公共地址以及应用程序的特定主机。



重要

此方法需要使用 **LoadBalancer** 服务类型公开 Kourier 网关。您可以通过在 **KnativeServing** 自定义资源定义 (CRD) 中添加以下 YAML 来配置：

```
...
spec:
  ingress:
    kourier:
      service-type: LoadBalancer
...
```

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 安装 OpenShift CLI (**oc**) 。
- 您已创建了 Knative 服务。

流程

1. 找到应用程序主机。请参阅*验证无服务器应用程序部署*中的相关内容。
2. 查找 ingress 网关的公共地址：

```
$ oc -n knative-serving-ingress get svc kourier
```

输**出示例**

```
NAME                TYPE           CLUSTER-IP      EXTERNAL-IP
PORT(S)
AGE
kourier   LoadBalancer   172.30.51.103   a83e86291bcdd11e993af02b7a65e514-
33544245.us-east-1.elb.amazonaws.com   80:31380/TCP,443:31390/TCP   67m
```

公共地址位于 **EXTERNAL-IP** 字段，在本例中是 **a83e86291bcdd11e993af02b7a65e514-33544245.us-east-1.elb.amazonaws.com**。

3. 手动在 HTTP 请求的主机标头中设置应用程序的主机，但将请求定向到 ingress 网关的公共地址。

```
$ curl -H "Host: hello-default.example.com" a83e86291bcdd11e993af02b7a65e514-33544245.us-east-1.elb.amazonaws.com
```

输出示例

```
Hello Serverless!
```

您还可以通过将授权设置为应用程序的主机来发出 gRPC 请求，同时将请求直接定向到 ingress 网关：

```
grpc.Dial(
    "a83e86291bcdd11e993af02b7a65e514-33544245.us-east-1.elb.amazonaws.com:80",
    grpc.WithAuthority("hello-default.example.com:80"),
    grpc.WithInsecure(),
)
```

注意

如上例所示，请确保将对应的端口（默认为 80）附加到两个主机中。

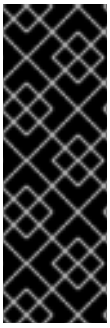
5.1.6. 在具有限制性网络策略的集群中启用与 Knative 应用程序通信

如果您使用多个用户可访问的集群，您的集群可能会使用网络策略来控制哪些 pod、服务和命名空间可以通过网络相互通信。如果您的集群使用限制性网络策略，Knative 系统 Pod 可能无法访问 Knative 应用程序。例如，如果您的命名空间具有以下网络策略（拒绝所有请求），Knative 系统 pod 无法访问您的 Knative 应用程序：

拒绝对命名空间的所有请求的 NetworkPolicy 对象示例

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: deny-by-default
  namespace: example-namespace
spec:
  podSelector:
  ingress: []
```

要允许从 Knative 系统 pod 访问应用程序，您必须为每个 Knative 系统命名空间添加标签，然后在应用程序命名空间中创建一个 **NetworkPolicy** 对象，以便为具有此标签的其他命名空间访问命名空间。



重要

拒绝对集群中非原生服务的请求的网络策略仍阻止访问这些服务。但是，通过允许从 Knative 系统命名空间访问 Knative 应用程序，您可以从集群中的所有命名空间中访问 Knative 应用程序。

如果您不想允许从集群中的所有命名空间中访问 Knative 应用程序，您可能需要为 Knative 服务使用 *JSON Web Token 身份验证* 。Knative 服务的 JSON Web 令牌身份验证需要 Service Mesh。

先决条件

- 安装 OpenShift CLI (**oc**) 。
- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。

流程

1. 将 **knative.openshift.io/system-namespace=true** 标签添加到需要访问应用程序的每个 Knative 系统命名空间：
   a. 标记 **knative-serving** 命名空间：

   ```
   $ oc label namespace knative-serving knative.openshift.io/system-namespace=true
   ```

   b. 标记 **knative-serving-ingress** 命名空间：

   ```
   $ oc label namespace knative-serving-ingress knative.openshift.io/system-namespace=true
   ```

   c. 标记 **knative-eventing** 命名空间：

   ```
   $ oc label namespace knative-eventing knative.openshift.io/system-namespace=true
   ```

   d. 标记 **knative-kafka** 命名空间：

   ```
   $ oc label namespace knative-kafka knative.openshift.io/system-namespace=true
   ```

2. 在应用程序命名空间中创建一个 **NetworkPolicy** 对象，允许从带有 **knative.openshift.io/system-namespace** 标签的命名空间访问：

   NetworkPolicy 对象示例

   ```
   apiVersion: networking.k8s.io/v1
   kind: NetworkPolicy
   metadata:
     name: <network_policy_name> 1
     namespace: <namespace> 2
   spec:
     ingress:
     - from:
       - namespaceSelector:
           matchLabels:
             knative.openshift.io/system-namespace: "true"
     podSelector: {}
     policyTypes:
     - Ingress
   ```

   1 为您的网络策略提供名称。

   2 应用程序所在的命名空间。

5.1.7. 配置 init 容器

Init 容器是 pod 中应用程序容器之前运行的专用容器。它们通常用于为应用程序实施初始化逻辑，其中可能包括运行设置脚本或下载所需的配置。

注意

Init 容器可能会导致应用程序的启动时间较长，应该谨慎地用于无服务器应用程序，这应该经常被扩展或缩减。

单个 Knative 服务 spec 支持多个 init 容器。如果没有提供模板名称，Knative 提供一个默认的可配置命名模板。通过在 Knative **Service** 对象 spec 中添加适当的值，可以设置 init 容器模板。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 在将 init 容器用于 Knative 服务前，管理员必须在 **KnativeServing** 自定义资源 (CR) 中添加 **kubernetes.podspec-init-containers** 标志。如需更多信息，请参阅 OpenShift Serverless "Global configuration" 文档。

流程

- 将 **initContainers** spec 添加到 Knative **Service** 对象中：

  服务规格示例

```
apiVersion: serving.knative.dev/v1
kind: Service
...
spec:
  template:
    spec:
      initContainers:
        - imagePullPolicy: IfNotPresent (1)
          image: <image_uri> (2)
          volumeMounts: (3)
            - name: data
              mountPath: /data
...
```

  (1) 镜像下载时的镜像拉取策略。

  (2) init 容器镜像的 URI。

  (3) 卷在容器文件系统中挂载的位置。

5.1.8. 每个服务的 HTTPS 重定向

您可以通过配置 **networking.knative.dev/http-option** 注解来为服务启用或禁用 HTTPS 重定向。以下示例演示了如何在 Knative **Service** YAML 对象中使用此注解：

```
apiVersion: serving.knative.dev/v1
```

```
kind: Service
metadata:
  name: example
  namespace: default
  annotations:
    networking.knative.dev/http-option: "redirected"
spec:
  ...
```

5.1.9. 其他资源

- Knative Serving CLI 命令
- 为 Knative 服务配置 JSON Web 令牌身份验证

5.2. 自动缩放

Knative Serving 为应用程序提供自动扩展功能（或 *autoscaling*），以满足传入的需求。例如，如果应用程序没有流量，并且启用了缩减到零，Knative Serving 将应用程序缩减为零个副本。如果缩减到零，则应用程序会缩减到为集群中的应用程序配置的最小副本数。如果应用流量增加，也可以向上扩展副本来满足需求。

Knative 服务的自动扩展设置可以是由集群管理员配置的全局设置，或为单个服务配置每个修订设置。您可以使用 OpenShift Container Platform Web 控制台修改服务的每个修订设置，方法是修改服务的 YAML 文件，或使用 Knative (**kn**) CLI 修改服务。

注意

您为服务设置的任何限制或目标均是针对应用程序的单个实例来衡量。例如，将 **target** 注解设置为 **50** 可将自动扩展器配置为缩放应用程序，以便每个修订一次处理 50 个请求。

5.2.1. 扩展范围

缩放范围决定了可在任意给定时间为应用程序服务的最小和最大副本数。您可以为应用设置规模绑定，以帮助防止冷启动和控制计算成本。

5.2.1.1. 最小扩展范围

为应用程序提供服务的最小副本数量由 **min-scale** 注解决定。如果没有启用缩减为零，则 **min-scale** 值默认为 **1**。

如果满足以下条件，**min-scale** 值默认为 **0** 个副本：

- 不设置 **min-scale** 注解
- 启用扩展到零
- 使用类 **KPA**

带有 min-scale 注解的 service spec 示例

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
```

```
  name: example-service
  namespace: default
spec:
  template:
    metadata:
      annotations:
        autoscaling.knative.dev/min-scale: "0"
...
```

5.2.1.1.1. 使用 Knative CLI 设置 min-scale 注解

使用 Knative (**kn**) CLI 设置 **min-scale** 注解，比直接修改 YAML 文件提供了一个更加精简且直观的用户界面。您可以使用带有 **--scale-min** 标志的 **kn service** 命令为服务创建或修改 **min-scale** 值。

先决条件

- 在集群中安装了 Knative Serving。
- 已安装 Knative (**kn**) CLI。

流程

- 使用 **--scale-min** 标志设置服务的最小副本数：

```
$ kn service create <service_name> --image <image_uri> --scale-min <integer>
```

示例命令

```
$ kn service create example-service --image quay.io/openshift-knative/knative-eventing-sources-event-display:latest --scale-min 2
```

5.2.1.2. 最大扩展范围

可提供应用程序的副本数量由 **max-scale** 注解决定。如果没有设置 **max-scale** 注解，则创建的副本数没有上限。

带有 max-scale 注解的 service spec 示例

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: example-service
  namespace: default
spec:
  template:
    metadata:
      annotations:
        autoscaling.knative.dev/max-scale: "10"
...
```

5.2.1.2.1. 使用 Knative CLI 设置 max-scale 注解

使用 Knative (**kn**) CLI 设置 **max-scale** 注解，比直接修改 YAML 文件提供了一个更精简且直观的用户界面。您可以使用带有 **--scale-max** 标志的 **kn service** 命令为服务创建或修改 **max-scale** 值。

先决条件

- 在集群中安装了 Knative Serving。
- 已安装 Knative (**kn**) CLI。

流程

- 使用 **--scale-max** 标志设置服务的最大副本数：

```
$ kn service create <service_name> --image <image_uri> --scale-max <integer>
```

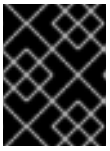
示例命令

```
$ kn service create example-service --image quay.io/openshift-knative/knative-eventing-sources-event-display:latest --scale-max 10
```

5.2.2. 并发

并发请求数决定了应用程序的每个副本可在任意给定时间处理的并发请求数。并发可以配置为*软限制*或*硬限制*：

- 软限制是目标请求限制，而不是严格实施的绑定。例如，如果流量突发，可以超过软限制目标。
- 硬限制是严格实施的上限请求限制。如果并发达到硬限制，则请求将被缓冲，必须等到有足够的可用容量来执行请求。



重要

只有在应用程序中明确用例时才建议使用硬限制配置。指定较少的硬限制可能会对应用程序的吞吐量和延迟造成负面影响，并可能导致冷启动。

添加软目标和硬限制意味着自动扩展以并发请求的软目标数为目标，但为请求的最大数量施加硬限制值。

如果硬限制值小于软限制值，则软限制值将降级，因为不需要将目标设定为多于实际处理的请求数。

5.2.2.1. 配置软并发目标

软限制是目标请求限制，而不是严格实施的绑定。例如，如果流量突发，可以超过软限制目标。您可以通过在 spec 中设置 **autoscaling.knative.dev/target** 注解，或者使用带有正确标记的 **kn service** 命令为 Knative 服务指定软并发目标。

流程

- 可选：在 **Service** 自定义资源的 spec 中为您的 Knative 服务设置 **autoscaling.knative.dev/target** 注解：

服务规格示例

```
apiVersion: serving.knative.dev/v1
```

```
kind: Service
metadata:
  name: example-service
  namespace: default
spec:
  template:
    metadata:
      annotations:
        autoscaling.knative.dev/target: "200"
```

- 可选： 使用 **kn service** 命令指定 **--concurrency-target** 标志：

```
$ kn service create <service_name> --image <image_uri> --concurrency-target <integer>
```

创建服务的示例，并发目标为 50 请求

```
$ kn service create example-service --image quay.io/openshift-knative/knative-eventing-sources-event-display:latest --concurrency-target 50
```

5.2.2.2. 配置硬并发限制

硬并发限制是严格强制执行上限的上限。如果并发达到硬限制，则请求将被缓冲，必须等到有足够的可用容量来执行请求。您可以通过修改 **containerConcurrency** spec 或使用带有正确标记的 **kn service** 命令为 Knative 服务指定硬并发限制。

流程

- 可选：在 **Service** 自定义资源的 spec 中为您的 Knative 服务设置 **containerConcurrency** spec：

  服务规格示例

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: example-service
  namespace: default
spec:
  template:
    spec:
      containerConcurrency: 50
```

  默认值为 **0**，这意味着允许同时访问服务的一个副本的请求数量没有限制。

  大于 **0** 的值指定允许一次传输到服务的一个副本的请求的确切数量。这个示例将启用 50 个请求的硬并发限制。

- 可选： 使用 **kn service** 命令指定 **--concurrency-limit** 标志：

```
$ kn service create <service_name> --image <image_uri> --concurrency-limit <integer>
```

创建服务且并发限制为 50 个请求的命令示例

```
$ kn service create example-service --image quay.io/openshift-knative/knative-eventing-sources-event-display:latest --concurrency-limit 50
```

5.2.2.3. 并发目标使用率

此值指定自动扩展实际的目标并发限制的百分比。这也称为指定运行副本的*热性（hotness）*，允许自动扩展在达到定义的硬限制前进行扩展。

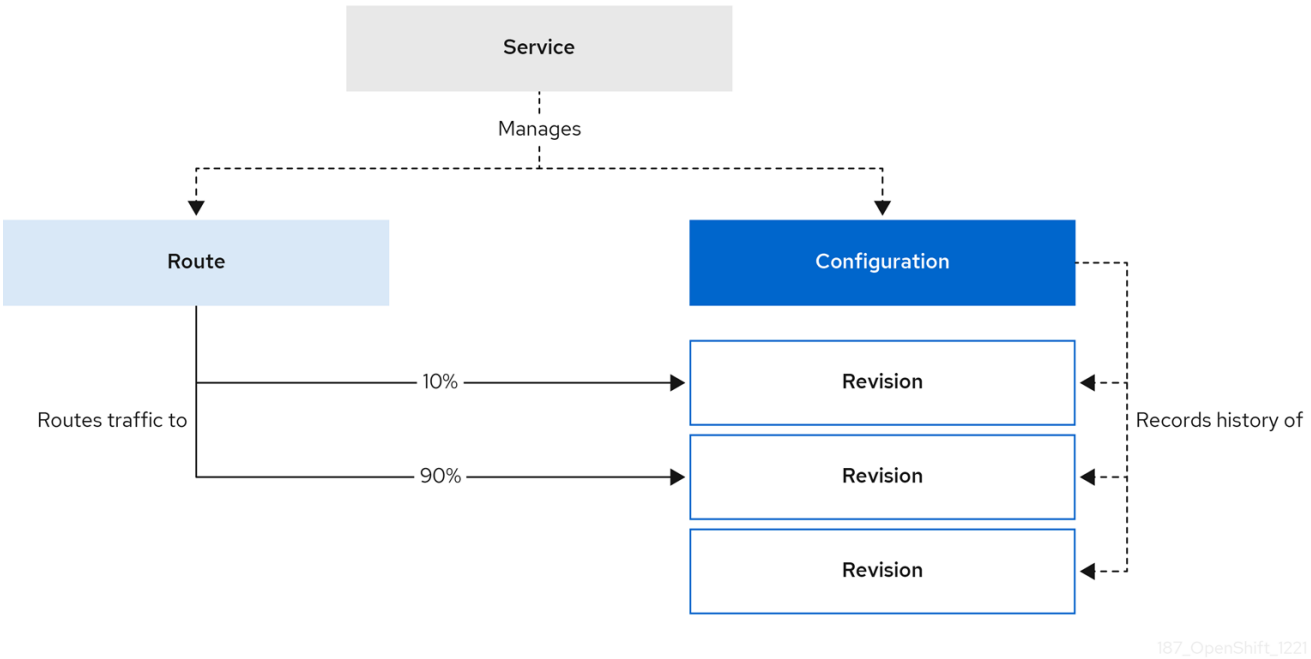
例如，如果 **containerConcurrency** 值设置为 10，并且 **target-utilization-percentage** 值设置为 70%，则自动扩展会在所有现有副本的平均并发请求数量达到 7 时创建一个新的副本。编号为 7 到 10 的请求仍然会被发送到现有的副本，但达到 **containerConcurrency** 值后会启动额外的副本。

使用 target-utilization-percentage 注解配置的服务示例

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: example-service
  namespace: default
spec:
  template:
    metadata:
      annotations:
        autoscaling.knative.dev/target-utilization-percentage: "70"
...
```

5.3. 流量管理

在 Knative 应用程序中，可以通过创建流量分割来管理流量。流量分割被配置为由 Knative 服务管理的路由的一部分。



配置路由允许将请求发送到服务的不同修订版本。此路由由 **Service** 对象的 **traffic** spec 决定。

traffic 规格声明由一个或多个修订版本组成，每个修订版本负责处理整个流量的一部分。路由到每个修订版本的流量百分比必须添加到 100%，由 Knative 验证确保。

traffic 规格中指定的修订版本可以是固定的、名为修订的修订版本，或者可以指向"latest"修订，该修订跟踪服务所有修订版本列表的头。"latest" 修订版本是一个浮动引用类型，它在创建了新修订版本时更新。每个修订版本都可以附加标签，为该修订版本创建一个额外访问 URL。

traffic 规格可通过以下方法修改：

- 直接编辑 **Service** 对象的 YAML。
- 使用 Knative (**kn**) CLI **--traffic** 标志。
- 使用 OpenShift Container Platform Web 控制台。

当您创建 Knative 服务时，它没有任何默认 **traffic** spec 设置。

5.3.1. traffic 规格示例

以下示例显示了一个 **traffic** 规格，其中 100% 的流量路由到该服务的最新修订版本。在 **status** 下，您可以看到 **latestRevision** 解析为的最新修订版本的名称：

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: example-service
  namespace: default
spec:
...
  traffic:
  - latestRevision: true
    percent: 100
status:
  ...
  traffic:
  - percent: 100
    revisionName: example-service
```

以下示例显示了一个 **traffic** 规格，其中 100% 的流量路由到当前标记为 **current** 修订版本，并且该修订版本的名称指定为 **example-service**。标记为 **latest** 的修订版本会保持可用，即使没有流量路由到它：

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: example-service
  namespace: default
spec:
...
  traffic:
  - tag: current
    revisionName: example-service
    percent: 100
  - tag: latest
    latestRevision: true
    percent: 0
```

以下示例演示了如何扩展 **traffic** 规格中的修订版本列表，以便在多个修订版本间分割流量。这个示例将 50% 的流量发送到标记为 **current** 修订版本，50% 的流量发送到标记为 **candidate** 的修订版本。标记为 **latest** 的修订版本会保持可用，即使没有流量路由到它：

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: example-service
  namespace: default
spec:
...
  traffic:
  - tag: current
    revisionName: example-service-1
    percent: 50
  - tag: candidate
    revisionName: example-service-2
    percent: 50
  - tag: latest
    latestRevision: true
    percent: 0
```

5.3.2. Knative CLI 流量管理标志

Knative (**kn**) CLI 支持作为 **kn service update** 命令的一部分对服务的流量块进行流量操作。

下表显示流量分割标志、值格式和标志执行的操作汇总。**Repetition** 列表示在 **kn service update** 命令中是否允许重复标志的特定值。

标记	值	操作	重复
--traffic	**RevisionName=Percent**	为 **RevisionName** 提供 **Percent** 的流量	是
--traffic	**Tag=Percent**	为带有 **Tag** 的修订版本提供 **Percent** 的流量	是
--traffic	**@latest=Percent**	为最新可用的修订版本提供 **Percent** 的流量	否
--tag	**RevisionName=Tag**	为 **RevisionName** 提供 **Tag**	是
--tag	**@latest=Tag**	为最新可用的修订版本提供 **Tag**	否
--untag	**Tag**	从修订中删除 **Tag**	是

5.3.2.1. 多个标志和顺序优先级

所有流量相关标志均可使用单一 **kn service update** 命令指定。**kn** 定义这些标志的优先级。不考虑使用命令时指定的标志顺序。

通过 **kn** 评估标志时，标志的优先级如下：

1. **--untag**：带有此标志的所有引用修订版本均将从流量块中移除。
2. **--tag**：修订版本将按照流量块中的指定进行标记。
3. **--traffic**：为引用的修订版本分配一部分流量分割。

您可以将标签添加到修订版本，然后根据您设置的标签来分割流量。

5.3.2.2. 修订版本的自定义 URL

使用 **kn service update** 命令为服务分配 **--tag** 标志，可为在更新服务时创建的修订版本创建一个自定义 URL。自定义 URL 使用 **https://<tag>-<service_name>-<namespace>.<domain>;** or **http://<tag>-<service_name>-<namespace>.<domain>;** 格式。

--tag 和 **--untag** 标志使用以下语法：

- 需要一个值。
- 在服务的流量块中表示唯一标签。
- 在一个命令中可多次指定.

5.3.2.2.1. 示例：将标签分配给修订版本

以下示例将标签 **latest** 分配给名为 **example-revision** 的修订版本：

```
$ kn service update <service_name> --tag @latest=example-tag
```

5.3.2.2.2. 示例：从修订中删除标签

您可以使用 **--untag** 标志来删除自定义 URL。

注意

如果修订版本删除了其标签，并分配了流量的 0%，则修订版本将完全从流量块中删除。

以下命令从名为 **example-revision** 的修订版本中删除所有标签：

```
$ kn service update <service_name> --untag example-tag
```

5.3.3. 使用 Knative CLI 创建流量分割

使用 Knative (**kn**) CLI 创建流量分割功能，通过直接修改 YAML 文件，提供更精简且直观的用户界面。您可以使用 **kn service update** 命令在服务修订版本间分割流量。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。

- 已安装 Knative (**kn**) CLI。
- 您已创建了 Knative 服务。

流程

- 使用带有标准 **kn service update** 命令的 **--traffic** 标签指定服务修订版本以及您要路由到它的流量百分比：

  示例命令

  ```
  $ kn service update <service_name> --traffic <revision>=<percentage>
  ```

  其中：

  - **<service_name>** 是您要为其配置流量路由的 Knative 服务的名称。
  - **<revision>** 是您要配置为接收流量百分比的修订版本。您可以使用 **--tag** 标志指定修订版本的名称，或指定分配给修订版本的标签。
  - **<percentage>** 是您要发送到指定修订版本的流量百分比。

- 可选： **--traffic** 标志可在一个命令中多次指定。例如，如果您有一个标记为 **@latest** 的修订版本以及名为 **stable** 的修订版本，您可以指定您要分割到每个修订版本的流量百分比：

  示例命令

  ```
  $ kn service update example-service --traffic @latest=20,stable=80
  ```

  如果您有多个修订版本，且没有指定应分割到最后一个修订版本的流量百分比，**--traffic** 标志可以自动计算此设置。例如，如果您有一个第三个版本名为 **example**，则使用以下命令：

  示例命令

  ```
  $ kn service update example-service --traffic @latest=10,stable=60
  ```

  剩余的 30% 的流量被分成 **example** 修订，即使未指定。

5.3.4. 使用 OpenShift Container Platform Web 控制台管理修订版本之间的流量

创建无服务器应用程序后，应用程序会在 OpenShift Container Platform Web 控制台中的 **Developer 视角** 的 **Topology 视图**中显示。应用程序修订版本由节点表示，Knative 服务由节点的四边形表示。

代码或服务配置中的任何新更改都会创建一个新修订版本，也就是给定时间点上代码的快照。对于服务，您可以根据需要通过分割服务修订版本并将其路由到不同的修订版本来管理服务间的流量。

先决条件

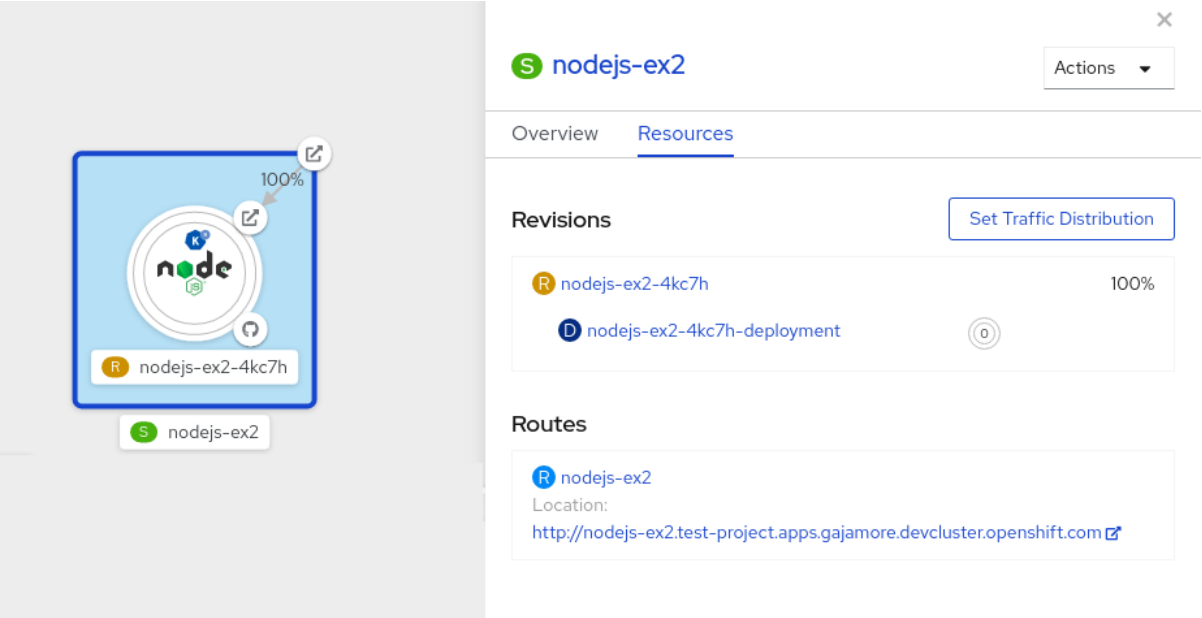
- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 已登陆到 OpenShift Container Platform Web 控制台。

流程

要在 **Topology** 视图中的多个应用程序修订版本间分割流量：

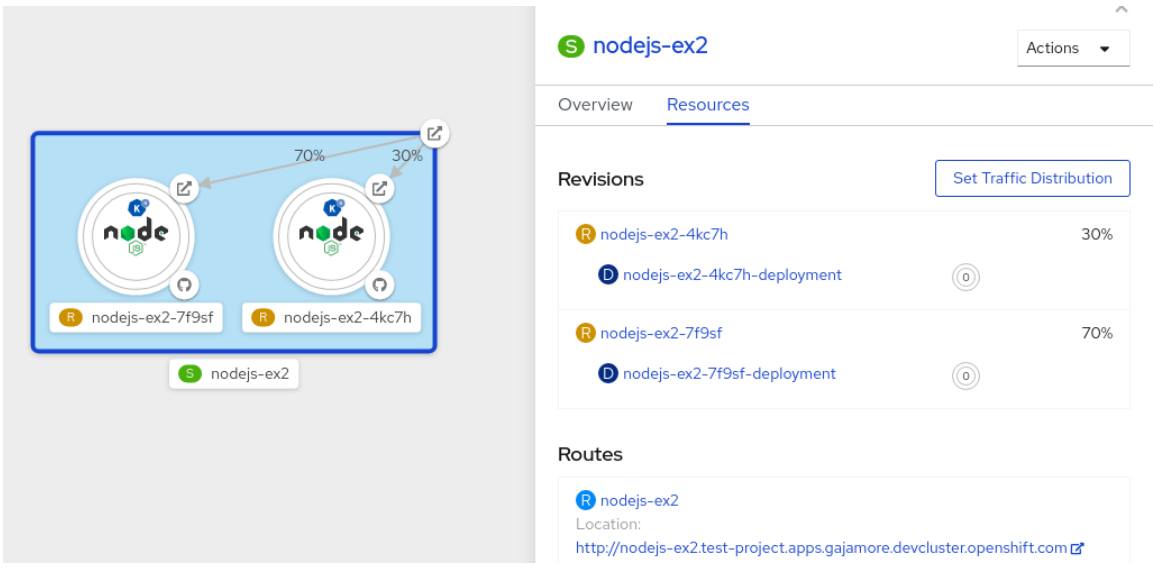
1. 点 Knative 服务在侧面面板中查看其概述信息。
2. 点 **Resources** 选项卡，查看服务的 **Revisions** 和 **Routes** 列表。

图 5.1. 无服务器应用程序



3. 点侧边面板顶部的由 **S** 图标代表的服务，查看服务详情概述。
4. 点 **YAML** 选项卡，在 YAML 编辑器中修改服务配置，然后点 **Save**。例如，将 **timeoutseconds** 从 300 改为 301。这个配置更改会触发新修订版本。在 **Topology** 视图中会显示最新的修订，服务 **Resources** 选项卡现在会显示两个修订版本。
5. 在 **Resources** 选项卡中，点 **Set Traffic Distribution** 查看流量分布对话框：
   a. 在 **Splits** 字段中为两个修订版本添加流量百分比。
   b. 添加标签以便为这两个修订版本创建自定义 URL。
   c. 点 **Save** 查看两个节点，分别代表 Topology 视图中的两个修订版本。

图 5.2. 无服务器应用程序修订



5.3.5. 使用蓝绿部署策略路由和管理流量

您可以使用蓝绿部署策略，安全地将流量从应用的生产版本重新路由到新版本。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 安装 OpenShift CLI (**oc**) 。

流程

1. 创建并部署应用程序作为 Knative 服务。
2. 通过查看以下命令的输出，查找部署服务时创建的第一个修订版本的名称：

```
$ oc get ksvc <service_name> -o=jsonpath='{.status.latestCreatedRevisionName}'
```

示例命令

```
$ oc get ksvc example-service -o=jsonpath='{.status.latestCreatedRevisionName}'
```

输**出示例**

```
$ example-service-00001
```

3. 在服务 **spec** 中添加以下 YAML 以将入站流量发送到修订版本：

```
...
spec:
  traffic:
    - revisionName: <first_revision_name>
      percent: 100 # All traffic goes to this revision
...
```

4. 验证您可以在 URL 输出中运行以下命令来查看您的应用程序：

```
$ oc get ksvc <service_name>
```

5. 通过修改服务的 **template** 规格中至少有一个字段来部署应用程序的第二个修订版本。例如，您可以修改服务的 **image** 或 **env** 环境变量。您可以通过应用服务 YAML 文件重新部署服务，如果安装了 Knative (**kn**) CLI，也可以使用 **kn service update** 命令。
6. 运行以下命令，查找您在重新部署服务时创建的第二个最新的修订版本的名称：

```
$ oc get ksvc <service_name> -o=jsonpath='{.status.latestCreatedRevisionName}'
```

此时，服务的第一个和第二个修订版本都已部署并运行。

7. 更新您的现有服务，以便为第二个修订版本创建新的测试端点，同时仍然将所有其他流量发送到第一个修订版本：

使用测试端点更新的服务 spec 示例

```
...
spec:
  traffic:
    - revisionName: <first_revision_name>
      percent: 100 # All traffic is still being routed to the first revision
    - revisionName: <second_revision_name>
      percent: 0 # No traffic is routed to the second revision
      tag: v2 # A named route
...
```

在通过重新应用 YAML 资源重新部署此服务后，应用的第二个修订现已被暂存。没有流量路由到主 URL 的第二个修订版本，Knative 会创建一个名为 **v2** 的新服务来测试新部署的修订版本。

8. 运行以下命令，获取第二个修订版本的新服务的 URL：

```
$ oc get ksvc <service_name> --output jsonpath="{.status.traffic[*].url}"
```

在将任何流量路由到之前，您可以使用此 URL 验证新版本的应用运行正常。

9. 再次更新您的现有服务，以便 50% 的流量发送到第一个修订版本，50% 发送到第二个修订版本：

更新的服务 spec 在修订版本间分割流量 50/50 的示例

```
...
spec:
  traffic:
    - revisionName: <first_revision_name>
      percent: 50
    - revisionName: <second_revision_name>
      percent: 50
      tag: v2
...
```

10. 当您准备好将所有流量路由到应用程序的新版本时，请再次更新该服务，将 100% 的流量发送到第二个修订版本：

更新的服务 spec 将所有流量发送到第二个修订版本的示例

```
...
spec:
  traffic:
    - revisionName: <first_revision_name>
      percent: 0
    - revisionName: <second_revision_name>
      percent: 100
      tag: v2
...
```

提示

如果您不计划回滚修订版本，您可以删除第一个修订版本，而不是将其设置为流量的 0%。然后，不可路由的修订版本对象会被垃圾回收。

11. 访问第一个修订版本的 URL，以验证没有更多流量发送到应用程序的旧版本。

5.4. 路由

Knative 利用 OpenShift Container Platform TLS 终止来为 Knative 服务提供路由。创建 Knative 服务时，会自动为该服务创建一个 OpenShift Container Platform 路由。此路由由 OpenShift Serverless Operator 管理。OpenShift Container Platform 路由通过与 OpenShift Container Platform 集群相同的域公开 Knative 服务。

您可以禁用 OpenShift Container Platform 路由的 Operator 控制，以便您可以配置 Knative 路由来直接使用 TLS 证书。

Knative 路由也可以与 OpenShift Container Platform 路由一起使用，以提供额外的精细路由功能，如流量分割。

5.4.1. 为 OpenShift Container Platform 路由自定义标签和注解

OpenShift Container Platform 路由支持使用自定义标签和注解，您可以通过修改 Knative 服务的**元数据规格来配置这些标签和注解**。自定义标签和注解从服务传播到 Knative 路由，然后传播到 Knative ingress，最后传播到 OpenShift Container Platform 路由。

先决条件

- 您必须已在 OpenShift Container Platform 集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 安装 OpenShift CLI (**oc**) 。

流程

1. 创建包含您要传播到 OpenShift Container Platform 路由的标签或注解的 Knative 服务：
   - 使用 YAML 创建服务：

     使用 YAML 创建的服务示例

     ```
     apiVersion: serving.knative.dev/v1
     kind: Service
     metadata:
       name: <service_name>
       labels:
         <label_name>: <label_value>
       annotations:
         <annotation_name>: <annotation_value>
     ...
     ```

   - 要使用 Knative (**kn**) CLI 创建服务，请输入：

     使用 kn 命令创建的服务示例

     ```
     $ kn service create <service_name> \
       --image=<image> \
       --annotation <annotation_name>=<annotation_value> \
       --label <label_value>=<label_value>
     ```

2. 通过检查以下命令的输出来验证 OpenShift Container Platform 路由是否已使用您添加的注解或标签创建：

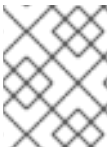
   验证**命令示例**

```
$ oc get routes.route.openshift.io \
    -l serving.knative.openshift.io/ingressName=<service_name> \ 1
    -l serving.knative.openshift.io/ingressNamespace=<service_namespace> \ 2
    -n knative-serving-ingress -o yaml \
       | grep -e "<label_name>: \"<label_value>\""  -e "<annotation_name>:
<annotation_value>" 3
```

1. 使用服务的名称。
2. 使用创建服务的命名空间。
3. 将您的值用于标签和注解名称和值。

5.4.2. 为 Knative 服务配置 OpenShift Container Platform 路由

如果要将 Knative 服务配置为在 OpenShift Container Platform 上使用 TLS 证书，则必须禁用 OpenShift Serverless Operator 为服务自动创建路由，而是手动为服务创建路由。



注意

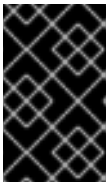
完成以下步骤时，不会创建 **knative-serving-ingress** 命名空间中的默认 OpenShift Container Platform 路由。但是，应用程序的 Knative 路由仍然在此命名空间中创建。

先决条件

- OpenShift Serverless Operator 和 Knative Serving 组件必须安装在 OpenShift Container Platform 集群中。
- 安装 OpenShift CLI (**oc**) 。

流程

1. 创建包含 **service.knative.openshift.io/disableRoute=true** 注解的 Knative 服务：



重要

service.knative.openshift.io/disableRoute=true 注解指示 OpenShift Serverless 不自动为您创建路由。但是，该服务仍然会显示 URL 并达到 **Ready** 状态。除非使用与 URL 中主机名相同的主机名创建自己的路由，此 URL 才能在外部工作。

   a. 创建 Knative **Service** 资源：

      资**源示例**

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
```

```
    name: <service_name>
    annotations:
      serving.knative.openshift.io/disableRoute: "true"
  spec:
    template:
      spec:
        containers:
        - image: <image>
  ...
```

b. 应用 **Service** 资源：

```
$ oc apply -f <filename>
```

c. 可选。使用 **kn service create** 命令创建 Knative 服务：

kn 命令示例

```
$ kn service create <service_name> \
  --image=gcr.io/knative-samples/helloworld-go \
  --annotation serving.knative.openshift.io/disableRoute=true
```

2. 验证没有为服务创建 OpenShift Container Platform 路由：

示例命令

```
$ $ oc get routes.route.openshift.io \
  -l serving.knative.openshift.io/ingressName=$KSERVICE_NAME \
  -l serving.knative.openshift.io/ingressNamespace=$KSERVICE_NAMESPACE \
  -n knative-serving-ingress
```

您将看到以下输出：

```
No resources found in knative-serving-ingress namespace.
```

3. 在 **knative-serving-ingress** 命名空间中创建 **Route** 资源：

```
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  annotations:
    haproxy.router.openshift.io/timeout: 600s (1)
  name: <route_name> (2)
  namespace: knative-serving-ingress (3)
spec:
  host: <service_host> (4)
  port:
    targetPort: http2
  to:
    kind: Service
    name: kourier
    weight: 100
  tls:
```

```
  insecureEdgeTerminationPolicy: Allow
  termination: edge 5
  key: |-
    -----BEGIN PRIVATE KEY-----
    [...]
    -----END PRIVATE KEY-----
  certificate: |-
    -----BEGIN CERTIFICATE-----
    [...]
    -----END CERTIFICATE-----
  caCertificate: |-
    -----BEGIN CERTIFICATE-----
    [...]
    -----END CERTIFICATE----
wildcardPolicy: None
```

1. OpenShift Container Platform 路由的超时值。您必须设置与 **max-revision-timeout-seconds** 设置相同的值（默认为 **600s**）。
2. OpenShift Container Platform 路由的名称。
3. OpenShift Container Platform 路由的命名空间。这必须是 **knative-serving-ingress**。
4. 用于外部访问的主机名。您可以将其设置为 **<service_name>-<service_namespace>.<domain>**。
5. 您要使用的证书。目前，只支持 边缘（**edge**） 终止。

4. 应用 **Route** 资源：

```
$ oc apply -f <filename>
```

5.4.3. 将集群可用性设置为集群本地

默认情况下，Knative 服务会发布到一个公共 IP 地址。被发布到一个公共 IP 地址意味着 Knative 服务是公共应用程序，并有一个公开访问的 URL。

可以从集群以外访问公开的 URL。但是，开发人员可能需要构建后端服务，这些服务只能从集群内部访问（称为 *私有服务*）。开发人员可以使用 **networking.knative.dev/visibility=cluster-local** 标签标记集群中的各个服务，使其私有。

重要

对于 OpenShift Serverless 1.15.0 及更新的版本，**service.knative.dev/visibility** 标签不再可用。您必须更新现有服务来改用 **networking.knative.dev/visibility** 标签。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 您已创建了 Knative 服务。

流程

- 通过添加 **networking.knative.dev/visibility=cluster-local** 标签来设置服务的可见性：

```
$ oc label ksvc <service_name> networking.knative.dev/visibility=cluster-local
```

验证

- 输入以下命令并查看输出结果，检查您的服务的 URL 是否现在格式为 **http://<service_name>.<namespace>.svc.cluster.local**：

```
$ oc get ksvc
```

输出示例

```
NAME    URL                                        LATESTCREATED   LATESTREADY   READY   REASON
hello   http://hello.default.svc.cluster.local     hello-tx2g7     hello-tx2g7   True
```

5.4.4. 其他资源

- 特定于路由的注解

5.5. 事件 SINK

创建事件源时，您可以指定事件从源发送到的接收器。sink 是一个可寻址或可调用的资源，可以从其他资源接收传入的事件。Knative 服务、频道和代理都是接收器示例。

可寻址的对象接收和确认通过 HTTP 发送的事件到其 **status.address.url** 字段中定义的地址。作为特殊情况，核心 Kubernetes **Service** 对象也履行可寻址的接口。

可调用的对象可以接收通过 HTTP 发送的事件并转换事件，并在 HTTP 响应中返回 **0** 或 **1** 新事件。这些返回的事件可能会象处理外部事件源中的事件一样进一步处理。

5.5.1. Knative CLI sink 标记

当使用 Knative (**kn**) CLI 创建事件源时，您可以使用 **--sink** 标志指定事件从该资源发送到的接收器。sink 可以是任何可寻址或可调用的资源，可以从其他资源接收传入的事件。

以下示例创建使用服务 **http://event-display.svc.cluster.local** 的接收器绑定作为接收器：

使用 sink 标记的命令示例

```
$ kn source binding create bind-heartbeat \
  --namespace sinkbinding-example \
  --subject "Job:batch/v1:app=heartbeat-cron" \
  --sink http://event-display.svc.cluster.local \ 1
  --ce-override "sink=bound"
```

1 **http://event-display.svc.cluster.local** 中的 **svc** 确定接收器是一个 Knative 服务。其他默认的接收器前缀包括 **channel** 和 **broker**。

提示

您可以通过自定义 **kn**，配置哪些 CR 可在 Knative (**kn**) CLI 命令中使用 **--sink** 标记。

5.5.2. 使用 Developer 视角将事件源连接到接收器（sink）

当使用 OpenShift Container Platform web 控制台创建事件源时，您可以指定事件从该资源发送到的接收器。sink 可以是任何可寻址或可调用的资源，可以从其他资源接收传入的事件。

先决条件

- OpenShift Serverless Operator、Knative Serving 和 Knative Eventing 已在 OpenShift Container Platform 集群中安装。
- 已登陆到 web 控制台，且处于 **Developer** 视角。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。
- 您已创建了 sink，如 Knative 服务、频道或代理。

流程

1. 进入 **+Add → Event Sources**，然后选择您要创建的事件源类型，创建任何类型的事件源。
2. 在 **Create Event Source** 表单视图的 **Sink** 部分，在 **Resource** 列表中选择您的接收器。
3. 点 **Create**。

验证

您可以通过查看 **Topology** 页面来验证事件源是否已创建并连接到 sink。在 **Developer** 视角中，导航到 **Topology**。

1. 查看事件源并点击连接的 sink 来查看侧面面板中的 sink 详情。

5.5.3. 将触发器连接到 sink

您可以将触发器连接到 sink，以便在将代理的事件发送到 sink 前过滤代理的事件。在 **Trigger** 对象的资源规格中，连接到触发器的 sink 会配置为 **订阅者**。

连接到 Kafka sink 的 Trigger 对象示例

```
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  name: <trigger_name> 1
spec:
...
  subscriber:
    ref:
      apiVersion: eventing.knative.dev/v1alpha1
      kind: KafkaSink
      name: <kafka_sink_name> 2
```

1. 连接到 sink 的触发器的名称。
2. **KafkaSink** 对象的名称。

5.6. 事件交付

您可以配置事件交付参数，当事件无法发送到事件 sink 时。配置事件交付参数，包括死信接收器，可确保重试任何无法发送到事件接收器的事件。否则，未验证的事件将被丢弃。

5.6.1. 频道和代理的事件交付行为模式

不同的频道和代理类型都有自己的行为模式，用于事件交付。

5.6.1.1. Knative Kafka 频道和代理

如果事件成功传送到 Kafka 频道或代理接收器，接收器会使用一个 **202** 状态代码进行响应，这意味着该事件已被安全地存储在 Kafka 主题中且不会丢失。

如果接收器使用任何其他状态代码响应，则事件不会被安全存储，用户必须执行步骤来解决这个问题。

5.6.2. 可配置事件交付参数

为事件交付配置以下参数：

死信接收器

您可以配置 **deadLetterSink** 交付参数，以便在事件无法发送时，它存储在指定的事件 sink 中。取消请求没有存储在死信接收器中的事件会被丢弃。死信接收器是符合 Knative Eventing sink 合同的任何可寻址对象，如 Knative 服务、Kubernetes 服务或一个 URI。

Retries

您可以通过使用整数值配置重试 delivery 参数，在事件发送到 dead letter sink 前重试交付的次数。

Back off 延迟

您可以设置 **backoffDelay** 交付参数，以在失败后尝试事件交付重试前指定延迟。**backoffDelay** 参数的持续时间使用 ISO 8601 格式指定。例如，**PT1S** 指定 1 秒延迟。

Back off 策略

backoffPolicy 交付参数可以用来指定重试避退策略。该策略可以指定为 **linear** 或 **exponential**。当使用 **linear** back off 策略时，back off 延迟等同于 **backoffDelay * <numberOfRetries>**。当使用 **exponential** backoff 策略时，back off 的延迟等于 **backoffDelay*2^<numberOfRetries>**。

5.6.3. 配置事件交付参数示例

您可以为 **Broker**、**Trigger**、**Channel** 和 **Subscription** 对象配置事件交付参数。如果您为代理或频道配置事件交付参数，这些参数会传播到为这些对象创建的触发器或订阅。您还可以为触发器或订阅设置事件交付参数，以覆盖代理或频道的设置。

Broker 对象示例

```
apiVersion: eventing.knative.dev/v1
kind: Broker
metadata:
...
spec:
```

```
  delivery:
    deadLetterSink:
      ref:
        apiVersion: eventing.knative.dev/v1alpha1
        kind: KafkaSink
        name: <sink_name>
    backoffDelay: <duration>
    backoffPolicy: <policy_type>
    retry: <integer>
...
```

Trigger 对象示例

```
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
...
spec:
  broker: <broker_name>
  delivery:
    deadLetterSink:
      ref:
        apiVersion: serving.knative.dev/v1
        kind: Service
        name: <sink_name>
    backoffDelay: <duration>
    backoffPolicy: <policy_type>
    retry: <integer>
...
```

Channel 对象示例

```
apiVersion: messaging.knative.dev/v1
kind: Channel
metadata:
...
spec:
  delivery:
    deadLetterSink:
      ref:
        apiVersion: serving.knative.dev/v1
        kind: Service
        name: <sink_name>
    backoffDelay: <duration>
    backoffPolicy: <policy_type>
    retry: <integer>
...
```

Subscription 对象示例

```
apiVersion: messaging.knative.dev/v1
kind: Subscription
metadata:
...
```

```
spec:
  channel:
    apiVersion: messaging.knative.dev/v1
    kind: Channel
    name: <channel_name>
  delivery:
    deadLetterSink:
      ref:
        apiVersion: serving.knative.dev/v1
        kind: Service
        name: <sink_name>
    backoffDelay: <duration>
    backoffPolicy: <policy_type>
    retry: <integer>
...
```

5.6.4. 为触发器配置事件交付顺序

如果使用 Kafka 代理，您可以将事件的交付顺序从触发器配置为事件 sink。

先决条件

- OpenShift Serverless Operator、Knative Eventing 和 Knative Kafka 安装在 OpenShift Container Platform 集群中。
- Kafka 代理被启用在集群中使用，您也创建了一个 Kafka 代理。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。
- 已安装 OpenShift (**oc**) CLI。

流程

1. 创建或修改 **Trigger** 对象并设置 **kafka.eventing.knative.dev/delivery.order** 注解：

```
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  name: <trigger_name>
  annotations:
    kafka.eventing.knative.dev/delivery.order: ordered
...
```

支持的消费者交付保证有：

unordered

未排序的消费者是一种非阻塞消费者，它能以未排序的方式提供消息，同时保持正确的偏移管理。

排序的

一个订购的消费者是一个按分区阻止消费者，在提供分区的下一个消息前等待来自 CloudEvent 订阅者成功响应。
默认排序保证是 **unordered**。

2. 应用 **Trigger** 对象：

```
$ oc apply -f <filename>
```

5.7. 列出事件源和事件源类型

可以查看存在的事件源或事件源类型的列表，也可以在 OpenShift Container Platform 集群中使用。您可以使用 OpenShift Container Platform Web 控制台中的 Knative (**kn**) CLI 或 **Developer** 视角列出可用事件源或事件源类型。

5.7.1. 使用 Knative CLI 列出可用事件源类型

使用 Knative (**kn**) CLI 提供了简化和直观的用户界面，用来在集群中查看可用事件源类型。您可以使用 **kn source list-types** CLI 命令列出集群中创建和使用的事件源类型。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Eventing。
- 已安装 Knative (**kn**) CLI。

流程

1. 列出终端中的可用事件源类型：

```
$ kn source list-types
```

输**出示例**

```
TYPE              NAME                                DESCRIPTION
ApiServerSource   apiserversources.sources.knative.dev   Watch and send Kubernetes API events to a sink
PingSource        pingsources.sources.knative.dev       Periodically send ping events to a sink
SinkBinding       sinkbindings.sources.knative.dev      Binding for connecting a PodSpecable to a sink
```

2. 可选：您也可以以 YAML 格式列出可用事件源类型：

```
$ kn source list-types -o yaml
```

5.7.2. 在 Developer 视角中查看可用事件源类型

您可以查看集群中所有可用事件源类型的列表。使用 OpenShift Container Platform Web 控制台提供了一个简化的用户界面，可用事件源类型。

先决条件

- 已登陆到 OpenShift Container Platform Web 控制台。
- OpenShift Serverless Operator 和 Knative Eventing 已安装在 OpenShift Container Platform 集群中。

- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。

流程

1. 访问 **Developer** 视角。
2. 点 **+Add**。
3. 点 **Event source**。
4. 查看可用的事件源类型。

5.7.3. 使用 Knative CLI 列出可用事件源

使用 Knative (**kn**) CLI 提供了简化和直观的用户界面，用来查看集群中的现有事件源。您可以使用 **kn source list** 命令列出现有的事件源。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Eventing。
- 已安装 Knative (**kn**) CLI。

流程

1. 列出终端中的现有事件源：

```
$ kn source list
```

输**出示例**

```
NAME   TYPE              RESOURCE                              SINK         READY
a1     ApiServerSource   apiserversources.sources.knative.dev   ksvc:eshow2   True
b1     SinkBinding       sinkbindings.sources.knative.dev       ksvc:eshow3   False
p1     PingSource        pingsources.sources.knative.dev        ksvc:eshow1   True
```

2. 可选： 您可以使用 **--type** 标志来只列出特定类型的事件源：

```
$ kn source list --type <event_source_type>
```

示例命令

```
$ kn source list --type PingSource
```

输**出示例**

```
NAME   TYPE              RESOURCE                              SINK         READY
p1     PingSource        pingsources.sources.knative.dev        ksvc:eshow1   True
```

5.8. 创建 API 服务器源

API 服务器源是一个事件源，可用于将事件接收器（sink），如 Knative 服务，连接到 Kubernetes API 服务器。API 服务器源监视 Kubernetes 事件并将其转发到 Knative Eventing 代理。

5.8.1. 使用 Web 控制台创建 API 服务器源

在集群中安装 Knative Eventing 后，您可以使用 web 控制台创建 API 服务器源。使用 OpenShift Container Platform Web 控制台提供了一个简化且直观的用户界面来创建事件源。

先决条件

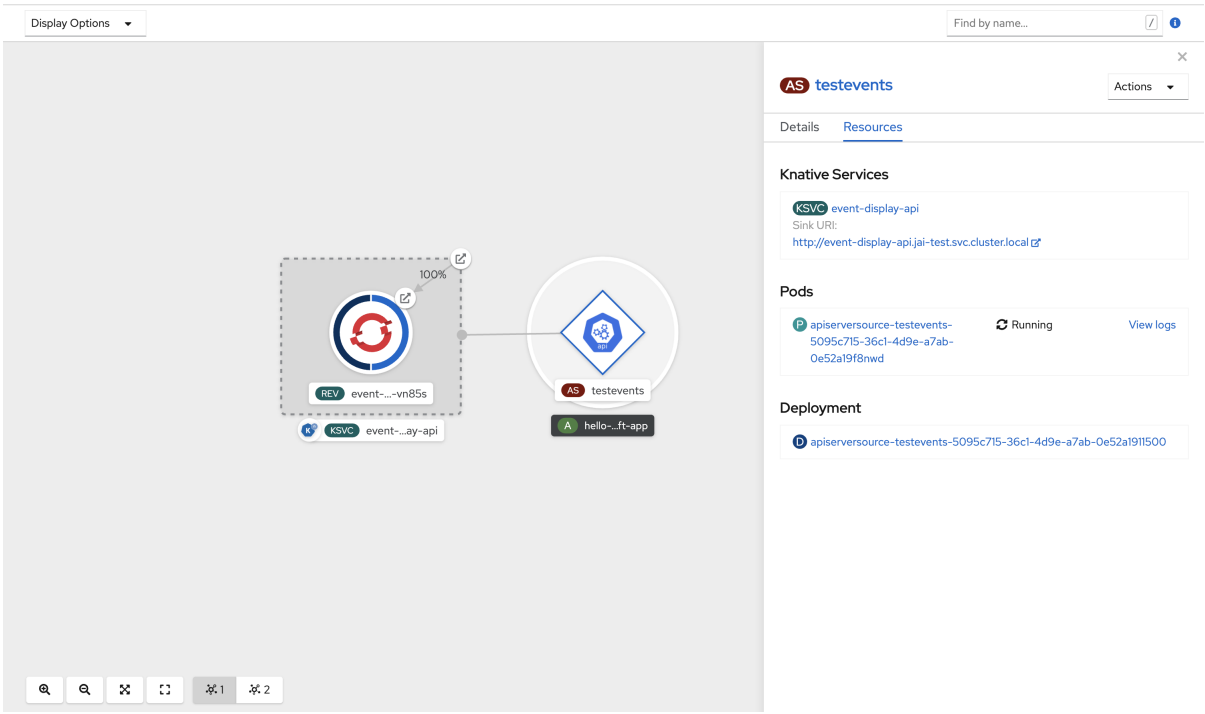
- 已登陆到 OpenShift Container Platform Web 控制台。
- 在集群中安装了 OpenShift Serverless Operator 和 Knative Eventing。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。
- 已安装 OpenShift CLI(**oc**)。

流程

1. 导航到 **Add** 页面并选择 **Event Source**。
2. 在 **Event Sources** 页面中，在 **Type** 部分选择 **ApiServerSource**。
3. 配置 **ApiServerSource** 设置：
   a. 输入 **v1** 作为 **APIVERSION**，**Event** 作为 **KIND**。
   b. 为您创建的服务帐户选择 **Service Account Name**。
   c. 为事件源选择 **Sink**。**Sink** 可以是一个 **资源**，如频道、代理或服务，也可以是一个 **URI**。
4. 点 **Create**。

验证

- 创建 API 服务器源后，您会在 **Topology** 视图中看到它连接到接收器的服务。

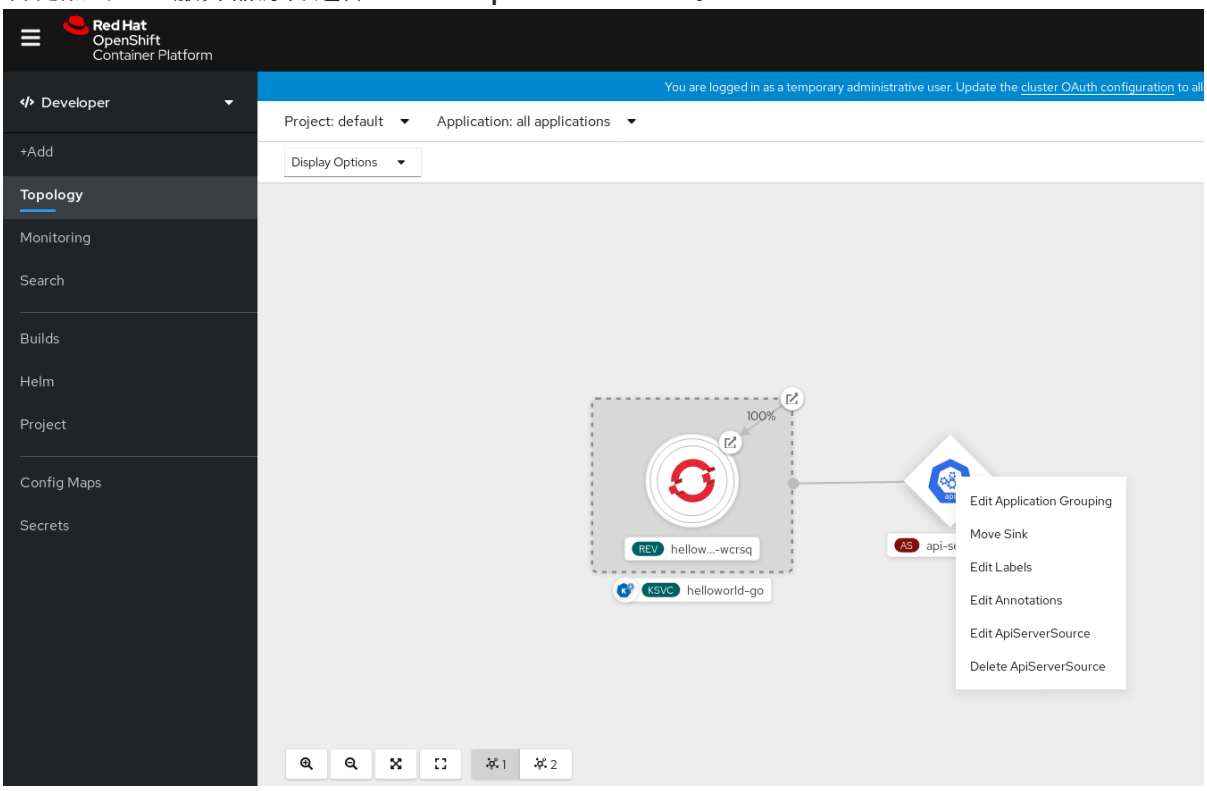


注意

如果使用 URI sink，请右键点击 **URI sink → Edit URI** 来修改 URI。

删除 API 服务器源

1. 导航到 **Topology** 视图。
2. 右键点击 API 服务器源并选择 **Delete ApiServerSource**。

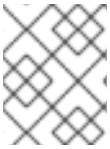


5.8.2. 使用 Knative CLI 创建 API 服务器源

您可以使用 **kn source apiserver create** 命令，使用 **kn** CLI 创建 API 服务器源。使用 **kn** CLI 创建 API 服务器源可提供比直接修改 YAML 文件更精简且直观的用户界面。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Eventing。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。
- 已安装 OpenShift CLI(**oc**)。
- 已安装 Knative (**kn**) CLI。



流程

如果要重新使用现有服务帐户，您可以修改现有的 **ServiceAccount** 资源，使其包含所需的权限，而不是创建新资源。

1. 以 YAML 文件形式,为事件源创建服务帐户、角色和角色绑定：

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: events-sa
  namespace: default 1

---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: event-watcher
  namespace: default 2
rules:
  - apiGroups:
      - ""
    resources:
      - events
    verbs:
      - get
      - list
      - watch

---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: k8s-ra-event-watcher
  namespace: default 3
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: event-watcher
subjects:
```

```
  - kind: ServiceAccount
    name: events-sa
    namespace: default 4
```

1 2 3 4 将这个命名空间更改为已选择安装事件源的命名空间。

2. 应用 YAML 文件：

```
$ oc apply -f <filename>
```

3. 创建具有事件 sink 的 API 服务器源。在以下示例中，sink 是一个代理：

```
$ kn source apiserver create <event_source_name> --sink broker:<broker_name> --
resource "event:v1" --service-account <service_account_name> --mode Resource
```

4. 要检查 API 服务器源是否已正确设置，请创建一个 Knative 服务，在日志中转储传入的信息：

```
$ kn service create <service_name> --image quay.io/openshift-knative/knative-eventing-
sources-event-display:latest
```

5. 如果您使用代理作为事件 sink，请创建一个触发器将事件从 **default** 代理过滤到服务：

```
$ kn trigger create <trigger_name> --sink ksvc:<service_name>
```

6. 通过在 default 命名空间中启动 pod 来创建事件：

```
$ oc create deployment hello-node --image quay.io/openshift-knative/knative-eventing-
sources-event-display:latest
```

7. 通过检查以下命令生成的输出来检查是否正确映射了控制器：

```
$ kn source apiserver describe <source_name>
```

输出示例

```
Name:               mysource
Namespace:          default
Annotations:        sources.knative.dev/creator=developer,
sources.knative.dev/lastModifier=developer
Age:                3m
ServiceAccountName:  events-sa
Mode:               Resource
Sink:
  Name:       default
  Namespace:  default
  Kind:       Broker (eventing.knative.dev/v1)
Resources:
  Kind:       event (v1)
  Controller:  false
Conditions:
  OK TYPE                  AGE REASON
  ++ Ready                 3m
```

```
++ Deployed                 3m
++ SinkProvided             3m
++ SufficientPermissions    3m
++ EventTypesProvided       3m
```

验证

您可以通过查看消息转储程序功能日志来验证 Kubernetes 事件是否已发送到 Knative。

1. 获取 pod:

```
$ oc get pods
```

2. 查看 pod 的消息转储程序功能日志：

```
$ oc logs $(oc get pod -o name | grep event-display) -c user-container
```

输出示例

```
☁  cloudevents.Event
Validation: valid
Context Attributes,
  specversion: 1.0
  type: dev.knative.apiserver.resource.update
  datacontenttype: application/json
  ...
Data,
  {
    "apiVersion": "v1",
    "involvedObject": {
      "apiVersion": "v1",
      "fieldPath": "spec.containers{hello-node}",
      "kind": "Pod",
      "name": "hello-node",
      "namespace": "default",
       .....
    },
    "kind": "Event",
    "message": "Started container",
    "metadata": {
      "name": "hello-node.159d7608e3a3572c",
      "namespace": "default",
      ....
    },
    "reason": "Started",
    ...
  }
```

删除 API 服务器源

1. 删除触发器：

```
$ kn trigger delete <trigger_name>
```

2. 删除事件源：

```
$ kn source apiserver delete <source_name>
```

3. 删除服务帐户、集群角色和集群绑定：

```
$ oc delete -f authentication.yaml
```

5.8.2.1. Knative CLI sink 标记

当使用 Knative (**kn**) CLI 创建事件源时，您可以使用 **--sink** 标志指定事件从该资源发送到的接收器。sink 可以是任何可寻址或可调用的资源，可以从其他资源接收传入的事件。

以下示例创建使用服务 **http://event-display.svc.cluster.local** 的接收器绑定作为接收器：

使用 sink 标记的命令示例

```
$ kn source binding create bind-heartbeat \
  --namespace sinkbinding-example \
  --subject "Job:batch/v1:app=heartbeat-cron" \
  --sink http://event-display.svc.cluster.local \ 1
  --ce-override "sink=bound"
```

1 **http://event-display.svc.cluster.local** 中的 **svc** 确定接收器是一个 Knative 服务。其他默认的接收器前缀包括 **channel** 和 **broker**。

5.8.3. 使用 YAML 文件创建 API 服务器源

使用 YAML 文件创建 Knative 资源使用声明性 API，它允许您以可重复的方式描述事件源。要使用 YAML 创建 API 服务器源，您必须创建一个 YAML 文件来定义 **ApiServerSource** 对象，然后使用 **oc apply** 命令应用它。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Eventing。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。
- 您已在与 API 服务器源 YAML 文件中定义的相同的命名空间中创建 **default** 代理。
- 安装 OpenShift CLI (**oc**) 。

流程

如果要重新使用现有服务帐户，您可以修改现有的 **ServiceAccount** 资源，使其包含所需的权限，而不是创建新资源。

1. 以 YAML 文件形式,为事件源创建服务帐户、角色和角色绑定：

```
apiVersion: v1
kind: ServiceAccount
```

```
metadata:
  name: events-sa
  namespace: default (1)

---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: event-watcher
  namespace: default (2)
rules:
  - apiGroups:
      - ""
    resources:
      - events
    verbs:
      - get
      - list
      - watch

---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: k8s-ra-event-watcher
  namespace: default (3)
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: event-watcher
subjects:
  - kind: ServiceAccount
    name: events-sa
    namespace: default (4)
```

(1)(2)(3)(4)将这个命名空间更改为已选择安装事件源的命名空间。

2. 应用 YAML 文件：

```
$ oc apply -f <filename>
```

3. 将 API 服务器源创建为 YAML 文件：

```
apiVersion: sources.knative.dev/v1alpha1
kind: ApiServerSource
metadata:
  name: testevents
spec:
  serviceAccountName: events-sa
  mode: Resource
  resources:
    - apiVersion: v1
      kind: Event
  sink:
```

```
    ref:
      apiVersion: eventing.knative.dev/v1
      kind: Broker
      name: default
```

4. 应用 **ApiServerSource** YAML 文件：

```
$ oc apply -f <filename>
```

5. 要检查 API 服务器源是否已正确设置，请创建一个 Knative 服务作为 YAML 文件，在日志中转储传入的信息：

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: event-display
  namespace: default
spec:
  template:
    spec:
      containers:
        - image: quay.io/openshift-knative/knative-eventing-sources-event-display:latest
```

6. 应用 **Service** YAML 文件：

```
$ oc apply -f <filename>
```

7. 创建一个 **Trigger** 对象作为一个 YAML 文件，该文件将事件从 **default** 代理过滤到上一步中创建的服务：

```
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  name: event-display-trigger
  namespace: default
spec:
  broker: default
  subscriber:
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: event-display
```

8. 应用 **Trigger** YAML 文件：

```
$ oc apply -f <filename>
```

9. 通过在 default 命名空间中启动 pod 来创建事件：

```
$ oc create deployment hello-node --image=quay.io/openshift-knative/knative-eventing-sources-event-display
```

10. 输入以下命令并检查输出，检查是否正确映射了控制器：

```
-
```

```
$ oc get apiserversource.sources.knative.dev testevents -o yaml
```

输**出示例**

```
apiVersion: sources.knative.dev/v1alpha1
kind: ApiServerSource
metadata:
  annotations:
  creationTimestamp: "2020-04-07T17:24:54Z"
  generation: 1
  name: testevents
  namespace: default
  resourceVersion: "62868"
  selfLink:
/apis/sources.knative.dev/v1alpha1/namespaces/default/apiserversources/testevents2
  uid: 1603d863-bb06-4d1c-b371-f580b4db99fa
spec:
  mode: Resource
  resources:
  - apiVersion: v1
    controller: false
    controllerSelector:
      apiVersion: ""
      kind: ""
      name: ""
      uid: ""
    kind: Event
    labelSelector: {}
  serviceAccountName: events-sa
  sink:
    ref:
      apiVersion: eventing.knative.dev/v1
      kind: Broker
      name: default
```

验证

要验证 Kubernetes 事件是否已发送到 Knative，您可以查看消息转储程序功能日志。

1. 输入以下命令来获取 pod：

```
$ oc get pods
```

2. 输入以下命令来查看 pod 的消息转储程序功能日志：

```
$ oc logs $(oc get pod -o name | grep event-display) -c user-container
```

输**出示例**

```
☁  cloudevents.Event
Validation: valid
Context Attributes,
  specversion: 1.0
  type: dev.knative.apiserver.resource.update
```

```
  datacontenttype: application/json
  ...
Data,
  {
    "apiVersion": "v1",
    "involvedObject": {
      "apiVersion": "v1",
      "fieldPath": "spec.containers{hello-node}",
      "kind": "Pod",
      "name": "hello-node",
      "namespace": "default",
       .....
    },
    "kind": "Event",
    "message": "Started container",
    "metadata": {
      "name": "hello-node.159d7608e3a3572c",
      "namespace": "default",
      ....
    },
    "reason": "Started",
    ...
  }
```

删除 API 服务器源

1. 删除触发器：

```
$ oc delete -f trigger.yaml
```

2. 删除事件源：

```
$ oc delete -f k8s-events.yaml
```

3. 删除服务帐户、集群角色和集群绑定：

```
$ oc delete -f authentication.yaml
```

5.9. 创建 PING 源

ping 源是一个事件源，可用于定期向事件消费者发送带有恒定有效负载的 ping 事件。ping 源可以用来调度发送事件，类似于计时器。

5.9.1. 使用 Web 控制台创建 ping 源

在集群中安装 Knative Eventing 后，您可以使用 web 控制台创建 ping 源。使用 OpenShift Container Platform Web 控制台提供了一个简化且直观的用户界面来创建事件源。

先决条件

- 已登陆到 OpenShift Container Platform Web 控制台。
- 在集群中安装了 OpenShift Serverless Operator、Knative Serving 和 Knative Eventing。

- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。

流程

1. 要验证 ping 源是否可以工作，请创建一个简单的 Knative 服务，在服务日志中转储传入的信息。
   a. 在 **Developer** 视角中，导航到 **+Add → YAML**。
   b. 复制 YAML 示例：

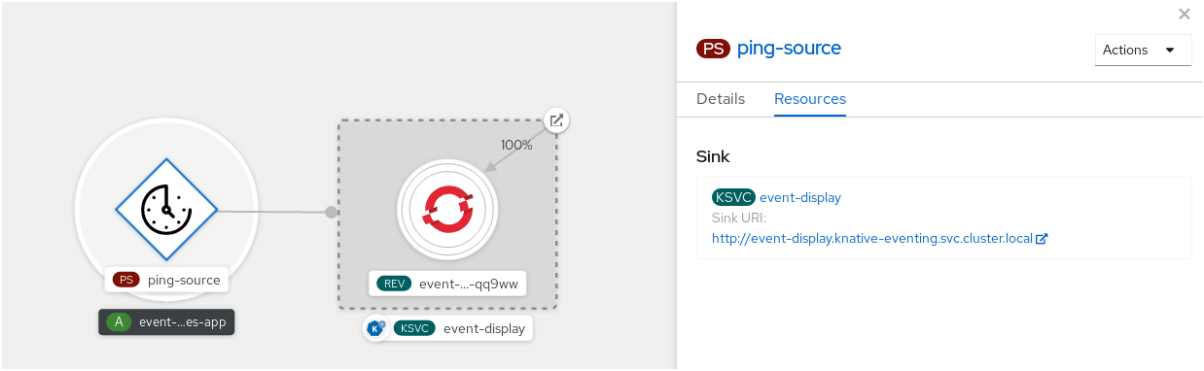
```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: event-display
spec:
  template:
    spec:
      containers:
        - image: quay.io/openshift-knative/knative-eventing-sources-event-display:latest
```

   c. 点 **Create**。
2. 在与上一步中创建的服务相同的命名空间中创建一个 ping 源，或您要将事件发送到的任何其他接收器。
   a. 在 **Developer** 视角中，导航到 **+Add → Event Source**。
   b. 选择 **Ping Source**。
   c. 可选：您可以为 **Data** 输入一个值，它是消息的有效负载。
   d. 为 **Schedule** 输入一个值。在本例中，值为 ***/2 * * * ***，它会创建一个 ping 源，每两分钟发送一条消息。
   e. 选择一个 **Sink**。这可以是 **Resource** 或一个 **URI**。在这个示例中，上一步中创建的 **event-display** 服务被用作 **Resource** sink。
   f. 点 **Create**。

验证

您可以通过查看 **Topology** 页面来验证 ping 源是否已创建并连接到接收器。

1. 在 **Developer** 视角中，导航到 **Topology**。
2. 查看 ping 源和接收器。



删除 ping 源

1. 导航到 **Topology** 视图。
2. 右键单击 API 服务器源，再选择 **Delete Ping Source**。

5.9.2. 使用 Knative CLI 创建 ping 源

您可以使用 **kn source ping create** 命令，通过 Knative (**kn**) CLI 创建 ping 源。使用 Knative CLI 创建事件源提供了比直接修改 YAML 文件更精简且直观的用户界面。

先决条件

- 在集群中安装了 OpenShift Serverless Operator、Knative Serving 和 Knative Eventing。
- 已安装 Knative (**kn**) CLI。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。
- 可选： 如果要使用此流程验证步骤，请安装 OpenShift CLI (**oc**) 。

流程

1. 要验证 ping 源是否可以工作，请创建一个简单的 Knative 服务，在服务日志中转储传入的信息：

```
$ kn service create event-display \
    --image quay.io/openshift-knative/knative-eventing-sources-event-display:latest
```

2. 对于您要请求的每一组 ping 事件，请在与事件消费者相同的命名空间中创建一个 ping 源：

```
$ kn source ping create test-ping-source \
    --schedule "*/2 * * * *" \
    --data '{"message": "Hello world!"}' \
    --sink ksvc:event-display
```

3. 输入以下命令并检查输出，检查是否正确映射了控制器：

```
$ kn source ping describe test-ping-source
```

输出示例

```
Name:         test-ping-source
Namespace:    default
Annotations:  sources.knative.dev/creator=developer,
sources.knative.dev/lastModifier=developer
Age:          15s
Schedule:     */2 * * * *
Data:         {"message": "Hello world!"}

Sink:
  Name:       event-display
  Namespace:  default
  Resource:   Service (serving.knative.dev/v1)
```

```
Conditions:
  OK TYPE                  AGE REASON
  ++ Ready                 8s
  ++ Deployed              8s
  ++ SinkProvided          15s
  ++ ValidSchedule         15s
  ++ EventTypeProvided     15s
  ++ ResourcesCorrect      15s
```

验证

您可以通过查看 sink pod 的日志来验证 Kubernetes 事件是否已发送到 Knative 事件。

默认情况下，如果在 60 秒内都没有流量，Knative 服务会终止其 Pod。本指南中演示的示例创建了一个 ping 源，每 2 分钟发送一条消息，因此每个消息都应该在新创建的 pod 中观察到。

1. 查看新创建的 pod：

```
$ watch oc get pods
```

2. 使用 Ctrl+C 取消查看 pod，然后查看所创建 pod 的日志：

```
$ oc logs $(oc get pod -o name | grep event-display) -c user-container
```

输**出示例**

```
☁  cloudevents.Event
Validation: valid
Context Attributes,
  specversion: 1.0
  type: dev.knative.sources.ping
  source: /apis/v1/namespaces/default/pingsources/test-ping-source
  id: 99e4f4f6-08ff-4bff-acf1-47f61ded68c9
  time: 2020-04-07T16:16:00.000601161Z
  datacontenttype: application/json
Data,
  {
    "message": "Hello world!"
  }
```

删除 ping 源

- 删除 ping 源：

```
$ kn delete pingsources.sources.knative.dev <ping_source_name>
```

5.9.2.1. Knative CLI sink 标记

当使用 Knative (**kn**) CLI 创建事件源时，您可以使用 **--sink** 标志指定事件从该资源发送到的接收器。sink 可以是任何可寻址或可调用的资源，可以从其他资源接收传入的事件。

以下示例创建使用服务 **http://event-display.svc.cluster.local** 的接收器绑定作为接收器：

使用 sink 标记的命令示例

```
$ kn source binding create bind-heartbeat \
  --namespace sinkbinding-example \
  --subject "Job:batch/v1:app=heartbeat-cron" \
  --sink http://event-display.svc.cluster.local \ 1
  --ce-override "sink=bound"
```

1 **http://event-display.svc.cluster.local** 中的 **svc** 确定接收器是一个 Knative 服务。其他默认的接收器前缀包括 **channel** 和 **broker**。

5.9.3. 使用 YAML 创建 ping 源

使用 YAML 文件创建 Knative 资源使用声明性 API，它允许您以可重复的方式描述事件源。要使用 YAML 创建无服务器 ping 源，您必须创建一个 YAML 文件来定义 **PingSource** 对象，然后使用 **oc apply** 来应用它。

PingSource 对象示例

```
apiVersion: sources.knative.dev/v1
kind: PingSource
metadata:
  name: test-ping-source
spec:
  schedule: "*/2 * * * *" 1
  data: '{"message": "Hello world!"}' 2
  sink: 3
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: event-display
```

1 事件指定的调度使用 CRON 格式。

2 事件消息正文以 JSON 编码的数据字符串表示。

3 这些是事件消费者的详情。在这个示例中，我们使用名为 **event-display** 的 Knative 服务。

先决条件

- 在集群中安装了 OpenShift Serverless Operator、Knative Serving 和 Knative Eventing。
- 安装 OpenShift CLI (**oc**) 。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。

流程

1. 要验证 ping 源是否可以工作，请创建一个简单的 Knative 服务，在服务日志中转储传入的信息。
   a. 创建服务 YAML 文件：

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: event-display
spec:
  template:
    spec:
      containers:
        - image: quay.io/openshift-knative/knative-eventing-sources-event-display:latest
```

b. 创建服务：

```
$ oc apply -f <filename>
```

2. 对于您要请求的每一组 ping 事件，请在与事件消费者相同的命名空间中创建一个 ping 源。

a. 为 ping 源创建 YAML 文件：

```
apiVersion: sources.knative.dev/v1
kind: PingSource
metadata:
  name: test-ping-source
spec:
  schedule: "*/2 * * * *"
  data: '{"message": "Hello world!"}'
  sink:
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: event-display
```

b. 创建 ping 源：

```
$ oc apply -f <filename>
```

3. 输入以下命令检查是否正确映射了控制器：

```
$ oc get pingsource.sources.knative.dev <ping_source_name> -oyaml
```

输出示例

```
apiVersion: sources.knative.dev/v1
kind: PingSource
metadata:
  annotations:
    sources.knative.dev/creator: developer
    sources.knative.dev/lastModifier: developer
  creationTimestamp: "2020-04-07T16:11:14Z"
  generation: 1
  name: test-ping-source
  namespace: default
  resourceVersion: "55257"
  selfLink: /apis/sources.knative.dev/v1/namespaces/default/pingsources/test-ping-source
  uid: 3d80d50b-f8c7-4c1b-99f7-3ec00e0a8164
```

```
spec:
  data: '{ value: "hello" }'
  schedule: '*/2 * * * *'
  sink:
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: event-display
      namespace: default
```

验证

您可以通过查看 sink pod 的日志来验证 Kubernetes 事件是否已发送到 Knative 事件。

默认情况下，如果在 60 秒内都没有流量，Knative 服务会终止其 Pod。本指南中演示的示例创建了一个 PingSource，每 2 分钟发送一条消息，因此每个消息都应该在新创建的 pod 中观察到。

1. 查看新创建的 pod：

```
$ watch oc get pods
```

2. 使用 Ctrl+C 取消查看 pod，然后查看所创建 pod 的日志：

```
$ oc logs $(oc get pod -o name | grep event-display) -c user-container
```

输**出示例**

```
☁  cloudevents.Event
Validation: valid
Context Attributes,
  specversion: 1.0
  type: dev.knative.sources.ping
  source: /apis/v1/namespaces/default/pingsources/test-ping-source
  id: 042ff529-240e-45ee-b40c-3a908129853e
  time: 2020-04-07T16:22:00.000791674Z
  datacontenttype: application/json
Data,
  {
    "message": "Hello world!"
  }
```

删除 ping 源

- 删除 ping 源：

```
$ oc delete -f <filename>
```

示例命令

```
$ oc delete -f ping-source.yaml
```

5.10. 自定义事件源

如果您需要从 Knative 中没有包含在 Knative 的事件制作者或发出没有 **CloudEvent** 格式的事件的制作者中入站事件，您可以通过创建自定义事件源来实现此目标。您可以使用以下方法之一创建自定义事件源：

- 通过创建接收器绑定，将 **PodSpecable** 对象用作事件源。
- 通过创建容器源，将容器用作事件源。

5.10.1. 接收器（sink）绑定

SinkBinding 对象支持将事件产品与交付寻址分离。接收器绑定用于将 *事件制作者* 连接到事件消费者（*sink*）。event producer 是一个 Kubernetes 资源，用于嵌入 **PodSpec** 模板并生成事件。sink 是一个可寻址的 Kubernetes 对象，可以接收事件。

SinkBinding 对象将环境变量注入到 sink 的 **PodTemplateSpec** 中，这意味着应用程序代码不需要直接与 Kubernetes API 交互来定位事件目的地。这些环境变量如下：

K_SINK

解析 sink 的 URL。

K_CE_OVERRIDES

指定出站事件覆盖的 JSON 对象。

注意

SinkBinding 对象目前不支持服务的自定义修订名称。

5.10.1.1. 使用 YAML 创建接收器绑定

使用 YAML 文件创建 Knative 资源使用声明性 API，它允许您以可重复的方式描述事件源。要使用 YAML 创建接收器绑定，您必须创建一个 YAML 文件来定义 **SinkBinding** 对象，然后使用 **oc apply** 命令应用它。

先决条件

- 在集群中安装了 OpenShift Serverless Operator、Knative Serving 和 Knative Eventing。
- 安装 OpenShift CLI (**oc**)。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。

流程

1. 要检查接收器绑定是否已正确设置，请创建一个 Knative 事件显示服务或事件接收器，在日志中转储传入的信息。
   a. 创建服务 YAML 文件：

服务 YAML 文件示例

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: event-display
spec:
```

```
  template:
    spec:
      containers:
        - image: quay.io/openshift-knative/knative-eventing-sources-event-display:latest
```

b. 创建服务：

```
$ oc apply -f <filename>
```

2. 创建将事件定向到该服务的接收器绑定实例。

a. 创建接收器绑定 YAML 文件：

服务 YAML 文件示例

```
apiVersion: sources.knative.dev/v1alpha1
kind: SinkBinding
metadata:
  name: bind-heartbeat
spec:
  subject:
    apiVersion: batch/v1
    kind: Job 1
    selector:
      matchLabels:
        app: heartbeat-cron

  sink:
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: event-display
```

1 在本例中，任何具有标签 **app: heartbeat-cron** 的作业都将被绑定到事件 sink。

b. 创建接收器绑定：

```
$ oc apply -f <filename>
```

3. 创建 **CronJob** 对象。

a. 创建 cron 任务 YAML 文件：

Cron Job YAML 文件示例

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: heartbeat-cron
spec:
  # Run every minute
  schedule: "* * * * *"
  jobTemplate:
    metadata:
```

```
    labels:
      app: heartbeat-cron
      bindings.knative.dev/include: "true"
  spec:
    template:
      spec:
        restartPolicy: Never
        containers:
          - name: single-heartbeat
            image: quay.io/openshift-knative/heartbeats:latest
            args:
              - --period=1
            env:
              - name: ONE_SHOT
                value: "true"
              - name: POD_NAME
                valueFrom:
                  fieldRef:
                    fieldPath: metadata.name
              - name: POD_NAMESPACE
                valueFrom:
                  fieldRef:
                    fieldPath: metadata.namespace
```

重要

要使用接收器绑定，您必须手动在 Knative 资源中添加 **bindings.knative.dev/include=true** 标签。

例如，要将此标签添加到 **CronJob** 资源，请将以下行添加到 **Job** 资源 YAML 定义中：

```
jobTemplate:
  metadata:
    labels:
      app: heartbeat-cron
      bindings.knative.dev/include: "true"
```

b. 创建 cron job：

```
$ oc apply -f <filename>
```

4. 输入以下命令并检查输出，检查是否正确映射了控制器：

```
$ oc get sinkbindings.sources.knative.dev bind-heartbeat -oyaml
```

输出示例

```
spec:
  sink:
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
```

```
    name: event-display
    namespace: default
subject:
  apiVersion: batch/v1
  kind: Job
  namespace: default
  selector:
    matchLabels:
      app: heartbeat-cron
```

验证

您可以通过查看消息 dumper 功能日志，来验证 Kubernetes 事件是否已发送到 Knative 事件。

1. 输入命令：

```
$ oc get pods
```

2. 输入命令：

```
$ oc logs $(oc get pod -o name | grep event-display) -c user-container
```

输**出示例**

```
☁  cloudevents.Event
Validation: valid
Context Attributes,
  specversion: 1.0
  type: dev.knative.eventing.samples.heartbeat
  source: https://knative.dev/eventing-contrib/cmd/heartbeats/#event-test/mypod
  id: 2b72d7bf-c38f-4a98-a433-608fbcdd2596
  time: 2019-10-18T15:23:20.809775386Z
  contenttype: application/json
Extensions,
  beats: true
  heart: yes
  the: 42
Data,
  {
    "id": 1,
    "label": ""
  }
```

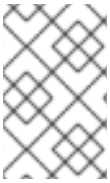
5.10.1.2. 使用 Knative CLI 创建接收器绑定

您可以使用 **kn source binding create** 命令通过 Knative (**kn**) CLI 创建接收器绑定。使用 Knative CLI 创建事件源提供了比直接修改 YAML 文件更精简且直观的用户界面。

先决条件

- 在集群中安装了 OpenShift Serverless Operator、Knative Serving 和 Knative Eventing。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。

- 安装 Knative (**kn**) CLI。
- 安装 OpenShift CLI (**oc**) 。



注意

以下操作过程要求您创建 YAML 文件。

如果更改了示例中使用的 YAML 文件的名称，则需要更新对应的 CLI 命令。

流程

1. 要检查接收器绑定是否已正确设置，请创建一个 Knative 事件显示服务或事件 sink，在日志中转储传入的信息：

```
$ kn service create event-display --image quay.io/openshift-knative/knative-eventing-sources-event-display:latest
```

2. 创建将事件定向到该服务的接收器绑定实例：

```
$ kn source binding create bind-heartbeat --subject Job:batch/v1:app=heartbeat-cron --sink ksvc:event-display
```

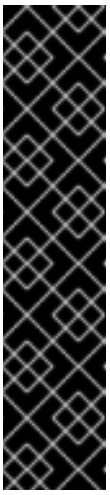
3. 创建 **CronJob** 对象。

   a. 创建 cron 任务 YAML 文件：

Cron Job YAML 文件示例

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: heartbeat-cron
spec:
  # Run every minute
  schedule: "* * * * *"
  jobTemplate:
    metadata:
      labels:
        app: heartbeat-cron
        bindings.knative.dev/include: "true"
    spec:
      template:
        spec:
          restartPolicy: Never
          containers:
            - name: single-heartbeat
              image: quay.io/openshift-knative/heartbeats:latest
              args:
                - --period=1
              env:
                - name: ONE_SHOT
                  value: "true"
                - name: POD_NAME
                  valueFrom:
```

```
      fieldRef:
        fieldPath: metadata.name
    - name: POD_NAMESPACE
      valueFrom:
        fieldRef:
          fieldPath: metadata.namespace
```



重要

要使用接收器绑定，您必须手动在 Knative CR 中添加 **bindings.knative.dev/include=true** 标签。

例如，要将此标签添加到 **CronJob** CR，请将以下行添加到 **Job** CR YAML 定义中：

```
jobTemplate:
  metadata:
    labels:
      app: heartbeat-cron
      bindings.knative.dev/include: "true"
```

b. 创建 cron job：

```
$ oc apply -f <filename>
```

4. 输入以下命令并检查输出，检查是否正确映射了控制器：

```
$ kn source binding describe bind-heartbeat
```

输**出示例**

```
Name:         bind-heartbeat
Namespace:    demo-2
Annotations:  sources.knative.dev/creator=minikube-user,
sources.knative.dev/lastModifier=minikub ...
Age:          2m
Subject:
  Resource:   job (batch/v1)
  Selector:
    app:      heartbeat-cron
Sink:
  Name:       event-display
  Resource:   Service (serving.knative.dev/v1)

Conditions:
  OK TYPE     AGE REASON
  ++ Ready    2m
```

验证

您可以通过查看消息 dumper 功能日志，来验证 Kubernetes 事件是否已发送到 Knative 事件。

- 您可以输入以下命令来查看消息转储程序功能日志：

```
$ oc get pods
```

```
$ oc logs $(oc get pod -o name | grep event-display) -c user-container
```

输出示例

```
☁  cloudevents.Event
Validation: valid
Context Attributes,
  specversion: 1.0
  type: dev.knative.eventing.samples.heartbeat
  source: https://knative.dev/eventing-contrib/cmd/heartbeats/#event-test/mypod
  id: 2b72d7bf-c38f-4a98-a433-608fbcdd2596
  time: 2019-10-18T15:23:20.809775386Z
  contenttype: application/json
Extensions,
  beats: true
  heart: yes
  the: 42
Data,
  {
    "id": 1,
    "label": ""
  }
```

5.10.1.2.1. Knative CLI sink 标记

当使用 Knative (**kn**) CLI 创建事件源时，您可以使用 **--sink** 标志指定事件从该资源发送到的接收器。sink 可以是任何可寻址或可调用的资源，可以从其他资源接收传入的事件。

以下示例创建使用服务 **http://event-display.svc.cluster.local** 的接收器绑定作为接收器：

使用 sink 标记的命令示例

```
$ kn source binding create bind-heartbeat \
  --namespace sinkbinding-example \
  --subject "Job:batch/v1:app=heartbeat-cron" \
  --sink http://event-display.svc.cluster.local \ 1
  --ce-override "sink=bound"
```

1 **http://event-display.svc.cluster.local** 中的 **svc** 确定接收器是一个 Knative 服务。其他默认的接收器前缀包括 **channel** 和 **broker**。

5.10.1.3. 使用 Web 控制台创建接收器绑定

在集群中安装 Knative Eventing 后，您可以使用 web 控制台创建接收器绑定。使用 OpenShift Container Platform Web 控制台提供了一个简化且直观的用户界面来创建事件源。

先决条件

- 已登陆到 OpenShift Container Platform Web 控制台。

- OpenShift Serverless Operator、Knative Serving 和 Knative Eventing 已在 OpenShift Container Platform 集群中安装。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。

流程

1. 创建 Knative 服务以用作接收器：
   a. 在 **Developer** 视角中，导航到 **+Add → YAML**。
   b. 复制 YAML 示例：

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: event-display
spec:
  template:
    spec:
      containers:
        - image: quay.io/openshift-knative/knative-eventing-sources-event-display:latest
```

   c. 点 **Create**。
2. 创建用作事件源的 **CronJob** 资源，并每分钟发送一个事件。
   a. 在 **Developer** 视角中，导航到 **+Add → YAML**。
   b. 复制 YAML 示例：

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: heartbeat-cron
spec:
  # Run every minute
  schedule: "*/1 * * * *"
  jobTemplate:
    metadata:
      labels:
        app: heartbeat-cron
        bindings.knative.dev/include: true (1)
    spec:
      template:
        spec:
          restartPolicy: Never
          containers:
            - name: single-heartbeat
              image: quay.io/openshift-knative/heartbeats
              args:
              - --period=1
              env:
                - name: ONE_SHOT
                  value: "true"
```

```
        - name: POD_NAME
          valueFrom:
            fieldRef:
              fieldPath: metadata.name
        - name: POD_NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
```

1 确保包含 **bindings.knative.dev/include: true** 标签。OpenShift Serverless 的默认命名空间选择行为使用包含模式。

c. 点 **Create**。

3. 在与上一步中创建的服务相同的命名空间中创建接收器绑定，或您要将事件发送到的任何其他接收器。

a. 在 **Developer** 视角中，导航到 **+Add → Event Source**。此时会显示 **Event Sources** 页面。

b. 可选：如果您的事件源有多个供应商，请从 **Providers** 列表中选择所需的供应商，以过滤供应商的可用事件源。

c. 选择 **Sink Binding**，然后单击 **Create Event Source**。此时会显示 **Create Event Source** 页面。

d. 在 **apiVersion** 字段中，输入 **batch/v1**。

e. 在 **Kind** 字段中，输入 **Job**。

注意

OpenShift Serverless sink 绑定不支持 **CronJob** kind，因此 **Kind** 字段必须以 cron 任务创建的 **Job** 对象为目标，而不是 cron 作业对象本身。

f. 选择一个 **Sink**。这可以是 **Resource** 或一个 **URI**。在这个示例中，上一步中创建的 **event-display** 服务被用作 **Resource** sink。

g. 在 **Match labels** 部分：

i. 在 **Name** 字段中输入 **app**。

ii. 在 **Value** 字段中输入 **heartbeat-cron**。

注意

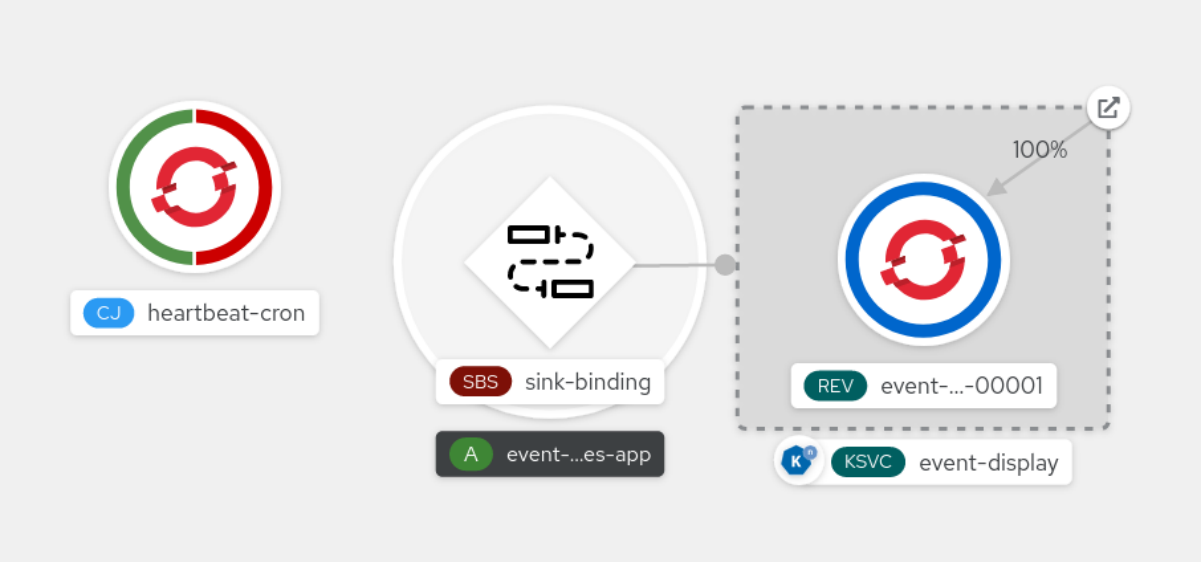
使用带有接收器绑定的 cron 任务时，需要标签选择器，而不是资源名称。这是因为，Cron Job 创建的作业没有可预测的名称，并在名称中包含随机生成的字符串。例如，**hearthbeat-cron-1cc23f**.

h. 点 **Create**。

验证

您可以通过查看 **Topology** 页面和 pod 日志来验证接收器绑定、接收器和 cron 任务是否已创建并正常工作。

1. 在 Developer 视角中，导航到 Topology。
2. 查看接收器绑定、接收器和心跳 cron 任务。



3. 观察在添加了接收器绑定后 cron 任务正在注册成功的作业。这意味着接收器绑定成功重新配置由 cron 任务创建的作业。
4. 浏览 **event-display** 服务 pod 的日志，以查看 heartbeats cron 作业生成的事件。

5.10.1.4. 接收器绑定引用

您可以通过创建接收器绑定，将 **PodSpecable** 对象用作事件源。您可以在创建 **SinkBinding** 对象时配置多个参数。

SinkBinding 对象支持以下参数：

字段	描述	必需或可选
apiVersion	指定 API 版本，如 **sources.knative.dev/v1**。	必需
kind	将此资源对象标识为 **SinkBinding** 对象。	必需
metadata	指定唯一标识 **SinkBinding** 对象的元数据。例如，**名称**。	必需
spec	指定此 **SinkBinding** 对象的配置信息。	必需
spec.sink	对解析为 URI 作为 sink 的对象的引用。	必需
spec.subject	提及通过绑定实施来增强运行时合同的资源。	必需

字段	描述	必需或可选
spec.ceOverrides	定义覆盖来控制发送到 sink 的事件的输出格式和修改。	选填

5.10.1.4.1. 主题参数

Subject 参数引用通过绑定实施来增强运行时合同的资源。您可以为 **Subject** 定义配置多个字段。

Subject 定义支持以下字段：

字段	描述	必需或可选
apiVersion	引用的 API 版本。	必需
kind	引用的类型。	必需
namespace	引用的命名空间。如果省略，则默认为对象的命名空间。	选填
name	引用的名称。	如果配置 **选择器**，请不要使用。
selector	引用的选择器。	如果配置 **名称**，请不要使用。
selector.matchExpressions	标签选择器要求列表。	仅使用 **matchExpressions** 或 **matchLabels** 中的一个。
selector.matchExpressions.key	选择器应用到的标签键。	使用 **matchExpressions** 时需要此项。
selector.matchExpressions.operator	代表键与一组值的关系。有效的运算符为 **In**、**NotIn**、**Exists** 和 **DoesNotExist**。	使用 **matchExpressions** 时需要此项。
selector.matchExpressions.values	字符串值数组。如果 **operator** 参数值是 **In** 或 **NotIn**，则值数组必须是非空的。如果 **operator** 参数值是 **Exists** 或 **DoesNotExist**，则值数组必须为空。这个数组会在策略性合并补丁中被替换。	使用 **matchExpressions** 时需要此项。
selector.matchLabels	键值对映射.**matchLabels** 映射中的每个键值对等同于 **matchExpressions** 元素，其中 key 字段是 **matchLabels.<key>**，**Operator** 为 **In**，**values** 数组仅包含 **matchLabels.<value>**。	仅使用 **matchExpressions** 或 **matchLabels** 中的一个。

主题参数示例

根据以下 YAML，选择 **default** 命名空间中名为 **mysubject** 的 **Deployment** 对象：

```
apiVersion: sources.knative.dev/v1
kind: SinkBinding
metadata:
  name: bind-heartbeat
spec:
  subject:
    apiVersion: apps/v1
    kind: Deployment
    namespace: default
    name: mysubject
  ...
```

根据以下 YAML，可以选择在 **default** 命名空间中带有 **working=example** 标签的 **Job** 对象：

```
apiVersion: sources.knative.dev/v1
kind: SinkBinding
metadata:
  name: bind-heartbeat
spec:
  subject:
    apiVersion: batch/v1
    kind: Job
    namespace: default
    selector:
      matchLabels:
        working: example
  ...
```

根据以下 YAML，可以选择在 **default** 命名空间中带有标签 **working=example** 或 **working=sample** 的 **Pod** 对象：

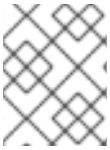
```
apiVersion: sources.knative.dev/v1
kind: SinkBinding
metadata:
  name: bind-heartbeat
spec:
  subject:
    apiVersion: v1
    kind: Pod
    namespace: default
    selector:
      - matchExpression:
        key: working
        operator: In
        values:
          - example
          - sample
  ...
```

5.10.1.4.2. CloudEvent 覆盖

ceOverrides 定义提供覆盖控制发送到 sink 的 CloudEvent 输出格式和修改。您可以为 **ceOverrides** 定义配置多个字段。

ceOverrides 定义支持以下字段：

字段	描述	必需或可选
extensions	指定在出站事件中添加或覆盖哪些属性。每个 **extensions** 键值对在事件上作为属性扩展进行独立设置。	选填



注意

仅允许有效的 **CloudEvent** 属性名称作为扩展。您无法从扩展覆盖配置设置 spec 定义的属性。例如，您无法修改 **type** 属性。

CloudEvent Overrides 示例

```
apiVersion: sources.knative.dev/v1
kind: SinkBinding
metadata:
  name: bind-heartbeat
spec:
  ...
  ceOverrides:
    extensions:
      extra: this is an extra attribute
      additional: 42
```

这会在 **主题** 上设置 **K_CE_OVERRIDES** 环境变量：

输出示例

```
{ "extensions": { "extra": "this is an extra attribute", "additional": "42" } }
```

5.10.1.4.3. include 标签

要使用接收器绑定，您需要为资源或包含资源的命名空间分配 **bindings.knative.dev/include: "true"** 标签。如果资源定义不包括该标签，集群管理员可以通过运行以下命令将它附加到命名空间：

```
$ oc label namespace <namespace> bindings.knative.dev/include=true
```

5.10.2. 容器源

容器源创建容器镜像来生成事件并将事件发送到 sink。您可以通过创建容器镜像和使用您的镜像 URI 的 **ContainerSource** 对象，使用容器源创建自定义事件源。

5.10.2.1. 创建容器镜像的指南

两个环境变量由容器源控制器注入：**K_SINK** 和 **K_CE_OVERRIDES**。这些变量分别从 **sink** 和 **andceOverrides** spec 解析。事件发送到 **K_SINK** 环境变量中指定的 sink URI。该消息必须使用 **CloudEvent** HTTP 格式作为 **POST** 发送。

容器镜像示例

以下是心跳容器镜像的示例：

```
package main

import (
 "context"
 "encoding/json"
 "flag"
 "fmt"
 "log"
 "os"
 "strconv"
 "time"

 duckv1 "knative.dev/pkg/apis/duck/v1"

 cloudevents "github.com/cloudevents/sdk-go/v2"
 "github.com/kelseyhightower/envconfig"
)

type Heartbeat struct {
 Sequence int    `json:"id"`
 Label    string `json:"label"`
}

var (
 eventSource string
 eventType   string
 sink        string
 label       string
 periodStr   string
)

func init() {
 flag.StringVar(&eventSource, "eventSource", "", "the event-source (CloudEvents)")
 flag.StringVar(&eventType, "eventType", "dev.knative.eventing.samples.heartbeat", "the event-type
(CloudEvents)")
 flag.StringVar(&sink, "sink", "", "the host url to heartbeat to")
 flag.StringVar(&label, "label", "", "a special label")
 flag.StringVar(&periodStr, "period", "5", "the number of seconds between heartbeats")
}

type envConfig struct {
 // Sink URL where to send heartbeat cloud events
 Sink string `envconfig:"K_SINK"`

 // CEOverrides are the CloudEvents overrides to be applied to the outbound event.
 CEOverrides string `envconfig:"K_CE_OVERRIDES"`

 // Name of this pod.
 Name string `envconfig:"POD_NAME" required:"true"`
```

```
 // Namespace this pod exists in.
 Namespace string `envconfig:"POD_NAMESPACE" required:"true"`

 // Whether to run continuously or exit.
 OneShot bool `envconfig:"ONE_SHOT" default:"false"`
}

func main() {
 flag.Parse()

 var env envConfig
 if err := envconfig.Process("", &env); err != nil {
  log.Printf("[ERROR] Failed to process env var: %s", err)
  os.Exit(1)
 }

 if env.Sink != "" {
  sink = env.Sink
 }

 var ceOverrides *duckv1.CloudEventOverrides
 if len(env.CEOverrides) > 0 {
  overrides := duckv1.CloudEventOverrides{}
  err := json.Unmarshal([]byte(env.CEOverrides), &overrides)
  if err != nil {
   log.Printf("[ERROR] Unparseable CloudEvents overrides %s: %v", env.CEOverrides, err)
   os.Exit(1)
  }
  ceOverrides = &overrides
 }

 p, err := cloudevents.NewHTTP(cloudevents.WithTarget(sink))
 if err != nil {
  log.Fatalf("failed to create http protocol: %s", err.Error())
 }

 c, err := cloudevents.NewClient(p, cloudevents.WithUUIDs(), cloudevents.WithTimeNow())
 if err != nil {
  log.Fatalf("failed to create client: %s", err.Error())
 }

 var period time.Duration
 if p, err := strconv.Atoi(periodStr); err != nil {
  period = time.Duration(5) * time.Second
 } else {
  period = time.Duration(p) * time.Second
 }

 if eventSource == "" {
  eventSource = fmt.Sprintf("https://knative.dev/eventing-contrib/cmd/heartbeats/#%s/%s",
env.Namespace, env.Name)
  log.Printf("Heartbeats Source: %s", eventSource)
 }

 if len(label) > 0 && label[0] == "" {
```

```
 label, _ = strconv.Unquote(label)
}
hb := &Heartbeat{
 Sequence: 0,
 Label:    label,
}
ticker := time.NewTicker(period)
for {
 hb.Sequence++

 event := cloudevents.NewEvent("1.0")
 event.SetType(eventType)
 event.SetSource(eventSource)
 event.SetExtension("the", 42)
 event.SetExtension("heart", "yes")
 event.SetExtension("beats", true)

 if ceOverrides != nil && ceOverrides.Extensions != nil {
  for n, v := range ceOverrides.Extensions {
   event.SetExtension(n, v)
  }
 }

 if err := event.SetData(cloudevents.ApplicationJSON, hb); err != nil {
  log.Printf("failed to set cloudevents data: %s", err.Error())
 }

 log.Printf("sending cloudevent to %s", sink)
 if res := c.Send(context.Background(), event); !cloudevents.IsACK(res) {
  log.Printf("failed to send cloudevent: %v", res)
 }

 if env.OneShot {
  return
 }

 // Wait for next tick
 <-ticker.C
}
}
```

以下是引用先前心跳容器镜像的容器源示例：

```
apiVersion: sources.knative.dev/v1
kind: ContainerSource
metadata:
  name: test-heartbeats
spec:
  template:
    spec:
      containers:
        # This corresponds to a heartbeats image URI that you have built and published
        - image: gcr.io/knative-releases/knative.dev/eventing/cmd/heartbeats
          name: heartbeats
          args:
            - --period=1
```

```
        env:
          - name: POD_NAME
            value: "example-pod"
          - name: POD_NAMESPACE
            value: "event-test"
  sink:
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: example-service
...
```

5.10.2.2. 使用 Knative CLI 创建和管理容器源

您可以使用 **kn source container** 命令来使用 Knative (**kn**) 创建和管理容器源。使用 Knative CLI 创建事件源提供了比直接修改 YAML 文件更精简且直观的用户界面。

创建容器源

```
$ kn source container create <container_source_name> --image <image_uri> --sink <sink>
```

删除容器源

```
$ kn source container delete <container_source_name>
```

描述容器源

```
$ kn source container describe <container_source_name>
```

列出现有容器源

```
$ kn source container list
```

以 YAML 格式列出现有容器源

```
$ kn source container list -o yaml
```

更新容器源

此命令为现有容器源更新镜像 URI：

```
$ kn source container update <container_source_name> --image <image_uri>
```

5.10.2.3. 使用 Web 控制台创建容器源

在集群中安装 Knative Eventing 后，您可以使用 Web 控制台创建容器源。使用 OpenShift Container Platform Web 控制台提供了一个简化且直观的用户界面来创建事件源。

先决条件

- 已登陆到 OpenShift Container Platform Web 控制台。

- OpenShift Serverless Operator、Knative Serving 和 Knative Eventing 已在 OpenShift Container Platform 集群中安装。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。

流程

1. 在 **Developer** 视角中，导航到 **+Add → Event Source**。此时会显示 **Event Sources** 页面。
2. 选择 **Container Source**。
3. 配置 **Container Source** 设置：
   a. 在 **Image** 字段中，输入您要在由容器源创建的容器中运行的镜像的 URI。
   b. 在 **Name** 字段中输入镜像的名称。
   c. 可选：在 **Arguments** 参数字段中，输入要传递给容器的任何参数。
   d. 可选：在 **Environment variables** 字段中，添加容器中要设置的任何环境变量。
   e. 在 **Sink** 部分，添加一个接收器，其中将容器源的事件路由到其中。
      i. 选择 **Resource** 使用频道、代理或服务作为事件源的接收器。
      ii. 选择 **URI**，以指定容器源的事件路由到的位置。
4. 配置完容器源后，点 **Create**。

5.10.2.4. 容器源参考

您可以通过创建 **ContainerSource** 对象来使用容器作为事件源。您可以在创建 **ContainerSource** 对象时配置多个参数。

ContainerSource 对象支持以下字段：

字段	描述	必需或可选
apiVersion	指定 API 版本，如 **sources.knative.dev/v1**。	必需
kind	将此资源对象标识为 **ContainerSource** 对象。	必需
metadata	指定唯一标识 **ContainerSource** 对象的元数据。例如，**name**。	必需
spec	指定此 **ContainerSource** 对象的配置信息。	必需
spec.sink	对解析为 URI 作为 sink 的对象的引用。	必需

字段	描述	必需或可选
spec.template	**ContainerSource** 对象的 **template** 规格。	必需
spec.ceOverrides	定义覆盖来控制发送到 sink 的事件的输出格式和修改。	选填

模板参数示例

```
apiVersion: sources.knative.dev/v1
kind: ContainerSource
metadata:
  name: test-heartbeats
spec:
  template:
    spec:
      containers:
        - image: quay.io/openshift-knative/heartbeats:latest
          name: heartbeats
          args:
            - --period=1
          env:
            - name: POD_NAME
              value: "mypod"
            - name: POD_NAMESPACE
              value: "event-test"
  ...
```

5.10.2.4.1. CloudEvent 覆盖

ceOverrides 定义提供覆盖控制发送到 sink 的 CloudEvent 输出格式和修改。您可以为 **ceOverrides** 定义配置多个字段。

ceOverrides 定义支持以下字段：

字段	描述	必需或可选
extensions	指定在出站事件中添加或覆盖哪些属性。每个 **extensions** 键值对在事件上作为属性扩展进行独立设置。	选填

注意

仅允许有效的 **CloudEvent** 属性名称作为扩展。您无法从扩展覆盖配置设置 spec 定义的属性。例如，您无法修改 **type** 属性。

CloudEvent Overrides 示例

```
apiVersion: sources.knative.dev/v1
kind: ContainerSource
metadata:
  name: test-heartbeats
spec:
  ...
  ceOverrides:
    extensions:
      extra: this is an extra attribute
      additional: 42
```

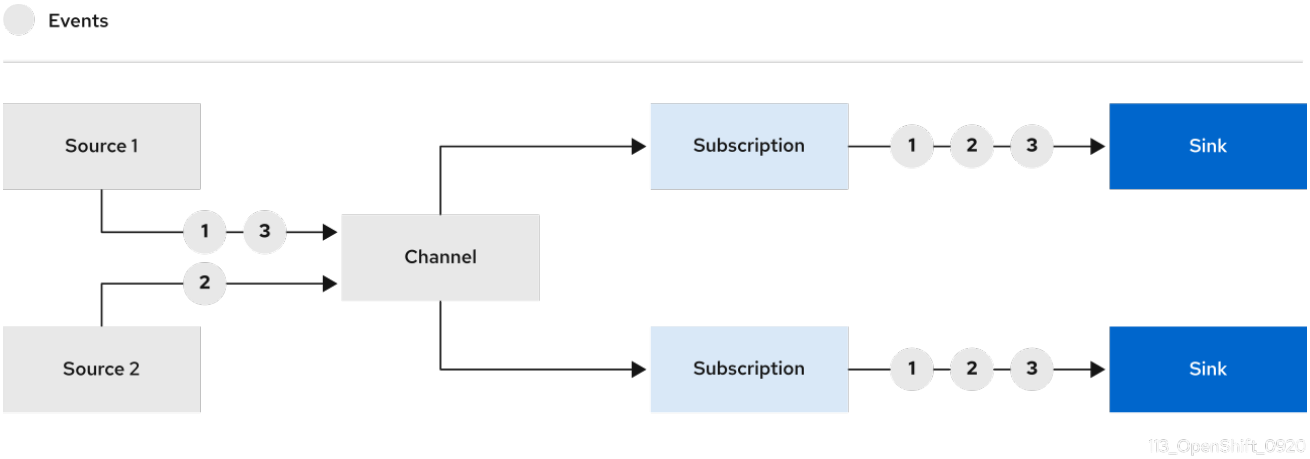
这会在 **主题** 上设置 **K_CE_OVERRIDES** 环境变量：

输**出示例**

```
{ "extensions": { "extra": "this is an extra attribute", "additional": "42" } }
```

5.11. 创建频道

频道是定义单一事件转发和持久层的自定义资源。事件源或生成程序在将事件发送到频道后，可使用订阅将这些事件发送到多个 Knative 服务或其他 sink。



您可以通过实例化受支持的 **Channel** 对象来创建频道，并通过修改 **Subscription** 对象中的 **delivery** 规格来配置重新发送尝试。

5.11.1. 使用 Web 控制台创建频道

使用 OpenShift Container Platform Web 控制台提供了一个简化的用户界面来创建频道。在集群中安装 Knative Eventing 后，您可以使用 web 控制台创建频道。

先决条件

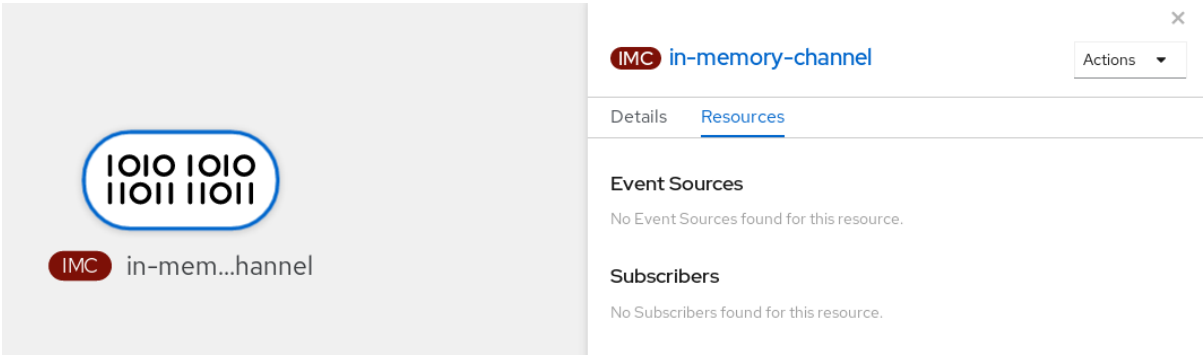
- 已登陆到 OpenShift Container Platform Web 控制台。
- OpenShift Serverless Operator 和 Knative Eventing 已安装在 OpenShift Container Platform 集群中。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。

流程

1. 在 **Developer** 视角中，导航到 **+Add → Channel**。
2. 选择您要在 **Type** 列表中创建的 **Channel** 对象类型。
3. 点 **Create**。

验证

- 通过导航到 **Topology** 页面确认频道现在存在。



5.11.2. 使用 Knative CLI 创建频道

使用 Knative (**kn**) 创建频道提供了比直接修改 YAML 文件更精简且直观的用户界面。您可以使用 **kn channel create** 命令创建频道。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Eventing。
- 已安装 Knative (**kn**) CLI。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。

流程

- 创建频道：

```
$ kn channel create <channel_name> --type <channel_type>
```

频道类型是可选的，但如果指定，则必须使用 **Group:Version:Kind** 格式。例如，您可以创建一个 **InMemoryChannel** 对象：

```
$ kn channel create mychannel --type messaging.knative.dev:v1:InMemoryChannel
```

输出示例

```
Channel 'mychannel' created in namespace 'default'.
```

验证

- 要确认该频道现在存在，请列出现有频道并检查输出：

```
$ kn channel list
```

输**出示例**

```
kn channel list
NAME TYPE URL AGE READY REASON
mychannel InMemoryChannel http://mychannel-kn-channel.default.svc.cluster.local 93s
True
```

删除频道

- 删除频道：

```
$ kn channel delete <channel_name>
```

5.11.3. 使用 YAML 创建默认实现频道

使用 YAML 文件创建 Knative 资源使用声明性 API，它允许您以声明性的方式描述频道，并以可重复的方式描述频道。要使用 YAML 创建无服务器频道，您必须创建一个 YAML 文件来定义 **Channel** 对象，然后使用 **oc apply** 命令应用它。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Eventing。
- 安装 OpenShift CLI (**oc**) 。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。

流程

1. 创建一个 **Channel** 对象作为一个 YAML 文件：

```
apiVersion: messaging.knative.dev/v1
kind: Channel
metadata:
  name: example-channel
  namespace: default
```

2. 应用 YAML 文件：

```
$ oc apply -f <filename>
```

5.11.4. 使用 YAML 创建 Kafka 频道

使用 YAML 文件创建 Knative 资源使用声明性 API，它允许您以声明性的方式描述频道，并以可重复的方式描述频道。您可以通过创建一个 Kafka 频道，创建由 Kafka 主题支持的 Knative Eventing 频道。要使用 YAML 创建 Kafka 频道，您必须创建一个 YAML 文件来定义 **KafkaChannel** 对象，然后使用 **oc apply** 命令应用它。

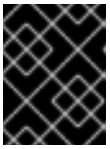
先决条件

- OpenShift Serverless Operator、Knative Eventing 和 **KnativeKafka** 自定义资源已安装在 OpenShift Container Platform 集群中。
- 安装 OpenShift CLI (**oc**) 。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。

流程

1. 创建一个 **KafkaChannel** 对象作为一个 YAML 文件：

```
apiVersion: messaging.knative.dev/v1beta1
kind: KafkaChannel
metadata:
  name: example-channel
  namespace: default
spec:
  numPartitions: 3
  replicationFactor: 1
```



重要

仅支持 OpenShift Serverless 上的 **KafkaChannel** 对象的 **v1beta1** API 版本。不要使用这个 API 的 **v1alpha1** 版本，因为这个版本现已弃用。

2. 应用 **KafkaChannel** YAML 文件：

```
$ oc apply -f <filename>
```

5.11.5. 后续步骤

- 创建频道后，创建一个订阅，允许事件 sink 订阅频道并接收事件。
- 配置事件交付参数，当事件无法发送到事件 sink 时。请参阅配置事件交付参数的示例。

5.12. 创建和管理订阅

创建频道和事件 sink 后，您可以创建一个订阅来启用事件交付。订阅是通过配置 **Subscription** 对象创建的，它指定频道和接收器（也称为 *订阅者*）来发送事件。

5.12.1. 使用 Web 控制台创建订阅

创建频道和事件 sink 后，您可以创建一个订阅来启用事件交付。使用 OpenShift Container Platform Web 控制台提供了一个简化且直观的用户界面来创建订阅。

先决条件

- OpenShift Serverless Operator、Knative Serving 和 Knative Eventing 已在 OpenShift Container Platform 集群中安装。
- 已登陆到 web 控制台。

- 您已创建了事件 sink，如 Knative 服务以及频道。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。

流程

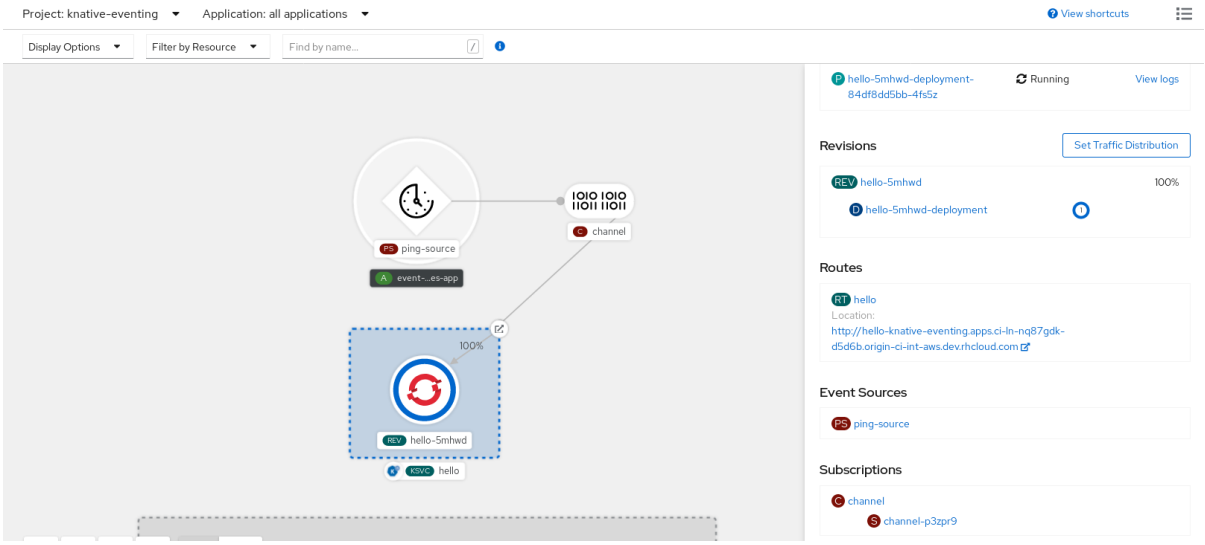
1. 在 **Developer** 视角中，进入 **Topology** 页。
2. 使用以下方法之一创建订阅：
   a. 将鼠标悬停在您要为其创建订阅的频道上，并拖动箭头。此时会显示 **Add Subscription** 选项。

      i. 在 **Subscriber** 列表中选择您的接收器。
      ii. 点 **Add**。

   b. 如果服务在与频道相同的命名空间或项目下的 **Topology** 视图中可用，点击您要为该频道创建订阅的频道，并将箭头直接拖到服务以立即从频道创建订阅到该服务。

验证

- 创建订阅后，您可以在 **Topology** 视图中将频道连接到该服务的行显示为：



5.12.2. 使用 YAML 创建订阅

创建频道和事件 sink 后，您可以创建一个订阅来启用事件交付。使用 YAML 文件创建 Knative 资源使用声明性 API，它允许您以声明性的方式描述订阅，并以可重复的方式描述订阅。要使用 YAML 创建订阅，您必须创建一个 YAML 文件来定义 **Subscription** 对象，然后使用 **oc apply** 命令应用它。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Eventing。
- 安装 OpenShift CLI (**oc**) 。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。

流程

- 创建 **Subscription** 对象：
  - 创建 YAML 文件并将以下示例代码复制到其中：

```
apiVersion: messaging.knative.dev/v1beta1
kind: Subscription
metadata:
  name: my-subscription (1)
  namespace: default
spec:
  channel: (2)
    apiVersion: messaging.knative.dev/v1beta1
    kind: Channel
    name: example-channel
  delivery: (3)
    deadLetterSink:
      ref:
        apiVersion: serving.knative.dev/v1
        kind: Service
        name: error-handler
  subscriber: (4)
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: event-display
```

(1) 订阅的名称。

(2) 订阅连接的频道的配置设置。

(3) 事件交付的配置设置。这会告诉订阅无法发送给订阅者的事件。配置后，消耗的事件会发送到 **deadLetterSink**。事件将被丢弃，不会尝试重新发送该事件，并在系统中记录错误。**deadLetterSink** 的值需要是一个 Destination。

(4) 订阅用户的配置设置。这是事件从频道发送的事件 sink。

  - 应用 YAML 文件：

```
$ oc apply -f <filename>
```

5.12.3. 使用 Knative CLI 创建订阅

创建频道和事件 sink 后，您可以创建一个订阅来启用事件交付。使用 Knative (**kn**) CLI 创建订阅提供了比直接修改 YAML 文件更精简且直观的用户界面。您可以使用带有适当标志的 **kn subscription create** 命令创建订阅。

先决条件

- OpenShift Serverless Operator 和 Knative Eventing 已安装在 OpenShift Container Platform 集群中。
- 已安装 Knative (**kn**) CLI。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。

流程

- 创建订阅以将接收器连接到频道：

```
$ kn subscription create <subscription_name> \
  --channel <group:version:kind>:<channel_name> \ 1
  --sink <sink_prefix>:<sink_name> \ 2
  --sink-dead-letter <sink_prefix>:<sink_name> 3
```

1. **--channel** 指定应处理的云事件的来源。您必须提供频道名称。如果您没有使用由 **Channel** 自定义资源支持的默认 **InMemoryChannel** 频道，您必须为指定频道类型添加 **<group:version:kind>** 前缀。例如：Kafka 支持的频道是 **messaging.knative.dev:v1beta1:KafkaChannel**。

2. **--sink** 指定事件要传送到的目标目的地。默认情况下，**<sink_name>** 解释为此名称的 Knative 服务，与订阅位于同一个命名空间中。您可以使用以下前缀之一指定接收器类型：

   ksvc
   : Knative 服务。

   channel
   : 作为目的地的频道。这里只能引用默认频道类型。

   broker
   : Eventing 代理。

3. 可选：**--sink-dead-letter** 是一个可选标志，可用于指定在无法发送事件时哪些事件应发送到的接收器。如需更多信息，请参阅 OpenShift Serverless *事件交付*文档。

   示例命令

   ```
   $ kn subscription create mysubscription --channel mychannel --sink ksvc:event-display
   ```

   输出示例

   ```
   Subscription 'mysubscription' created in namespace 'default'.
   ```

验证

- 要确认频道已连接到事件接收器或 *subscriber*，使用一个订阅列出现有订阅并检查输出：

```
$ kn subscription list
```

输出示例

```
NAME           CHANNEL           SUBSCRIBER          REPLY   DEAD LETTER SINK
READY   REASON
mysubscription   Channel:mychannel   ksvc:event-display                      True
```

删除订阅

- 删除订阅：

```
$ kn subscription delete <subscription_name>
```

5.12.4. 使用 Knative CLI 描述订阅

您可以使用 **kn subscription describe** 命令在终端中使用 Knative (**kn**) 打印有关订阅的信息。使用 Knative CLI 描述订阅可提供比直接查看 YAML 文件更精简且直观的用户界面。

先决条件

- 已安装 Knative (**kn**) CLI。
- 您已在集群中创建了订阅。

流程

- 描述订阅：

```
$ kn subscription describe <subscription_name>
```

输出示例

```
Name:            my-subscription
Namespace:       default
Annotations:     messaging.knative.dev/creator=openshift-user,
messaging.knative.dev/lastModifier=min ...
Age:             43s
Channel:         Channel:my-channel (messaging.knative.dev/v1)
Subscriber:
  URI:           http://edisplay.default.example.com
Reply:
  Name:          default
  Resource:      Broker (eventing.knative.dev/v1)
DeadLetterSink:
  Name:          my-sink
  Resource:      Service (serving.knative.dev/v1)

Conditions:
  OK TYPE                  AGE REASON
  ++ Ready                 43s
```

```
++ AddedToChannel       43s
++ ChannelReady         43s
++ ReferencesResolved   43s
```

5.12.5. 使用 Knative CLI 列出订阅

您可以使用 **kn subscription list** 命令通过 Knative (**kn**) CLI 列出集群中的现有订阅。使用 Knative CLI 列出订阅提供了精简且直观的用户界面。

先决条件

- 已安装 Knative (**kn**) CLI。

流程

- 列出集群中的订阅：

```
$ kn subscription list
```

输**出示例**

```
NAME           CHANNEL           SUBSCRIBER          REPLY   DEAD LETTER SINK
READY   REASON
mysubscription   Channel:mychannel   ksvc:event-display                      True
```

5.12.6. 使用 Knative CLI 更新订阅

您可以使用 **kn subscription update** 命令以及使用 Knative (**kn**) CLI 从终端更新订阅的适当标志。使用 Knative CLI 更新订阅可提供比直接更新 YAML 文件更精简且直观的用户界面。

先决条件

- 已安装 Knative (**kn**) CLI。
- 您已创建了订阅。

流程

- 更新订阅：

```
$ kn subscription update <subscription_name> \
  --sink <sink_prefix>:<sink_name> \ 1
  --sink-dead-letter <sink_prefix>:<sink_name> 2
```

1 **--sink** 指定要将事件传送到的更新目标目的地。您可以使用以下前缀之一指定接收器类型：

ksvc
: Knative 服务。

channel
: 作为目的地的频道。这里只能引用默认频道类型。

broker
: Eventing 代理。

2 可选： **--sink-dead-letter** 是一个可选标志，可用于指定在无法发送事件时哪些事件应发送到的接收器。如需更多信息，请参阅 OpenShift Serverless *事件交付*文档。

示例命令

```
$ kn subscription update mysubscription --sink ksvc:event-display
```

5.12.7. 后续步骤

- 配置事件交付参数，当事件无法发送到事件 sink 时。请参阅配置事件交付参数的示例。

5.13. 创建代理

Knative 提供基于频道的默认代理实现。这个基于频道的代理可用于开发和测试目的，但不为生产环境提供适当的事件交付保证。

如果集群管理员将 OpenShift Serverless 部署配置为使用 Kafka 作为 default 代理类型，使用默认设置创建代理会创建一个基于 Kafka 的代理。

如果您的 OpenShift Serverless 部署没有配置为使用 Kafka 代理作为 default 代理类型，则按照以下流程中的默认设置时会创建基于频道的代理。

5.13.1. 使用 Knative CLI 创建代理

代理可与触发器结合使用，用于将事件源发送到事件 sink。通过使用 Knative (**kn**) CLI 创建代理，通过直接修改 YAML 文件来提供更简化的、直观的用户界面。您可以使用 **kn broker create** 命令创建代理。

先决条件

- OpenShift Serverless Operator 和 Knative Eventing 已安装在 OpenShift Container Platform 集群中。
- 已安装 Knative (**kn**) CLI。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。

流程

- 创建代理：

```
$ kn broker create <broker_name>
```

验证

1. 使用 **kn** 命令列出所有现有代理：

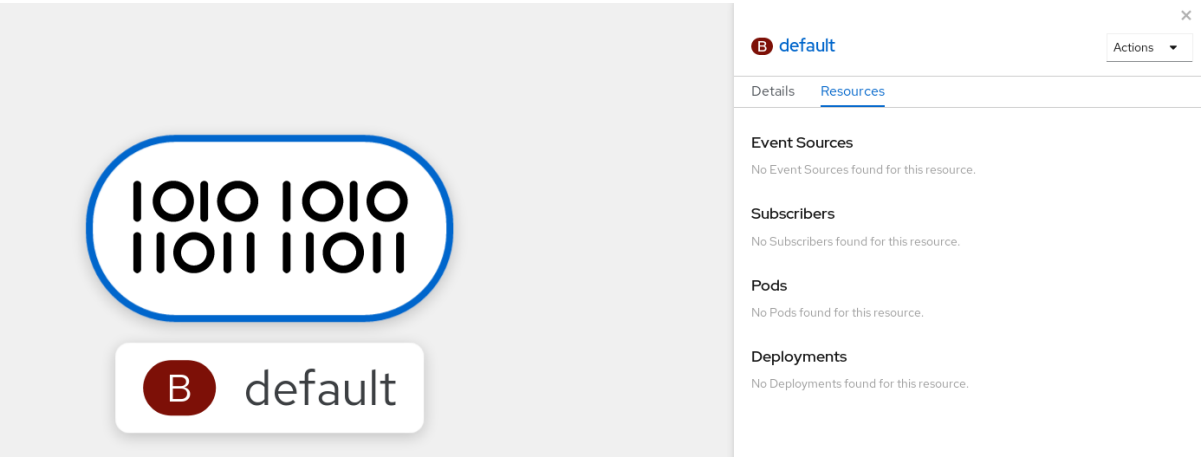
```
$ kn broker list
```

输出示例

```
NAME    URL                                                AGE   CONDITIONS   READY
```

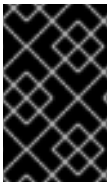
```
REASON
default   http://broker-ingress.knative-eventing.svc.cluster.local/test/default   45s   5 OK / 5
True
```

2. 可选：如果使用 OpenShift Container Platform Web 控制台，在 **Developer** 视角中进入 **Topology** 视图来查看存在的代理：



5.13.2. 通过注解触发器来创建代理

代理可与触发器结合使用，用于将事件源发送到事件 sink。您可以通过将 **eventing.knative.dev/injection: enabled** 注解添加到 **Trigger** 对象来创建代理。



重要

如果您使用 **eventing.knative.dev/injection: enabled** 注解创建代理，则在没有集群管理员权限的情况下无法删除该代理。如果您在集群管理员还没有删除此注解前删除了代理，则代理会在删除后再次被创建。

先决条件

- OpenShift Serverless Operator 和 Knative Eventing 已安装在 OpenShift Container Platform 集群中。
- 安装 OpenShift CLI (**oc**) 。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。

流程

1. 创建一个 **Trigger** 对象作为 YAML 文件，该文件带有 **eventing.knative.dev/injection: enabled** 注解：

```
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  annotations:
    eventing.knative.dev/injection: enabled
  name: <trigger_name>
spec:
  broker: default
  subscriber: 1
```

```
  ref:
    apiVersion: serving.knative.dev/v1
    kind: Service
    name: <service_name>
```

1 指定触发器将事件发送到的事件 sink 或 *subscriber*。

2. 应用 **Trigger** YAML 文件：

```
$ oc apply -f <filename>
```

验证

您可以使用 **oc** CLI，或使用 web 控制台中的 **Topology** 视图来验证代理是否已成功创建。

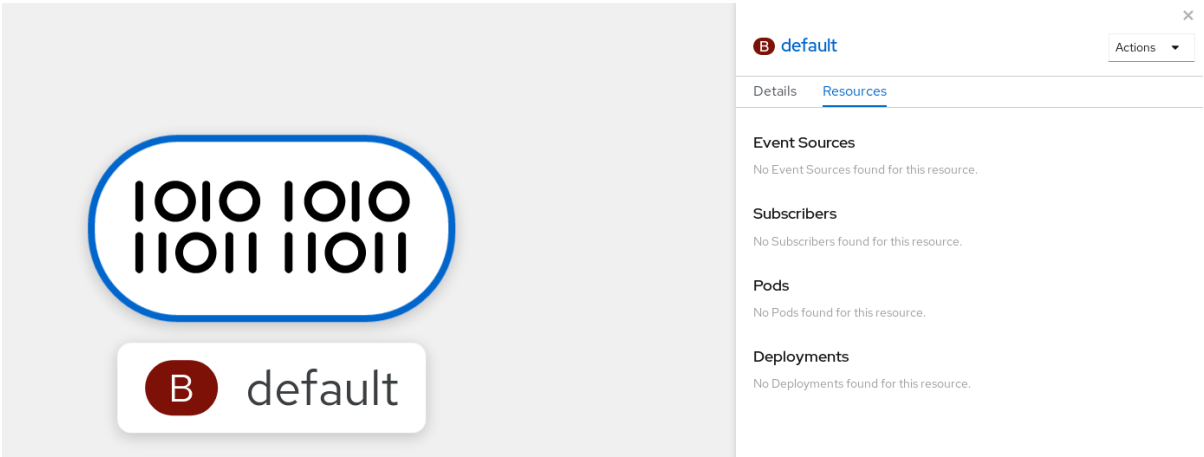
1. 输入以下 **oc** 命令来获取代理：

```
$ oc -n <namespace> get broker default
```

输**出示例**

```
NAME    READY    REASON    URL                                                                    AGE
default   True               http://broker-ingress.knative-eventing.svc.cluster.local/test/default
3m56s
```

2. 可选：如果使用 OpenShift Container Platform Web 控制台，在 **Developer** 视角中进入 **Topology** 视图来查看存在的代理：



5.13.3. 通过标记命名空间来创建代理

代理可与触发器结合使用，用于将事件源发送到事件 sink。您可以通过标记您拥有的命名空间或具有写入权限来自动创建 **default** 代理。

注意

如果您删除该标签，则不会删除使用这个方法创建的代理。您必须手动删除它们。

先决条件

- OpenShift Serverless Operator 和 Knative Eventing 已安装在 OpenShift Container Platform 集群中。
- 安装 OpenShift CLI (**oc**) 。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。

流程

- 使用 **eventing.knative.dev/injection=enabled** 标识一个命名空间：

```
$ oc label namespace <namespace> eventing.knative.dev/injection=enabled
```

验证

您可以使用 **oc** CLI，或使用 web 控制台中的 **Topology** 视图来验证代理是否已成功创建。

1. 使用 **oc** 命令获取代理：

```
$ oc -n <namespace> get broker <broker_name>
```

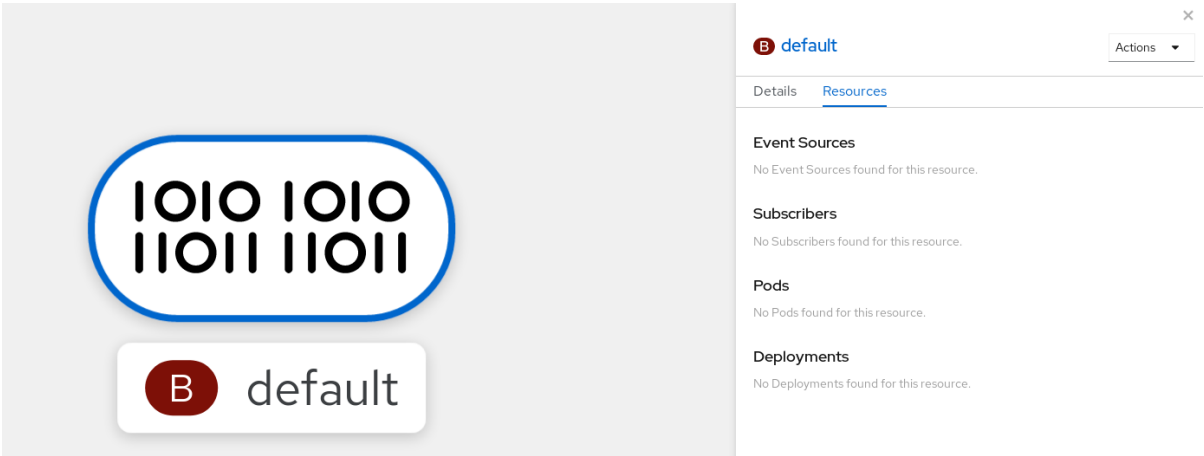
示例命令

```
$ oc -n default get broker default
```

输**出示例**

```
NAME    READY    REASON    URL                                                                    AGE
default   True             http://broker-ingress.knative-eventing.svc.cluster.local/test/default
3m56s
```

2. 可选：如果使用 OpenShift Container Platform Web 控制台，在 **Developer** 视角中进入 **Topology** 视图来查看存在的代理：



5.13.4. 删除通过注入创建的代理

如果通过注入创建了一个代理并在以后需要删除它时，您必须手动删除它。如果删除了标签或注解，则使用命名空间标签或触发器注解创建的代理不会被永久删除。

先决条件

- 安装 OpenShift CLI (**oc**) 。

流程

1. 从命名空间中删除 **eventing.knative.dev/injection=enabled** 标识：

```
$ oc label namespace <namespace> eventing.knative.dev/injection-
```

移除注解可防止 Knative 在删除代理后重新创建代理。

2. 从所选命名空间中删除代理：

```
$ oc -n <namespace> delete broker <broker_name>
```

验证

- 使用 **oc** 命令获取代理：

```
$ oc -n <namespace> get broker <broker_name>
```

示例命令

```
$ oc -n default get broker default
```

输**出示例**

```
No resources found.
Error from server (NotFound): brokers.eventing.knative.dev "default" not found
```

5.13.5. 当配置为 default 代理类型时，创建 Kafka 代理

如果您的 OpenShift Serverless 部署没有配置为使用 Kafka 代理作为默认代理类型，您仍可使用以下步骤创建基于 Kafka 的代理。

5.13.5.1. 使用 YAML 创建 Kafka 代理

使用 YAML 文件创建 Knative 资源使用声明性 API，它允许您以声明性的方式描述应用程序，并以可重复的方式描述应用程序。要使用 YAML 创建 Kafka 代理，您必须创建一个 YAML 文件来定义 **Broker** 对象，然后使用 **oc apply** 命令应用它。

先决条件

- OpenShift Serverless Operator、Knative Eventing 和 **KnativeKafka** 自定义资源已安装在 OpenShift Container Platform 集群中。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。
- 已安装 OpenShift CLI(**oc**)。

流程

1. 创建一个基于 Kafka 的代理作为 YAML 文件：

```
apiVersion: eventing.knative.dev/v1
kind: Broker
metadata:
  annotations:
    eventing.knative.dev/broker.class: Kafka (1)
  name: example-kafka-broker
spec:
  config:
    apiVersion: v1
    kind: ConfigMap
    name: kafka-broker-config (2)
    namespace: knative-eventing
```

(1) 代理类。如果没有指定，代理使用由集群管理员配置的默认类。要使用 Kafka 代理，这个值必须是 **Kafka**。

(2) Knative Kafka 代理的默认配置映射。当集群管理员在集群中启用 Kafka 代理功能时，会创建此配置映射。

2. 应用基于 Kafka 的代理 YAML 文件：

```
$ oc apply -f <filename>
```

5.13.5.2. 创建使用外部管理的 Kafka 主题的 Kafka 代理

如果要在不创建自己的内部主题的情况下使用 Kafka 代理，您可以使用外部管理的 Kafka 主题。要做到这一点，您必须创建一个使用 **kafka.eventing.knative.dev/external.topic** 注解的 Kafka **Broker** 对象。

先决条件

- OpenShift Serverless Operator、Knative Eventing 和 **KnativeKafka** 自定义资源已安装在 OpenShift Container Platform 集群中。
- 您可以访问一个 Kafka 实例，如 Red Hat AMQ Streams，并创建了 Kafka 主题。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。
- 已安装 OpenShift CLI(**oc**)。

流程

1. 创建一个基于 Kafka 的代理作为 YAML 文件：

```
apiVersion: eventing.knative.dev/v1
kind: Broker
metadata:
  annotations:
    eventing.knative.dev/broker.class: Kafka (1)
    kafka.eventing.knative.dev/external.topic: <topic_name> (2)
...
```

1. 代理类。如果没有指定，代理使用由集群管理员配置的默认类。要使用 Kafka 代理，这个值必须是 **Kafka**。
2. 要使用的 Kafka 主题的名称。

2. 应用基于 Kafka 的代理 YAML 文件：

```
$ oc apply -f <filename>
```

5.13.6. 管理代理

Knative (**kn**) CLI 提供了可用于描述和列出现有代理的命令。

5.13.6.1. 使用 Knative CLI 列出现有代理

使用 Knative (**kn**) CLI 列出代理提供了精简且直观的用户界面。您可以使用 **kn broker list** 命令列出集群中的现有代理。

先决条件

- OpenShift Serverless Operator 和 Knative Eventing 已安装在 OpenShift Container Platform 集群中。
- 已安装 Knative (**kn**) CLI。

流程

- 列出所有存在的代理：

```
$ kn broker list
```

输出示例

```
NAME    URL                                                    AGE   CONDITIONS   READY
REASON
default   http://broker-ingress.knative-eventing.svc.cluster.local/test/default   45s   5 OK / 5
True
```

5.13.6.2. 使用 Knative CLI 描述现有代理

使用 Knative (**kn**) 描述代理提供了精简且直观的用户界面。您可以使用 **kn broker describe** 命令通过 Knative CLI 输出集群中现有代理的信息。

先决条件

- OpenShift Serverless Operator 和 Knative Eventing 已安装在 OpenShift Container Platform 集群中。
- 已安装 Knative (**kn**) CLI。

流程

- 描述现有代理：

```
$ kn broker describe <broker_name>
```

使用 default broker 的命令示例

```
$ kn broker describe default
```

输**出示例**

```
Name:         default
Namespace:    default
Annotations:  eventing.knative.dev/broker.class=MTChannelBasedBroker,
eventing.knative.dev/creato ...
Age:          22s

Address:
  URL:    http://broker-ingress.knative-eventing.svc.cluster.local/default/default

Conditions:
  OK TYPE                  AGE REASON
  ++ Ready                 22s
  ++ Addressable           22s
  ++ FilterReady           22s
  ++ IngressReady          22s
  ++ TriggerChannelReady   22s
```

5.13.7. 后续步骤

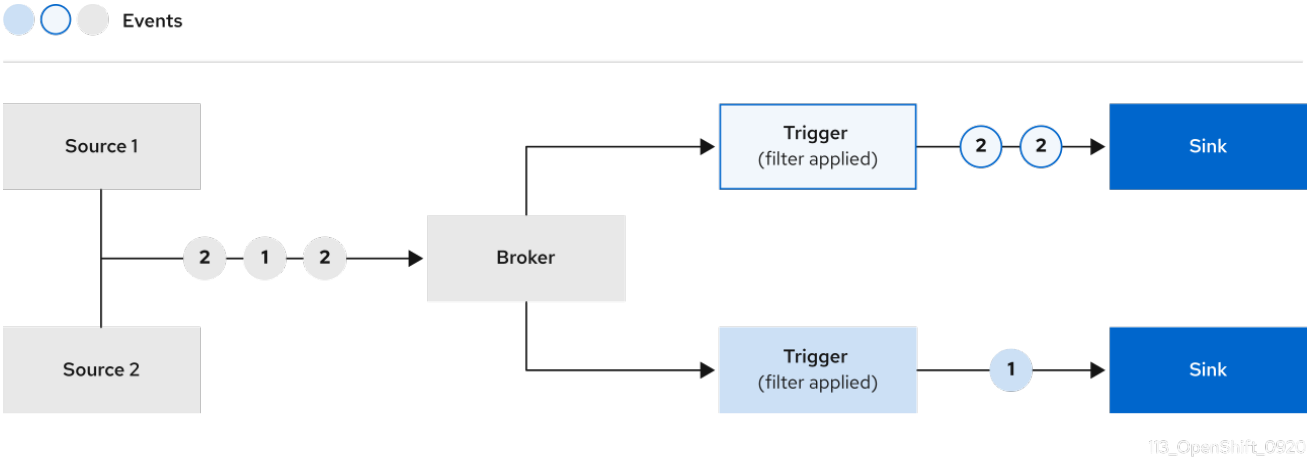
- 配置事件交付参数，当事件无法发送到事件 sink 时。请参阅配置事件交付参数的示例。

5.13.8. 其他资源

- 配置默认代理类
- Triggers Event sources
- 事件交付
- Kafka 代理
- 配置 Knative Kafka

5.14. 触发器

代理可与触发器结合使用，用于将事件源发送到事件 sink。事件从事件源发送到代理，作为 HTTP **POST** 请求。事件进入代理后，可使用触发器根据 CloudEvent 属性 进行过滤，并作为 HTTP **POST** 请求发送到事件 sink。



如果使用 Kafka 代理，您可以将事件的交付顺序从触发器配置为事件 sink。请参阅为触发器配置事件交付顺序。

5.14.1. 使用 Web 控制台创建触发器

使用 OpenShift Container Platform Web 控制台提供了一个简化且直观的用户界面来创建触发器。在集群中安装 Knative Eventing 并创建了代理后，您可以使用 web 控制台创建触发器。

先决条件

- OpenShift Serverless Operator、Knative Serving 和 Knative Eventing 已在 OpenShift Container Platform 集群中安装。
- 已登陆到 web 控制台。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。
- 您已创建了代理和 Knative 服务或其他事件 sink 以连接触发器。

流程

1. 在 **Developer** 视角中，进入 **Topology** 页。
2. 将鼠标悬停在您要创建触发器的代理上，并拖动箭头。此时会显示 **Add Trigger** 选项。
3. 点 **Add Trigger**。
4. 在 **Subscriber** 列表中选择您的接收器。
5. 点 **Add**。

验证

- 创建订阅后，您可以在 **Topology** 页面中查看它，其中它是一个将代理连接到事件 sink 的行。

删除触发器

1. 在 **Developer** 视角中，进入 **Topology** 页。
2. 点您要删除的触发器。

3. 在 **Actions** 上下文菜单中，选择 **Delete Trigger**。

5.14.2. 使用 Knative CLI 创建触发器

使用 Knative (**kn**) CLI 创建触发器通过直接修改 YAML 文件，提供更精简且直观的用户界面。您可以使用 **kn trigger create** 命令创建触发器。

先决条件

- OpenShift Serverless Operator 和 Knative Eventing 已安装在 OpenShift Container Platform 集群中。
- 已安装 Knative (**kn**) CLI。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。

流程

- 创建触发器：

```
$ kn trigger create <trigger_name> --broker <broker_name> --filter <key=value> --sink
<sink_name>
```

或者，您可以创建触发器并使用代理注入同时创建 **default** 代理：

```
$ kn trigger create <trigger_name> --inject-broker --filter <key=value> --sink <sink_name>
```

默认情况下，触发器会将发送到代理的所有事件转发到订阅到该代理的 sink。通过对触发器使用 **--filter** 属性，您可以从代理过滤事件，这样订阅者才会根据您定义的标准接收一小部分事件。

5.14.3. 使用 Knative CLI 列出触发器

使用 Knative (**kn**) CLI 列出触发器提供精简、直观的用户界面。您可以使用 **kn trigger list** 命令列出集群中的现有触发器。

先决条件

- OpenShift Serverless Operator 和 Knative Eventing 已安装在 OpenShift Container Platform 集群中。
- 已安装 Knative (**kn**) CLI。

流程

1. 显示可用触发器列表：

```
$ kn trigger list
```

输**出示例**

```
NAME    BROKER    SINK          AGE   CONDITIONS   READY   REASON
email   default   ksvc:edisplay   4s    5 OK / 5     True
ping    default   ksvc:edisplay   32s   5 OK / 5     True
```

2. 可选：以 JSON 格式输出触发器列表：

```
$ kn trigger list -o json
```

5.14.4. 使用 Knative CLI 描述触发器

使用 Knative (**kn**) CLI 描述触发器，提供了一个简化且直观的用户界面。您可以通过 **kn trigger describe** 命令使用 Knative CLI 输出集群中现有触发器的信息。

先决条件

- OpenShift Serverless Operator 和 Knative Eventing 已安装在 OpenShift Container Platform 集群中。
- 已安装 Knative (**kn**) CLI。
- 您已创建了触发器。

流程

- 输入命令：

```
$ kn trigger describe <trigger_name>
```

输**出示例**

```
Name:         ping
Namespace:    default
Labels:       eventing.knative.dev/broker=default
Annotations:  eventing.knative.dev/creator=kube:admin,
eventing.knative.dev/lastModifier=kube:admin
Age:          2m
Broker:       default
Filter:
  type:       dev.knative.event

Sink:
  Name:       edisplay
  Namespace:  default
  Resource:   Service (serving.knative.dev/v1)

Conditions:
  OK TYPE                 AGE REASON
  ++ Ready                2m
  ++ BrokerReady          2m
  ++ DependencyReady      2m
  ++ Subscribed           2m
  ++ SubscriberResolved   2m
```

5.14.5. 使用 Knative CLI 使用触发器过滤事件

使用 Knative (**kn**) CLI 使用 kn CLI 通过触发器过滤事件，可提供精简且直观的用户界面。您可以使用 **kn trigger create** 命令和适当的标记来通过使用触发器过滤事件。

在以下触发器示例中，只有带有属性 **type: dev.knative.samples.helloworld** 的事件才会发送到事件 sink:

```
$ kn trigger create <trigger_name> --broker <broker_name> --filter
type=dev.knative.samples.helloworld --sink ksvc:<service_name>
```

您还可以使用多个属性过滤事件。以下示例演示了如何使用类型、源和扩展属性过滤事件：

```
$ kn trigger create <trigger_name> --broker <broker_name> --sink ksvc:<service_name> \
--filter type=dev.knative.samples.helloworld \
--filter source=dev.knative.samples/helloworldsource \
--filter myextension=my-extension-value
```

5.14.6. 使用 Knative CLI 更新触发器

使用 Knative (**kn**) CLI 更新触发器提供精简、直观的用户界面。您可以使用带有特定标志的 **kn trigger update** 命令来更新触发器的属性。

先决条件

- OpenShift Serverless Operator 和 Knative Eventing 已安装在 OpenShift Container Platform 集群中。
- 已安装 Knative (**kn**) CLI。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。

流程

- 更新触发器：

```
$ kn trigger update <trigger_name> --filter <key=value> --sink <sink_name> [flags]
```

  - 您可以更新触发器来过滤与传入事件匹配的事件属性。例如，使用 **type** 属性：

```
$ kn trigger update <trigger_name> --filter type=knative.dev.event
```

  - 您可以从触发器中删除过滤器属性。例如，您可以使用键 **type** 来删除过滤器属性：

```
$ kn trigger update <trigger_name> --filter type-
```

  - 您可以使用 **--sink** 参数来更改触发器的事件 sink:

```
$ kn trigger update <trigger_name> --sink ksvc:my-event-sink
```

5.14.7. 使用 Knative CLI 删除触发器

使用 Knative (**kn**) CLI 删除触发器提供精简而直观的用户界面。您可以使用 **kn trigger delete** 命令删除触发器。

先决条件

- OpenShift Serverless Operator 和 Knative Eventing 已安装在 OpenShift Container Platform 集群中。
- 已安装 Knative (**kn**) CLI。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。

流程

- 删除触发器：

```
$ kn trigger delete <trigger_name>
```

验证

1. 列出现有触发器：

```
$ kn trigger list
```

2. 验证触发器不再存在：

   输**出示例**

```
No triggers found.
```

5.14.8. 为触发器配置事件交付顺序

如果使用 Kafka 代理，您可以将事件的交付顺序从触发器配置为事件 sink。

先决条件

- OpenShift Serverless Operator、Knative Eventing 和 Knative Kafka 安装在 OpenShift Container Platform 集群中。
- Kafka 代理被启用在集群中使用，您也创建了一个 Kafka 代理。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。
- 已安装 OpenShift (**oc**) CLI。

流程

1. 创建或修改 **Trigger** 对象并设置 **kafka.eventing.knative.dev/delivery.order** 注解：

```
apiVersion: eventing.knative.dev/v1
kind: Trigger
```

```
metadata:
  name: <trigger_name>
  annotations:
    kafka.eventing.knative.dev/delivery.order: ordered
...
```

支持的消费者交付保证有：

unordered

未排序的消费者是一种非阻塞消费者，它能以未排序的方式提供消息，同时保持正确的偏移管理。

排序的

一个订购的消费者是一个按分区阻止消费者，在提供分区的下一个消息前等待来自 CloudEvent 订阅者成功响应。
默认排序保证是 **unordered**。

2. 应用 **Trigger** 对象：

```
$ oc apply -f <filename>
```

5.14.9. 后续步骤

- 配置事件交付参数，当事件无法发送到事件 sink 时。请参阅配置事件交付参数的示例。

5.15. 使用 KNATIVE KAFKA

Knative Kafka 提供集成选项，供您在 OpenShift Serverless 中使用支持的 Apache Kafka 消息流平台。Kafka 为事件源、频道、代理和事件 sink 功能提供选项。

如果集群管理员安装了 **KnativeKafka** 自定义资源，则 OpenShift Serverless 安装中就会提供 Knative Kafka 功能。

注意

IBM Z 和 IBM Power Systems 目前不支持 Knative Kafka。

Knative Kafka 提供了额外的选项，例如：

- Kafka 源
- Kafka 频道
- Kafka 代理
- Kafka 接收器

5.15.1. Kafka 事件交付和重试

在事件驱动的架构中使用 Kafka 组件会提供"至少一次"事件交付。这意味着，会在收到返回代码值前重试操作。这使您的应用对丢失的事件更具弹性，但可能会导致发送重复的事件。

对于 Kafka 事件源，默认会尝试发送事件的固定次数。对于 Kafka 频道，只有在 Kafka 频道 **Delivery** 规格中配置了它们时才会进行重试。

有关交付保证的更多信息，请参阅 事件交付 文档。

5.15.2. Kafka 源

您可以创建一个 Kafka 源从 Apache Kafka 集群中读取事件，并将这些事件传递给接收器。您可以使用 OpenShift Container Platform web 控制台、Knative (**kn**) CLI 或直接创建 **KafkaSource** 对象并使用 OpenShift CLI (**oc**) 创建 Kafka 源来应用它。

5.15.2.1. 使用 Web 控制台创建 Kafka 事件源

在集群中安装了 Knative Kafka 后，您可以使用 Web 控制台创建 Kafka 源。使用 OpenShift Container Platform Web 控制台提供了一个简化的用户界面来创建 Kafka 源。

先决条件

- OpenShift Serverless Operator、Knative Eventing 和 **KnativeKafka** 自定义资源已安装在集群中。
- 已登陆到 web 控制台。
- 您可以访问 Red Hat AMQ Streams（Kafka）集群，该集群会生成您要导入的 Kafka 信息。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。

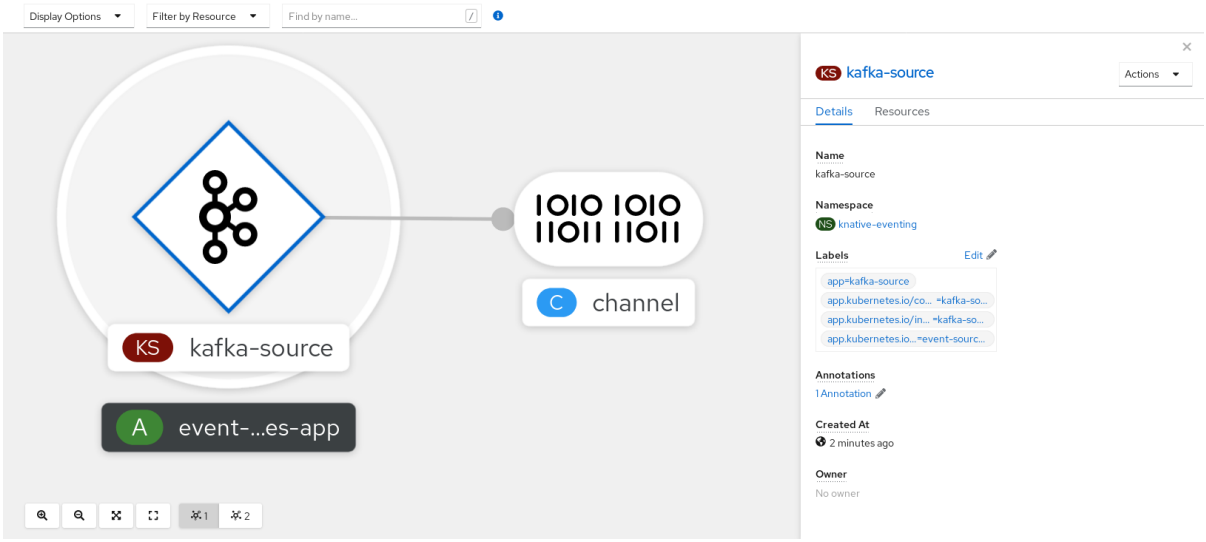
流程

1. 在 **Developer** 视角中，导航到 **Add** 页面并选择 **Event Source**。
2. 在 **Event Sources** 页面中，在 **Type** 部分选择 **Kafka Source**。
3. 配置 **Kafka Source** 设置：
   a. 添加用逗号分开的 **Bootstrap 服务器**列表。
   b. 添加以逗号分隔的**标题**列表。
   c. 添加一个 **消费者组**。
   d. 为您创建的服务帐户选择 **Service Account Name**。
   e. 为事件源选择 **Sink**。**Sink** 可以是一个 **资源**，如频道、代理或服务，也可以是一个 **URI**。
   f. 输入 Kafka 事件源的 **名称**。
4. 点 **Create**。

验证

您可以通过查看 **Topology** 页面来验证 Kafka 事件源是否已创建并连接到接收器。

1. 在 **Developer** 视角中，导航到 **Topology**。
2. 查看 Kafka 事件源和接收器。



5.15.2.2. 使用 Knative CLI 创建 Kafka 事件源

您可以使用 **kn source kafka create** 命令，使用 Knative (**kn**) CLI 创建 Kafka 源。使用 Knative CLI 创建事件源提供了比直接修改 YAML 文件更精简且直观的用户界面。

先决条件

- OpenShift Serverless Operator、Knative Eventing、Knative Serving 和 **KnativeKafka** 自定义资源（CR）已安装在集群中。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。
- 您可以访问 Red Hat AMQ Streams（Kafka）集群，该集群会生成您要导入的 Kafka 信息。
- 已安装 Knative (**kn**) CLI。
- 可选：如果您想要使用此流程中的验证步骤，已安装 OpenShift CLI (**oc**)。

流程

1. 要验证 Kafka 事件源是否可以工作，请创建一个 Knative 服务，在服务日志中转储传入的事件：

```
$ kn service create event-display \
    --image quay.io/openshift-knative/knative-eventing-sources-event-display
```

2. 创建 **KafkaSource** CR：

```
$ kn source kafka create <kafka_source_name> \
    --servers <cluster_kafka_bootstrap>.kafka.svc:9092 \
    --topics <topic_name> --consumergroup my-consumer-group \
    --sink event-display
```

注意

将此命令中的占位符值替换为源名称、引导服务器和主题的值。

--servers、**--topics** 和 **--consumergroup** 选项指定到 Kafka 集群的连接参数。**--consumergroup** 选项是可选的。

3. 可选：查看您创建的 **KafkaSource** CR 的详情：

```
$ kn source kafka describe <kafka_source_name>
```

输出示例

```
Name:              example-kafka-source
Namespace:         kafka
Age:               1h
BootstrapServers:  example-cluster-kafka-bootstrap.kafka.svc:9092
Topics:            example-topic
ConsumerGroup:     example-consumer-group

Sink:
  Name:       event-display
  Namespace:  default
  Resource:   Service (serving.knative.dev/v1)

Conditions:
  OK TYPE           AGE REASON
  ++ Ready          1h
  ++ Deployed       1h
  ++ SinkProvided   1h
```

验证步骤

1. 触发 Kafka 实例将信息发送到主题：

```
$ oc -n kafka run kafka-producer \
    -ti --image=quay.io/strimzi/kafka:latest-kafka-2.7.0 --rm=true \
    --restart=Never -- bin/kafka-console-producer.sh \
    --broker-list <cluster_kafka_bootstrap>:9092 --topic my-topic
```

在提示符后输入信息。这个命令假设：

- Kafka 集群安装在 **kafka** 命名空间中。
- **KafkaSource** 对象已被配置为使用 **my-topic** 主题。

2. 通过查看日志来验证消息是否显示：

```
$ oc logs $(oc get pod -o name | grep event-display) -c user-container
```

输出示例

```
☁  cloudevents.Event
Validation: valid
Context Attributes,
  specversion: 1.0
  type: dev.knative.kafka.event
  source: /apis/v1/namespaces/default/kafkasources/example-kafka-source#example-topic
```

```
  subject: partition:46#0
  id: partition:46/offset:0
  time: 2021-03-10T11:21:49.4Z
Extensions,
  traceparent: 00-161ff3815727d8755848ec01c866d1cd-7ff3916c44334678-00
Data,
  Hello!
```

5.15.2.2.1. Knative CLI sink 标记

当使用 Knative (**kn**) CLI 创建事件源时，您可以使用 **--sink** 标志指定事件从该资源发送到的接收器。sink 可以是任何可寻址或可调用的资源，可以从其他资源接收传入的事件。

以下示例创建使用服务 **http://event-display.svc.cluster.local** 的接收器绑定作为接收器：

使用 sink 标记的命令示例

```
$ kn source binding create bind-heartbeat \
  --namespace sinkbinding-example \
  --subject "Job:batch/v1:app=heartbeat-cron" \
  --sink http://event-display.svc.cluster.local \ 1
  --ce-override "sink=bound"
```

1 **http://event-display.svc.cluster.local** 中的 **svc** 确定接收器是一个 Knative 服务。其他默认的接收器前缀包括 **channel** 和 **broker**。

5.15.2.3. 使用 YAML 创建 Kafka 事件源

使用 YAML 文件创建 Knative 资源使用声明性 API，它允许您以声明性的方式描述应用程序，并以可重复的方式描述应用程序。要使用 YAML 创建 Kafka 源，您必须创建一个 YAML 文件来定义 **KafkaSource** 对象，然后使用 **oc apply** 命令应用它。

先决条件

- OpenShift Serverless Operator、Knative Eventing 和 **KnativeKafka** 自定义资源已安装在集群中。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。
- 您可以访问 Red Hat AMQ Streams（Kafka）集群，该集群会生成您要导入的 Kafka 信息。
- 安装 OpenShift CLI (**oc**) 。

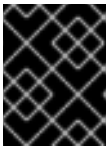
流程

1. 创建 **KafkaSource** 对象作为 YAML 文件：

```
apiVersion: sources.knative.dev/v1beta1
kind: KafkaSource
metadata:
  name: <source_name>
spec:
```

```
consumerGroup: <group_name> 1
bootstrapServers:
- <list_of_bootstrap_servers>
topics:
- <list_of_topics> 2
sink:
- <list_of_sinks> 3
```

1. 用户组是一组使用相同组群 ID 的用户，并消耗一个标题中的数据。
2. 主题提供数据存储的目的地。每个主题都被分成一个或多个分区。
3. sink 指定事件从源发送到的位置。



重要

仅支持 OpenShift Serverless 上的 **KafkaSource** 对象的 **v1beta1** API 版本。不要使用这个 API 的 **v1alpha1** 版本，因为这个版本现已弃用。

KafkaSource 对象示例

```
apiVersion: sources.knative.dev/v1beta1
kind: KafkaSource
metadata:
  name: kafka-source
spec:
  consumerGroup: knative-group
  bootstrapServers:
  - my-cluster-kafka-bootstrap.kafka:9092
  topics:
  - knative-demo-topic
  sink:
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: event-display
```

2. 应用 **KafkaSource** YAML 文件：

```
$ oc apply -f <filename>
```

验证

- 输入以下命令验证 Kafka 事件源是否已创建：

```
$ oc get pods
```

输出示例

```
NAME                                 READY   STATUS    RESTARTS   AGE
kafkasource-kafka-source-5ca0248f-...   1/1     Running   0          13m
```

5.15.3. Kafka 代理

对于生产环境就绪的 Knative Eventing 部署，红帽建议您使用 Knative Kafka 代理实现。Kafka 代理是 Knative 代理的 Apache Kafka 原生实现，它将 CloudEvents 直接发送到 Kafka 实例。

重要

Kafka 代理禁用联邦信息处理标准 (FIPS) 模式。

Kafka 代理具有与 Kafka 的原生集成，用于存储和路由事件。它可以更好地与 Kafka 集成用于代理，并在其他代理类型中触发模型，并减少网络跃点。Kafka 代理实现的其他优点包括：

- 最少一次的交付保证
- 根据 CloudEvents 分区扩展排序事件交付
- 数据平面的高可用性
- 水平扩展数据平面

Knative Kafka 代理使用二进制内容模式将传入的 CloudEvents 存储为 Kafka 记录。这意味着，所有 CloudEvent 属性和扩展都会在 Kafka 记录上映射，而 CloudEvent 的 **data** 规格与 Kafka 记录的值对应。

有关使用 Kafka 代理的详情，请参考创建代理。

5.15.4. 使用 YAML 创建 Kafka 频道

使用 YAML 文件创建 Knative 资源使用声明性 API，它允许您以声明性的方式描述频道，并以可重复的方式描述频道。您可以通过创建一个 Kafka 频道，创建由 Kafka 主题支持的 Knative Eventing 频道。要使用 YAML 创建 Kafka 频道，您必须创建一个 YAML 文件来定义 **KafkaChannel** 对象，然后使用 **oc apply** 命令应用它。

先决条件

- OpenShift Serverless Operator、Knative Eventing 和 **KnativeKafka** 自定义资源已安装在 OpenShift Container Platform 集群中。
- 安装 OpenShift CLI (**oc**) 。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。

流程

1. 创建一个 **KafkaChannel** 对象作为一个 YAML 文件：

```
apiVersion: messaging.knative.dev/v1beta1
kind: KafkaChannel
metadata:
  name: example-channel
  namespace: default
spec:
  numPartitions: 3
  replicationFactor: 1
```

重要

仅支持 OpenShift Serverless 上的 **KafkaChannel** 对象的 **v1beta1** API 版本。不要使用这个 API 的 **v1alpha1** 版本，因为这个版本现已弃用。

2. 应用 **KafkaChannel** YAML 文件：

```
$ oc apply -f <filename>
```

5.15.5. Kafka 接收器

如果集群管理员在集群中启用了 Kafka，则 Kafka sink 是事件 sink 类型。您可以使用 Kafka sink 将事件从事件源发送到 Kafka 主题。

5.15.5.1. 使用 Kafka sink

您可以创建一个称为 Kafka sink 的事件 sink，用于将事件发送到 Kafka 主题。使用 YAML 文件创建 Knative 资源使用声明性 API，它允许您以声明性的方式描述应用程序，并以可重复的方式描述应用程序。要使用 YAML 创建 Kafka sink，您必须创建一个 YAML 文件来定义 **KafkaSink** 对象，然后使用 **oc apply** 命令应用它。

先决条件

- 在集群中安装了 OpenShift Serverless Operator、Knative Eventing 和 **KnativeKafka** 自定义资源 (CR) 。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。
- 您可以访问 Red Hat AMQ Streams（Kafka）集群，该集群会生成您要导入的 Kafka 信息。
- 安装 OpenShift CLI (**oc**) 。

流程

1. 创建一个 **KafkaSink** 对象定义作为一个 YAML 文件：

Kafka sink YAML

```
apiVersion: eventing.knative.dev/v1alpha1
kind: KafkaSink
metadata:
  name: <sink-name>
  namespace: <namespace>
spec:
  topic: <topic-name>
  bootstrapServers:
   - <bootstrap-server>
```

2. 要创建 Kafka sink，请应用 **KafkaSink** YAML 文件：

```
$ oc apply -f <filename>
```

3. 配置事件源，以便在其 spec 中指定 sink:

连接到 API 服务器源的 Kafka sink 示例

```
apiVersion: sources.knative.dev/v1alpha2
kind: ApiServerSource
metadata:
  name: <source-name> 1
  namespace: <namespace> 2
spec:
  serviceAccountName: <service-account-name> 3
  mode: Resource
  resources:
  - apiVersion: v1
    kind: Event
  sink:
    ref:
      apiVersion: eventing.knative.dev/v1alpha1
      kind: KafkaSink
      name: <sink-name> 4
```

1. 事件源的名称。
2. 事件源的命名空间。
3. 事件源的服务帐户。
4. Kafka sink 名称。

5.15.6. 其他资源

- Red Hat AMQ Streams 文档
- Kafka 文档中的 Red Hat AMQ Streams TLS 和 SASL
- 事件交付
- Knative Kafka 集群管理员文档

第 6 章 管理

6.1. 全局配置

OpenShift Serverless Operator 管理 Knative 安装的全局配置，包括将 **KnativeServing** 和 **KnativeEventing** 自定义资源的值传播到系统配置映射。任何手动应用的配置映射更新都会被 Operator 覆盖。但是，通过修改 Knative 自定义资源，您可以为这些配置映射设置值。

Knative 具有多个配置映射，它们使用前缀 **config-** 命名。所有 Knative 配置映射都与它们应用到的自定义资源在同一命名空间中创建。例如，如果在 **knative-serving** 命名空间中创建 **KnativeServing** 自定义资源，则也会在此命名空间中创建所有 Knative Serving 配置映射。

Knative 自定义资源中的 **spec.config** 为每个配置映射有一个 **<name>** 条目，名为 **config-<name>**，其值为配置映射的 **data**。

6.1.1. 配置默认频道实施

default-ch-webhook 配置映射可以用来指定 Knative Eventing 的默认频道实现。您可以为整个集群或一个或多个命名空间指定默认频道实现。目前支持 **InMemoryChannel** 和 **KafkaChannel** 频道类型。

先决条件

- 在 OpenShift Container Platform 上具有管理员权限。
- 在集群中安装了 OpenShift Serverless Operator 和 Knative Eventing。
- 如果要使用 Kafka 频道作为默认频道实现，还必须在集群中安装 **KnativeKafka** CR。

流程

- 修改 **KnativeEventing** 自定义资源，以添加 **default-ch-webhook** 配置映射的配置详情：

```
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeEventing
metadata:
  name: knative-eventing
  namespace: knative-eventing
spec:
  config: (1)
    default-ch-webhook: (2)
      default-ch-config: |
        clusterDefault: (3)
          apiVersion: messaging.knative.dev/v1
          kind: InMemoryChannel
          spec:
            delivery:
              backoffDelay: PT0.5S
              backoffPolicy: exponential
              retry: 5
        namespaceDefaults: (4)
          my-namespace:
            apiVersion: messaging.knative.dev/v1beta1
            kind: KafkaChannel
```

```
    spec:
      numPartitions: 1
      replicationFactor: 1
```

1. 在 **spec.config** 中，您可以指定您要为修改的配置添加的配置映射。
2. **default-ch-webhook** 配置映射可以用来指定集群的默认频道实施，也可以用于一个或多个命名空间。
3. 集群范围的默认频道类型配置。在本例中，集群的默认频道实现是 **InMemoryChannel**。
4. 命名空间范围的默认频道类型配置。在本例中, **my-namespace** 命名空间的默认频道实现是 **KafkaChannel**。

重要

配置特定于命名空间的默认设置会覆盖任何集群范围的设置。

6.1.2. 配置默认代理支持频道

如果您使用基于频道的代理，您可以将代理的默认后备频道类型设置为 **InMemoryChannel** 或 **KafkaChannel**。

先决条件

- 在 OpenShift Container Platform 上具有管理员权限。
- 在集群中安装了 OpenShift Serverless Operator 和 Knative Eventing。
- 已安装 OpenShift (**oc**) CLI。
- 如果要使用 Kafka 频道作为默认后备频道类型，还必须在集群中安装 **KnativeKafka** CR。

流程

1. 修改 **KnativeEventing** 自定义资源 (CR)以添加 **config-br-default-channel** 配置映射的配置详情：

```
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeEventing
metadata:
  name: knative-eventing
  namespace: knative-eventing
spec:
  config: 1
    config-br-default-channel:
      channel-template-spec: |
        apiVersion: messaging.knative.dev/v1beta1
        kind: KafkaChannel 2
        spec:
          numPartitions: 6 3
          replicationFactor: 3 4
```

1. 在 **spec.config** 中，您可以指定您要为修改的配置添加的配置映射。
2. 默认后备频道类型配置。在本例中，集群的默认频道实现是 **KafkaChannel**。
3. 支持代理的 Kafka 频道的分区数量。
4. 支持代理的 Kafka 频道的复制因素。

2. 应用更新的 **KnativeEventing** CR：

```
$ oc apply -f <filename>
```

6.1.3. 配置默认代理类

您可以使用 **config-br-defaults** 配置映射来指定 Knative Eventing 的默认代理类设置。您可以为整个集群或一个或多个命名空间指定默认代理类。目前，支持 **MTChannelBasedBroker** 和 **Kafka** 代理类型。

先决条件

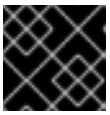
- 在 OpenShift Container Platform 上具有管理员权限。
- 在集群中安装了 OpenShift Serverless Operator 和 Knative Eventing。
- 如果要使用 Kafka 代理作为默认代理实现，还必须在集群中安装 **KnativeKafka** CR。

流程

- 修改 **KnativeEventing** 自定义资源，以添加 **config-br-defaults** 配置映射的配置详情：

```
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeEventing
metadata:
  name: knative-eventing
  namespace: knative-eventing
spec:
  defaultBrokerClass: Kafka (1)
  config: (2)
    config-br-defaults: (3)
      default-br-config: |
        clusterDefault: (4)
          brokerClass: Kafka
          apiVersion: v1
          kind: ConfigMap
          name: kafka-broker-config (5)
          namespace: knative-eventing (6)
        namespaceDefaults: (7)
          my-namespace:
            brokerClass: MTChannelBasedBroker
            apiVersion: v1
            kind: ConfigMap
            name: config-br-default-channel (8)
            namespace: knative-eventing (9)
...
```

1. Knative Eventing 的默认代理类。
2. 在 **spec.config** 中，您可以指定您要为修改的配置添加的配置映射。
3. **config-br-defaults** 配置映射指定任何没有指定 **spec.config** 设置或代理类的代理的默认设置。
4. 集群范围的默认代理类配置。在本例中，集群的默认代理类实现是 **Kafka**。
5. **kafka-broker-config** 配置映射指定 Kafka 代理的默认设置。请参阅 "Additional resources" 部分的"配置 Kafka 代理设置"。
6. 存在 **kafka-broker-config** 配置映射的命名空间。
7. 命名空间范围的默认代理类配置。在本例中，**my-namespace** 命名空间的默认代理类实现是 **MTChannelbasedBroker**。您可以为多个命名空间指定默认代理类实现。
8. **config-br-default-channel** 配置映射指定代理的默认后备频道。请参阅"Additional resources"部分的"配置默认代理支持频道"部分。
9. **config-br-default-channel** 配置映射所在的命名空间。



重要

配置特定于命名空间的默认设置会覆盖任何集群范围的设置。

其他资源

- 配置 Kafka 代理设置
- 配置默认代理支持频道

6.1.4. 启用 scale-to-zero

Knative Serving 为应用程序提供自动扩展功能（或 *autoscaling*），以满足传入的需求。您可以使用 **enable-scale-to-zero** spec，为集群中的应用程序全局启用或禁用 scale-to-zero。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 有集群管理员权限。
- 使用默认的 Knative Pod Autoscaler。如果使用 Kubernetes Horizontal Pod Autoscaler，则缩减为零功能将不可用。

流程

- 在 **KnativeServing** 自定义资源 (CR) 中修改 **enable-scale-to-zero** spec：

  KnativeServing CR 示例

```
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeServing
```

```
metadata:
  name: knative-serving
spec:
  config:
    autoscaler:
      enable-scale-to-zero: "false" 1
```

1 **enable-scale-to-zero** spec 可以是 **"true"** 或 **"false"**。如果设置为 true，则会启用 scale-to-zero。如果设置为 false，应用程序将缩减至配置的*最小扩展绑定*。默认值为 **"true"**。

6.1.5. 配置 scale-to-zero 宽限期

Knative Serving 为应用程序提供自动缩放为零个 pod。您可以使用 **scale-to-zero-grace-period** spec 定义上限，Knative 在删除应用程序的最后一个副本前等待 scale-to-zero machinery 原位。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 有集群管理员权限。
- 使用默认的 Knative Pod Autoscaler。如果使用 Kubernetes Horizontal Pod Autoscaler，则缩减为零功能将不可用。

流程

- 在 **KnativeServing** 自定义资源 (CR) 中修改 **scale-to-zero-grace-period** spec：

  KnativeServing CR 示例

```
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeServing
metadata:
  name: knative-serving
spec:
  config:
    autoscaler:
      scale-to-zero-grace-period: "30s" 1
```

1 宽限期（以秒为单位）。默认值为 30 秒。

6.1.6. 覆盖系统部署配置

您可以通过修改 **KnativeServing** 和 **KnativeEventing** 自定义资源 (CR) 中的 **deployment** spec 来覆盖某些特定部署的默认配置。

6.1.6.1. 覆盖 Knative Serving 系统部署配置

您可以通过修改 **KnativeServing** 自定义资源 (CR) 中的 **deployments** spec 来覆盖某些特定部署的默认配置。目前，支持覆盖默认配置的字段包括 **resources**, **replicas**, **labels**, **annotations**, 和 **nodeSelector**。

在以下示例中，**KnativeServing** CR 会覆盖 **Webhook** 部署，以便：

- 部署指定了 CPU 和内存资源限制。
- 部署有 3 个副本。
- 添加 **example-label: label** 标签。
- 添加 **example-annotation:** 注解。
- **nodeSelector** 字段被设置为选择带有 **disktype: hdd** 标签的节点。

注意

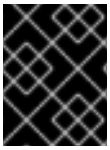
KnativeServing CR 标签和注解设置覆盖部署本身和生成的 Pod 的部署标签和注解。

KnativeServing CR 示例

```
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeServing
metadata:
  name: ks
  namespace: knative-serving
spec:
  high-availability:
    replicas: 2
  deployments:
  - name: webhook
    resources:
    - container: webhook
      requests:
        cpu: 300m
        memory: 60Mi
      limits:
        cpu: 1000m
        memory: 1000Mi
    replicas: 3
    labels:
      example-label: label
    annotations:
      example-annotation: annotation
    nodeSelector:
      disktype: hdd
```

6.1.6.2. 覆盖 Knative Eventing 系统部署配置

您可以通过修改 **KnativeEventing** 自定义资源 (CR) 中的 **deployments** spec 来覆盖某些特定部署的默认配置。目前，对于 **eventing-controller**、**eventing-webhook** 和 **imc-controller** 字段支持覆盖默认配置设置。



重要

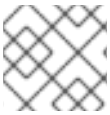
replicas spec 无法覆盖使用 Horizontal Pod Autoscaler (HPA) 的部署副本数，且不适用于 **eventing-webhook** 部署。

在以下示例中，**KnativeEventing** CR 覆盖 **eventing-controller** 部署，以便：

- 部署指定了 CPU 和内存资源限制。
- 部署有 3 个副本。
- 添加 **example-label: label** 标签。
- 添加 **example-annotation:** 注解。
- **nodeSelector** 字段被设置为选择带有 **disktype: hdd** 标签的节点。

KnativeEventing CR 示例

```
apiVersion: operator.knative.dev/v1beta1
kind: KnativeEventing
metadata:
  name: knative-eventing
  namespace: knative-eventing
spec:
  deployments:
  - name: eventing-controller
    resources:
    - container: eventing-controller
      requests:
        cpu: 300m
        memory: 100Mi
      limits:
        cpu: 1000m
        memory: 250Mi
    replicas: 3
    labels:
      example-label: label
    annotations:
      example-annotation: annotation
    nodeSelector:
      disktype: hdd
```



注意

KnativeEventing CR 标签和注解设置覆盖部署本身和生成的 Pod 的部署标签和注解。

6.1.7. 配置 EmptyDir 扩展

emptyDir 卷是创建 pod 时创建的空卷，用来提供临时工作磁盘空间。当为其创建 pod 被删除时，**emptyDir** 卷会被删除。

kubernetes.podspec-volumes-emptydir 扩展控制 **emptyDir** 卷是否与 Knative Serving 搭配使用。要使用 **emptyDir** 卷启用，您必须修改 **KnativeServing** 自定义资源 (CR) 使其包含以下 YAML：

KnativeServing CR 示例

```
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeServing
metadata:
```

```
  name: knative-serving
spec:
  config:
    features:
      kubernetes.podspec-volumes-emptydir: enabled
...
```

6.1.8. HTTPS 重定向全局设置

HTTPS 重定向为传入的 HTTP 请求提供重定向。这些重定向的 HTTP 请求会被加密。您可以通过为 **KnativeServing** 自定义资源 (CR) 配置 **httpProtocol** spec，为集群中的所有服务启用 HTTPS 重定向。

启用 HTTPS 重定向的 KnativeServing CR 示例

```
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeServing
metadata:
  name: knative-serving
spec:
  config:
    network:
      httpProtocol: "redirected"
...
```

6.1.9. 为外部路由设置 URL 方案

用于增强安全性，外部路由的 URL 方案默认为 HTTPS。这个方案由 **KnativeServing** 自定义资源 (CR) spec 中的 **default-external-scheme** 键决定。

默认规格

```
...
spec:
  config:
    network:
      default-external-scheme: "https"
...
```

您可以通过修改 **default-external-scheme** 键来覆盖默认的 spec 以使用 HTTP：

HTTP 覆盖规格

```
...
spec:
  config:
    network:
      default-external-scheme: "http"
...
```

6.1.10. 设置 Kourier 网关服务类型

Kourier 网关默认作为 **ClusterIP** 服务类型公开。此服务类型由 **KnativeServing** 自定义资源 (CR) 中的 **service-type** ingress spec 决定。

默认规格

```
...
spec:
  ingress:
    kourier:
      service-type: ClusterIP
...
```

您可以通过修改 **service-type** spec 来覆盖默认服务类型来使用负载均衡器服务类型：

LoadBalancer 覆盖规格

```
...
spec:
  ingress:
    kourier:
      service-type: LoadBalancer
...
```

6.1.11. 启用 PVC 支持

有些无服务器应用程序需要持久性数据存储。要做到这一点，您可以为 Knative 服务配置持久性卷声明 (PVC) 。

重要

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

有关红帽技术预览功能支持范围的详情，请参考 https://access.redhat.com/support/offerings/techpreview/。

流程

1. 要启用 Knative Serving 使用 PVC 并写入它们，请修改 **KnativeServing** 自定义资源 (CR) 使其包含以下 YAML：

   启用具有写入访问的 PVC

   ```
   ...
   spec:
     config:
       features:
         "kubernetes.podspec-persistent-volume-claim": enabled
         "kubernetes.podspec-persistent-volume-write": enabled
   ...
   ```

   - **kubernetes.podspec-persistent-volume-claim** 扩展控制持久性卷 (PV) 是否可以用于 Knative Serving。

- **kubernetes.podspec-persistent-volume-write** 扩展控制 Knative Serving 是否使用写入访问权限。

2. 要声明 PV，请修改您的服务使其包含 PV 配置。例如，您可能具有以下配置的持久性卷声明：

注意

使用支持您请求的访问模式的存储类。例如，您可以使用 **ReadWriteMany** 访问模式的 **ocs-storagecluster-cephfs** 类。

PersistentVolumeClaim 配置

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: example-pv-claim
  namespace: my-ns
spec:
  accessModes:
    - ReadWriteMany
  storageClassName: ocs-storagecluster-cephfs
  resources:
    requests:
      storage: 1Gi
```

在这种情况下，若要声明具有写访问权限的 PV，请修改服务，如下所示：

Knative 服务 PVC 配置

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  namespace: my-ns
...
spec:
 template:
   spec:
     containers:
         ...
         volumeMounts: (1)
           - mountPath: /data
             name: mydata
             readOnly: false
     volumes:
       - name: mydata
         persistentVolumeClaim: (2)
           claimName: example-pv-claim
           readOnly: false (3)
```

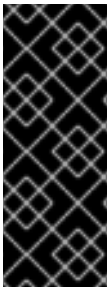
1. 卷挂载规格。
2. 持久性卷声明规格。
3. 启用只读访问的标记。

注意

要在 Knative 服务中成功使用持久性存储，您需要额外的配置，如 Knative 容器用户的用户权限。

6.1.12. 启用 init 容器

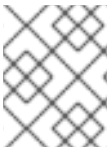
Init 容器是 pod 中应用程序容器之前运行的专用容器。它们通常用于为应用程序实施初始化逻辑，其中可能包括运行设置脚本或下载所需的配置。您可以通过修改 **KnativeServing** 自定义资源 (CR) 来启用 init 容器用于 Knative 服务。



重要

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

有关红帽技术预览功能支持范围的详情，请参考 https://access.redhat.com/support/offerings/techpreview/。



注意

Init 容器可能会导致应用程序的启动时间较长，应该谨慎地用于无服务器应用程序，这应该经常被扩展或缩减。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 有集群管理员权限。

流程

- 通过在 **KnativeServing** CR 中添加 **kubernetes.podspec-init-containers** 标记来启用 init 容器的使用：

  KnativeServing CR 示例

```
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeServing
metadata:
  name: knative-serving
spec:
  config:
    features:
      kubernetes.podspec-init-containers: enabled
...
```

6.1.13. tag-to-digest 解析

如果 Knative Serving 控制器可以访问容器 registry，Knative Serving 会在创建服务的修订时将镜像标签解析为摘要。这被称为 *tag-to-digest 解析*，有助于为部署提供一致性。

要让控制器访问 OpenShift Container Platform 上的容器 registry，您必须创建一个 secret，然后配置控制器自定义证书。您可以通过修改 **KnativeServing** 自定义资源 (CR) 中的 **controller-custom-certs** spec 来配置控制器自定义证书。secret 必须位于与 **KnativeServing** CR 相同的命名空间中。

如果 **KnativeServing** CR 中不包含 secret，此设置默认为使用公钥基础架构 (PKI) 。在使用 PKI 时，集群范围的证书会使用 **config-service-sa** 配置映射自动注入到 Knative Serving 控制器。OpenShift Serverless Operator 使用集群范围证书填充 **config-service-sa** 配置映射，并将配置映射作为卷挂载到控制器。

6.1.13.1. 使用 secret 配置 tag-to-digest 解析

如果 **controller-custom-certs** spec 使用 **Secret** 类型，secret 将被挂载为 secret 卷。Knative 组件直接使用 secret，假设 secret 具有所需的证书。

先决条件

- 在 OpenShift Container Platform 上具有集群管理员权限。
- 您已在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。

流程

1. 创建 secret：

   示例命令

   ```
   $ oc -n knative-serving create secret generic custom-secret --from-file=<secret_name>.crt=
   <path_to_certificate>
   ```

2. 配置 **KnativeServing** 自定义资源 (CR) 中的 **controller-custom-certs** 规格以使用 **Secret** 类型：

   KnativeServing CR 示例

   ```
   apiVersion: operator.knative.dev/v1alpha1
   kind: KnativeServing
   metadata:
     name: knative-serving
     namespace: knative-serving
   spec:
     controller-custom-certs:
       name: custom-secret
       type: Secret
   ```

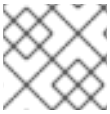
6.1.14. 其他资源

- 管理自定义资源定义中的资源
- 了解持久性存储
- 配置自定义 PKI

6.2. 配置 KNATIVE KAFKA

Knative Kafka 提供集成选项，供您在 OpenShift Serverless 中使用支持的 Apache Kafka 消息流平台。Kafka 为事件源、频道、代理和事件 sink 功能提供选项。

除了作为 OpenShift Serverless 核心安装一部分的 Knative Eventing 组件外，集群管理员还可安装 **KnativeKafka** 自定义资源 (CR) 。



注意

IBM Z 和 IBM Power Systems 目前不支持 Knative Kafka。

KnativeKafka CR 为用户提供其他选项，例如：

- Kafka 源
- Kafka 频道
- Kafka 代理
- Kafka 接收器

6.2.1. 安装 Knative Kafka

Knative Kafka 提供集成选项，供您在 OpenShift Serverless 中使用支持的 Apache Kafka 消息流平台。如果您已安装 KnativeKafka 自定义资源，则 OpenShift Serverless 安装中提供了 **Knative Kafka** 功能。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Eventing。
- 您可以访问 Red Hat AMQ Streams 集群。
- 如果要使用验证步骤，请安装 OpenShift CLI (**oc**) 。
- 在 OpenShift Container Platform 上具有集群管理员权限。
- 已登陆到 OpenShift Container Platform Web 控制台。

流程

1. 在 **Administrator** 视角中，进入 **Operators → Installed Operators**。
2. 检查页面顶部的 **Project** 下拉菜单是否已设置为 **Project: knative-eventing**。
3. 在 OpenShift Serverless Operator 的 **Provided APIs** 列表中，找到 **Knative Kafka** 复选框并点 **Create Instance**。
4. 在 **Create Knative Kafka** 页面中配置 **KnativeKafka** 对象。

重要

要在集群中使用 Kafka 频道、源、代理或 sink，您需要将要使用的属性的 **enabled** 选项设置为 **true**。这些交换机默认设置为 **false**。另外，要使用 Kafka 频道、代理或接收器，您必须指定 bootstrap 服务器。

KnativeKafka 自定义资源示例

```
apiVersion: operator.serverless.openshift.io/v1alpha1
kind: KnativeKafka
metadata:
  name: knative-kafka
  namespace: knative-eventing
spec:
  channel:
    enabled: true (1)
    bootstrapServers: <bootstrap_servers> (2)
  source:
    enabled: true (3)
  broker:
    enabled: true (4)
    defaultConfig:
      bootstrapServers: <bootstrap_servers> (5)
      numPartitions: <num_partitions> (6)
      replicationFactor: <replication_factor> (7)
  sink:
    enabled: true (8)
```

1. 让开发人员在集群中使用 **KafkaChannel** 频道类型。
2. 以逗号分隔的 AMQ Streams 集群中的 bootstrap 服务器列表。
3. 让开发人员在集群中使用 **KafkaSource** 事件源类型。
4. 让开发人员在集群中使用 Knative Kafka 代理实现。
5. 来自 Red Hat AMQ Streams 集群的 bootstrap 服务器的逗号分隔列表。
6. 定义 Kafka 主题分区的数量，由 **Broker** 对象支持。默认值为 **10**。
7. 定义 Kafka 主题的复制因素，由 **Broker** 对象支持。默认值为 **3**。
8. 让开发人员在集群中使用 Kafka sink。

注意

replicationFactor 值必须小于或等于 Red Hat AMQ Streams 集群的节点数量。

a. 建议您在不需要完全控制 **KnativeKafka** 对象创建的简单配置中使用该表单。

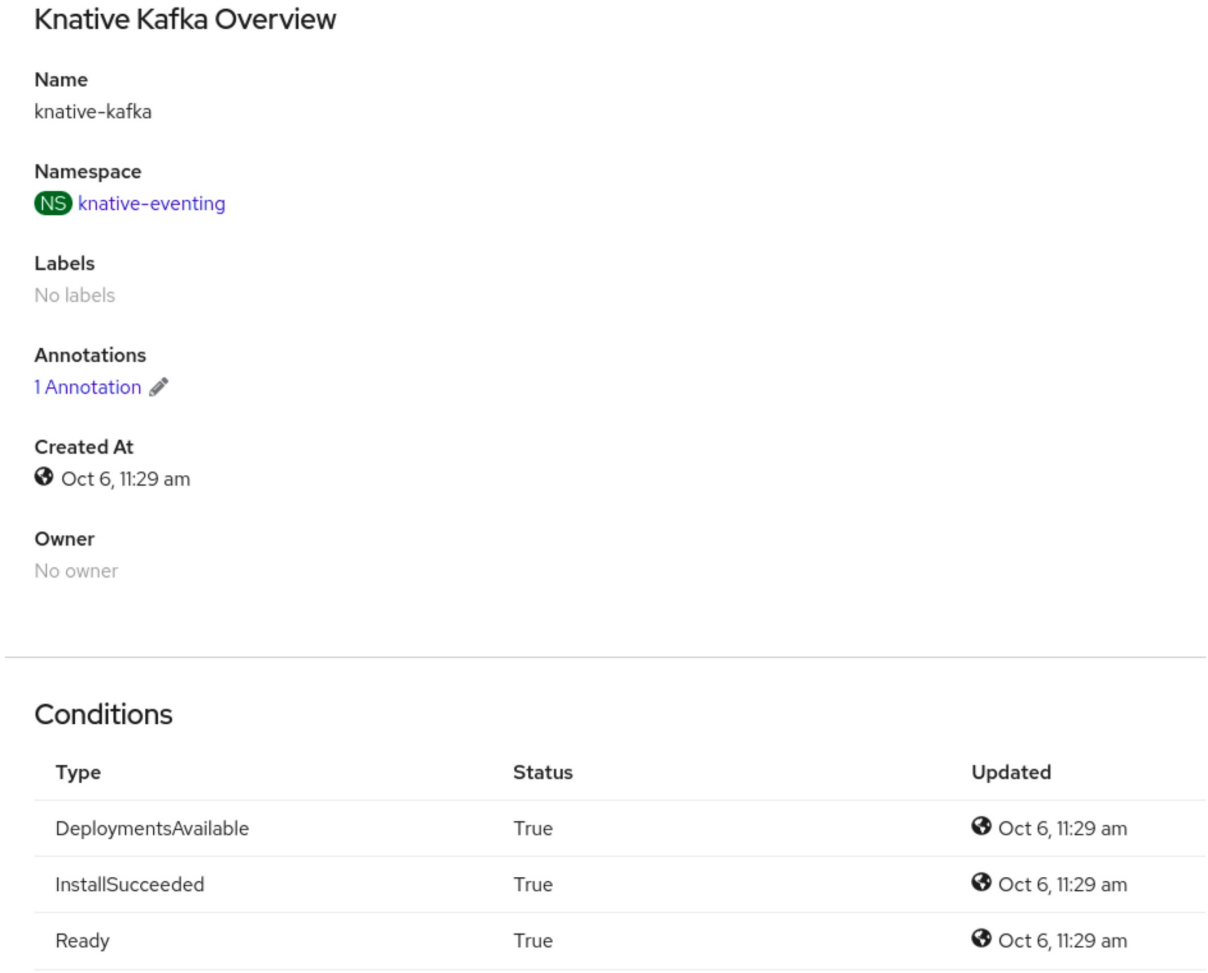
b. 对于更复杂的配置，建议编辑 YAML，这可以完全控制 **KnativeKafka** 对象的创建。您可以通过点 **Create Knative Kafka** 页面右上角的 **Edit YAML** 链接来访问 YAML。

5. 完成 Kafka 的任何可选配置后，点 **Create**。您会自动定向到 **Knative Kafka** 标签页，其中 **knative-kafka** 在资源列表中。

验证

1. 点 **Knative Kafka** 选项卡中的 **knative-kafka** 资源。您会自动定向到 **Knative Kafka Overview** 页面。

2. 查看资源的 **Conditions** 列表，并确认其状态为 **True**。



如果条件的状态为 **Unknown** 或 **False**，请等待几分钟刷新页面。

3. 检查是否已创建 Knative Kafka 资源：

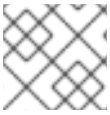
```
$ oc get pods -n knative-eventing
```

输**出示例**

```
NAME                                        READY   STATUS    RESTARTS   AGE
kafka-broker-dispatcher-7769fbbcbb-xgffn    2/2     Running   0          44s
kafka-broker-receiver-5fb56f7656-fhq8d      2/2     Running   0          44s
kafka-channel-dispatcher-84fd6cb7f9-k2tjv   2/2     Running   0          44s
kafka-channel-receiver-9b7f795d5-c76xr      2/2     Running   0          44s
kafka-controller-6f95659bf6-trd6r           2/2     Running   0          44s
kafka-source-dispatcher-6bf98bdfff-8bcsn    2/2     Running   0          44s
kafka-webhook-eventing-68dc95d54b-825xs     2/2     Running   0          44s
```

6.2.2. Knative Kafka 的安全配置

Kafka 集群通常使用 TLS 或 SASL 身份验证方法进行保护。您可以使用 TLS 或 SASL 将 Kafka 代理或频道配置为针对受保护的 Red Hat AMQ Streams 集群进行操作。



注意

红帽建议您同时启用 SASL 和 TLS。

6.2.2.1. 为 Kafka 代理配置 TLS 身份验证

Apache Kafka 客户端和服务器使用 *传输层安全性* (TLS) 来加密 Knative 和 Kafka 之间的流量，以及用于身份验证。TLS 是 Knative Kafka 唯一支持的流量加密方法。

先决条件

- 在 OpenShift Container Platform 上具有集群管理员权限。
- OpenShift Serverless Operator、Knative Eventing 和 **KnativeKafka** CR 已安装在 OpenShift Container Platform 集群中。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。
- 您有一个 Kafka 集群 CA 证书存储为一个 **.pem** 文件。
- 您有一个 Kafka 集群客户端证书，并存储为 **.pem** 文件的密钥。
- 安装 OpenShift CLI (**oc**) 。

流程

1. 在 **knative-eventing** 命名空间中创建证书文件作为 secret：

```
$ oc create secret -n knative-eventing generic <secret_name> \
  --from-literal=protocol=SSL \
  --from-file=ca.crt=caroot.pem \
  --from-file=user.crt=certificate.pem \
  --from-file=user.key=key.pem
```

重要

使用密钥名称 **ca.crt**、**user.crt** 和 **user.key**。不要更改它们。

2. 编辑 **KnativeKafka** CR，并在 **broker** spec 中添加对 secret 的引用：

```
apiVersion: operator.serverless.openshift.io/v1alpha1
kind: KnativeKafka
metadata:
  namespace: knative-eventing
  name: knative-kafka
spec:
  broker:
    enabled: true
    defaultConfig:
      authSecretName: <secret_name>
...
```

6.2.2.2. 为 Kafka 代理配置 SASL 身份验证

Apache Kafka 使用 *简单身份验证和安全层* (SASL) 进行身份验证。如果在集群中使用 SASL 身份验证，用户则必须向 Knative 提供凭证才能与 Kafka 集群通信，否则无法生成或消耗事件。

先决条件

- 在 OpenShift Container Platform 上具有集群管理员权限。
- OpenShift Serverless Operator、Knative Eventing 和 **KnativeKafka** CR 已安装在 OpenShift Container Platform 集群中。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。
- 您有一个 Kafka 集群的用户名和密码。
- 您已选择使用 SASL 机制，例如 **PLAIN**、**SCRAM-SHA-256** 或 **SCRAM-SHA-512**。
- 如果启用了 TLS，您还需要 Kafka 集群的 **ca.crt** 证书文件。
- 安装 OpenShift CLI (**oc**) 。

流程

1. 在 **knative-eventing** 命名空间中创建证书文件作为 secret：

```
$ oc create secret -n knative-eventing generic <secret_name> \
  --from-literal=protocol=SASL_SSL \
  --from-literal=sasl.mechanism=<sasl_mechanism> \
  --from-file=ca.crt=caroot.pem \
  --from-literal=password="SecretPassword" \
  --from-literal=user="my-sasl-user"
```

   - 使用键名 **ca.crt**, **password**, 和 **sasl.mechanism**.不要更改它们。
   - 如果要将 SASL 与公共 CA 证书搭配使用，您必须在创建 secret 时使用 **tls.enabled=true** 标志，而不是使用 **ca.crt** 参数。例如：

```
$ oc create secret -n <namespace> generic <kafka_auth_secret> \
  --from-literal=tls.enabled=true \
  --from-literal=password="SecretPassword" \
  --from-literal=saslType="SCRAM-SHA-512" \
  --from-literal=user="my-sasl-user"
```

2. 编辑 **KnativeKafka** CR，并在 **broker** spec 中添加对 secret 的引用：

```
apiVersion: operator.serverless.openshift.io/v1alpha1
kind: KnativeKafka
metadata:
  namespace: knative-eventing
  name: knative-kafka
spec:
  broker:
    enabled: true
```

```
    defaultConfig:
      authSecretName: <secret_name>
...
```

6.2.2.3. 为 Kafka 频道配置 TLS 验证

Apache Kafka 客户端和服务器使用 *传输层安全性* (TLS) 来加密 Knative 和 Kafka 之间的流量，以及用于身份验证。TLS 是 Knative Kafka 唯一支持的流量加密方法。

先决条件

- 在 OpenShift Container Platform 上具有集群管理员权限。
- OpenShift Serverless Operator、Knative Eventing 和 **KnativeKafka** CR 已安装在 OpenShift Container Platform 集群中。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。
- 您有一个 Kafka 集群 CA 证书存储为一个 **.pem** 文件。
- 您有一个 Kafka 集群客户端证书，并存储为 **.pem** 文件的密钥。
- 安装 OpenShift CLI (**oc**) 。

流程

1. 在所选命名空间中创建证书文件作为 secret：

```
$ oc create secret -n <namespace> generic <kafka_auth_secret> \
  --from-file=ca.crt=caroot.pem \
  --from-file=user.crt=certificate.pem \
  --from-file=user.key=key.pem
```

重要

使用密钥名称 **ca.crt**、**user.crt** 和 **user.key**。不要更改它们。

2. 编辑 **KnativeKafka** 自定义资源：

```
$ oc edit knativekafka
```

3. 引用您的 secret 和 secret 的命名空间：

```
apiVersion: operator.serverless.openshift.io/v1alpha1
kind: KnativeKafka
metadata:
  namespace: knative-eventing
  name: knative-kafka
spec:
  channel:
    authSecretName: <kafka_auth_secret>
    authSecretNamespace: <kafka_auth_secret_namespace>
    bootstrapServers: <bootstrap_servers>
```

```
    enabled: true
  source:
    enabled: true
```

注意

确保指定 bootstrap 服务器中的匹配端口。

例如：

```
apiVersion: operator.serverless.openshift.io/v1alpha1
kind: KnativeKafka
metadata:
  namespace: knative-eventing
  name: knative-kafka
spec:
  channel:
    authSecretName: tls-user
    authSecretNamespace: kafka
    bootstrapServers: eventing-kafka-bootstrap.kafka.svc:9094
    enabled: true
  source:
    enabled: true
```

6.2.2.4. 为 Kafka 频道配置 SASL 验证

Apache Kafka 使用 *简单身份验证和安全层* (SASL) 进行身份验证。如果在集群中使用 SASL 身份验证，用户则必须向 Knative 提供凭证才能与 Kafka 集群通信，否则无法生成或消耗事件。

先决条件

- 在 OpenShift Container Platform 上具有集群管理员权限。
- OpenShift Serverless Operator、Knative Eventing 和 **KnativeKafka** CR 已安装在 OpenShift Container Platform 集群中。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。
- 您有一个 Kafka 集群的用户名和密码。
- 您已选择使用 SASL 机制，例如 **PLAIN**、**SCRAM-SHA-256** 或 **SCRAM-SHA-512**。
- 如果启用了 TLS，您还需要 Kafka 集群的 **ca.crt** 证书文件。
- 安装 OpenShift CLI (**oc**) 。

流程

1. 在所选命名空间中创建证书文件作为 secret：

```
$ oc create secret -n <namespace> generic <kafka_auth_secret> \
  --from-file=ca.crt=caroot.pem \
  --from-literal=password="SecretPassword" \
```

```
  --from-literal=saslType="SCRAM-SHA-512" \
  --from-literal=user="my-sasl-user"
```

- 使用键名 **ca.crt**, **password**, 和 **sasl.mechanism**.不要更改它们。
- 如果要将 SASL 与公共 CA 证书搭配使用，您必须在创建 secret 时使用 **tls.enabled=true** 标志，而不是使用 **ca.crt** 参数。例如：

```
$ oc create secret -n <namespace> generic <kafka_auth_secret> \
  --from-literal=tls.enabled=true \
  --from-literal=password="SecretPassword" \
  --from-literal=saslType="SCRAM-SHA-512" \
  --from-literal=user="my-sasl-user"
```

2. 编辑 **KnativeKafka** 自定义资源：

```
$ oc edit knativekafka
```

3. 引用您的 secret 和 secret 的命名空间：

```
apiVersion: operator.serverless.openshift.io/v1alpha1
kind: KnativeKafka
metadata:
  namespace: knative-eventing
  name: knative-kafka
spec:
  channel:
    authSecretName: <kafka_auth_secret>
    authSecretNamespace: <kafka_auth_secret_namespace>
    bootstrapServers: <bootstrap_servers>
    enabled: true
  source:
    enabled: true
```

注意

确保指定 bootstrap 服务器中的匹配端口。

例如：

```
apiVersion: operator.serverless.openshift.io/v1alpha1
kind: KnativeKafka
metadata:
  namespace: knative-eventing
  name: knative-kafka
spec:
  channel:
    authSecretName: scram-user
    authSecretNamespace: kafka
    bootstrapServers: eventing-kafka-bootstrap.kafka.svc:9093
    enabled: true
  source:
    enabled: true
```

6.2.2.5. 为 Kafka 源配置 SASL 身份验证

Apache Kafka 使用 *简单身份验证和安全层* (SASL) 进行身份验证。如果在集群中使用 SASL 身份验证，用户则必须向 Knative 提供凭证才能与 Kafka 集群通信，否则无法生成或消耗事件。

先决条件

- 在 OpenShift Container Platform 上具有集群或专用管理员权限。
- OpenShift Serverless Operator、Knative Eventing 和 **KnativeKafka** CR 已安装在 OpenShift Container Platform 集群中。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。
- 您有一个 Kafka 集群的用户名和密码。
- 您已选择使用 SASL 机制，例如 **PLAIN**、**SCRAM-SHA-256** 或 **SCRAM-SHA-512**。
- 如果启用了 TLS，您还需要 Kafka 集群的 **ca.crt** 证书文件。
- 已安装 OpenShift (**oc**) CLI。

流程

1. 在所选命名空间中创建证书文件作为 secret：

```
$ oc create secret -n <namespace> generic <kafka_auth_secret> \
  --from-file=ca.crt=caroot.pem \
  --from-literal=password="SecretPassword" \
  --from-literal=saslType="SCRAM-SHA-512" \ 1
  --from-literal=user="my-sasl-user"
```

1 SASL 类型可以是 **PLAIN**、**SCRAM-SHA-256** 或 **SCRAM-SHA-512**。

2. 创建或修改 Kafka 源，使其包含以下 **spec** 配置：

```
apiVersion: sources.knative.dev/v1beta1
kind: KafkaSource
metadata:
  name: example-source
spec:
...
  net:
    sasl:
      enable: true
      user:
        secretKeyRef:
          name: <kafka_auth_secret>
          key: user
      password:
        secretKeyRef:
          name: <kafka_auth_secret>
          key: password
      type:
```

```
      secretKeyRef:
        name: <kafka_auth_secret>
        key: saslType
    tls:
      enable: true
      caCert: 1
        secretKeyRef:
          name: <kafka_auth_secret>
          key: ca.crt
...
```

1 如果您使用公共云 Kafka 服务，则不需要 **caCert** spec，如 Red Hat OpenShift Streams for Apache Kafka。

6.2.2.6. 为 Kafka sink 配置安全性

Apache Kafka 客户端和服务器使用 *传输层安全性* (TLS) 来加密 Knative 和 Kafka 之间的流量，以及用于身份验证。TLS 是 Knative Kafka 唯一支持的流量加密方法。

Apache Kafka 使用 *简单身份验证和安全层* (SASL) 进行身份验证。如果在集群中使用 SASL 身份验证，用户则必须向 Knative 提供凭证才能与 Kafka 集群通信，否则无法生成或消耗事件。

先决条件

- OpenShift Serverless Operator、Knative Eventing 和 **KnativeKafka** 自定义资源（CR）已安装在 OpenShift Container Platform 集群中。
- 在 **KnativeKafka** CR 中启用了 Kafka sink。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。
- 您有一个 Kafka 集群 CA 证书存储为一个 **.pem** 文件。
- 您有一个 Kafka 集群客户端证书，并存储为 **.pem** 文件的密钥。
- 已安装 OpenShift (**oc**) CLI。
- 您已选择使用 SASL 机制，例如 **PLAIN**、**SCRAM-SHA-256** 或 **SCRAM-SHA-512**。

流程

1. 在与 **KafkaSink** 对象相同的命名空间中创建一个 secret：

重要

证书和密钥必须采用 PEM 格式。

- 对于使用 SASL 时没有加密的身份验证：

```
$ oc create secret -n <namespace> generic <secret_name> \
  --from-literal=protocol=SASL_PLAINTEXT \
  --from-literal=sasl.mechanism=<sasl_mechanism> \
```

```
  --from-literal=user=<username> \
  --from-literal=password=<password>
```

- 对于使用 TLS 的 SASL 和加密进行身份验证：

```
$ oc create secret -n <namespace> generic <secret_name> \
  --from-literal=protocol=SASL_SSL \
  --from-literal=sasl.mechanism=<sasl_mechanism> \
  --from-file=ca.crt=<my_caroot.pem_file_path> \ 1
  --from-literal=user=<username> \
  --from-literal=password=<password>
```

1 如果您使用公共云管理 Kafka 服务，可以省略 **ca.crt** 来使用系统的根 CA，如用于 Apache Kafka 的 Red Hat OpenShift Streams。

- 使用 TLS 进行身份验证和加密：

```
$ oc create secret -n <namespace> generic <secret_name> \
  --from-literal=protocol=SSL \
  --from-file=ca.crt=<my_caroot.pem_file_path> \ 1
  --from-file=user.crt=<my_cert.pem_file_path> \
  --from-file=user.key=<my_key.pem_file_path>
```

1 如果您使用公共云管理 Kafka 服务，可以省略 **ca.crt** 来使用系统的根 CA，如用于 Apache Kafka 的 Red Hat OpenShift Streams。

2. 创建或修改 **KafkaSink** 对象，并在 **auth** spec 中添加对 secret 的引用：

```
apiVersion: eventing.knative.dev/v1alpha1
kind: KafkaSink
metadata:
  name: <sink_name>
  namespace: <namespace>
spec:
...
  auth:
    secret:
      ref:
        name: <secret_name>
...
```

3. 应用 **KafkaSink** 对象：

```
$ oc apply -f <filename>
```

6.2.3. 配置 Kafka 代理设置

您可以通过创建配置映射并在 Kafka **Broker** 对象中引用此配置映射，配置复制因素、bootstrap 服务器和 Kafka 代理的主题分区数量。

先决条件

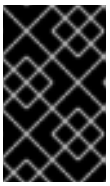
- 在 OpenShift Container Platform 上具有集群或专用管理员权限。
- OpenShift Serverless Operator、Knative Eventing 和 **KnativeKafka** 自定义资源（CR）已安装在 OpenShift Container Platform 集群中。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。
- 已安装 OpenShift CLI(**oc**)。

流程

1. 修改 **kafka-broker-config** 配置映射，或创建自己的配置映射来包含以下配置：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: <config_map_name> 1
  namespace: <namespace> 2
data:
  default.topic.partitions: <integer> 3
  default.topic.replication.factor: <integer> 4
  bootstrap.servers: <list_of_servers> 5
```

1. 配置映射名称。
2. 配置映射所在的命名空间。
3. Kafka 代理的主题分区数量。这控制如何将事件发送到代理的速度。更多分区需要更多计算资源。
4. 主题消息的复制因素。这可防止数据丢失。更高的复制因素需要更大的计算资源和更多存储。
5. 以逗号分隔的 bootstrap 服务器列表。这可以位于 OpenShift Container Platform 集群内部或外部，是代理从发送事件发送到的 Kafka 集群列表。



重要

default.topic.replication.factor 值必须小于或等于集群中的 Kafka 代理实例数量。例如，如果您只有一个 Kafka 代理，则 **default.topic.replication.factor** 值应该不超过 **"1"**。

Kafka 代理配置映射示例

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: kafka-broker-config
  namespace: knative-eventing
data:
```

```
    default.topic.partitions: "10"
    default.topic.replication.factor: "3"
    bootstrap.servers: "my-cluster-kafka-bootstrap.kafka:9092"
```

2. 应用配置映射：

```
$ oc apply -f <config_map_filename>
```

3. 指定 Kafka **Broker** 对象的配置映射：

Broker 对象示例

```
apiVersion: eventing.knative.dev/v1
kind: Broker
metadata:
  name: <broker_name> 1
  namespace: <namespace> 2
  annotations:
    eventing.knative.dev/broker.class: Kafka 3
spec:
  config:
    apiVersion: v1
    kind: ConfigMap
    name: <config_map_name> 4
    namespace: <namespace> 5
...
```

1. 代理名称。
2. 代理存在的命名空间。
3. 代理类注解。在本例中，代理是使用类值 Kafka 的 **Kafka** 代理。
4. 配置映射名称。
5. 配置映射所在的命名空间。

4. 应用代理：

```
$ oc apply -f <broker_filename>
```

其他资源

- 创建代理

6.2.4. 其他资源

- Red Hat AMQ Streams 文档
- Kafka 上的 TLS 和 SASL

6.3. ADMINISTRATOR 视角中的 SERVERLESS 组件

如果您不想在 OpenShift Container Platform web 控制台中切换到 **Developer** 视角，或使用 Knative (**kn**) CLI 或 YAML 文件，您可以使用 OpenShift Container Platform Web 控制台的 **Administator** 视角创建 Knative 组件。

6.3.1. 使用管理员视角创建无服务器应用程序

无服务器应用程序已创建并部署为 Kubernetes 服务，由路由和配置定义，并包含在 YAML 文件中。要使用 OpenShift Serverless 部署无服务器应用程序，您必须创建一个 Knative **Service** 对象。

Knative Service 对象 YAML 文件示例

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: hello 1
  namespace: default 2
spec:
  template:
    spec:
      containers:
        - image: docker.io/openshift/hello-openshift 3
          env:
            - name: RESPONSE 4
              value: "Hello Serverless!"
```

1. 应用程序的名称。
2. 应用程序使用的命名空间。
3. 应用程序的镜像。
4. 示例应用程序输出的环境变量。

创建服务并部署应用程序后，Knative 会为应用程序的这个版本创建一个不可变的修订版本。Knative 还将执行网络操作，为您的应用程序创建路由、入口、服务和负载平衡器，并根据流量自动扩展或缩减 pod。

先决条件

要使用 **管理员** 视角创建无服务器应用程序，请确定您已完成了以下步骤。

- 安装了 OpenShift Serverless Operator 和 Knative Serving。
- 您已登录到 Web 控制台，且处于 **Administrator** 视角。

流程

1. 进入 **Serverless → Serving** 页面。
2. 在 **Create** 列表中，选择 **Service**。
3. 手动输入 YAML 或 JSON 定义，或者将文件拖放到编辑器中。
4. 点 **Create**。

6.3.2. 其他资源

- 无服务器应用程序

6.4. 将 SERVICE MESH 与 OPENSHIFT SERVERLESS 集成

OpenShift Serverless Operator 提供 Kourier 作为 Knative 的默认入口。但是，无论是否启用了 Kourier，您都可以在 OpenShift Serverless 中使用 Service Mesh。禁用 Kourier 集成后，您可以配置 Kourier ingress 不支持的额外网络和路由选项，如 mTLS 功能。

重要

OpenShift Serverless 只支持使用本指南中明确记录的 Red Hat OpenShift Service Mesh 功能，且不支持其他未记录的功能。

6.4.1. 先决条件

- 以下流程中的示例使用域 **example.com**。这个域的示例证书被用作为子域证书签名的证书颁发机构（CA）。
  要在部署中完成并验证这些步骤，您需要由广泛信任的公共 CA 签名的证书或您的机构提供的 CA。根据您的域、子域和 CA 调整命令示例。

- 您必须配置通配符证书，以匹配 OpenShift Container Platform 集群的域。例如，如果您的 OpenShift Container Platform 控制台地址是 **https://console-openshift-console.apps.openshift.example.com**，您必须配置通配符证书，以便域为 ***.apps.openshift.example.com**。有关配置通配符证书的更多信息，请参阅 *创建证书来加密传入的外部流量*。

- 如果要使用任何域名，包括不是默认 OpenShift Container Platform 集群域子域的域名，您必须为这些域设置域映射。如需更多信息，请参阅有关创建自定义域映射的 OpenShift Serverless 文档。

6.4.2. 创建证书来加密传入的外部流量

默认情况下，Service Mesh mTLS 功能只会保护 Service Mesh 本身内部的流量，在 ingress 网关和带有 sidecar 的独立 pod 间的安全。要在流向 OpenShift Container Platform 集群时对流量进行加密，您必须先生成证书，然后才能启用 OpenShift Serverless 和 Service Mesh 集成。

先决条件

- 您可以访问具有集群管理员权限的 OpenShift Container Platform 帐户。
- 安装了 OpenShift Serverless Operator 和 Knative Serving。
- 安装 OpenShift CLI (**oc**)。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。

流程

1. 创建为 Knative 服务签名的 root 证书和私钥：

```
$ openssl req -x509 -sha256 -nodes -days 365 -newkey rsa:2048 \
```

```
    -subj '/O=Example Inc./CN=example.com' \
    -keyout root.key \
    -out root.crt
```

2. 创建通配符证书：

```
$ openssl req -nodes -newkey rsa:2048 \
    -subj "/CN=*.apps.openshift.example.com/O=Example Inc." \
    -keyout wildcard.key \
    -out wildcard.csr
```

3. 为通配符证书签名：

```
$ openssl x509 -req -days 365 -set_serial 0 \
    -CA root.crt \
    -CAkey root.key \
    -in wildcard.csr \
    -out wildcard.crt
```

4. 使用通配符证书创建 secret：

```
$ oc create -n istio-system secret tls wildcard-certs \
    --key=wildcard.key \
    --cert=wildcard.crt
```

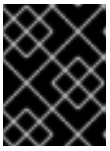
此证书由 OpenShift Serverless 与 Service Mesh 集成时创建的网关获取，以便入口网关使用此证书提供流量。

6.4.3. 将 Service Mesh 与 OpenShift Serverless 集成

您可以在不使用 Kourier 作为默认入口的情况下将 Service Mesh 与 OpenShift Serverless 集成。要做到这一点，在完成以下步骤前不要安装 Knative Serving 组件。在创建 **KnativeServing** 自定义资源定义 (CRD) 以将 Knative Serving 与 Service Mesh 集成时，还需要额外的步骤，这没有在一般的 Knative Serving 安装过程中覆盖。如果您要将 Service Mesh 集成为 OpenShift Serverless 安装的默认的且唯一的 ingress，这个步骤可能很有用。

先决条件

- 您可以访问具有集群管理员权限的 OpenShift Container Platform 帐户。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。
- 安装 Red Hat OpenShift Service Mesh Operator，并在 **istio-system** 命名空间中创建 **ServiceMeshControlPlane** 资源。如果要使用 mTLS 功能，还必须将 **ServiceMeshControlPlane** 资源的 **spec.security.dataPlane.mtls** 字段设置为 **true**。



重要

在 Service Mesh 中使用 OpenShift Serverless 只支持 Red Hat OpenShift Service Mesh 2.0.5 或更高版本。

- 安装 OpenShift Serverless Operator。

- 安装 OpenShift CLI (**oc**)。

流程

1. 将您要与 Service Mesh 集成的命名空间作为成员添加到 **ServiceMeshMemberRoll** 对象中：

```
apiVersion: maistra.io/v1
kind: ServiceMeshMemberRoll
metadata:
  name: default
  namespace: istio-system
spec:
  members: 1
    - knative-serving
    - <namespace>
```

1 要与 Service Mesh 集成的命名空间列表。

重要

此命名空间列表必须包含 **knative-serving** 命名空间。

2. 应用 **ServiceMeshMemberRoll** 资源：

```
$ oc apply -f <filename>
```

3. 创建必要的网关，以便 Service Mesh 可以接受流量：

使用 HTTP 的 knative-local-gateway 对象示例

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: knative-ingress-gateway
  namespace: knative-serving
spec:
  selector:
    istio: ingressgateway
  servers:
    - port:
        number: 443
        name: https
        protocol: HTTPS
      hosts:
        - "*"
      tls:
        mode: SIMPLE
        credentialName: <wildcard_certs> 1
---
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
 name: knative-local-gateway
```

```
 namespace: knative-serving
spec:
 selector:
   istio: ingressgateway
 servers:
   - port:
       number: 8081
       name: http
       protocol: HTTP 2
     hosts:
       - "*"
---
apiVersion: v1
kind: Service
metadata:
 name: knative-local-gateway
 namespace: istio-system
 labels:
   experimental.istio.io/disable-gateway-port-translation: "true"
spec:
 type: ClusterIP
 selector:
   istio: ingressgateway
 ports:
   - name: http2
     port: 80
     targetPort: 8081
```

1 添加包含通配符证书的 secret 名称。

2 **knative-local-gateway** 提供 HTTP 流量。使用 HTTP 表示来自 Service Mesh 外部的流量不会加密，但使用内部主机名，如 **example.default.svc.cluster.local**。您可以通过创建另一个通配符证书和使用不同的 **protocol** spec 的额外网关来为这个路径设置加密。

使用 HTTPS 的 knative-local-gateway 对象示例

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: knative-local-gateway
  namespace: knative-serving
spec:
  selector:
    istio: ingressgateway
  servers:
    - port:
        number: 443
        name: https
        protocol: HTTPS
      hosts:
        - "*"
      tls:
        mode: SIMPLE
        credentialName: <wildcard_certs>
```

4. 应用 **Gateway** 资源：

```
$ oc apply -f <filename>
```

5. 通过创建以下 **KnativeServing** 自定义资源定义 (CRD) 来安装 Knative Serving，该定义还启用了 Istio 集成：

```
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeServing
metadata:
  name: knative-serving
  namespace: knative-serving
spec:
  ingress:
    istio:
      enabled: true 1
  deployments: 2
  - name: activator
    annotations:
      "sidecar.istio.io/inject": "true"
      "sidecar.istio.io/rewriteAppHTTPProbers": "true"
  - name: autoscaler
    annotations:
      "sidecar.istio.io/inject": "true"
      "sidecar.istio.io/rewriteAppHTTPProbers": "true"
```

1 启用 Istio 集成。

2 为 Knative Serving data plane pod 启用 sidecar 注入。

6. 应用 **KnativeServing** 资源：

```
$ oc apply -f <filename>
```

7. 创建一个启用了 sidecar 注入并使用 pass-through 路由的 Knative Service：

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: <service_name>
  namespace: <namespace> 1
  annotations:
    serving.knative.openshift.io/enablePassthrough: "true" 2
spec:
  template:
    metadata:
      annotations:
        sidecar.istio.io/inject: "true" 3
        sidecar.istio.io/rewriteAppHTTPProbers: "true"
    spec:
      containers:
      - image: <image_url>
```

1. 作为 Service Mesh member roll 一部分的命名空间。
2. 指示 Knative Serving 生成 OpenShift Container Platform 直通启用路由，以便您已生成的证书直接通过 ingress 网关提供。
3. 将 Service Mesh sidecar 注入 Knative 服务 pod。

8. 应用 **Service** 资源：

```
$ oc apply -f <filename>
```

验证

- 使用 CA 信任的安全连接访问无服务器应用程序：

```
$ curl --cacert root.crt <service_url>
```

示例命令

```
$ curl --cacert root.crt https://hello-default.apps.openshift.example.com
```

输出示例

```
Hello Openshift!
```

6.4.4. 在使用带有 mTLS 的 Service Mesh 时启用 Knative Serving 指标

如果启用了 mTLS 的 Service Mesh，则默认禁用 Knative Serving 的指标，因为 Service Mesh 会防止 Prometheus 提取指标。本节介绍在使用 Service Mesh 和 mTLS 时如何启用 Knative Serving 指标。

先决条件

- 您已在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 已安装了启用了 mTLS 功能的 Red Hat OpenShift Service Mesh。
- 您可以访问具有集群管理员权限的 OpenShift Container Platform 帐户。
- 安装 OpenShift CLI (**oc**) 。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。

流程

1. 在 Knative Serving 自定义资源 (CR) 的 **observability** spec 中将 **prometheus** 指定为 **metrics.backend-destination** ：

```
apiVersion: operator.knative.dev/v1beta1
kind: KnativeServing
metadata:
  name: knative-serving
```

```
spec:
  config:
    observability:
      metrics.backend-destination: "prometheus"
...
```

此步骤可防止默认禁用指标。

2. 应用以下网络策略来允许来自 Prometheus 命名空间中的流量：

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-from-openshift-monitoring-ns
  namespace: knative-serving
spec:
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          name: "openshift-monitoring"
  podSelector: {}
...
```

3. 修改并重新应用 **istio-system** 命名空间中的默认 Service Mesh control plane，使其包含以下 spec：

```
...
spec:
  proxy:
    networking:
      trafficControl:
        inbound:
          excludedPorts:
          - 8444
...
```

6.4.5. 在启用了 Kourier 时将 Service Mesh 与 OpenShift Serverless 集成

即使已经启用了 Kourier，您也可以在 OpenShift Serverless 中使用 Service Mesh。如果您已在启用了 Kourier 的情况下安装了 Knative Serving，但决定在以后添加 Service Mesh 集成，这个过程可能会很有用。

先决条件

- 您可以访问具有集群管理员权限的 OpenShift Container Platform 帐户。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。
- 安装 OpenShift CLI (**oc**) 。
- 在集群上安装 OpenShift Serverless Operator 和 Knative Serving。

- 安装 Red Hat OpenShift Service Mesh。带有 Service Mesh 和 Kourier 的 OpenShift Serverless 支持与 Red Hat OpenShift Service Mesh 1.x 和 2.x 版本搭配使用。

流程

1. 将您要与 Service Mesh 集成的命名空间作为成员添加到 **ServiceMeshMemberRoll** 对象中：

```
apiVersion: maistra.io/v1
kind: ServiceMeshMemberRoll
metadata:
  name: default
  namespace: istio-system
spec:
  members:
    - <namespace> 1
...
```

1 要与 Service Mesh 集成的命名空间列表。

2. 应用 **ServiceMeshMemberRoll** 资源：

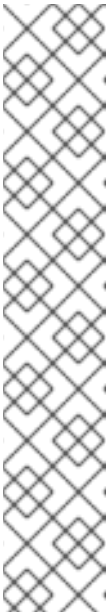
```
$ oc apply -f <filename>
```

3. 创建允许 Knative 系统 Pod 到 Knative 服务流量的网络策略：

   a. 对于您要与 Service Mesh 集成的每个命名空间，创建一个 **NetworkPolicy** 资源：

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-from-serving-system-namespace
  namespace: <namespace> 1
spec:
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          knative.openshift.io/part-of: "openshift-serverless"
  podSelector: {}
  policyTypes:
  - Ingress
...
```

1 添加您要与 Service Mesh 集成的命名空间。



注意

knative.openshift.io/part-of: "openshift-serverless" 标签添加到 OpenShift Serverless 1.22.0 中。如果使用 OpenShift Serverless 1.21.1 或更早版本，请将 **knative.openshift.io/part-of** 标签添加到 **knative-serving** 和 **knative-serving-ingress** 命名空间。

将标签添加到 **knative-serving** 命名空间：

```
$ oc label namespace knative-serving knative.openshift.io/part-of=openshift-serverless
```

将标签添加到 **knative-serving-ingress** 命名空间：

```
$ oc label namespace knative-serving-ingress knative.openshift.io/part-of=openshift-serverless
```

b. 应用 **NetworkPolicy** 资源：

```
$ oc apply -f <filename>
```

6.4.6. 为 Service Mesh 使用 secret 过滤来改进内存用量

默认情况下，Kubernetes **client-go** 库的 informers 实施会获取特定类型的所有资源。当有很多资源可用时，这可能会导致大量资源出现大量开销，这可能会导致 Knative **net-istio** 入口控制器因为内存泄漏而在大型集群中失败。但是，一个过滤机制可用于 Knative **net-istio** ingress 控制器，它可让控制器只获取 Knative 相关的 secret。您可以通过在 **KnativeServing** 自定义资源 (CR) 中添加注解来启用此机制。

先决条件

- 您可以访问具有集群管理员权限的 OpenShift Container Platform 帐户。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。
- 安装 Red Hat OpenShift Service Mesh。带有 Service Mesh 的 OpenShift Serverless 仅支持与 Red Hat OpenShift Service Mesh 2.0.5 或更高版本搭配使用。
- 安装 OpenShift Serverless Operator 和 Knative Serving。
- 安装 OpenShift CLI (**oc**) 。

流程

- 将 **serverless.openshift.io/enable-secret-informer-filtering** 注解添加到 **KnativeServing** CR：

  KnativeServing CR 示例

```
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeServing
metadata:
  name: knative-serving
  namespace: knative-serving
  annotations:
```

```
    serverless.openshift.io/enable-secret-informer-filtering: "true" (1)
spec:
  ingress:
    istio:
      enabled: true
  deployments:
    - annotations:
        sidecar.istio.io/inject: "true"
        sidecar.istio.io/rewriteAppHTTPProbers: "true"
      name: activator
    - annotations:
        sidecar.istio.io/inject: "true"
        sidecar.istio.io/rewriteAppHTTPProbers: "true"
      name: autoscaler
```

(1) 添加此注解会将 enviroment 变量 **ENABLE_SECRET_INFORMER_FILTERING_BY_CERT_UID=true** 注入到 **net-istio** 控制器 pod。

6.5. SERVERLESS 管理员指标

指标 (metrics) 可以让集群管理员监控 OpenShift Serverless 集群组件和工作负载的执行情况。

您可以通过在 OpenShift Container Platform Web 控制台 **Administrator** 视角中导航到 **Dashboards** 来查看 OpenShift Serverless 的不同指标。

6.5.1. 先决条件

- 如需有关为集群启用指标的信息，请参阅 OpenShift Container Platform 文档中有关管理指标的内容。
- 要在 OpenShift Container Platform 上查看 Knative 组件的指标，您需要集群管理员权限，并访问 Web 控制台 **Administrator** 视角。

警告

如果使用 mTLS 启用 Service Mesh，则 Knative Serving 的指标会被默认禁用，因为 Service Mesh 会防止 Prometheus 提取指标。

有关解决这个问题的详情，请参阅在使用带有 mTLS 的 Service Mesh 时启用 Knative Serving 指标。

提取指标不会影响 Knative 服务的自动扩展，因为提取请求不会通过激活器。因此，如果没有 pod 正在运行，则不会进行提取。

6.5.2. 控制器指标

以下指标由实施控制器逻辑的任何组件提供。这些指标显示协调操作的详细信息，以及将协调请求添加到工作队列的工作队列行为。

指标名称	描述	类型	Tags	单位
work_queue_depth	工作队列的深度。	量表	**reconciler**	整数（无单位）
reconcile_count	协调操作的数量。	计数	**reconciler**, **success**	整数（无单位）
reconcile_latency	协调操作的延迟。	Histogram	**reconciler**, **success**	Milliseconds
workqueue_adds_total	由工作队列处理的添加操作总数。	计数	**name**	整数（无单位）
workqueue_queue_latency_seconds	在请求之前，项目保留在工作队列中的时长。	Histogram	**name**	秒
workqueue_retries_total	工作队列处理的重试总数。	计数	**name**	整数（无单位）
workqueue_work_duration_seconds	处理和从工作队列中项目所需的时间。	Histogram	**name**	秒
workqueue_unfinished_work_seconds	未完成的工作队列项目的时间长度。	Histogram	**name**	秒
workqueue_longest_running_processor_seconds	在处理中的、未完成的工作队列项的最长时间。	Histogram	**name**	秒

6.5.3. Webhook 指标

Webhook 指标报告有关操作的有用信息。例如，如果大量操作失败，这可能表示用户创建的资源出现问题。

指标名称	描述	类型	Tags	单位

指标名称	描述	类型	Tags	单位
request_count	路由到 webhook 的请求数。	计数	**admission_allowed**, **kind_group**, **kind_kind**, **kind_version**, **request_operation**, **resource_group**, **resource_namespace**, **resource_resource**, **resource_version**	整数（无单位）
request_latencies	Webhook 请求的响应时间。	Histogram	**admission_allowed**, **kind_group**, **kind_kind**, **kind_version**, **request_operation**, **resource_group**, **resource_namespace**, **resource_resource**, **resource_version**	Milliseconds

6.5.4. Knative Eventing 指标

集群管理员可查看 Knative Eventing 组件的以下指标。

通过聚合 HTTP 代码的指标，事件可以分为两类：成功事件 (2xx) 和失败的事件 (5xx) 。

6.5.4.1. 代理入口指标

您可以使用以下指标调试代理 ingress，请参阅它的执行方式，以及哪些事件由 ingress 组件分配。

指标名称	描述	类型	Tags	单位

指标名称	描述	类型	Tags	单位
event_count	代理接收的事件数。	计数	**broker_name**, **event_type**, **namespace_name**, **response_code**, **response_code_class**, **unique_name**	整数（无单位）
event_dispatch_latencies	将事件发送到频道的时间。	Histogram	**broker_name**, **event_type**, **namespace_name**, **response_code**, **response_code_class**, **unique_name**	Milliseconds

6.5.4.2. 代理过滤指标

您可以使用以下指标调试代理过滤器，查看它们的执行方式，以及过滤器正在分配哪些事件。您还可以测量事件的过滤操作的延迟。

指标名称	描述	类型	Tags	单位
event_count	代理接收的事件数。	计数	**broker_name**, **container_name**, **filter_type**, **namespace_name**, **response_code**, **response_code_class**, **trigger_name**, **unique_name**	整数（无单位）
event_dispatch_latencies	将事件发送到频道的时间。	Histogram	**broker_name**, **container_name**, **filter_type**, **namespace_name**, **response_code**, **response_code_class**, **trigger_name**, **unique_name**	Milliseconds

指标名称	描述	类型	Tags	单位
event_processing_latencies	将事件分配给触发器订阅者前处理事件所需的时间。	Histogram	**broker_name**, **container_name**, **filter_type**, **namespace_name**, **trigger_name**, **unique_name**	Milliseconds

6.5.4.3. InMemoryChannel 分配程序指标

您可以使用以下指标调试 **InMemoryChannel** 频道，查看它们的运行方式，并查看频道正在分配哪些事件。

指标名称	描述	类型	Tags	单位
event_count	**InMemoryChannel** 频道发送的事件数量。	计数	**broker_name**, **container_name**, **filter_type**, **namespace_name**, **response_code**, **response_code_class**, **trigger_name**, **unique_name**	整数（无单位）
event_dispatch_latencies	从 **InMemoryChannel** 频道分配事件的时间。	Histogram	**broker_name**, **container_name**, **filter_type**, **namespace_name**, **response_code**, **response_code_class**, **trigger_name**, **unique_name**	Milliseconds

6.5.4.4. 事件源指标

您可以使用以下指标验证事件是否从事件源发送到连接的事件接收器（sink）。

指标名称	描述	类型	Tags	单位

指标名称	描述	类型	Tags	单位
event_count	事件源发送的事件数。	计数	**broker_name**, **container_name**, **filter_type**, **namespace_name**, **response_code**, **response_code_class**, **trigger_name**, **unique_name**	整数（无单位）
retry_event_count	事件源在最初发送失败后发送的重试事件数量。	计数	**event_source**, **event_type**, **name**, **namespace_name**, **resource_group**, **response_code**, **response_code_class**, **response_error**, **response_timeout**	整数（无单位）

6.5.5. Knative Serving 指标

集群管理员可查看 Knative Serving 组件的以下指标。

6.5.5.1. 激活器指标

您可以使用以下指标了解应用在流量通过激活器时如何响应。

指标名称	描述	类型	Tags	单位
request_concurrency	路由到激活器的并发请求数，或者报告周期内平均并发请求数。	量表	**configuration_name**, **container_name**, **namespace_name**, **pod_name**, **revision_name**, **service_name**	整数（无单位）

指标名称	描述	类型	Tags	单位
request_count	要激活的请求数。这些是从活动器处理程序实现的请求。	计数	**configuration_name**, **container_name**, **namespace_name**, **pod_name**, **response_code**, **response_code_class**, **revision_name**, **service_name**,	整数（无单位）
request_latencies	已实现的路由请求的响应时间（毫秒）。	Histogram	**configuration_name**, **container_name**, **namespace_name**, **pod_name**, **response_code**, **response_code_class**, **revision_name**, **service_name**	Milliseconds

6.5.5.2. 自动缩放器指标

自动缩放器组件会公开多个与每个修订版本自动扩展行为相关的指标。例如，在任何给定时间，您可以监控自动扩展尝试为服务分配的目标 pod 数量，在 stable 窗口中每秒请求平均数量，或者如果您使用 Knative pod 自动缩放器 (KPA)，自动扩展是否处于 panic 模式。

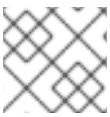
指标名称	描述	类型	Tags	单位
desired_pods	自动缩放器尝试为服务分配的 pod 数量。	量表	**configuration_name**, **namespace_name**, **revision_name**, **service_name**	整数（无单位）
excess_burst_capacity	过量激增容量在稳定窗口中提供。	量表	**configuration_name**, **namespace_name**, **revision_name**, **service_name**	整数（无单位）

指标名称	描述	类型	Tags	单位
stable_request_concurrency	每个通过稳定窗口观察到的 pod 的平均请求数。	量表	**configuration_name**, **namespace_name**, **revision_name**, **service_name**	整数（无单位）
panic_request_concurrency	每个观察到的 pod 的平均请求数通过 panic 窗口。	量表	**configuration_name**, **namespace_name**, **revision_name**, **service_name**	整数（无单位）
target_concurrency_per_pod	自动缩放器尝试发送到每个容器集的并发请求数。	量表	**configuration_name**, **namespace_name**, **revision_name**, **service_name**	整数（无单位）
stable_requests_per_second	通过 stable 窗口中每个观察到的 pod 的平均请求数每秒数。	量表	**configuration_name**, **namespace_name**, **revision_name**, **service_name**	整数（无单位）
panic_requests_per_second	每个通过 panic 窗口观察到的 pod 平均请求数每秒数。	量表	**configuration_name**, **namespace_name**, **revision_name**, **service_name**	整数（无单位）
target_requests_per_second	自动缩放器针对每个 Pod 的目标请求数。	量表	**configuration_name**, **namespace_name**, **revision_name**, **service_name**	整数（无单位）
panic_mode	如果自动扩展器处于 panic 模式，则这个值为 **1**，如果自动扩展器没有处于 panic 模式，则代表 **0**。	量表	**configuration_name**, **namespace_name**, **revision_name**, **service_name**	整数（无单位）

指标名称	描述	类型	Tags	单位
requested_pods	自动缩放器从 Kubernetes 集群请求的 pod 数量。	量表	**configuration_name**, **namespace_name**, **revision_name**, **service_name**	整数（无单位）
actual_pods	分配且当前具有就绪状态的 pod 数量。	量表	**configuration_name**, **namespace_name**, **revision_name**, **service_name**	整数（无单位）
not_ready_pods	处于未就绪状态的 pod 数量。	量表	**configuration_name**, **namespace_name**, **revision_name**, **service_name**	整数（无单位）
pending_pods	当前待处理的 pod 数量。	量表	**configuration_name**, **namespace_name**, **revision_name**, **service_name**	整数（无单位）
terminating_pods	当前终止的 pod 数量。	量表	**configuration_name**, **namespace_name**, **revision_name**, **service_name**	整数（无单位）

6.5.5.3. Go 运行时指标

每个 Knative Serving control plane 进程会发出多个 Go 运行时内存统计 (MemStats) 。



注意

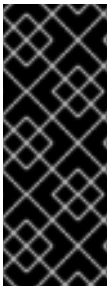
每个指标的 **name** 标签是一个空标签。

指标名称	描述	类型	Tags	单位

指标名称	描述	类型	Tags	单位
go_alloc	分配的堆对象的字节数。这个指标与 **heap_alloc** 相同。	量表	**name**	整数（无单位）
go_total_alloc	为堆对象分配的累积字节。	量表	**name**	整数（无单位）
go_sys	从操作系统获得的内存总量。	量表	**name**	整数（无单位）
go_lookups	运行时执行的指针查找数量。	量表	**name**	整数（无单位）
go_mallocs	分配的堆对象的累计数。	量表	**name**	整数（无单位）
go_frees	已释放的堆对象的累计数。	量表	**name**	整数（无单位）
go_heap_alloc	分配的堆对象的字节数。	量表	**name**	整数（无单位）
go_heap_sys	从操作系统获得的堆内存字节数。	量表	**name**	整数（无单位）
go_heap_idle	空闲、未使用的字节数。	量表	**name**	整数（无单位）
go_heap_in_use	当前正在使用的字节数。	量表	**name**	整数（无单位）
go_heap_released	返回到操作系统的物理内存字节数。	量表	**name**	整数（无单位）
go_heap_objects	分配的堆对象数量。	量表	**name**	整数（无单位）
go_stack_in_use	堆栈中当前正在使用的字节数。	量表	**name**	整数（无单位）
go_stack_sys	从操作系统获得的堆栈内存字节数。	量表	**name**	整数（无单位）

指标名称	描述	类型	Tags	单位
go_mspan_in_use	分配的 **mspan** 结构的字节数。	量表	**name**	整数（无单位）
go_mspan_sys	从操作系统获得的用于 **mspan** 结构的内存字节数。	量表	**name**	整数（无单位）
go_mcache_in_use	分配的 **mcache** 结构的字节数。	量表	**name**	整数（无单位）
go_mcache_sys	从操作系统获取的用于 **mcache** 结构的内存字节数。	量表	**name**	整数（无单位）
go_bucket_hash_sys	分析 bucket 哈希表中的内存字节数。	量表	**name**	整数（无单位）
go_gc_sys	垃圾回收元数据中的字节内存数量。	量表	**name**	整数（无单位）
go_other_sys	其它非堆运行时分配的内存字节数。	量表	**name**	整数（无单位）
go_next_gc	下一个垃圾回收周期的目标堆大小。	量表	**name**	整数（无单位）
go_last_gc	最后一次垃圾回收完成的时间（*Epoch* 或 Unix 时间）。	量表	**name**	Nanoseconds
go_total_gc_pause_ns	自程序启动以来，垃圾回收的 *stop-the-world* 暂停的累积时间。	量表	**name**	Nanoseconds
go_num_gc	完成的垃圾回收周期数量。	量表	**name**	整数（无单位）
go_num_forced_gc	由于应用调用垃圾回收功能而强制执行的垃圾回收周期数量。	量表	**name**	整数（无单位）
go_gc_cpu_fraction	程序启动后，被垃圾收集器使用的程序可用 CPU 时间的比例。	量表	**name**	整数（无单位）

6.6. 使用 OPENSHIFT SERVERLESS 的 METERING



重要

Metering 是一个已弃用的功能。弃用的功能仍然包含在 OpenShift Container Platform 中，并将继续被支持。但是，这个功能会在以后的发行版本中被删除，且不建议在新的部署中使用。

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

作为集群管理员，您可使用 Metering 来分析 OpenShift Serverless 集群中的情况。

如需有关 OpenShift Container Platform 的 metering 的更多信息，请参阅 关于 metering。

注意

Metering 目前不支持 IBM Z 和 IBM Power 系统。

6.6.1. 安装 metering

有关在 OpenShift Container Platform 上安装 metering 的详情请参考 Installing Metering。

6.6.2. Knative Serving metering 的数据源报告

以下数据源报告是 OpenShift Container Platform metering 如何使用 Knative Serving 的示例。

6.6.2.1. 数据源报告 Knative Serving 中的 CPU 使用量

此数据源报告提供了在报告期间每个 Knative 服务使用的总 CPU 秒数。

YAML 文件示例

```
apiVersion: metering.openshift.io/v1
kind: ReportDataSource
metadata:
  name: knative-service-cpu-usage
spec:
  prometheusMetricsImporter:
    query: >
      sum
        by(namespace,
          label_serving_knative_dev_service,
          label_serving_knative_dev_revision)
        (

label_replace(rate(container_cpu_usage_seconds_total{container!="POD",container!="",pod!=""}
[1m]), "pod", "$1", "pod", "(.*)")
          *
          on(pod, namespace)
          group_left(label_serving_knative_dev_service, label_serving_knative_dev_revision)
          kube_pod_labels{label_serving_knative_dev_service!=""}
        )
```

6.6.2.2. 数据源报告 Knative Serving 中的内存用量

此数据源报告提供每个 Knative 服务在报告期间的平均内存消耗。

YAML 文件示例

```
apiVersion: metering.openshift.io/v1
kind: ReportDataSource
metadata:
  name: knative-service-memory-usage
spec:
  prometheusMetricsImporter:
    query: >
      sum
          by(namespace,
             label_serving_knative_dev_service,
             label_serving_knative_dev_revision)
          (
            label_replace(container_memory_usage_bytes{container!="POD", container!="",pod!=""},
"pod", "$1", "pod", "(.*)")
            *
            on(pod, namespace)
            group_left(label_serving_knative_dev_service, label_serving_knative_dev_revision)
            kube_pod_labels{label_serving_knative_dev_service!=""}
          )
```

6.6.2.3. 为 Knative Serving metering 应用数据源报告

您可以使用以下命令应用数据源报告：

```
$ oc apply -f <data_source_report_name>.yaml
```

示例命令

```
$ oc apply -f knative-service-memory-usage.yaml
```

6.6.3. 对 Knative Serving metering 的查询

以下 **ReportQuery** 资源引用提供的 **ReportDataSource** 资源示例：

在 Knativelatesting 中查询 CPU 用量

```
apiVersion: metering.openshift.io/v1
kind: ReportQuery
metadata:
  name: knative-service-cpu-usage
spec:
  inputs:
  - name: ReportingStart
    type: time
  - name: ReportingEnd
    type: time
  - default: knative-service-cpu-usage
```

```
    name: KnativeServiceCpuUsageDataSource
    type: ReportDataSource
  columns:
  - name: period_start
    type: timestamp
    unit: date
  - name: period_end
    type: timestamp
    unit: date
  - name: namespace
    type: varchar
    unit: kubernetes_namespace
  - name: service
    type: varchar
  - name: data_start
    type: timestamp
    unit: date
  - name: data_end
    type: timestamp
    unit: date
  - name: service_cpu_seconds
    type: double
    unit: cpu_core_seconds
  query: |
    SELECT
      timestamp '{| default .Report.ReportingStart .Report.Inputs.ReportingStart| prestoTimestamp |}'
AS period_start,
      timestamp '{| default .Report.ReportingEnd .Report.Inputs.ReportingEnd | prestoTimestamp |}' AS
period_end,
      labels['namespace'] as project,
      labels['label_serving_knative_dev_service'] as service,
      min("timestamp") as data_start,
      max("timestamp") as data_end,
      sum(amount * "timeprecision") AS service_cpu_seconds
    FROM {| dataSourceTableName .Report.Inputs.KnativeServiceCpuUsageDataSource |}
    WHERE "timestamp" >= timestamp '{| default .Report.ReportingStart .Report.Inputs.ReportingStart
| prestoTimestamp |}'
    AND "timestamp" < timestamp '{| default .Report.ReportingEnd .Report.Inputs.ReportingEnd |
prestoTimestamp |}'
    GROUP BY labels['namespace'],labels['label_serving_knative_dev_service']
```

在 Knative Serving 中查询内存用量

```
apiVersion: metering.openshift.io/v1
kind: ReportQuery
metadata:
  name: knative-service-memory-usage
spec:
  inputs:
  - name: ReportingStart
    type: time
  - name: ReportingEnd
    type: time
  - default: knative-service-memory-usage
    name: KnativeServiceMemoryUsageDataSource
    type: ReportDataSource
```

```
  columns:
  - name: period_start
    type: timestamp
    unit: date
  - name: period_end
    type: timestamp
    unit: date
  - name: namespace
    type: varchar
    unit: kubernetes_namespace
  - name: service
    type: varchar
  - name: data_start
    type: timestamp
    unit: date
  - name: data_end
    type: timestamp
    unit: date
  - name: service_usage_memory_byte_seconds
    type: double
    unit: byte_seconds
  query: |
    SELECT
      timestamp '{| default .Report.ReportingStart .Report.Inputs.ReportingStart| prestoTimestamp |}'
AS period_start,
      timestamp '{| default .Report.ReportingEnd .Report.Inputs.ReportingEnd | prestoTimestamp |}' AS
period_end,
      labels['namespace'] as project,
      labels['label_serving_knative_dev_service'] as service,
      min("timestamp") as data_start,
      max("timestamp") as data_end,
      sum(amount * "timeprecision") AS service_usage_memory_byte_seconds
    FROM {| dataSourceTableName .Report.Inputs.KnativeServiceMemoryUsageDataSource |}
    WHERE "timestamp" >= timestamp '{| default .Report.ReportingStart .Report.Inputs.ReportingStart
| prestoTimestamp |}'
    AND "timestamp" < timestamp '{| default .Report.ReportingEnd .Report.Inputs.ReportingEnd |
prestoTimestamp |}'
    GROUP BY labels['namespace'],labels['label_serving_knative_dev_service']
```

6.6.3.1. 为 Knative Serving metering 应用查询

1. 应用 **ReportQuery** 资源：

```
$ oc apply -f <query_name>.yaml
```

示例命令

```
$ oc apply -f knative-service-memory-usage.yaml
```

6.6.4. Knative Serving 的 metering 报告

您可以通过创建 **Report** 资源来针对 Knative Serving 运行 metering 报告。在您运行报告前，您必须修改 **Report** 资源中的输入参数，以指定报告周期的开始和结束日期。

Report 资源示例

```
apiVersion: metering.openshift.io/v1
kind: Report
metadata:
  name: knative-service-cpu-usage
spec:
  reportingStart: '2019-06-01T00:00:00Z' 1
  reportingEnd: '2019-06-30T23:59:59Z' 2
  query: knative-service-cpu-usage 3
runImmediately: true
```

1. 报告的开始日期，格式为 ISO 8601。
2. 报告的结束日期，格式为 ISO 8601。
3. 用于 CPU 用量报告的 **knative-service-cpu-usage**，或用于内存用量报告的 **knative-service-memory-usage**。

6.6.4.1. 运行 metering 报告

1. 运行报告：

```
$ oc apply -f <report_name>.yml
```

2. 然后您可以检查报告：

```
$ oc get report
```

输出示例

```
NAME                        QUERY                       SCHEDULE   RUNNING    FAILED   LAST
REPORT TIME       AGE
knative-service-cpu-usage   knative-service-cpu-usage              Finished           2019-06-
30T23:59:59Z   10h
```

6.7. 高可用性

高可用性 (HA) 是 Kubernetes API 的标准功能，有助于确保在出现中断时 API 保持正常运行。在 HA 部署中，如果活跃控制器崩溃或被删除，另一个控制器就可以使用。此控制器会接管处理由现在不可用的控制器提供服务的 API。

OpenShift Serverless 中的 HA 可通过领导选举机制获得，该机制会在安装 Knative Serving 和 Eventing control plane 后默认启用。在使用领导选举 HA 模式时，控制器实例在需要前应该已在集群内调度并运行。这些控制器实例争用共享资源，即领导选举锁定。在任何给定时间可以访问领导选举机制锁定资源的控制器实例被称为领导 (leader)。

6.7.1. 为 Knative Serving 配置高可用性副本

默认情况下，Knative Serving **activator**, **autoscaler**, **autoscaler-hpa**, **controller**, **webhook**, **kourier-control**, 和 **kourier-gateway** 组件支持高可用性（HA）功能，它们默认被配置为有两个副本。您可以通

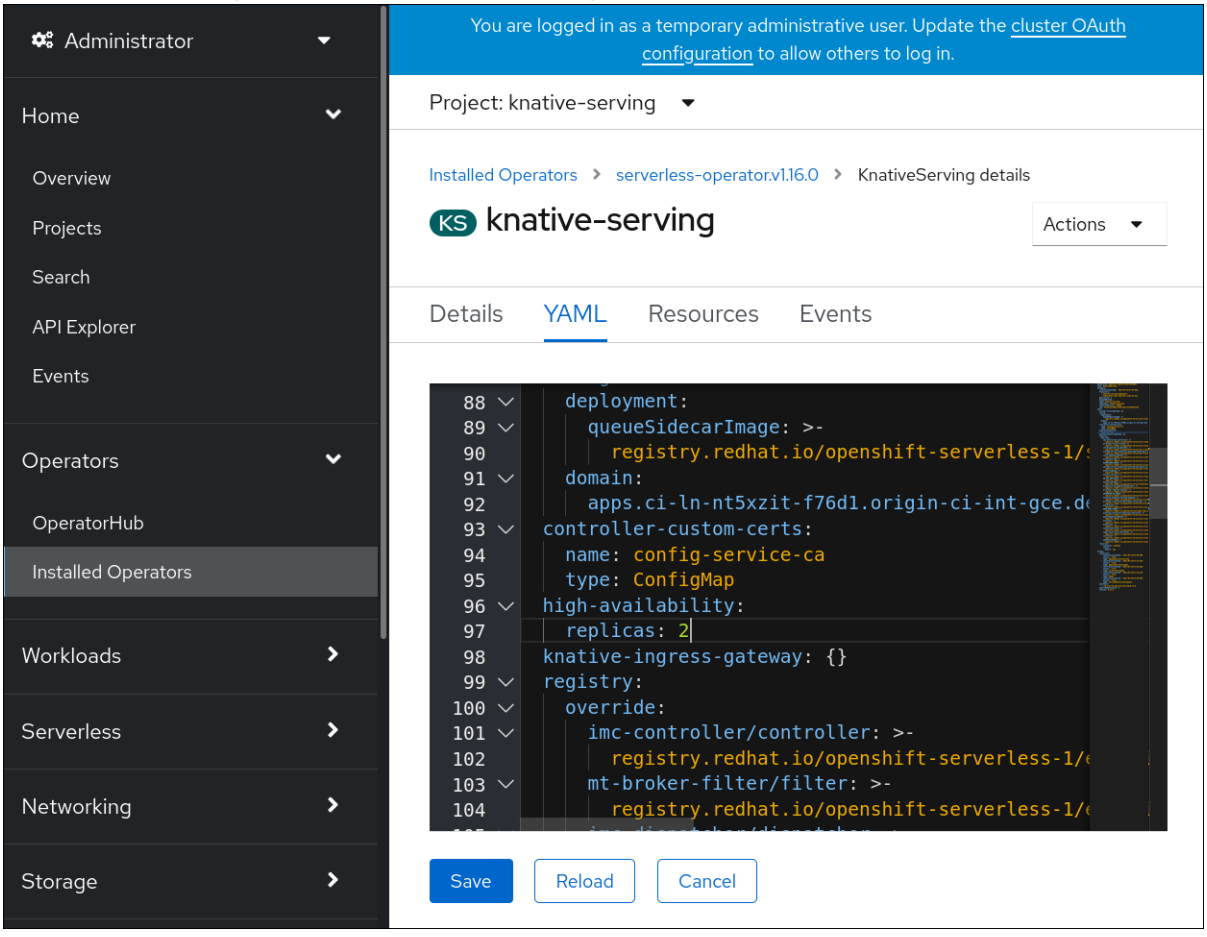
过修改 **KnativeServing** 自定义资源 (CR) 中的 **spec.high-availability.replicas** 值来更改这些组件的副本数。

先决条件

- 您可以使用集群管理员权限访问 OpenShift Container Platform 集群。
- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 已登录到 web 控制台。

流程

1. 在 OpenShift Container Platform web 控制台的 **Administrator** 视角中，进入 **OperatorHub → Installed Operators**。
2. 选择 **knative-serving** 命名空间。
3. 点 OpenShift Serverless Operator 的 **Provided APIs** 列表中的 **Knative Serving** 来进入 **Knative Serving** 选项卡。
4. 点 **knative-serving**，然后使用 **knative-serving** 页面中的 **YAML** 选项卡。



5. 修改 **KnativeServing** CR 中的副本数量：

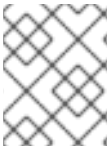
YAML 示例

```
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeServing
metadata:
```

```
  name: knative-serving
  namespace: knative-serving
spec:
  high-availability:
    replicas: 3
```

6.7.2. 为 Knative Eventing 配置高可用性副本

默认情况下，Knative Eventing **eventing-controller**、**eventing-webhook**、**imc-controller**、**imc-dispatcher** 和 **mt-broker-controller** 组件都会具有高可用性 (HA) 。您可以通过修改 **KnativeEventing** 自定义资源 (CR) 中的 **spec.high-availability.replicas** 值来更改这些组件的副本数。



注意

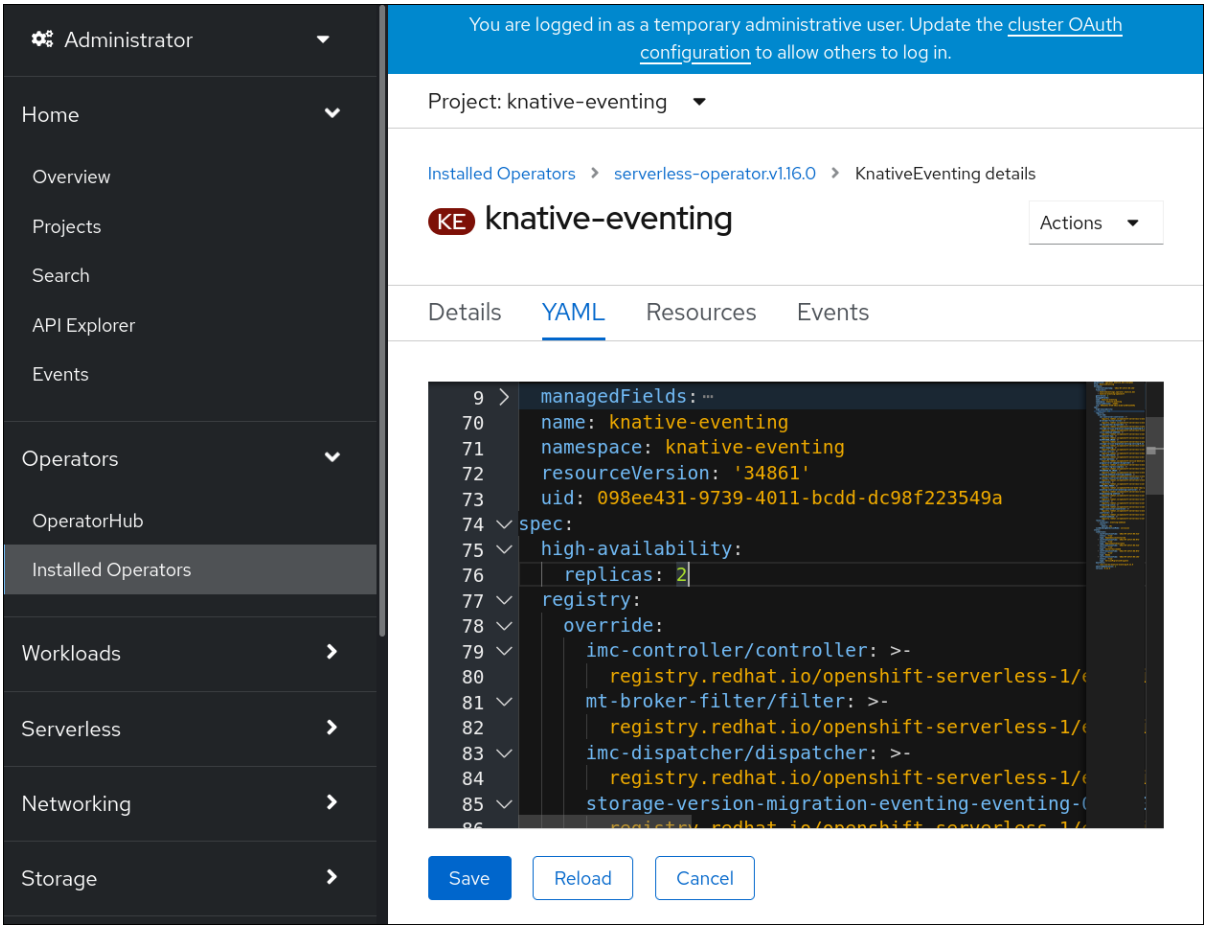
对于 Knative Eventing，HA 不会扩展 **mt-broker-filter** 和 **mt-broker-ingress** 部署。如果需要多个部署，请手动扩展这些组件。

先决条件

- 您可以使用集群管理员权限访问 OpenShift Container Platform 集群。
- 在集群中安装了 OpenShift Serverless Operator 和 Knative Eventing。

流程

1. 在 OpenShift Container Platform web 控制台的 **Administrator** 视角中，进入 **OperatorHub → Installed Operators**。
2. 选择 **knative-eventing** 命名空间。
3. 点 OpenShift Serverless Operator 的 **Provided APIs** 列表中的 **Knative Eventing** 来进入 **Knative Eventing** 选项卡。
4. 点 **knative-eventing**，然后进入 **knative-eventing** 页面中的 **YAML** 选项卡。



5. 修改 **KnativeEventing** CR 中的副本数量：

YAML 示例

```
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeEventing
metadata:
  name: knative-eventing
  namespace: knative-eventing
spec:
  high-availability:
    replicas: 3
```

6.7.3. 为 Knative Kafka 配置高可用性副本

默认情况下，Knative Kafka **kafka-controller** 和 **kafka-webhook-eventing** 组件都会具有高可用性 (HA)，这些组件默认配置为默认具有两个副本。您可以通过修改 **KnativeKafka** 自定义资源 (CR) 中的 **spec.high-availability.replicas** 值来更改这些组件的副本数。

先决条件

- 您可以使用集群管理员权限访问 OpenShift Container Platform 集群。
- 在集群中安装了 OpenShift Serverless Operator 和 Knative Kafka。

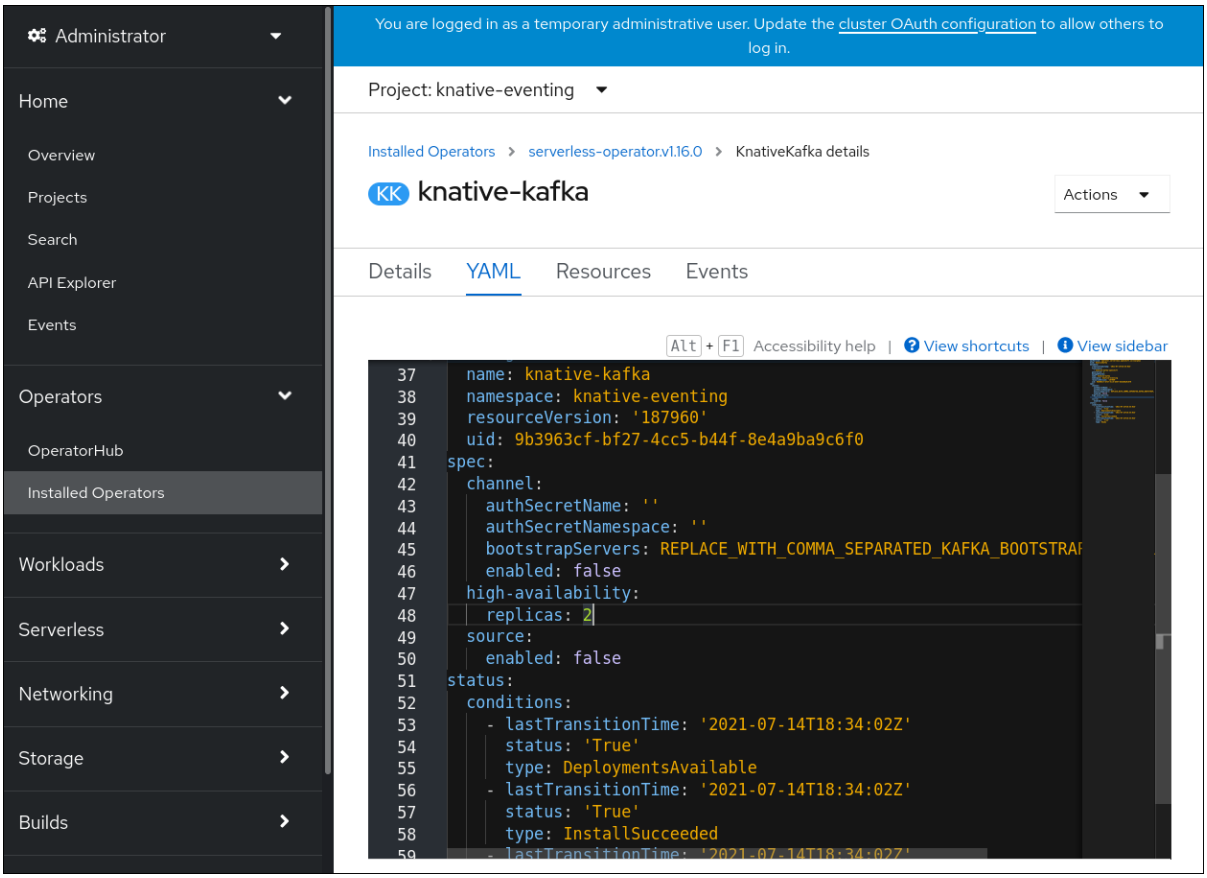
流程

1. 在 OpenShift Container Platform web 控制台的 **Administrator** 视角中，进入 **OperatorHub → Installed Operators**。

2. 选择 **knative-eventing** 命名空间。

3. 点 OpenShift Serverless Operator 的 **Provided APIs** 列表中的 **Knative Kafka** 进入 **Knative Kafka** 标签页。

4. 点 **knative-kafka**，然后进入 **knative-kafka** 页面中的 **YAML** 选项卡。



5. 修改 **KnativeKafka** CR 中的副本数量：

YAML 示例

```
apiVersion: operator.serverless.openshift.io/v1alpha1
kind: KnativeKafka
metadata:
  name: knative-kafka
  namespace: knative-eventing
spec:
  high-availability:
    replicas: 3
```

第 7 章 MONITOR

7.1. 在 OPENSHIFT SERVERLESS 中使用 OPENSHIFT LOGGING

7.1.1. 关于部署集群日志记录

OpenShift Container Platform 集群管理员可以使用 CLI 命令和 OpenShift Container Platform web 控制台安装 Elasticsearch Operator 和 Cluster Logging Operator，以此部署集群日志记录。安装 Operator 后，可创建 **ClusterLogging** 自定义资源（CR）以调度集群日志记录 pod 和支持集群日志记录所需的其他资源。Operator 负责部署、升级和维护集群日志记录。

ClusterLogging CR 定义包括日志记录堆栈的所有组件在内的完整集群日志记录环境，以收集、存储和视觉化日志。Cluster Logging Operator 监视集群日志记录 CR 并相应地调整日志记录部署。

管理员和应用程序开发人员可以查看他们具有查看访问权限的项目的日志。

7.1.2. 关于部署和配置集群日志记录

OpenShift Container Platform 集群日志记录已设计为可搭配默认配置使用，该配置针对中小型 OpenShift Container Platform 集群进行了调优。

以下安装说明包括一个示例 **ClusterLogging** 自定义资源（CR），您可以使用它来创建集群日志记录实例并配置集群日志记录环境。

如果要使用默认集群日志记录安装，可直接使用示例 CR。

如果要自定义部署，请根据需要对示例 CR 进行更改。下文介绍了在安装集群日志记录实例或安装后修改时可以进行的配置。请参阅“配置”部分来了解有关使用各个组件的更多信息，包括可以在 **ClusterLogging** 自定义资源之外进行的修改。

7.1.2.1. 配置和调优集群日志记录

您可以通过修改 **openshift-logging** 项目中部署的 **ClusterLogging** 自定义资源来配置集群日志记录环境。

您可以在安装时或安装后修改以下任何组件：

内存和 CPU

您可以使用有效的内存和 CPU 值修改 **resources** 块，以此调整各个组件的 CPU 和内存限值：

```
spec:
  logStore:
    elasticsearch:
      resources:
        limits:
          cpu:
          memory: 16Gi
        requests:
          cpu: 500m
          memory: 16Gi
      type: "elasticsearch"
  collection:
    logs:
      fluentd:
```

```
      resources:
        limits:
          cpu:
          memory:
        requests:
          cpu:
          memory:
      type: "fluentd"
  visualization:
    kibana:
      resources:
        limits:
          cpu:
          memory:
        requests:
          cpu:
          memory:
     type: kibana
  curation:
    curator:
      resources:
        limits:
          memory: 200Mi
        requests:
          cpu: 200m
          memory: 200Mi
      type: "curator"
```

Elasticsearch 存储

您可以使用 **storageClass name** 和 **size** 参数，为 Elasticsearch 集群配置持久性存储类和大小。Cluster Logging Operator 基于这些参数，为 Elasticsearch 集群中的每个数据节点创建一个持久性卷声明（PVC）。

```
spec:
  logStore:
    type: "elasticsearch"
    elasticsearch:
      nodeCount: 3
      storage:
        storageClassName: "gp2"
        size: "200G"
```

本例中指定，集群中的每个数据节点将绑定到请求 200G 的 gp2 存储的 PVC。每个主分片将由单个副本支持。

注意

省略 **storage** 块会导致部署中仅包含临时存储。

```
spec:
  logStore:
    type: "elasticsearch"
    elasticsearch:
      nodeCount: 3
      storage: {}
```

Elasticsearch 复制策略

您可以通过设置策略来定义如何在集群中的数据节点之间复制 Elasticsearch 分片：

- **FullRedundancy**。各个索引的分片完整复制到每个数据节点上。
- **MultipleRedundancy**。各个索引的分片分布到一半数据节点上。
- **SingleRedundancy**。各个分片具有单个副本。只要存在至少两个数据节点，日志就能始终可用且可恢复。
- **ZeroRedundancy**。所有分片均无副本。如果节点关闭或发生故障，则可能无法获得日志数据。

Curator 调度

以 cron 格式指定 Curator 的调度。

```
spec:
  curation:
  type: "curator"
  resources:
  curator:
    schedule: "30 3 * * *"
```

7.1.2.2. 修改后的 ClusterLogging 自定义资源示例

以下是使用前面描述的选项修改的 **ClusterLogging** 自定义资源的示例。

修改后的 ClusterLogging 自定义资源示例

```
apiVersion: "logging.openshift.io/v1"
kind: "ClusterLogging"
metadata:
  name: "instance"
  namespace: "openshift-logging"
spec:
  managementState: "Managed"
  logStore:
    type: "elasticsearch"
    retentionPolicy:
      application:
        maxAge: 1d
      infra:
```

```
      maxAge: 7d
    audit:
      maxAge: 7d
  elasticsearch:
    nodeCount: 3
    resources:
      limits:
        memory: 32Gi
      requests:
        cpu: 3
        memory: 32Gi
      storage:
        storageClassName: "gp2"
        size: "200G"
    redundancyPolicy: "SingleRedundancy"
visualization:
  type: "kibana"
  kibana:
    resources:
      limits:
        memory: 1Gi
      requests:
        cpu: 500m
        memory: 1Gi
    replicas: 1
curation:
  type: "curator"
  curator:
    resources:
      limits:
        memory: 200Mi
      requests:
        cpu: 200m
        memory: 200Mi
    schedule: "*/5 * * * *"
collection:
  logs:
    type: "fluentd"
    fluentd:
      resources:
        limits:
          memory: 1Gi
        requests:
          cpu: 200m
          memory: 1Gi
```

7.1.3. 使用集群日志来查找 Knative Serving 组件的日志

先决条件

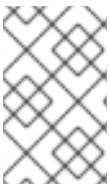
- 安装 OpenShift CLI (**oc**) 。

流程

1. 获取 Kibana 路由：

```
$ oc -n openshift-logging get route kibana
```

2. 使用路由的 URL 导航到 Kibana 仪表板并登录。
3. 检查是否将索引设置为 **.all**。如果索引未设置为 **.all**，则只会列出 OpenShift Container Platform 系统日志。
4. 使用 **knative-serving** 命名空间过滤日志。在搜索框中输入 **kubernetes.namespace_name:knative-serving** 以过滤结果。



注意

Knative Serving 默认使用结构化日志记录。您可以通过自定义集群日志记录 Fluentd 设置来启用这些日志的解析。这可使日志更易搜索，并且能够在日志级别进行过滤以快速识别问题。

7.1.4. 使用集群日志来查找通过 Knative Serving 部署的服务的日志

使用 OpenShift Cluster Logging，应用程序写入控制台的日志将在 Elasticsearch 中收集。以下流程概述了如何使用 Knative Serving 将这些功能应用到所部署的应用程序中。

先决条件

- 安装 OpenShift CLI (**oc**) 。

流程

1. 获取 Kibana 路由：

```
$ oc -n openshift-logging get route kibana
```

2. 使用路由的 URL 导航到 Kibana 仪表板并登录。
3. 检查是否将索引设置为 **.all**。如果索引未设置为 **.all**，则只会列出 OpenShift 系统日志。
4. 使用 **knative-serving** 命名空间过滤日志。在搜索框中输入服务的过滤器来过滤结果。

   过滤**器示例**

```
kubernetes.namespace_name:default AND kubernetes.labels.serving_knative_dev\/service:{service_name}
```

   除此之外还可使用 **/configuration** 或 **/revision** 来过滤。

5. 您可使用 **kubernetes.container_name:<user-container>** 来缩小搜索范围，只显示由您的应用程序生成的日志。否则，会显示来自 queue-proxy 的日志。

注意

在应用程序中使用基于 JSON 的结构化日志记录，以便在生产环境中快速过滤这些日志。

7.2. SERVERLESS 开发人员指标

指标 (metrics) 使开发人员能够监控 Knative 服务的运行情况。您可以使用 OpenShift Container Platform 监控堆栈记录并查看 Knative 服务的健康检查和指标。

您可以通过在 OpenShift Container Platform Web 控制台 **Developer** 视角中导航到 **Dashboards** 来查看 OpenShift Serverless 的不同指标。

警告

如果使用 mTLS 启用 Service Mesh，则 Knative Serving 的指标会被默认禁用，因为 Service Mesh 会防止 Prometheus 提取指标。

有关解决这个问题的详情，请参阅在使用带有 mTLS 的 Service Mesh 时启用 Knative Serving 指标。

提取指标不会影响 Knative 服务的自动扩展，因为提取请求不会通过激活器。因此，如果没有 pod 正在运行，则不会进行提取。

7.2.1. Knative 服务指标默认公开

表 7.1. 在端口 9090 上为每个 Knative 服务公开的指标

指标名称、单元和类型	描述	指标标签
queue_requests_per_second 指标单元：无维度 指标类型：量表	每秒到达队列代理的请求数。 公式：**stats.RequestCount / r.reportingPeriodSeconds** **stats.RequestCount** 直接从给定报告持续时间的网络 **pkg** stats 中计算。	destination_configuration="event-display", destination_namespace="pingsource1", destination_pod="event-display-00001-deployment-6b455479cb-75p6w", destination_revision="event-display-00001"
queue_proxied_operations_per_second 指标单元：无维度 指标类型：量表	每秒代理请求数. 公式：**stats.ProxiedRequestCount / r.reportingPeriodSeconds** **stats.ProxiedRequestCount** 直接从给定报告持续时间的网络 **pkg** stats 中计算。	

指标名称、单元和类型	描述	指标标签
queue_average_concurrent_requests 指标单元：无维度 指标类型：量表	此 pod 当前处理的请求数。 在网络 **pkg** 侧计算平均并发，如下所示： ● 当发生 **req** 更改时，将计算更改之间的时间传送时间。根据结果，计算超过 delta 的当前并发数，并添加到当前计算的并发数。此外，还保留了 deltas 的总和。 增量型的当前并发性计算如下： **global_concurrency** delta ● 每次完成报告时，都会重置总和和当前计算的并发性。 ● 在报告平均并发时，当前计算的并发性被除以 deltas 的总和。 ● 当一个新请求进入时，全局并发计数器会增加。当请求完成后，计数器会减少。	destination_configuration="event-display", destination_namespace="pingsource1", destination_pod="event-display-00001-deployment-6b455479cb-75p6w", destination_revision="event-display-00001"
queue_average_proxied_concurrent_requests 指标单元：无维度 指标类型：量表	当前由此 pod 处理的代理请求数： **stats.AverageProxiedConcurrency**	destination_configuration="event-display", destination_namespace="pingsource1", destination_pod="event-display-00001-deployment-6b455479cb-75p6w", destination_revision="event-display-00001"
process_uptime 指标单元： 秒 指标类型：量表	进程启动的秒数。	destination_configuration="event-display", destination_namespace="pingsource1", destination_pod="event-display-00001-deployment-6b455479cb-75p6w", destination_revision="event-display-00001"

表 7.2. 在端口 9091 上为每个 Knative 服务公开的指标

指标名称、单元和类型	描述	指标标签
request_count 指标单元：无维度 指标类型：计数器	路由到 **queue-proxy** 的请求数。	configuration_name="event-display", container_name="queue-proxy", namespace_name="apiserversource1", pod_name="event-display-00001-deployment-658fd4f9cf-qcnr5", response_code="200", response_code_class="2xx", revision_name="event-display-00001", service_name="event-display"
request_latencies 指标单元：毫秒 指标类型：togram	以毫秒为单位的响应时间。	configuration_name="event-display", container_name="queue-proxy", namespace_name="apiserversource1", pod_name="event-display-00001-deployment-658fd4f9cf-qcnr5", response_code="200", response_code_class="2xx", revision_name="event-display-00001", service_name="event-display"
app_request_count 指标单元：无维度 指标类型：计数器	路由到 **user-container** 的请求数。	configuration_name="event-display", container_name="queue-proxy", namespace_name="apiserversource1", pod_name="event-display-00001-deployment-658fd4f9cf-qcnr5", response_code="200", response_code_class="2xx", revision_name="event-display-00001", service_name="event-display"
app_request_latencies 指标单元：毫秒 指标类型：togram	以毫秒为单位的响应时间。	configuration_name="event-display", container_name="queue-proxy", namespace_name="apiserversource1", pod_name="event-display-00001-deployment-658fd4f9cf-qcnr5", response_code="200", response_code_class="2xx", revision_name="event-display-00001", service_name="event-display"

指标名称、单元和类型	描述	指标标签
queue_depth 指标单元：无维度 指标类型：量表	服务和等待队列中的当前项目数，或者如果无限并发，则不报告。使用 **breaker.inFlight**。	configuration_name="event-display", container_name="queue-proxy", namespace_name="apiserversource1", pod_name="event-display-00001-deployment-658fd4f9cf-qcnr5", response_code="200", response_code_class="2xx", revision_name="event-display-00001", service_name="event-display"

7.2.2. 带有自定义应用程序指标的 Knative 服务

您可以扩展 Knative 服务导出的指标集合。具体的实施取决于您的应用和使用的语言。

以下列表实施了一个 Go 应用示例，它导出处理的事件计数自定义指标。

```
package main

import (
  "fmt"
  "log"
  "net/http"
  "os"

  "github.com/prometheus/client_golang/prometheus" (1)
  "github.com/prometheus/client_golang/prometheus/promauto"
  "github.com/prometheus/client_golang/prometheus/promhttp"
)

var (
  opsProcessed = promauto.NewCounter(prometheus.CounterOpts{ (2)
    Name: "myapp_processed_ops_total",
    Help: "The total number of processed events",
  })
)


func handler(w http.ResponseWriter, r *http.Request) {
  log.Print("helloworld: received a request")
  target := os.Getenv("TARGET")
  if target == "" {
    target = "World"
  }
  fmt.Fprintf(w, "Hello %s!\n", target)
  opsProcessed.Inc() (3)
}

func main() {
  log.Print("helloworld: starting server...")
```

```
  port := os.Getenv("PORT")
  if port == "" {
    port = "8080"
  }

  http.HandleFunc("/", handler)

  // Separate server for metrics requests
  go func() { 4
    mux := http.NewServeMux()
    server := &http.Server{
      Addr: fmt.Sprintf(":%s", "9095"),
      Handler: mux,
    }
    mux.Handle("/metrics", promhttp.Handler())
    log.Printf("prometheus: listening on port %s", 9095)
    log.Fatal(server.ListenAndServe())
  }()

   // Use same port as normal requests for metrics
  //http.Handle("/metrics", promhttp.Handler()) 5
  log.Printf("helloworld: listening on port %s", port)
  log.Fatal(http.ListenAndServe(fmt.Sprintf(":%s", port), nil))
}
```

1 包含 Prometheus 软件包。

2 定义 **opsProcessed** 指标。

3 递增 **opsProcessed** 指标。

4 将配置为将单独的服务器用于指标请求.

5 将配置为使用与指标和**指标**子路径正常请求相同的端口。

7.2.3. 配置提取自定义指标

自定义指标提取由专门用于用户工作负载监控的 Prometheus 实例执行。启用用户工作负载监控并创建应用程序后，您需要一个配置来定义监控堆栈提取指标的方式。

以下示例配置为您的应用程序定义了 **ksvc** 并配置服务监控器。确切的配置取决于您的应用程序以及它如何导出指标。

```
apiVersion: serving.knative.dev/v1 1
kind: Service
metadata:
  name: helloworld-go
spec:
  template:
    metadata:
      labels:
        app: helloworld-go
      annotations:
```

```
    spec:
      containers:
      - image: docker.io/skonto/helloworld-go:metrics
        resources:
          requests:
            cpu: "200m"
        env:
        - name: TARGET
          value: "Go Sample v1"
---
apiVersion: monitoring.coreos.com/v1 (2)
kind: ServiceMonitor
metadata:
  labels:
  name: helloworld-go-sm
spec:
  endpoints:
  - port: queue-proxy-metrics
    scheme: http
  - port: app-metrics
    scheme: http
  namespaceSelector: {}
  selector:
    matchLabels:
      name:  helloworld-go-sm
---
apiVersion: v1 (3)
kind: Service
metadata:
  labels:
    name:  helloworld-go-sm
  name:  helloworld-go-sm
spec:
  ports:
  - name: queue-proxy-metrics
    port: 9091
    protocol: TCP
    targetPort: 9091
  - name: app-metrics
    port: 9095
    protocol: TCP
    targetPort: 9095
  selector:
    serving.knative.dev/service: helloworld-go
  type: ClusterIP
```

1. 应用程序规格。
2. 配置提取应用程序的指标。
3. 提取指标的方式的配置。

7.2.4. 检查服务的指标

在将应用配置为导出指标和监控堆栈以提取它们后，您可以在 web 控制台中查看指标数据。

先决条件

- 已登陆到 OpenShift Container Platform Web 控制台。
- 安装了 OpenShift Serverless Operator 和 Knative Serving。

流程

1. 可选：针对应用程序运行请求，您可以在指标中看到：

```
$ hello_route=$(oc get ksvc helloworld-go -n ns1 -o jsonpath='{.status.url}') && \
    curl $hello_route
```

输**出示例**

```
Hello Go Sample v1!
```

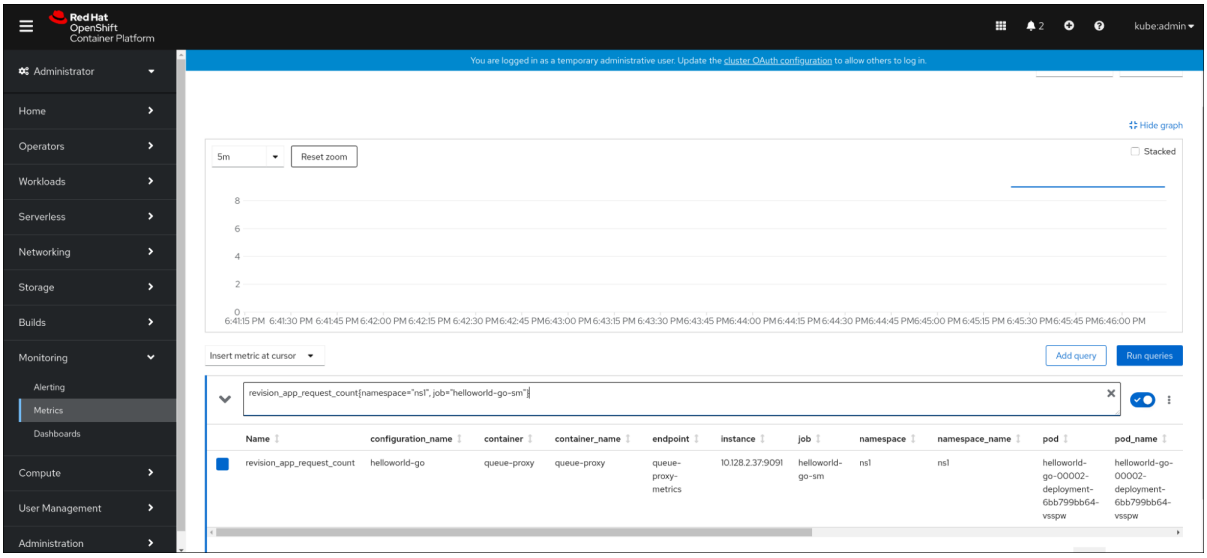
2. 在 Web 控制台中，进入到 **Monitoring → Metrics** 界面。
3. 在输入字段中，输入您要观察到的指标的查询，例如：

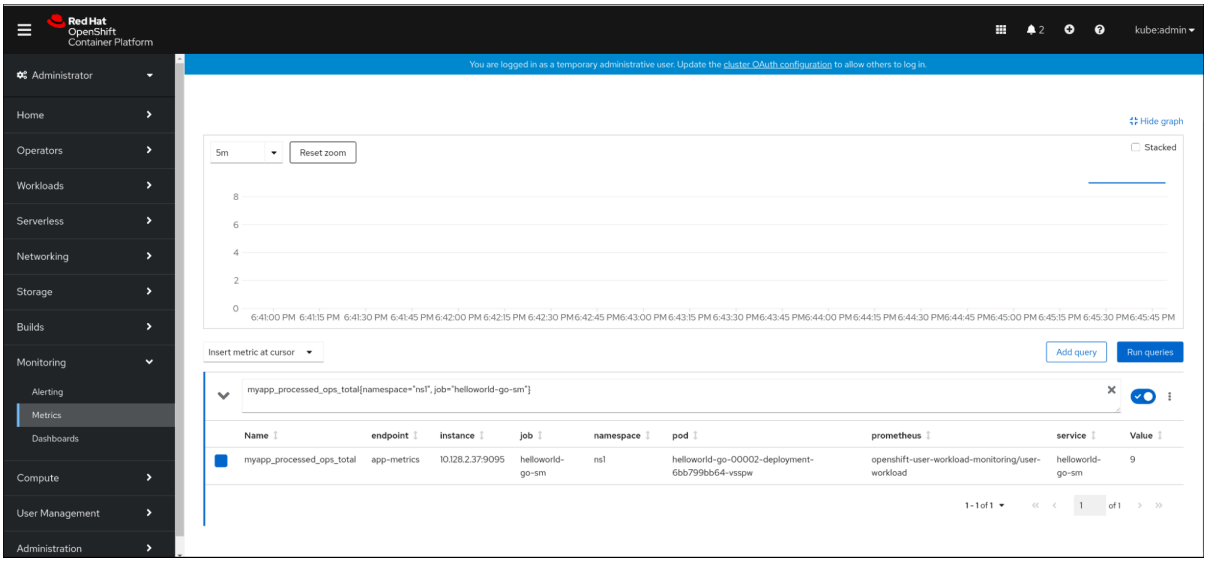
```
revision_app_request_count{namespace="ns1", job="helloworld-go-sm"}
```

另一个示例：

```
myapp_processed_ops_total{namespace="ns1", job="helloworld-go-sm"}
```

4. 观察视觉化的指标：





7.2.4.1. 队列代理指标

每个 Knative 服务都有一个代理容器，用于代理到应用程序容器的连接。报告多个用于队列代理性能的指标。

您可以使用以下指标来测量请求是否排入代理端，并在应用一侧服务请求的实际延迟。

指标名称	描述	类型	Tags	单位
revision_request_count	路由到 **queue-proxy** pod 的请求数。	计数	**configuration_name**, **container_name**, **namespace_name**, **pod_name**, **response_code**, **response_code_class**, **revision_name**, **service_name**	整数（无单位）
revision_request_latencies	修订请求的响应时间。	Histogram	**configuration_name**, **container_name**, **namespace_name**, **pod_name**, **response_code**, **response_code_class**, **revision_name**, **service_name**	Milliseconds

指标名称	描述	类型	Tags	单位
revision_app_request_count	路由到 **user-container** 容器集的请求数。	计数	**configuration_name**, **container_name**, **namespace_name**, **pod_name**, **response_code**, **response_code_class**, **revision_name**, **service_name**	整数（无单位）
revision_app_request_latencies	修订应用程序请求的响应时间。	Histogram	**configuration_name**, **namespace_name**, **pod_name**, **response_code**, **response_code_class**, **revision_name**, **service_name**	Milliseconds
revision_queue_depth	当前在 **serving** 和 **waiting** 队列中的项的数量。如果配置了无限并发，则不会报告此指标。	量表	**configuration_name**, **event-display**, **container_name**, **namespace_name**, **pod_name**, **response_code_class**, **revision_name**, **service_name**	整数（无单位）

7.2.5. 在仪表板中检查服务的指标

您可以使用专用的仪表板来按命名空间聚合队列代理指标，以检查指标。

先决条件

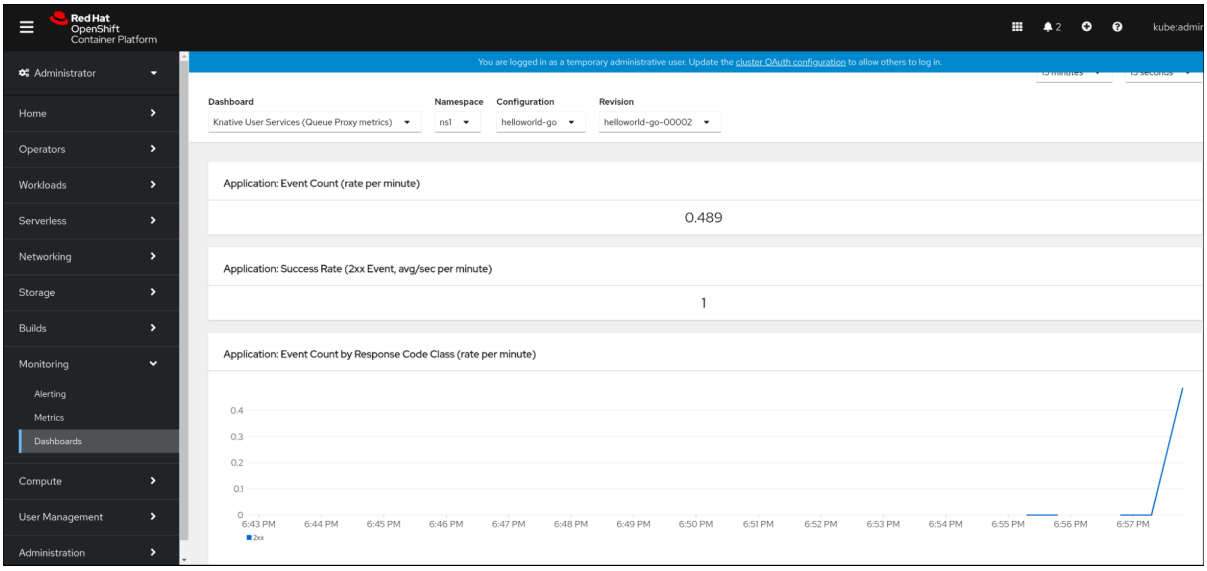
- 已登陆到 OpenShift Container Platform Web 控制台。
- 安装了 OpenShift Serverless Operator 和 Knative Serving。

流程

1. 在 Web 控制台中，进入到 **Monitoring → Metrics** 界面。
2. 选择 **Knative User Services(Queue Proxy metrics)** 仪表板。

3. 选择与应用程序对应的 **Namespace**、**Configuration** 和 **Revision**。

4. 观察视觉化的指标：



7.2.6. 其他资源

- 监控概述
- 为用户定义的项目启用监控
- 指定如何监控服务

第 8 章 跟踪请求

分布式追踪记录了一个请求在组成一个应用程序的多个微服务间的路径。它被用来将不同工作单元的信息串联在一起，理解分布式事务中整个事件链。工作单元可能会在不同进程或主机中执行。

8.1. 分布式追踪概述

作为服务所有者，您可以使用分布式追踪来检测您的服务，以收集与服务架构相关的信息。您可以使用分布式追踪来监控、网络性能分析，并对现代、云原生的基于微服务的应用中组件之间的交互进行故障排除。

通过分布式追踪，您可以执行以下功能：

- 监控分布式事务
- 优化性能和延迟时间
- 执行根原因分析

Red Hat OpenShift distributed tracing 包括两个主要组件：

- **Red Hat OpenShift distributed tracing Platform** - 此组件基于开源 Jaeger 项目。
- **Red Hat OpenShift distributed tracing 数据收集** - 此组件基于开源 OpenTelemetry 项目。

这两个组件都基于厂商中立的 OpenTracing API 和工具。

8.2. 使用 RED HAT OPENSHIFT DISTRIBUTED TRACING 启用分布式追踪

Red Hat OpenShift distributed tracing 由几个组件组成，它们一起收集、存储和显示追踪数据。您可以使用 Red Hat OpenShift Serverless 的 Red Hat OpenShift distributed tracing 监控无服务器应用程序并进行故障排除。

先决条件

- 您可以访问具有集群管理员权限的 OpenShift Container Platform 帐户。
- 还没有安装 OpenShift Serverless Operator 和 Knative Serving。这些需要在 Red Hat OpenShift distributed tracing 安装后安装。
- 您已按照 OpenShift Container Platform "Installing distributed tracing" 文档 安装了 Red Hat OpenShift distributed tracing。
- 已安装 OpenShift CLI(**oc**)。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。

流程

1. 创建 **OpenTelemetryCollector** 自定义资源 (CR)：

   OpenTelemetryCollector CR 示例

   ```
   apiVersion: opentelemetry.io/v1alpha1
   ```

```
kind: OpenTelemetryCollector
metadata:
  name: cluster-collector
  namespace: <namespace>
spec:
  mode: deployment
  config: |
    receivers:
      zipkin:
    processors:
    exporters:
      jaeger:
        endpoint: jaeger-all-in-one-inmemory-collector-headless.tracing-system.svc:14250
        tls:
          ca_file: "/var/run/secrets/kubernetes.io/serviceaccount/service-ca.crt"
      logging:
    service:
      pipelines:
        traces:
          receivers: [zipkin]
          processors: []
          exporters: [jaeger, logging]
```

2. 验证有两个 pod 在安装了 Red Hat OpenShift distributed tracing 的命名空间中运行：

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

输**出示例**

```
NAME                                  READY   STATUS    RESTARTS   AGE
cluster-collector-collector-85c766b5c-b5g99   1/1     Running   0          5m56s
jaeger-all-in-one-inmemory-ccbc9df4b-ndkl5    2/2     Running   0          15m
```

3. 验证是否已创建以下无头服务：

```
$ oc get svc -n <namespace> | grep headless
```

输**出示例**

```
cluster-collector-collector-headless          ClusterIP   None            <none>        9411/TCP
7m28s
jaeger-all-in-one-inmemory-collector-headless   ClusterIP   None            <none>
9411/TCP,14250/TCP,14267/TCP,14268/TCP   16m
```

这些服务用于配置 Jaeger 和 Knative Serving。Jaeger 服务的名称可能会有所不同。

4. 按照"安装 OpenShift Serverless Operator"文档安装 OpenShift Serverless Operator。

5. 通过创建以下 **KnativeServing** CR 来安装 Knative Serving：

KnativeServing CR 示例

```
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeServing
```

```
metadata:
   name: knative-serving
   namespace: knative-serving
spec:
 config:
   tracing:
     backend: "zipkin"
     zipkin-endpoint: "http://cluster-collector-collector-headless.tracing-
system.svc:9411/api/v2/spans"
     debug: "true"
     sample-rate: "0.1" 1
```

1 **sample-rate** 定义抽样概率。**sample-rate: "0.1"** 表示在 10 个 trace 中会抽样 1 个。

6. 创建 Knative 服务：

服务示例

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
 name: helloworld-go
spec:
 template:
   metadata:
     labels:
       app: helloworld-go
     annotations:
       autoscaling.knative.dev/minScale: "1"
       autoscaling.knative.dev/target: "1"
   spec:
     containers:
     - image: quay.io/openshift-knative/helloworld:v1.2
       imagePullPolicy: Always
       resources:
         requests:
           cpu: "200m"
       env:
       - name: TARGET
         value: "Go Sample v1"
```

7. 向服务发出一些请求：

HTTPS 请求示例

```
$ curl https://helloworld-go.example.com
```

8. 获取 Jaeger web 控制台的 URL：

示例命令

```
$ oc get route jaeger-all-in-one-inmemory  -o jsonpath='{.spec.host}' -n <namespace>
```

现在，您可以使用 Jaeger 控制台检查 trace。

8.3. 使用 JAEGER 启用分布式追踪

如果您不想安装 Red Hat OpenShift distributed tracing 的所有组件，您仍可使用带有 OpenShift Serverless 的 OpenShift Container Platform 上的分布式追踪。要做到这一点，您必须安装并配置 Jaeger 作为独立集成。

先决条件

- 您可以访问具有集群管理员权限的 OpenShift Container Platform 帐户。
- 安装了 OpenShift Serverless Operator 和 Knative Serving。
- 已安装 Red Hat OpenShift distributed tracing platform Operator。
- 已安装 OpenShift CLI(**oc**)。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。

流程

1. 创建并应用包含以下内容的 **Jaeger** 自定义资源（CR）：

   Jaeger CR

   ```
   apiVersion: jaegertracing.io/v1
   kind: Jaeger
   metadata:
     name: jaeger
     namespace: default
   ```

2. 通过编辑 **KnativeServing** CR 并添加用于追踪的 YAML 配置来启用 Knative Serving 的追踪：

   追踪 YAML 示例

   ```
   apiVersion: operator.knative.dev/v1alpha1
   kind: KnativeServing
   metadata:
     name: knative-serving
     namespace: knative-serving
   spec:
     config:
       tracing:
         sample-rate: "0.1" 1
         backend: zipkin 2
         zipkin-endpoint: "http://jaeger-collector.default.svc.cluster.local:9411/api/v2/spans" 3
         debug: "false" 4
   ```

   1. **sample-rate** 定义抽样概率。**sample-rate: "0.1"** 表示在 10 个 trace 中会抽样 1 个。
   2. **后端** 必须设为 **zipkin**。

3 **zipkin-endpoint** 必须指向您的 **jaeger-collector** 服务端点。要获取此端点，请替换应用 Jaeger CR 的命名空间。

4 调试（debugging）应设为 **false**。通过设置 **debug: "true"** 启用调试模式，可绕过抽样将所有 span 发送到服务器。

验证

您可以使用 **jaeger** 路由来访问 Jaeger web 控制台以查看追踪数据。

1. 获取 **jaeger** 路由的主机名：

```
$ oc get route jaeger -n default
```

输**出示例**

```
NAME     HOST/PORT                         PATH   SERVICES       PORT    TERMINATION
WILDCARD
jaeger   jaeger-default.apps.example.com          jaeger-query   <all>   reencrypt     None
```

2. 在浏览器中使用端点地址来查看控制台。

8.4. 其他资源

- Red Hat OpenShift distributed tracing 架构
- 安装分布式追踪

第 9 章 OPENSHIFT SERVERLESS 支持

如果您在执行本文档所述的某个流程时遇到问题，请访问红帽客户门户网站 http://access.redhat.com。您可以使用红帽客户门户网站搜索或浏览有关红帽产品的技术支持文章。您还可以向红帽全球支持服务 (GSS) 提交支持问题单，或者访问其他产品文档。

如果您对本文档有任何改进建议，或发现了任何错误，您可以提交一个与相关文档组件的 Jira 问题。请提供具体信息，如章节号、指南名称和 OpenShift Serverless 版本，以便我们可以快速地找到相关内容。

9.1. 关于红帽知识库

红帽知识库提供丰富的内容以帮助您最大程度地利用红帽的产品和技术。红帽知识库包括文章、产品文档和视频，概述了安装、配置和使用红帽产品的最佳实践。另外，您还可以搜索已知问题的解决方案，其提供简洁的根原因描述和补救措施。

9.2. 搜索红帽知识库

如果出现 OpenShift Container Platform 问题，您可以先进行搜索，以确定红帽知识库中是否已存在相关的解决方案。

先决条件

- 您有红帽客户门户网站帐户。

流程

1. 登录到 红帽客户门户网站。
2. 在主红帽客户门户网站搜索字段中，输入与问题相关的关键字和字符串，包括：
   - OpenShift Container Platform 组件（如 **etcd**）
   - 相关步骤（比如 **安装**）
   - 警告、错误消息和其他与输出与特定的问题相关
3. 点 **Search**。
4. 选择 **OpenShift Container Platform** 产品过滤器。
5. 在内容类型过滤中选择 **Knowledgebase**。

9.3. 提交支持问题单

先决条件

- 已安装 OpenShift CLI（**oc**）。
- 您有红帽客户门户网站帐户。
- 您可以访问 OpenShift Cluster Manager。

流程

1. 登录到 红帽客户门户网站 并选择 **SUPPORT CASES → Open a case**。
2. 为您的问题选择适当的类别（如 **Defect / Bug**）、产品(**OpenShift Container Platform**)和产品版本（如果还没有自动填充则为**4.6** ）。
3. 查看推荐的红帽知识库解决方案列表，它们可能会与您要报告的问题相关。如果建议的文章没有解决这个问题，请点 **Continue**。
4. 输入一个简洁但描述性的问题概述，以及问题症状的详细信息，以及您预期的结果。
5. 查看更新的推荐红帽知识库解决方案列表，它们可能会与您要报告的问题相关。这个列表的范围会缩小，因为您在创建问题单的过程中提供了更多信息。如果建议的文章没有解决这个问题，请点 **Continue**。
6. 请确保提供的帐户信息是正确的，如果需要，请相应调整。
7. 检查自动填充的 OpenShift Container Platform 集群 ID 是否正确。如果不正确，请手动提供集群 ID。
   - 使用 OpenShift Container Platform Web 控制台手动获得集群 ID：
     a. 导航到 **Home → Dashboards → Overview**。
     b. 该值包括在 **Details** 中的 **Cluster ID** 项中。
   - 另外，也可以通过 OpenShift Container Platform Web 控制台直接创建新的支持问题单，并自动填充集群 ID。
     a. 从工具栏导航至 **（?） help → Open Support Case**。
     b. **Cluster ID** 的值会被自动填充 。
   - 要使用 OpenShift CLI（**oc**）获取集群 ID，请运行以下命令：

     ```
     $ oc get clusterversion -o jsonpath='{.items[].spec.clusterID}{"\n"}'
     ```

8. 完成以下提示的问题，点 **Continue**:
   - 您在哪里遇到了这个问题？什么环境？
   - 这个行为在什么时候发生？发生频率？重复发生？是否只在特定时间发生？
   - 请提供这个问题对您的业务的影响及与时间相关的信息？
9. 上传相关的诊断数据文件并点击 **Continue**。建议您将使用 **oc adm must-gather** 命令收集的数据作为起点，并提供这个命令没有收集的与您的具体问题相关的其他数据。
10. 输入相关问题单管理详情，点 **Continue**。
11. 预览问题单详情，点 **Submit**。

9.4. 为支持收集诊断信息

在提交问题单时同时提供您的集群信息，可以帮助红帽支持为您进行排除故障。您可使用 **must-gather** 工具来收集有关 OpenShift Container Platform 集群的诊断信息，包括与 OpenShift Serverless 相关的数据。为了获得快速支持，请提供 OpenShift Container Platform 和 OpenShift Serverless 的诊断信息。

9.4.1. 关于 must-gather 工具

oc adm must-gather CLI 命令可收集最有助于解决问题的集群信息，包括：

- 资源定义
- 服务日志

默认情况下，**oc adm must-gather** 命令使用默认的插件镜像，并写入 **./must-gather.local**。

另外，您可以使用适当的参数运行命令来收集具体信息，如以下部分所述：

- 要收集与一个或多个特定功能相关的数据，请使用 **--image** 参数和镜像，如以下部分所述。
  例如：

```
$ oc adm must-gather  --image=registry.redhat.io/container-native-virtualization/cnv-must-gather-rhel8:v4.9.0
```

- 要收集审计日志，请使用 **-- /usr/bin/gather_audit_logs** 参数，如以下部分所述。
  例如：

```
$ oc adm must-gather -- /usr/bin/gather_audit_logs
```

注意

作为默认信息集合的一部分，不会收集审计日志来减小文件的大小。

当您运行 **oc adm must-gather** 时，集群的新项目中会创建一个带有随机名称的新 pod。在该 pod 上收集数据，并保存至以 **must-gather.local** 开头的一个新目录中。此目录在当前工作目录中创建。

例如：

```
NAMESPACE                  NAME              READY   STATUS      RESTARTS      AGE
...
openshift-must-gather-5drcj    must-gather-bklx4    2/2     Running     0          72s
openshift-must-gather-5drcj    must-gather-s8sdh    2/2     Running     0          72s
...
```

9.4.2. 关于收集 OpenShift Serverless 数据

您可使用 **oc adm must-gather** CLI 命令来收集有关集群的信息，包括与 OpenShift Serverless 相关的功能和对象。要使用 **must-gather** 来收集 OpenShift Serverless 数据，您必须为已安装的 OpenShift Serverless 版本指定 OpenShift Serverless 镜像和镜像标签。

先决条件

- 安装 OpenShift CLI (**oc**) 。

流程

- 使用 **oc adm must-gather** 命令收集数据：

```
$ oc adm must-gather --image=registry.redhat.io/openshift-serverless-1/svls-must-gather-rhel8:<image_version_tag>
```

示例命令

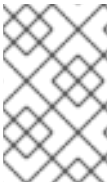
```
$ oc adm must-gather --image=registry.redhat.io/openshift-serverless-1/svls-must-gather-rhel8:1.14.0
```

第 10 章 安全性

10.1. 配置 TLS 身份验证

您可以使用 *传输层安全* (TLS) 加密 Knative 流量并进行身份验证。

TLS 是 Knative Kafka 唯一支持的流量加密方法。红帽建议将 SASL 和 TLS 同时用于 Knative Kafka 资源。



注意

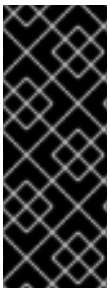
如果要使用 Red Hat OpenShift Service Mesh 集成启用内部 TLS，您必须使用 mTLS 启用 Service Mesh，而不是按照以下流程所述的内部加密。请参阅在使用带有 mTLS 的 Service Mesh 时启用 Knative Serving 指标的文档。

10.1.1. 为内部流量启用 TLS 身份验证

OpenShift Serverless 默认支持 TLS 边缘终止，以便最终用户的 HTTPS 流量加密。但是，OpenShift 路由后面的内部流量使用普通数据转发到应用。通过为内部流量启用 TLS，组件间发送的流量会进行加密，从而使此流量更加安全。

注意

如果要使用 Red Hat OpenShift Service Mesh 集成启用内部 TLS，您必须使用 mTLS 启用 Service Mesh，而不是按照以下流程所述的内部加密。



重要

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

有关红帽技术预览功能支持范围的详情，请参考 https://access.redhat.com/support/offerings/techpreview/。

先决条件

- 安装了 OpenShift Serverless Operator 和 Knative Serving。
- 已安装 OpenShift (**oc**) CLI。

流程

1. 在 spec 中创建一个包含 **internal-encryption: "true"** 字段的 Knative 服务：

```
...
spec:
  config:
    network:
      internal-encryption: "true"
...
```

2. 重启 **knative-serving** 命名空间中的 activator pod 来加载证书：

```
$ oc delete pod -n knative-serving --selector app=activator
```

10.1.2. 为集群本地服务启用 TLS 身份验证

对于集群本地服务，使用 Kourier 本地网关 **kourier-internal**。如果要针对 Kourier 本地网关使用 TLS 流量，则必须在本地网关中配置您自己的服务器证书。

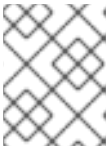
先决条件

- 安装了 OpenShift Serverless Operator 和 Knative Serving。
- 有管理员权限。
- 已安装 OpenShift (**oc**) CLI。

流程

1. 在 **knative-serving-ingress** 命名空间中部署服务器证书：

```
$ export san="knative"
```



注意

需要主题备用名称(SAN)验证，以便这些证书能够向 **<app_name>.<namespace>.svc.cluster.local** 提供请求。

2. 生成 root 密钥和证书：

```
$ openssl req -x509 -sha256 -nodes -days 365 -newkey rsa:2048 \
    -subj '/O=Example/CN=Example' \
    -keyout ca.key \
    -out ca.crt
```

3. 生成使用 SAN 验证的服务器密钥：

```
$ openssl req -out tls.csr -newkey rsa:2048 -nodes -keyout tls.key \
  -subj "/CN=Example/O=Example" \
  -addext "subjectAltName = DNS:$san"
```

4. 创建服务器证书：

```
$ openssl x509 -req -extfile <(printf "subjectAltName=DNS:$san") \
  -days 365 -in tls.csr \
  -CA ca.crt -CAkey ca.key -CAcreateserial -out tls.crt
```

5. 为 Kourier 本地网关配置 secret：

   a. 从前面的步骤创建的证书，在 **knative-serving-ingress** 命名空间中部署 secret：

```
$ oc create -n knative-serving-ingress secret tls server-certs \
    --key=tls.key \
    --cert=tls.crt --dry-run=client -o yaml | oc apply -f -
```

b. 更新 **KnativeServing** 自定义资源 (CR) spec，以使用 Kourier 网关创建的 secret：

KnativeServing CR 示例

```
...
spec:
  config:
    kourier:
      cluster-cert-secret: server-certs
...
```

Kourier 控制器在不重启该服务的情况下设置证书，因此您不需要重启 pod。

您可以通过端口 **443** 访问 Kourier 内部服务，方法是从客户端挂载并使用 **ca.crt**。

其他资源

- 在使用带有 mTLS 的 Service Mesh 时启用 Knative Serving 指标

10.1.3. 使用 TLS 证书保护带有自定义域的服务

为 Knative 服务配置了自定义域后，您可以使用 TLS 证书来保护映射的服务。要做到这一点，您必须创建一个 Kubernetes TLS secret，然后更新 **DomainMapping** CR 以使用您创建的 TLS secret。

先决条件

- 为 Knative 服务配置了自定义域，并有一个正常工作的 **DomainMapping** CR。
- 您有来自证书授权机构供应商或自签名证书的 TLS 证书。
- 您已从证书授权中心（CA）提供商或自签名证书获取 **cert** 和 **key** 文件。
- 安装 OpenShift CLI (**oc**) 。

流程

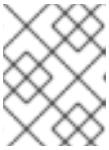
1. 创建 Kubernetes TLS secret：

```
$ oc create secret tls <tls_secret_name> --cert=<path_to_certificate_file> --key=
<path_to_key_file>
```

2. 如果您使用 Red Hat OpenShift Service Mesh 作为 OpenShift Serverless 安装的 ingress，请使用以下内容标记 Kubernetes TLS secret：

```
“networking.internal.knative.dev/certificate-uid": “<value>”
```

如果使用第三方 secret 供应商（如 cert-manager），您可以配置 secret manager 来自动标记 Kubernetes TLS secret。cert-manager 用户可以使用提供的 secret 模板自动生成带有正确标签的 secret。在本例中，secret 过滤仅基于键，但这个值可以存储有用的信息，如 secret 包含的证书 ID。



注意

{cert-manager-operator} 是一个技术预览功能。如需更多信息，请参阅**安装 {cert-manager-operator}** 文档。

3. 更新 **DomainMapping** CR，以使用您创建的 TLS secret：

```
apiVersion: serving.knative.dev/v1alpha1
kind: DomainMapping
metadata:
  name: <domain_name>
  namespace: <namespace>
spec:
  ref:
    name: <service_name>
    kind: Service
    apiVersion: serving.knative.dev/v1
# TLS block specifies the secret to be used
  tls:
    secretName: <tls_secret_name>
```

验证

1. 验证 **DomainMapping** CR 状态是否为 **True**，输出中的 **URL** 列显示了使用 scheme **https** 的映射域：

```
$ oc get domainmapping <domain_name>
```

输**出示例**

```
NAME                URL                         READY   REASON
example.com         https://example.com         True
```

2. 可选：如果服务公开，请运行以下命令验证该服务是否可用：

```
$ curl https://<domain_name>
```

如果证书是自签名的，请通过在 **curl** 命令中添加 **-k** 标志来跳过验证。

10.1.4. 为 Kafka 代理配置 TLS 身份验证

Apache Kafka 客户端和服务器使用 *传输层安全性* (TLS) 来加密 Knative 和 Kafka 之间的流量，以及用于身份验证。TLS 是 Knative Kafka 唯一支持的流量加密方法。

先决条件

- 在 OpenShift Container Platform 上具有集群管理员权限。
- OpenShift Serverless Operator、Knative Eventing 和 **KnativeKafka** CR 已安装在 OpenShift Container Platform 集群中。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。

- 您有一个 Kafka 集群 CA 证书存储为一个 **.pem** 文件。
- 您有一个 Kafka 集群客户端证书，并存储为 **.pem** 文件的密钥。
- 安装 OpenShift CLI (**oc**) 。

流程

1. 在 **knative-eventing** 命名空间中创建证书文件作为 secret：

```
$ oc create secret -n knative-eventing generic <secret_name> \
  --from-literal=protocol=SSL \
  --from-file=ca.crt=caroot.pem \
  --from-file=user.crt=certificate.pem \
  --from-file=user.key=key.pem
```

重要

使用密钥名称 **ca.crt**、**user.crt** 和 **user.key**。不要更改它们。

2. 编辑 **KnativeKafka** CR，并在 **broker** spec 中添加对 secret 的引用：

```
apiVersion: operator.serverless.openshift.io/v1alpha1
kind: KnativeKafka
metadata:
  namespace: knative-eventing
  name: knative-kafka
spec:
  broker:
    enabled: true
    defaultConfig:
      authSecretName: <secret_name>
...
```

10.1.5. 为 Kafka 频道配置 TLS 验证

Apache Kafka 客户端和服务器使用 *传输层安全性* (TLS) 来加密 Knative 和 Kafka 之间的流量，以及用于身份验证。TLS 是 Knative Kafka 唯一支持的流量加密方法。

先决条件

- 在 OpenShift Container Platform 上具有集群管理员权限。
- OpenShift Serverless Operator、Knative Eventing 和 **KnativeKafka** CR 已安装在 OpenShift Container Platform 集群中。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。
- 您有一个 Kafka 集群 CA 证书存储为一个 **.pem** 文件。
- 您有一个 Kafka 集群客户端证书，并存储为 **.pem** 文件的密钥。
- 安装 OpenShift CLI (**oc**) 。

流程

1. 在所选命名空间中创建证书文件作为 secret：

```
$ oc create secret -n <namespace> generic <kafka_auth_secret> \
  --from-file=ca.crt=caroot.pem \
  --from-file=user.crt=certificate.pem \
  --from-file=user.key=key.pem
```

重要

使用密钥名称 **ca.crt**、**user.crt** 和 **user.key**。不要更改它们。

2. 编辑 **KnativeKafka** 自定义资源：

```
$ oc edit knativekafka
```

3. 引用您的 secret 和 secret 的命名空间：

```
apiVersion: operator.serverless.openshift.io/v1alpha1
kind: KnativeKafka
metadata:
  namespace: knative-eventing
  name: knative-kafka
spec:
  channel:
    authSecretName: <kafka_auth_secret>
    authSecretNamespace: <kafka_auth_secret_namespace>
    bootstrapServers: <bootstrap_servers>
    enabled: true
  source:
    enabled: true
```

注意

确保指定 bootstrap 服务器中的匹配端口。

例如：

```
apiVersion: operator.serverless.openshift.io/v1alpha1
kind: KnativeKafka
metadata:
  namespace: knative-eventing
  name: knative-kafka
spec:
  channel:
    authSecretName: tls-user
    authSecretNamespace: kafka
    bootstrapServers: eventing-kafka-bootstrap.kafka.svc:9094
    enabled: true
  source:
    enabled: true
```

10.2. 为 KNATIVE 服务配置 JSON WEB 令牌身份验证

OpenShift Serverless 当前没有用户定义的授权功能。要为部署添加用户定义的授权，您必须将 OpenShift Serverless 与 Red Hat OpenShift Service Mesh 集成，然后为 Knative 服务配置 JSON Web Token (JWT) 身份验证和 sidecar 注入。

10.2.1. 在 Service Mesh 2.x 和 OpenShift Serverless 中使用 JSON Web 令牌身份验证

您可以使用 Service Mesh 2.x 和 OpenShift Serverless 在 Knative 服务中使用 JSON Web Token (JWT) 身份验证。要做到这一点，您必须在作为 **ServiceMeshMemberRoll** 对象成员的应用程序命名空间中创建身份验证请求和策略。您还必须为该服务启用 sidecar 注入。

重要

在启用了 Kourier 时，不支持在系统命名空间中向 pod 添加 sidecar 注入，如 **knative-serving** 和 **knative-serving-ingress**。

如果需要对这些命名空间中的 pod 进行 sidecar 注入，请参阅 OpenShift Serverless 文档中的*原生将 Service Mesh 与 OpenShift Serverless 集成*。

先决条件

- 您已在集群中安装了 OpenShift Serverless Operator、Knative Serving 和 Red Hat OpenShift Service Mesh。
- 安装 OpenShift CLI (**oc**)。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。

流程

1. 在您的服务中添加 **sidecar.istio.io/inject="true"** 注解：

服务示例

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: <service_name>
spec:
  template:
    metadata:
      annotations:
        sidecar.istio.io/inject: "true" 1
        sidecar.istio.io/rewriteAppHTTPProbers: "true" 2
...
```

1. 添加 **sidecar.istio.io/inject="true"** 注解。
2. 您必须在 Knative 服务中将注解 **sidecar.istio.io/rewriteAppHTTPProbers: "true"** 设置为 OpenShift Serverless 版本 1.14.0 或更新的版本，然后使用 HTTP 探测作为 Knative 服务的就绪度探测。

2. 应用 **Service** 资源：

```
$ oc apply -f <filename>
```

3. 在 **ServiceMeshMemberRoll** 对象的每个无服务器应用程序命名空间中创建一个 **RequestAuthentication** 资源：

```
apiVersion: security.istio.io/v1beta1
kind: RequestAuthentication
metadata:
  name: jwt-example
  namespace: <namespace>
spec:
  jwtRules:
  - issuer: testing@secure.istio.io
    jwksUri: https://raw.githubusercontent.com/istio/istio/release-
1.8/security/tools/jwt/samples/jwks.json
```

4. 应用 **RequestAuthentication** 资源：

```
$ oc apply -f <filename>
```

5. 通过创建以下 **AuthorizationPolicy** 资源，允许从 **ServiceMeshMemberRoll** 对象中的每个无服务器应用程序命名空间的系统 pod 访问 **RequestAuthenticaton** 资源：

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: allowlist-by-paths
  namespace: <namespace>
spec:
  action: ALLOW
  rules:
  - to:
    - operation:
        paths:
        - /metrics (1)
        - /healthz (2)
```

(1) 由系统 pod 收集指标的应用程序上的路径。

(2) 系统 pod 探测到应用程序的路径。

6. 应用 **AuthorizationPolicy** 资源：

```
$ oc apply -f <filename>
```

7. 对于作为 **ServiceMeshMemberRoll** 对象中成员的每个无服务器应用程序命名空间，请创建以下 **AuthorizationPolicy** 资源：

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
```

```
  name: require-jwt
  namespace: <namespace>
spec:
  action: ALLOW
  rules:
  - from:
    - source:
       requestPrincipals: ["testing@secure.istio.io/testing@secure.istio.io"]
```

8. 应用 **AuthorizationPolicy** 资源：

```
$ oc apply -f <filename>
```

验证

1. 如果您尝试使用 **curl** 请求来获取 Knative 服务 URL，则会被拒绝：

   示例命令

```
$ curl http://hello-example-1-default.apps.mycluster.example.com/
```

   输**出示例**

```
RBAC: access denied
```

2. 使用有效 JWT 验证请求。

   a. 获取有效的 JWT 令牌：

```
$ TOKEN=$(curl https://raw.githubusercontent.com/istio/istio/release-1.8/security/tools/jwt/samples/demo.jwt -s) && echo "$TOKEN" | cut -d '.' -f2 - | base64 --decode -
```

   b. 使用 **curl** 请求标头中的有效令牌访问该服务：

```
$ curl -H "Authorization: Bearer $TOKEN"  http://hello-example-1-default.apps.example.com
```

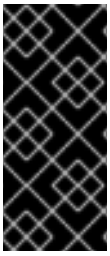
   现在允许请求：

   输**出示例**

```
Hello OpenShift!
```

10.2.2. 在 Service Mesh 1.x 和 OpenShift Serverless 中使用 JSON Web 令牌身份验证

您可以使用 Service Mesh 1.x 和 OpenShift Serverless 在 Knative 服务中使用 JSON Web Token (JWT) 身份验证。要做到这一点，您必须在作为 **ServiceMeshMemberRoll** 对象的成员的应用程序命名空间中创建策略。您还必须为该服务启用 sidecar 注入。



重要

在启用了 Kourier 时，不支持在系统命名空间中向 pod 添加 sidecar 注入，如 **knative-serving** 和 **knative-serving-ingress**。

如果需要对这些命名空间中的 pod 进行 sidecar 注入，请参阅 OpenShift Serverless 文档中的*原生将 Service Mesh 与 OpenShift Serverless 集成*。

先决条件

- 您已在集群中安装了 OpenShift Serverless Operator、Knative Serving 和 Red Hat OpenShift Service Mesh。
- 安装 OpenShift CLI (**oc**)。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。

流程

1. 在您的服务中添加 **sidecar.istio.io/inject="true"** 注解：

   服务示例

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: <service_name>
spec:
  template:
    metadata:
      annotations:
        sidecar.istio.io/inject: "true" 1
        sidecar.istio.io/rewriteAppHTTPProbers: "true" 2
...
```

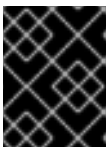
   1 添加 **sidecar.istio.io/inject="true"** 注解。

   2 您必须在 Knative 服务中将注解 **sidecar.istio.io/rewriteAppHTTPProbers: "true"** 设置为 OpenShift Serverless 版本 1.14.0 或更新的版本，然后使用 HTTP 探测作为 Knative 服务的就绪度探测。

2. 应用 **Service** 资源：

```
$ oc apply -f <filename>
```

3. 在作为 **ServiceMeshMemberRoll** 对象的成员的无服务器应用程序命名空间中创建策略，该策略只允许具有有效 JSON Web Tokens（JWT）的请求：



重要

路径 **/metrics** 和 **/healthz** 必须包含在 **excludePaths** 中，因为它们是从 **knative-serving** 命名空间中的系统 pod 访问的。

```
apiVersion: authentication.istio.io/v1alpha1
kind: Policy
metadata:
  name: default
  namespace: <namespace>
spec:
  origins:
  - jwt:
      issuer: testing@secure.istio.io
      jwksUri: "https://raw.githubusercontent.com/istio/istio/release-
1.6/security/tools/jwt/samples/jwks.json"
      triggerRules:
      - excludedPaths:
        - prefix: /metrics 1
        - prefix: /healthz 2
  principalBinding: USE_ORIGIN
```

1 由系统 pod 收集指标的应用程序上的路径。

2 系统 pod 探测到应用程序的路径。

4. 应用 **Policy** 资源：

```
$ oc apply -f <filename>
```

验证

1. 如果您尝试使用 **curl** 请求来获取 Knative 服务 URL，则会被拒绝：

```
$ curl http://hello-example-default.apps.mycluster.example.com/
```

输**出示例**

```
Origin authentication failed.
```

2. 使用有效 JWT 验证请求。

   a. 获取有效的 JWT 令牌：

   ```
   $ TOKEN=$(curl https://raw.githubusercontent.com/istio/istio/release-
   1.6/security/tools/jwt/samples/demo.jwt -s) && echo "$TOKEN" | cut -d '.' -f2 - | base64 --
   decode -
   ```

   b. 使用 **curl** 请求标头中的有效令牌访问该服务：

   ```
   $ curl http://hello-example-default.apps.mycluster.example.com/ -H "Authorization:
   Bearer $TOKEN"
   ```

   现在允许请求：

   输**出示例**

```
Hello OpenShift!
```

10.3. 为 KNATIVE 服务配置自定义域

Knative 服务会自动根据集群配置分配默认域名。例如，**<service_name>-<namespace>.example.com**。您可以通过将您自己的自定义域名映射到 Knative 服务来自定义 Knative 服务域。

您可以通过为服务创建 **DomainMapping** 资源来完成此操作。您还可以创建多个 **DomainMapping** 资源，将多个域和子域映射到单个服务。

10.3.1. 创建自定义域映射

您可以通过将您自己的自定义域名映射到 Knative 服务来自定义 Knative 服务域。要将自定义域名映射到自定义资源（CR），您必须创建一个映射到可寻址目标 CR 的 **DomainMapping** CR，如 Knative 服务或 Knative 路由。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 安装 OpenShift CLI (**oc**) 。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。
- 您已创建了 Knative 服务，并控制要映射到该服务的自定义域。

注意

您的自定义域必须指向 OpenShift Container Platform 集群的 IP 地址。

流程

1. 在与您要映射的目标 CR 相同的命名空间中创建一个包含 **DomainMapping** CR 的 YAML 文件：

```
apiVersion: serving.knative.dev/v1alpha1
kind: DomainMapping
metadata:
 name: <domain_name> 1
 namespace: <namespace> 2
spec:
 ref:
   name: <target_name> 3
   kind: <target_type> 4
   apiVersion: serving.knative.dev/v1
```

1. 要映射到目标 CR 的自定义域名。
2. **DomainMapping** CR 和目标 CR 的命名空间。
3. 映射到自定义域的目标 CR 名称。

4 映射到自定义域的 CR 类型。

服务域映射示例

```
apiVersion: serving.knative.dev/v1alpha1
kind: DomainMapping
metadata:
 name: example.com
 namespace: default
spec:
 ref:
   name: example-service
   kind: Service
   apiVersion: serving.knative.dev/v1
```

路由域映射示例

```
apiVersion: serving.knative.dev/v1alpha1
kind: DomainMapping
metadata:
 name: example.com
 namespace: default
spec:
 ref:
   name: example-route
   kind: Route
   apiVersion: serving.knative.dev/v1
```

2. 将 **DomainMapping** CR 应用为 YAML 文件：

```
$ oc apply -f <filename>
```

10.3.2. 使用 Knative CLI 创建自定义域映射

您可以通过将您自己的自定义域名映射到 Knative 服务来自定义 Knative 服务域。您可以使用 Knative (**kn**) CLI 创建映射到可寻址目标 CR 的 **DomainMapping** 自定义资源 (CR)，如 Knative 服务或 Knative 路由。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 您已创建了 Knative 服务或路由，并控制要映射到该 CR 的自定义域。

注意

您的自定义域必须指向 OpenShift Container Platform 集群的 DNS。

- 已安装 Knative (**kn**) CLI。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。

流程

- 将域映射到当前命名空间中的 CR：

```
$ kn domain create <domain_mapping_name> --ref <target_name>
```

示例命令

```
$ kn domain create example.com --ref example-service
```

--ref 标志为域映射指定一个可寻址的目标 CR。

如果使用 **--ref** 标志时没有提供前缀，则会假定目标为当前命名空间中的 Knative 服务。

- 将域映射到指定命名空间中的 Knative 服务：

```
$ kn domain create <domain_mapping_name> --ref
<ksvc:service_name:service_namespace>
```

示例命令

```
$ kn domain create example.com --ref ksvc:example-service:example-namespace
```

- 将域映射到 Knative 路由：

```
$ kn domain create <domain_mapping_name> --ref <kroute:route_name>
```

示例命令

```
$ kn domain create example.com --ref kroute:example-route
```

10.3.3. 使用 TLS 证书保护带有自定义域的服务

为 Knative 服务配置了自定义域后，您可以使用 TLS 证书来保护映射的服务。要做到这一点，您必须创建一个 Kubernetes TLS secret，然后更新 **DomainMapping** CR 以使用您创建的 TLS secret。

先决条件

- 为 Knative 服务配置了自定义域，并有一个正常工作的 **DomainMapping** CR。
- 您有来自证书授权机构供应商或自签名证书的 TLS 证书。
- 您已从证书授权中心（CA）提供商或自签名证书获取 **cert** 和 **key** 文件。
- 安装 OpenShift CLI (**oc**) 。

流程

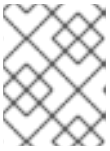
1. 创建 Kubernetes TLS secret：

```
$ oc create secret tls <tls_secret_name> --cert=<path_to_certificate_file> --key=
<path_to_key_file>
```

2. 如果您使用 Red Hat OpenShift Service Mesh 作为 OpenShift Serverless 安装的 ingress，请使用以下内容标记 Kubernetes TLS secret：

```
“networking.internal.knative.dev/certificate-uid": “<value>”
```

如果使用第三方 secret 供应商（如 cert-manager），您可以配置 secret manager 来自动标记 Kubernetes TLS secret。cert-manager 用户可以使用提供的 secret 模板自动生成带有正确标签的 secret。在本例中，secret 过滤仅基于键，但这个值可以存储有用的信息，如 secret 包含的证书 ID。



注意

{cert-manager-operator} 是一个技术预览功能。如需更多信息，请参阅**安装 {cert-manager-operator}** 文档。

3. 更新 **DomainMapping** CR，以使用您创建的 TLS secret：

```
apiVersion: serving.knative.dev/v1alpha1
kind: DomainMapping
metadata:
  name: <domain_name>
  namespace: <namespace>
spec:
  ref:
    name: <service_name>
    kind: Service
    apiVersion: serving.knative.dev/v1
# TLS block specifies the secret to be used
  tls:
    secretName: <tls_secret_name>
```

验证

1. 验证 **DomainMapping** CR 状态是否为 **True**，输出中的 **URL** 列显示了使用 scheme **https** 的映射域：

```
$ oc get domainmapping <domain_name>
```

输**出示例**

```
NAME                URL                         READY   REASON
example.com         https://example.com         True
```

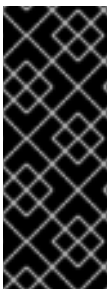
2. 可选：如果服务公开，请运行以下命令验证该服务是否可用：

```
$ curl https://<domain_name>
```

如果证书是自签名的，请通过在 **curl** 命令中添加 **-k** 标志来跳过验证。

第 11 章 FUNCTIONS

11.1. 设置 OPENSHIFT SERVERLESS 功能



重要

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

有关红帽技术预览功能支持范围的详情，请参考 https://access.redhat.com/support/offerings/techpreview/。

为改进应用程序代码部署的过程，您可以使用 OpenShift Serverless 部署无状态、事件驱动的功能，作为 OpenShift Container Platform 上的 Knative 服务。如果要开发功能，您必须完成设置步骤。

11.1.1. 先决条件

要在集群中启用 OpenShift Serverless 功能，您必须完成以下步骤：

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。

注意

功能部署为 Knative 服务。如果要将事件驱动的架构与您的功能搭配使用，还必须安装 Knative Eventing。

- 已安装 **oc** CLI。
- 已安装 Knative (**kn**) CLI。安装 Knative CLI 可让您使用 **kn func** 命令来创建和管理功能。
- 已安装 Docker Container Engine 或 podman 版本 3.4.7 或更高版本，并可以访问可用的镜像 registry，如 OpenShift Container Registry。
- 如果您使用 Quay.io 作为镜像 registry，您必须确保存储库不是私有的，或者按照 OpenShift Container Platform 文档中有关允许 Pod 引用其他安全 registry 中的镜像的内容进行操作。
- 如果使用 OpenShift Container Registry，集群管理员必须公开 registry。

11.1.2. 设置 podman

要使用高级容器管理功能，您可能需要将 podman 与 OpenShift Serverless 功能搭配使用。要做到这一点，您需要启动 podman 服务并配置 Knative (**kn**) CLI 来连接它。

流程

1. 在 **${XDG_RUNTIME_DIR}/podman/podman.sock** 的 UNIX 套接字上启动提供 Docker API 的 podman 服务：

```
$ systemctl start --user podman.socket
```

注意

在大多数系统中，此套接字位于 **/run/user/$ (id -u) /podman/podman.sock**。

2. 建立用于构建功能的环境变量：

```
$ export DOCKER_HOST="unix://${XDG_RUNTIME_DIR}/podman/podman.sock"
```

3. 在功能项目目录中使用 **-v** 标记运行构建命令，以查看详细的输出。您应该看到到本地 UNIX 套接字的连接：

```
$ kn func build -v
```

11.1.3. 在 macOS 中设置 podman

要使用高级容器管理功能，您可能需要将 podman 与 OpenShift Serverless 功能搭配使用。要在 macOS 中这样做，您需要启动 podman 机器并配置 Knative (**kn**) CLI 以连接它。

流程

1. 创建 podman 机器：

```
$ podman machine init --memory=8192 --cpus=2 --disk-size=20
```

2. 启动 podman 机器，它将在 UNIX 套接字上提供 Docker API：

```
$ podman machine start
Starting machine "podman-machine-default"
Waiting for VM ...
Mounting volume... /Users/myuser:/Users/user

[...truncated output...]

You can still connect Docker API clients by setting DOCKER_HOST using the
following command in your terminal session:

 export
DOCKER_HOST='unix:///Users/myuser/.local/share/containers/podman/machine/podman-
machine-default/podman.sock'

Machine "podman-machine-default" started successfully
```

注意

在大多数 macOS 系统上，此套接字位于 **/Users/myuser/.local/share/containers/podman/machine/podman-machine-default/podman.sock**。

3. 建立用于构建功能的环境变量：

```
$ export
DOCKER_HOST='unix:///Users/myuser/.local/share/containers/podman/machine/podman-
machine-default/podman.sock'
```

4. 在功能项目目录中使用 **-v** 标记运行构建命令，以查看详细的输出。您应该看到到本地 UNIX 套接字的连接：

```
$ kn func build -v
```

11.1.4. 后续步骤

- 有关 Docker Container Engine 或 podman 的更多信息，请参阅容器构建工具选项。
- 请参阅开始使用功能。

11.2. 功能入门

功能生命周期管理包括创建、构建和部署功能。另外，您还可以通过调用它来测试部署的功能。您可以使用 **kn func** 工具在 OpenShift Serverless 上完成所有这些操作。

重要

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

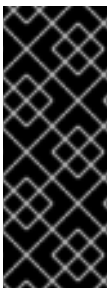
有关红帽技术预览功能支持范围的详情，请参考 https://access.redhat.com/support/offerings/techpreview/。

11.2.1. 先决条件

在完成以下步骤前，您必须确定您已完成了设置 OpenShift Serverless 功能的所有先决条件任务。

11.2.2. 创建功能

在构建和部署功能之前，您必须使用 Knative (**kn**) CLI 创建功能。您可以在命令行中指定路径、运行时、模板和镜像 registry，也可以使用 **-c** 标志在终端中启动交互式体验。



重要

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

有关红帽技术预览功能支持范围的详情，请参考 https://access.redhat.com/support/offerings/techpreview/。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 已安装 Knative (**kn**) CLI。

流程

- 创建功能项目：

```
$ kn func create -r <repository> -l <runtime> -t <template> <path>
```

  - 可接受的运行时值包括 **quarkus**、**node**、**typescript**、**go**、**python**、**springboot** 和 **rust**。
  - 可接受的模板值包括 **http** 和 **cloudevents**。

    示例命令

```
$ kn func create -l typescript -t cloudevents examplefunc
```

    输出示例

```
Created typescript function in /home/user/demo/examplefunc
```

  - 或者，您可以指定包含自定义模板的存储库。

    示例命令

```
$ kn func create -r https://github.com/boson-project/templates/ -l node -t hello-world
examplefunc
```

    输出示例

```
Created node function in /home/user/demo/examplefunc
```

11.2.3. 在本地运行一个函数

您可以使用 **kn func run** 命令在当前目录中本地运行函数，或者在 **--path** 标志指定的目录中运行。如果您运行的函数之前没有被构建，或者项目文件自上次构建以来已修改过，**kn func run** 命令将在运行它前构建该函数。

在当前目录中运行函数的命令示例

```
$ kn func run
```

在指定为路径的目录中运行函数的示例

```
$ kn func run --path=<directory_path>
```

您也可以在运行该函数前强制重建现有镜像，即使项目文件没有更改项目文件，则使用 **--build** 标志：

使用 build 标记的 run 命令示例

```
$ kn func run --build
```

如果将 **build** 标志设置为 false，这将禁用构建镜像，并使用之前构建的镜像运行该功能：

使用 build 标记的 run 命令示例

```
$ kn func run --build=false
```

您可以使用 help 命令了解更多有关 **kn func run** 命令选项的信息：

构建 help 命令

```
$ kn func help run
```

11.2.4. 构建函数

在运行功能前，您必须构建 function 项目。如果使用 **kn func run** 命令，则该函数会自动构建。但是，您可以使用 **kn func build** 命令在不运行的情况下构建函数，这对于高级用户或调试场景非常有用。

kn func build 命令创建可在您的计算机或 OpenShift Container Platform 集群中运行的 OCI 容器镜像。此命令使用功能项目名称和镜像 registry 名称为您的功能构建完全限定镜像名称。

11.2.4.1. 镜像容器类型

默认情况下，**kn func build** 使用 Red Hat Source-to-Image (S2I) 技术创建一个容器镜像。

使用 Red Hat Source-to-Image (S2I) 的 build 命令示例.

```
$ kn func build
```

您可以通过在命令中添加 **--builder** 标志并指定 **pack** 策略，来使用 CNCF Cloud Native Buildpacks 技术：

使用 CNCF Cloud Native Buildpacks 的 build 命令示例

```
$ kn func build --builder pack
```

11.2.4.2. 镜像 registry 类型

OpenShift Container Registry 默认用作存储功能镜像的镜像 registry。

使用 OpenShift Container Registry 的 build 命令示例

```
$ kn func build
```

输出示例

```
Building function image
Function image has been built, image: registry.redhat.io/example/example-function:latest
```

您可以使用 **--registry** 标志覆盖使用 OpenShift Container Registry 作为默认镜像 registry：

build 命令覆盖 OpenShift Container Registry 以使用 quay.io

```
$ kn func build --registry quay.io/username
```

输出示例

```
Building function image
Function image has been built, image: quay.io/username/example-function:latest
```

11.2.4.3. push 标记

您可以将 **--push** 标志添加到 **kn func build** 命令中，以便在成功构建后自动推送功能镜像：

使用 OpenShift Container Registry 的 build 命令示例

```
$ kn func build --push
```

11.2.4.4. help 命令

您可以使用 help 命令了解更多有关 **kn func build** 命令选项的信息：

构建 help 命令

```
$ kn func help build
```

11.2.5. 部署功能

您可以使用 **kn func deploy** 命令将功能部署到集群中，作为 Knative 服务。如果已经部署了目标功能，则会使用推送到容器镜像 registry 的新容器镜像进行更新，并更新 Knative 服务。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 已安装 Knative (**kn**) CLI。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。
- 您必须已创建并初始化要部署的功能。

流程

- 部署功能：

  ```
  $ kn func deploy [-n <namespace> -p <path> -i <image>]
  ```

  输出示例

  ```
  Function deployed at: http://func.example.com
  ```

  - 如果没有指定 **namespace**，则该函数部署到当前命名空间中。
  - 此函数从当前目录中部署，除非指定了 **path**。
  - Knative 服务名称派生自项目名称，无法使用此命令进行更改。

11.2.6. 使用测试事件调用部署的功能

您可以使用 **kn func invoke** CLI 命令发送测试请求，在本地或 OpenShift Container Platform 集群中调用功能。您可以使用此命令测试功能是否正常工作并且能够正确接收事件。本地调用函数可用于在功能开发期间进行快速测试。在测试与生产环境更接近的测试时，在集群中调用函数非常有用。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 已安装 Knative (**kn**) CLI。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。
- 您必须已部署了要调用的功能。

流程

- 调用函数：

```
$ kn func invoke
```

  - **kn func invoke** 命令仅在当前运行本地容器镜像时或在集群中部署功能时才有效。
  - **kn func invoke** 命令默认在本地目录上执行，并假定此目录是一个功能项目。

11.2.7. 删除函数

您可以使用 **kn func delete** 命令删除功能。当不再需要某个函数时，这很有用，并有助于在集群中保存资源。

流程

- 删除函数：

```
$ kn func delete [<function_name> -n <namespace> -p <path>]
```

  - 如果没有指定要删除的功能的名称或路径，则会搜索当前目录以查找用于决定要删除的功能的 **func.yaml** 文件。
  - 如果没有指定命名空间，则默认为 **func.yaml** 文件中的 **namespace** 值。

11.2.8. 其他资源

- 手动公开默认 registry
- Intellij Knative 插件的 marketplace 页面
- Visual Studio Code Knative 插件的 marketplace 页面

11.3. ON-CLUSTER 功能构建和部署

您可以直接在集群中构建功能，而不是在本地构建功能。在本地开发机器上使用此工作流时，您只需要使用功能源代码。例如，当您无法在集群功能构建工具（如 docker 或 podman）安装时，这非常有用。

11.3.1. 在集群中构建和部署功能

您可以使用 Knative (**kn**) CLI 启动功能项目构建，然后直接将功能部署到集群中。若要以这种方式构建功能项目，您的功能项目的源代码必须存在于可供集群访问的 Git 存储库分支中。

重要

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

有关红帽技术预览功能支持范围的详情，请参考 https://access.redhat.com/support/offerings/techpreview/。

先决条件

- 在集群中必须安装 Red Hat OpenShift Pipelines。
- 已安装 OpenShift CLI(**oc**)。
- 已安装 Knative (**kn**) CLI。

流程

1. 在您要运行 Pipelines 和部署功能的每个命名空间中，您必须创建以下资源：
   a. 创建 **s2i** Tekton 任务，以便在管道中使用 Source-to-Image：

```
$ oc apply -f https://raw.githubusercontent.com/openshift-knative/kn-plugin-func/serverless-1.25.0/pipelines/resources/tekton/task/func-s2i/0.1/func-s2i.yaml
```

   b. 创建 **kn func** 部署 Tekton 任务，以便在管道中部署该功能：

```
$ oc apply -f https://raw.githubusercontent.com/openshift-knative/kn-plugin-func/serverless-1.25.0/pipelines/resources/tekton/task/func-deploy/0.1/func-deploy.yaml
```

2. 创建功能：

```
$ kn func create <function_name> -l <runtime>
```

3. 在创建了新的功能项目后，您必须将项目添加到 Git 存储库，并确保该存储库可供集群使用。关于此 Git 存储库的信息用于在下一步中更新 **func.yaml** 文件。

4. 更新功能项目的 **func.yaml** 文件中的配置，以便为 Git 仓库启用 on-cluster 构建：

```
...
git:
  url: <git_repository_url> 1
  revision: main 2
  contextDir: <directory_path> 3
...
```

1 必需。指定包含功能源代码的 Git 存储库。

2. 可选。指定要使用的 Git 存储库修订。这可以是分支、标签或提交。
3. 可选。如果函数没有位于 Git 存储库根文件夹中，请指定函数的目录的路径。

5. 实施您功能的业务逻辑。然后，使用 Git 提交并推送更改。
6. 部署功能：

```
$ kn func deploy --remote
```

如果您没有登录到功能配置中引用的容器 registry，系统会提示您为托管功能镜像的远程容器 registry 提供凭证：

输出和提示示例

```
 Creating Pipeline resources
Please provide credentials for image registry used by Pipeline.
? Server: https://index.docker.io/v1/
? Username: my-repo
? Password: ********
  Function deployed at URL: http://test-function.default.svc.cluster.local
```

7. 要更新您的功能，使用 Git 提交并推送新的更改，然后再次运行 **kn func deploy --remote** 命令。

11.3.2. 指定功能修订

在集群中构建和部署功能时，您必须通过指定存储库中的 Git 存储库、分支和子目录来指定功能代码的位置。如果使用 **main** 分支，则不需要指定分支。同样，如果功能位于存储库的根目录，则不需要指定子目录。您可以在 **func.yaml** 配置文件中指定这些参数，或使用带有 **kn func deploy** 命令的标志。

先决条件

- 在集群中必须安装 Red Hat OpenShift Pipelines。
- 已安装 OpenShift (**oc**) CLI。
- 已安装 Knative (**kn**) CLI。

流程

- 部署功能：

```
$ kn func deploy --remote \ 1
            --git-url <repo-url> \ 2
            [--git-branch <branch>] \ 3
            [--git-dir <function-dir>] 4
```

1. 使用 **--remote** 标志时，构建远程运行。
2. 将 **<repo-url>** 替换为 Git 存储库的 URL。
3. 将 **<branch>** 替换为 Git 分支、标签或提交。如果使用 **main** 分支的最新提交，您可以跳过此标志。

4 如果 **<function-dir>** 与存储库根目录不同，请将 <function-dir> 替换为包含该函数的目录。

例如：

```
$ kn func deploy --remote \
                 --git-url https://example.com/alice/myfunc.git \
                 --git-branch my-feature \
                 --git-dir functions/example-func/
```

11.4. 开发 NODE.JS 功能

重要

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

有关红帽技术预览功能支持范围的详情，请参考 https://access.redhat.com/support/offerings/techpreview/。

创建 Node.js 功能项目 后，您可以修改提供的模板文件，以将业务逻辑添加到您的功能中。这包括配置功能调用和返回的标头和状态代码。

11.4.1. 先决条件

- 在开发功能前，您必须完成设置 OpenShift Serverless 功能 的步骤。

11.4.2. Node.js 功能模板结构

使用 Knative (**kn**) CLI 创建 Node.js 功能时，项目目录类似于典型的 Node.js 项目。唯一的例外是额外的 **func.yaml** 文件，用于配置函数。

http 和 **event** 触发器功能具有相同的模板结构：

模板结构

```
.
├── func.yaml 1
├── index.js 2
├── package.json 3
├── README.md
└── test 4
    ├── integration.js
    └── unit.js
```

1 **func.yaml** 配置文件用于决定镜像名称和 registry。

2 您的项目必须包含可导出单一功能的 **index.js** 文件。

3

您不限于模板 **package.json** 文件中提供的依赖项。您可以添加其他依赖项，如在任何其他 Node.js 项目中一样。

添加 npm 依赖项示例

```
npm install --save opossum
```

为部署构建项目时，这些依赖项将包含在创建的运行时容器镜像中。

4 集成和单元测试脚本作为功能模板的一部分提供。

11.4.3. 关于调用 Node.js 功能

当使用 Knative (**kn**) CLI 创建功能项目时，您可以生成一个响应 CloudEvents 的项目，或者响应简单 HTTP 请求的项目。Knative 中的 CloudEvents 作为 POST 请求通过 HTTP 传输，因此两种功能类型都侦听并响应传入的 HTTP 事件。

Node.js 功能可以通过简单的 HTTP 请求调用。收到传入请求后，将通过 **上下文** 对象作为第一个参数来调用函数。

11.4.3.1. Node.js 上下文对象

通过提供 **上下文** 对象作为第一个参数来调用函数。此对象提供对传入 HTTP 请求信息的访问。

上下文对象示例

```
function handle(context, data)
```

此信息包括 HTTP 请求方法、通过请求发送的任何查询字符串或标头、HTTP 版本和请求正文。传入包含 CloudEvent 的请求将进入 **CloudEvent** 实例附加到上下文对象，以便使用 **context.cloudevent** 访问它。

11.4.3.1.1. 上下文对象方法

上下文（context）对象具有单一方法 **cloudEventResponse ()**，它接受数据值并返回 CloudEvent。

在 Knative 系统中，如果发送 CloudEvent 的事件代理调用将部署为服务的功能，代理会检查响应。如果响应是 CloudEvent，则此事件由代理处理。

上下文对象方法示例

```
// Expects to receive a CloudEvent with customer data
function handle(context, customer) {
  // process the customer
  const processed = handle(customer);
  return context.cloudEventResponse(customer)
    .source('/handle')
    .type('fn.process.customer')
    .response();
}
```

11.4.3.1.2. CloudEvent 数据

如果传入的请求为 CloudEvent，则从事件中提取与 CloudEvent 相关的任何数据，并作为第二个参数提供。例如，如果收到在它的数据属性中包含类似如下的 JSON 字符串的 CloudEvent：

```
{
  "customerId": "0123456",
  "productId": "6543210"
}
```

在调用时。函数的第二个参数（在**上下文** 对象后），将是带有 **customerId** 和 **productId** 属性的 JavaScript 对象。

签**名示例**

```
function handle(context, data)
```

本例中 **data** 参数是一个 JavaScript 对象，其中包含 **customerId** 和 **productId** 属性。

11.4.4. Node.js 功能返回值

功能可以返回任何有效的 JavaScript 类型，或者没有返回值。当函数没有指定返回值且未指示失败时，调用者会收到 **204 No Content** 响应。

功能也可以返回 CloudEvent 或一个 **Message** 对象，以便将事件推送到 Knative Eventing 系统。在这种情况下，开发人员不需要了解或实施 CloudEvent 消息传递规范。使用响应提取并发送返回值中的标头和其他相关信息。

示例

```
function handle(context, customer) {
  // process customer and return a new CloudEvent
  return new CloudEvent({
    source: 'customer.processor',
    type: 'customer.processed'
  })
}
```

11.4.4.1. 返回的标头

您可以通过在 **return** 对象中添加 **headers** 属性来设置响应标头。这些标头会提取并发送至调用者。

响应标头**示例**

```
function handle(context, customer) {
  // process customer and return custom headers
  // the response will be '204 No content'
  return { headers: { customerid: customer.id } };
}
```

11.4.4.2. 返回状态代码

您可以通过在返回对象中添加 **statusCode** 属性来设置 **return** 到调用者的状态代码：

状态代码示例

```
function handle(context, customer) {
  // process customer
  if (customer.restricted) {
    return { statusCode: 451 }
  }
}
```

也可以为函数创建和丢弃的错误设置状态代码：

错误**状**态**代**码**示例**

```
function handle(context, customer) {
  // process customer
  if (customer.restricted) {
    const err = new Error('Unavailable for legal reasons');
    err.statusCode = 451;
    throw err;
  }
}
```

11.4.5. 测试 Node.js 功能

Node.js 功能可以在您的计算机上本地测试。在使用 **kn func create** 创建功能时创建的默认项目中，有一个 **test** 文件夹，其中包含一些简单的单元和集成测试。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 已安装 Knative (**kn**) CLI。
- 已使用 **kn func create** 创建功能。

流程

1. 导航到您的功能的 **test** 文件夹。
2. 运行测试：

```
$ npm test
```

11.4.6. 后续步骤

- 请参阅 Node.js 上下文对象参考文档。
- 构建和部署功能.

11.5. 开发类型脚本功能

重要

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

有关红帽技术预览功能支持范围的详情，请参考 https://access.redhat.com/support/offerings/techpreview/。

创建 TypeScript 功能项目 后，您可以修改提供的模板文件，以将业务逻辑添加到您的功能中。这包括配置功能调用和返回的标头和状态代码。

11.5.1. 先决条件

- 在开发功能前，您必须完成设置 OpenShift Serverless 功能 的步骤。

11.5.2. TypeScript 功能模板结构

使用 Knative (**kn**) CLI 创建 TypeScript 功能时，项目目录类似于典型的 TypeScript 项目。唯一的例外是额外的 **func.yaml** 文件，用于配置函数。

http 和 **event** 触发器功能具有相同的模板结构：

模板结构

```
.
├── func.yaml 1
├── package.json 2
├── package-lock.json
├── README.md
├── src
│   └── index.ts 3
├── test 4
│   ├── integration.ts
│   └── unit.ts
└── tsconfig.json
```

1. **func.yaml** 配置文件用于决定镜像名称和 registry。
2. 您不限于模板 **package.json** 文件中提供的依赖项。您可以添加额外的依赖项，如任何其他 TypeScript 项目中一样。

   添加 npm 依赖项示例

   ```
   npm install --save opossum
   ```

   为部署构建项目时，这些依赖项将包含在创建的运行时容器镜像中。
3. 您的项目必须包含 **src/index.js** 文件，该文件导出名为 **handle** 的函数。
4. 集成和单元测试脚本作为功能模板的一部分提供。

11.5.3. 关于调用 TypeScript 函数

当使用 Knative (**kn**) CLI 创建功能项目时，您可以生成一个响应 CloudEvents 的项目，或者响应简单 HTTP 请求的项目。Knative 中的 CloudEvents 作为 POST 请求通过 HTTP 传输，因此两种功能类型都侦听并响应传入的 HTTP 事件。

TypeScript 函数可通过简单的 HTTP 请求调用。收到传入请求后，将通过 **上下文** 对象作为第一个参数来调用函数。

11.5.3.1. TypeScript 上下文对象

若要调用函数，您可以提供一个 **context** 对象作为第一个参数。访问 **context** 对象的属性可以提供有关传入 HTTP 请求的信息。

上下文对象示例

```
function handle(context:Context): string
```

此信息包括 HTTP 请求方法、通过请求发送的任何查询字符串或标头、HTTP 版本和请求正文。传入包含 CloudEvent 的请求将进入 **CloudEvent** 实例附加到上下文对象，以便使用 **context.cloudevent** 访问它。

11.5.3.1.1. 上下文对象方法

上下文（context）对象具有单一方法 **cloudEventResponse ()**，它接受数据值并返回 CloudEvent。

在 Knative 系统中，如果发送 CloudEvent 的事件代理调用将部署为服务的功能，代理会检查响应。如果响应是 CloudEvent，则此事件由代理处理。

上下文对象方法示例

```
// Expects to receive a CloudEvent with customer data
export function handle(context: Context, cloudevent?: CloudEvent): CloudEvent {
  // process the customer
  const customer = cloudevent.data;
  const processed = processCustomer(customer);
  return context.cloudEventResponse(customer)
    .source('/customer/process')
    .type('customer.processed')
    .response();
}
```

11.5.3.1.2. 上下文类型

TypeScript 类型定义文件导出以下类型以便在您的功能中使用。

导出类型定义

```
// Invokable is the expeted Function signature for user functions
export interface Invokable {
    (context: Context, cloudevent?: CloudEvent): any
}

// Logger can be used for structural logging to the console
export interface Logger {
```

```
  debug: (msg: any) => void,
  info:  (msg: any) => void,
  warn:  (msg: any) => void,
  error: (msg: any) => void,
  fatal: (msg: any) => void,
  trace: (msg: any) => void,
}

// Context represents the function invocation context, and provides
// access to the event itself as well as raw HTTP objects.
export interface Context {
    log: Logger;
    req: IncomingMessage;
    query?: Record<string, any>;
    body?: Record<string, any>|string;
    method: string;
    headers: IncomingHttpHeaders;
    httpVersion: string;
    httpVersionMajor: number;
    httpVersionMinor: number;
    cloudevent: CloudEvent;
    cloudEventResponse(data: string|object): CloudEventResponse;
}

// CloudEventResponse is a convenience class used to create
// CloudEvents on function returns
export interface CloudEventResponse {
    id(id: string): CloudEventResponse;
    source(source: string): CloudEventResponse;
    type(type: string): CloudEventResponse;
    version(version: string): CloudEventResponse;
    response(): CloudEvent;
}
```

11.5.3.1.3. CloudEvent 数据

如果传入的请求为 CloudEvent，则从事件中提取与 CloudEvent 相关的任何数据，并作为第二个参数提供。例如，如果收到在它的数据属性中包含类似如下的 JSON 字符串的 CloudEvent：

```
{
  "customerId": "0123456",
  "productId": "6543210"
}
```

在调用时。函数的第二个参数（在**上下文** 对象后），将是带有 **customerId** 和 **productId** 属性的 JavaScript 对象。

签名示例

```
function handle(context: Context, cloudevent?: CloudEvent): CloudEvent
```

本例中的 **cloudevent** 参数是一个 JavaScript 对象，包含 **customerId** 和 **productId** 属性。

11.5.4. TypeScript 功能返回值

功能可以返回任何有效的 JavaScript 类型，或者没有返回值。当函数没有指定返回值且未指示失败时，调用者会收到 **204 No Content** 响应。

功能也可以返回 CloudEvent 或一个 **Message** 对象，以便将事件推送到 Knative Eventing 系统。在这种情况下，开发人员不需要了解或实施 CloudEvent 消息传递规范。使用响应提取并发送返回值中的标头和其他相关信息。

示例

```
export const handle: Invokable = function (
  context: Context,
  cloudevent?: CloudEvent
): Message {
  // process customer and return a new CloudEvent
  const customer = cloudevent.data;
  return HTTP.binary(
    new CloudEvent({
      source: 'customer.processor',
      type: 'customer.processed'
    })
  );
};
```

11.5.4.1. 返回的标头

您可以通过在 **return** 对象中添加 **headers** 属性来设置响应标头。这些标头会提取并发送至调用者。

响应标头**示例**

```
export function handle(context: Context, cloudevent?: CloudEvent): Record<string, any> {
  // process customer and return custom headers
  const customer = cloudevent.data as Record<string, any>;
  return { headers: { 'customer-id': customer.id } };
}
```

11.5.4.2. 返回状态代码

您可以通过在返回对象中添加 **statusCode** 属性来设置 **return** 到调用者的状态代码：

状态代码示例

```
export function handle(context: Context, cloudevent?: CloudEvent): Record<string, any> {
  // process customer
  const customer = cloudevent.data as Record<string, any>;
  if (customer.restricted) {
    return {
      statusCode: 451
    }
  }
  // business logic, then
  return {
    statusCode: 240
  }
}
```

也可以为函数创建和丢弃的错误设置状态代码：

错误**状态代码示例**

```
export function handle(context: Context, cloudevent?: CloudEvent): Record<string, string> {
  // process customer
  const customer = cloudevent.data as Record<string, any>;
  if (customer.restricted) {
    const err = new Error('Unavailable for legal reasons');
    err.statusCode = 451;
    throw err;
  }
}
```

11.5.5. 测试类型脚本功能

TypeScript 功能可在您的计算机上本地测试。在使用 **kn func create** 创建功能时创建的默认项目中，有一个 **test** 目录，其中包含一些简单的单元和集成测试。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 已安装 Knative (**kn**) CLI。
- 已使用 **kn func create** 创建功能。

流程

1. 如果您之前还没有运行测试，请首先安装依赖项：

```
$ npm install
```

2. 导航到您的功能的 **test** 文件夹。
3. 运行测试：

```
$ npm test
```

11.5.6. 后续步骤

- 请参阅 TypeScript 上下文对象参考文档。
- 构建和部署功能.
- 有关使用功能进行日志记录的更多信息，请参阅 Pino API 文档。

11.6. 开发 GO 功能

重要

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

有关红帽技术预览功能支持范围的详情，请参考 https://access.redhat.com/support/offerings/techpreview/。

创建 Go 功能项目后，您可以修改提供的模板文件，以将业务逻辑添加到您的功能中。这包括配置功能调用和返回的标头和状态代码。

11.6.1. 先决条件

- 在开发功能前，您必须完成设置 OpenShift Serverless 功能的步骤。

11.6.2. Go 功能模板结构

当使用 Knative (**kn**) CLI 创建 Go 功能时，项目目录类似于典型的 Go 项目。唯一的例外是额外的 **func.yaml** 配置文件，用于指定镜像。

Go 功能有一些限制。唯一的要求是您的项目必须在 **function** 模块中定义，并且必须导出功能 **Handle ()** 。

http 和 **event** 触发器功能具有相同的模板结构：

模板结构

```
fn
├── README.md
├── func.yaml 1
├── go.mod 2
├── go.sum
├── handle.go
└── handle_test.go
```

1. **func.yaml** 配置文件用于决定镜像名称和 registry。
2. 您可以在 **go.mod** 文件中添加任何需要的依赖项，该文件可以包括额外的本地 Go 文件。为部署构建项目时，生成的运行时容器镜像中会包含这些依赖项。

   添加依赖项**示例**

   ```
   $ go get gopkg.in/yaml.v2@v2.4.0
   ```

11.6.3. 关于调用 Go 功能

当使用 Knative (**kn**) CLI 创建功能项目时，您可以生成一个响应 CloudEvents 的项目，或者响应简单 HTTP 请求的项目。Go 函数通过不同的方法调用，具体取决于它们是由 HTTP 请求还是 CloudEvent 触发。

11.6.3.1. HTTP 请**求触**发**的功能**

收到传入的 HTTP 请求时，将通过标准 Go Context 作为第一个参数来调用函数，后跟 **http.ResponseWriter** 和 **http.Request** 参数。您可以使用标准 Go 技术访问请求，并为您的功能设置对应的 HTTP 响应。

HTTP 响应示例

```
func Handle(ctx context.Context, res http.ResponseWriter, req *http.Request) {
  // Read body
  body, err := ioutil.ReadAll(req.Body)
  defer req.Body.Close()
  if err != nil {
 http.Error(res, err.Error(), 500)
 return
  }
  // Process body and function logic
  // ...
}
```

11.6.3.2. 云事件触发的功能

收到传入的云事件时，由 CloudEvents Go SDK 调用该事件。调用使用 **Event** 类型作为参数。

您可以使用 Go Context 作为功能合同中的可选参数，如支持的功能签名列表中所示：

支持的功能签名

```
Handle()
Handle() error
Handle(context.Context)
Handle(context.Context) error
Handle(cloudevents.Event)
Handle(cloudevents.Event) error
Handle(context.Context, cloudevents.Event)
Handle(context.Context, cloudevents.Event) error
Handle(cloudevents.Event) *cloudevents.Event
Handle(cloudevents.Event) (*cloudevents.Event, error)
Handle(context.Context, cloudevents.Event) *cloudevents.Event
Handle(context.Context, cloudevents.Event) (*cloudevents.Event, error)
```

11.6.3.2.1. CloudEvent 触发器示例

接收云事件，其中包含 data 属性中的 JSON 字符串：

```
{
  "customerId": "0123456",
  "productId": "6543210"
}
```

若要访问此数据，必须定义一个结构，用于映射云事件数据中的属性，并从传入事件检索数据。以下示例使用 **Purchase** 结构：

```
type Purchase struct {
  CustomerId string `json:"customerId"`
  ProductId  string `json:"productId"`
```

```
}
func Handle(ctx context.Context, event cloudevents.Event) (err error) {

  purchase := &Purchase{}
  if err = event.DataAs(purchase); err != nil {
 fmt.Fprintf(os.Stderr, "failed to parse incoming CloudEvent %s\n", err)
 return
  }
  // ...
}
```

另外，一个 Go **encoding/json** 软件包也可用于以字节阵列的形式直接以 JSON 形式访问云事件：

```
func Handle(ctx context.Context, event cloudevents.Event) {
  bytes, err := json.Marshal(event)
  // ...
}
```

11.6.4. Go 功能返回值

HTTP 请求触发的功能可以直接设置响应。您可以使用 Go http.ResponseWriter 将该功能配置为执行此操作。

HTTP 响应示例

```
func Handle(ctx context.Context, res http.ResponseWriter, req *http.Request) {
  // Set response
  res.Header().Add("Content-Type", "text/plain")
  res.Header().Add("Content-Length", "3")
  res.WriteHeader(200)
  _, err := fmt.Fprintf(res, "OK\n")
  if err != nil {
 fmt.Fprintf(os.Stderr, "error or response write: %v", err)
  }
}
```

云事件触发的功能可能会返回任何内容、**error** 或 **CloudEvent**，从而将事件推送到 Knative Eventing 系统。在这种情况下，您必须为云事件设置唯一 **ID**、正确的 **Source** 和 **Type**。数据可以从定义的结构或者从一个 **映射（map）** 填充。

CloudEvent 响应示例

```
func Handle(ctx context.Context, event cloudevents.Event) (resp *cloudevents.Event, err error) {
  // ...
  response := cloudevents.NewEvent()
  response.SetID("example-uuid-32943bac6fea")
  response.SetSource("purchase/getter")
  response.SetType("purchase")
  // Set the data from Purchase type
  response.SetData(cloudevents.ApplicationJSON, Purchase{
 CustomerId: custId,
 ProductId:  prodId,
  })
  // OR set the data directly from map
```

```
  response.SetData(cloudevents.ApplicationJSON, map[string]string{"customerId": custId, "productId":
prodId})
  // Validate the response
  resp = &response
  if err = resp.Validate(); err != nil {
 fmt.Printf("invalid event created. %v", err)
  }
  return
}
```

11.6.5. 测试 Go 功能

Go 功能可以在您的计算机上进行本地测试。在使用 **kn func create** 创建函数时创建的默认项目中，有一个 **handle_test.go** 文件包含一些基本测试。这些测试可以根据需要扩展。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 已安装 Knative (**kn**) CLI。
- 已使用 **kn func create** 创建功能。

流程

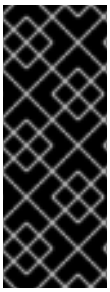
1. 导航到您的功能的 **test** 文件夹。
2. 运行测试：

```
$ go test
```

11.6.6. 后续步骤

- 构建和部署功能.

11.7. 开发 PYTHON 功能



重要

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

有关红帽技术预览功能支持范围的详情，请参考 https://access.redhat.com/support/offerings/techpreview/。

创建 PythonG 功能项目后，您可以修改提供的模板文件，以将业务逻辑添加到您的功能中。这包括配置功能调用和返回的标头和状态代码。

11.7.1. 先决条件

- 在开发功能前，您必须完成设置 OpenShift Serverless 功能 的步骤。

11.7.2. Python 功能模板结构

使用 Knative (**kn**) CLI 创建 Python 功能时，项目目录类似于典型的 Python 项目。Python 功能的限制非常少。唯一的要求是项目包含一个 **func.py** 文件，其中包含一个 **main ()** 函数，以及一个 **func.yaml** 配置文件。

开发人员不限于模板 **requirements.txt** 文件中提供的依赖项。可以像在任何其他 Python 项目中一样添加其他依赖项。为部署构建项目时，这些依赖项将包含在创建的运行时容器镜像中。

http 和 **event** 触发器功能具有相同的模板结构：

模板结构

```
fn
├── func.py 1
├── func.yaml 2
├── requirements.txt 3
└── test_func.py 4
```

1. 包含 **main ()** 函数。
2. 用于确定镜像名称和 registry。
3. 与在任何其他 Python 项目中一样，可以向 **requirements.txt** 文件中添加其他依赖项。
4. 包含一个简单的单元测试，可用于在本地测试您的功能。

11.7.3. 关于调用 Python 功能

Python 功能可以通过简单的 HTTP 请求调用。收到传入请求后，将通过 **上下文** 对象作为第一个参数来调用函数。

上下文 对象是一个 Python 类，具有两个属性：

- **request** 属性始终存在，包含 Flask **请求（request）** 对象。
- 如果传入请求是 **CloudEvent** 对象，则第二个属性 **cloud_event** 会被填充。

开发人员可以从上下文对象访问任何 **CloudEvent** 数据。

上下文对象示例

```
def main(context: Context):
    """
    The context parameter contains the Flask request object and any
    CloudEvent received with the request.
    """
    print(f"Method: {context.request.method}")
    print(f"Event data {context.cloud_event.data}")
    # ... business logic here
```

11.7.4. Python 功能返回值

功能可以返回 Flask 支持的任何值。这是因为调用框架将这些值直接代理到 Flask 服务器。

示例

```
def main(context: Context):
    body = { "message": "Howdy!" }
    headers = { "content-type": "application/json" }
    return body, 200, headers
```

功能可以将标头和响应代码设置为从函数调用的次要和第三响应值。

11.7.4.1. 返回 CloudEvents

开发人员可以使用 **@event** decorator 告知调用器，在发送响应前，函数返回值必须转换为 CloudEvent。

示例

```
@event("event_source"="/my/function", "event_type"="my.type")
def main(context):
    # business logic here
    data = do_something()
    # more data processing
    return data
```

这个示例发送 CloudEvent 作为响应值，类型为 **"my.type"**，源是 **"/my/function"**。CloudEvent **data**属性设置为返回的 **data** 变量。**event_source** 和 **event_type** decorator 属性都是可选的。

11.7.5. 测试 Python 功能

您可以在计算机上本地测试 Python 功能。default 项目包含一个 **test_func.py** 文件，它为函数提供了一个简单的单元测试。

注意

Python 功能的默认测试框架是 **unittest**。如果您愿意，可以使用不同的测试框架。

先决条件

- 要在本地运行 Python 功能测试，您必须安装所需的依赖项：

```
$ pip install -r requirements.txt
```

流程

1. 导航到包含 **test_func.py** 文件的函数的文件夹。
2. 运行测试：

```
$ python3 test_func.py
```

11.7.6. 后续步骤

- 构建和部署功能.

11.8. 开发 QUARKUS 功能

重要

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

有关红帽技术预览功能支持范围的详情，请参考 https://access.redhat.com/support/offerings/techpreview/。

创建 Quarkus 功能项目后，您可以修改提供的模板文件，以将业务逻辑添加到您的功能中。这包括配置功能调用和返回的标头和状态代码。

11.8.1. 先决条件

- 在开发功能前，您必须完成设置 OpenShift Serverless 功能中 的设置步骤。

11.8.2. Quarkus 功能模板结构

使用 Knative (**kn**) CLI 创建 Quarkus 功能时，项目目录类似于典型的 Maven 项目。另外，项目还包含用于配置功能的 **func.yaml** 文件。

http 和 **event** 触发器功能具有相同的模板结构：

模板结构

```
.
├── func.yaml 1
├── mvnw
├── mvnw.cmd
├── pom.xml 2
├── README.md
└── src
    ├── main
    │   ├── java
    │   │   └── functions
    │   │       ├── Function.java 3
    │   │       ├── Input.java
    │   │       └── Output.java
    │   └── resources
    │       └── application.properties
    └── test
        └── java
            └── functions 4
                ├── FunctionTest.java
                └── NativeFunctionIT.java
```

1 用于确定镜像名称和 registry。

2 项目对象模型（POM）文件包含项目配置，如依赖项的相关信息。您可以通过修改此文件来添加额外的依赖项。

外的依赖项。

其他依赖项示例

```
...
 <dependencies>
  <dependency>
   <groupId>junit</groupId>
   <artifactId>junit</artifactId>
   <version>4.11</version>
   <scope>test</scope>
  </dependency>
  <dependency>
   <groupId>org.assertj</groupId>
   <artifactId>assertj-core</artifactId>
   <version>3.8.0</version>
   <scope>test</scope>
  </dependency>
 </dependencies>
...
```

依赖项在第一次编译时下载。

3 功能项目必须包含标有 **@Funq** 的 Java 方法。您可以将此方法放置在 **Function.java** 类中。

4 包含可用于在本地测试功能的简单测试案例。

11.8.3. 关于调用 Quarkus 功能

您可以创建一个 Quarkus 项目来响应云事件，或创建响应简单 HTTP 请求的 Quarkus 项目。Knative 中的云事件作为 POST 请求通过 HTTP 传输，因此任一功能类型都可以侦听和响应传入的 HTTP 请求。

收到传入请求时，通过允许类型的实例调用 Quarkus 函数。

表 11.1. 功能调用选项

调用方法	实例中包含的数据类型	数据示例
HTTP POST 请求	请求正文中的 JSON 对象	**{ "customerId": "0123456", "productId": "6543210" }**
HTTP GET 请求	查询字符串中的数据	**?customerId=0123456&productId=6543210**
CloudEvent	**data** 属性中的 JSON 对象	**{ "customerId": "0123456", "productId": "6543210" }**

以下示例显示了接收并处理上表中列出的 **customerId** 和 **productId** 购买数据的函数：

Quarkus 功能示例

```
public class Functions {
```

```
    @Funq
    public void processPurchase(Purchase purchase) {
        // process the purchase
    }
}
```

包含购买数据的对应 **Purchase** JavaBean 类如下：

类**示例**

```
public class Purchase {
    private long customerId;
    private long productId;
    // getters and setters
}
```

11.8.3.1. 调用示例

以下示例代码定义了名为 **withBeans**、**withCloudEvent** 和 **withBinary** 的三个功能：

示例

```
import io.quarkus.funqy.Funq;
import io.quarkus.funqy.knative.events.CloudEvent;

public class Input {
    private String message;

    // getters and setters
}

public class Output {
    private String message;

    // getters and setters
}

public class Functions {
    @Funq
    public Output withBeans(Input in) {
        // function body
    }

    @Funq
    public CloudEvent<Output> withCloudEvent(CloudEvent<Input> in) {
        // function body
    }

    @Funq
    public void withBinary(byte[] in) {
        // function body
    }
}
```

Functions 类的 **withBeans** 功能可以通过以下方法调用：

- 带有 JSON 正文的 HTTP POST 请求：

```
$ curl "http://localhost:8080/withBeans" -X POST \
    -H "Content-Type: application/json" \
    -d '{"message": "Hello there."}'
```

- 带有查询参数的 HTTP GET 请求：

```
$ curl "http://localhost:8080/withBeans?message=Hello%20there." -X GET
```

- 二进制编码中的 **CloudEvent** 对象：

```
$ curl "http://localhost:8080/" -X POST \
  -H "Content-Type: application/json" \
  -H "Ce-SpecVersion: 1.0" \
  -H "Ce-Type: withBeans" \
  -H "Ce-Source: cURL" \
  -H "Ce-Id: 42" \
  -d '{"message": "Hello there."}'
```

- 结构化编码中的 **CloudEvent** 对象：

```
$ curl http://localhost:8080/ \
    -H "Content-Type: application/cloudevents+json" \
    -d '{ "data": {"message":"Hello there."},
          "datacontenttype": "application/json",
          "id": "42",
          "source": "curl",
          "type": "withBeans",
          "specversion": "1.0"}'
```

与 **withBeans** 函数类似，可以利用 **CloudEvent** 对象来调用 **Functions** 类的 **withCloudEvent** 功能。但是，与 **Beans** 不同，**CloudEvent** 无法通过普通 HTTP 请求来调用。

Functions 类的 **withBinary** 功能可通过以下方式调用：

- 二进制编码中的 **CloudEvent** 对象：

```
$ curl "http://localhost:8080/" -X POST \
  -H "Content-Type: application/octet-stream" \
  -H "Ce-SpecVersion: 1.0"\
  -H "Ce-Type: withBinary" \
  -H "Ce-Source: cURL" \
  -H "Ce-Id: 42" \
  --data-binary '@img.jpg'
```

- 结构化编码中的 **CloudEvent** 对象：

```
$ curl http://localhost:8080/ \
  -H "Content-Type: application/cloudevents+json" \
  -d "{ \"data_base64\": \"$(base64 --wrap=0 img.jpg)\",
        \"datacontenttype\": \"application/octet-stream\",
```

```
\"id\": \"42\",
\"source\": \"curl\",
\"type\": \"withBinary\",
\"specversion\": \"1.0\"}"
```

11.8.4. CloudEvent 属性

如果您需要读取或写入 CloudEvent 的属性，如 **type** 或 **subject**，您可以使用 **CloudEvent<T>** 通用接口和 **CloudEventBuilder** 构建器。**<T>** 类型参数必须是允许的类型之一。

在以下示例中，**CloudEventBuilder** 用于返回处理订购的成功或失败：

```
public class Functions {

    private boolean _processPurchase(Purchase purchase) {
        // do stuff
    }

    public CloudEvent<Void> processPurchase(CloudEvent<Purchase> purchaseEvent) {
        System.out.println("subject is: " + purchaseEvent.subject());

        if (!_processPurchase(purchaseEvent.data())) {
            return CloudEventBuilder.create()
                    .type("purchase.error")
                    .build();
        }
        return CloudEventBuilder.create()
                .type("purchase.success")
                .build();
    }
}
```

11.8.5. Quarkus 功能返回值

功能可以从允许类型列表中返回任何类型的实例。另外，他们可以返回 **Uni<T>** 类型，其中 **<T>** 类型参数可以是允许的类型的任何类型。

如果函数调用异步 API，因为返回的对象以与接收对象相同的格式序列化，**Uni<T>** 类型很有用。例如：

- 如果函数收到 HTTP 请求，则返回的对象将在 HTTP 响应的正文中发送。
- 如果函数通过二进制编码收到 **CloudEvent** 对象，则返回的对象将在二进制编码的 **CloudEvent** 对象的 data 属性中发送。

以下示例显示了获取购买列表的功能：

示例命令

```
public class Functions {
    @Funq
    public List<Purchase> getPurchasesByName(String name) {
      // logic to retrieve purchases
    }
}
```

- 通过 HTTP 请求调用此功能将生成 HTTP 响应，其中包含响应正文中的订购列表。
- 通过传入的 **CloudEvent** 对象调用此功能可生成 **CloudEvent** 响应，并在 **data** 属性中包括一个订购列表。

11.8.5.1. 允许的类型

功能的输入和输出可以是 **void**、**String**、或 **byte[]** 类型。此外，它们也可以是原语类型及其打包程序，例如 **int** 和 **Integer**。它们也可以是以下复杂的对象：Javabeans、映射、列表、数组和特殊的 **CloudEvents<T>** 类型。

映射、列出、数组、**CloudEvents<T>** 类型的 **<T>** 类型参数以及 Javabeans 的属性只能是此处列出的类型。

示例

```
public class Functions {
    public List<Integer> getIds();
    public Purchase[] getPurchasesByName(String name);
    public String getNameById(int id);
    public Map<String,Integer> getNameIdMapping();
    public void processImage(byte[] img);
}
```

11.8.6. 测试 Quarkus 功能

Quarkus 功能可以在您的计算机上进行本地测试。在使用 **kn func create** 创建功能时创建的默认项目中，有 **src/test/** 目录，其中包含基本的 Maven 测试。这些测试可以根据需要扩展。

先决条件

- 您已创建了 Quarkus 功能。
- 已安装 Knative (**kn**) CLI。

流程

1. 导航到您的功能的项目文件夹。
2. 运行 Maven 测试：

```
$ ./mvnw test
```

11.8.7. 后续步骤

- 构建和部署功能.

11.9. FUNC.YAML 中的功能项目配置

func.yaml 文件包含功能项目的配置。执行 **kn func** 命令时使用 **func.yaml** 中指定的值。例如，当运行 **kn func build** 命令时，会使用 **build** 字段中的值。在某些情况下，您可以使用命令行标志或环境变量覆盖这些值。

11.9.1. func.yaml 中的可配置字段

在创建、构建和部署您的功能时，**func.yaml** 中的许多字段会自动生成。但是，您也可以手动修改以更改操作，如函数名称或镜像名称。

11.9.1.1. buildEnvs

buildEnvs 字段允许您设置环境变量，供构建您的功能的环境使用。与使用 **envs** 设置的变量不同，使用 **buildEnv** 的变量集合在函数运行时不可用。

您可以直接从值设置 **buildEnv** 变量。在以下示例中，名为 **EXAMPLE1** 的 **buildEnv** 变量被直接分配为 **one** 值：

```
buildEnvs:
- name: EXAMPLE1
  value: one
```

您还可以从本地环境变量设置 **buildEnv** 变量。在以下示例中，名为 **EXAMPLE2** 的 **buildEnv** 变量被分配了 **LOCAL_ENV_VAR** 本地环境变量的值：

```
buildEnvs:
- name: EXAMPLE1
  value: '{{ env:LOCAL_ENV_VAR }}'
```

11.9.1.2. envs

envs 字段允许您在运行时设置环境变量供您的功能使用。您可以通过几种不同方式设置环境变量：

1. 直接来自一个值。
2. 来自分配给本地环境变量的值）。如需更多信息，请参阅"引用来自 func.yaml 字段中的本地环境变量"。
3. 从存储在 secret 或配置映射中的键值对。
4. 您还可以导入存储在 secret 或配置映射中的所有键值对，其键用作所创建的环境变量的名称。

这个示例演示了设置环境变量的不同方法：

```
name: test
namespace: ""
runtime: go
...
envs:
- name: EXAMPLE1 (1)
  value: value
- name: EXAMPLE2 (2)
  value: '{{ env:LOCAL_ENV_VALUE }}'
- name: EXAMPLE3 (3)
  value: '{{ secret:mysecret:key }}'
- name: EXAMPLE4 (4)
  value: '{{ configMap:myconfigmap:key }}'
- value: '{{ secret:mysecret2 }}' (5)
- value: '{{ configMap:myconfigmap2 }}' (6)
```

1. 直接从值设置的环境变量。
2. 从分配给本地环境变量的值设置的环境变量。
3. 从存储在 secret 中的键值对分配的环境变量。
4. 从配置映射中存储的键值对分配的环境变量。
5. 从 secret 的键值对导入的一组环境变量。
6. 从配置映射的键值对导入的一组环境变量。

11.9.1.3. builder

builder 字段指定函数用于构建镜像的策略。它接受 **pack** 或 **s2i** 的值。

11.9.1.4. build

build 字段指示如何构建函数。**local** 值表示该函数在您的机器上本地构建。**git** 值表示函数使用 **git** 字段中指定的值来在集群中构建。

11.9.1.5. 卷

volumes 字段允许您将 secret 和配置映射作为可在指定路径的函数访问的卷挂载，如下例所示：

```
name: test
namespace: ""
runtime: go
...
volumes:
- secret: mysecret (1)
  path: /workspace/secret
- configMap: myconfigmap (2)
  path: /workspace/configmap
```

1. **mysecret** secret 作为驻留于 **/workspace/secret** 的卷挂载。
2. **myconfigmap** 配置映射作为驻留于 **/workspace/configmap** 的卷挂载。

11.9.1.6. 选项

options 字段允许您修改部署的功能的 Knative Service 属性，如自动扩展。如果未设置这些选项，则使用默认的选项。

这些选项可用：

- **scale**
  - **min**：最小副本数。必须是一个非负的整数。默认值为 0。
  - **max** ：最大副本数。必须是一个非负的整数。默认值为 0，这代表没有限制。

- **metric**：定义 Autoscaler 监视哪一指标类型。它可以被设置为 **concurrency**（默认），或 **rps**。
- **target**：建议根据同时传入的请求数量，何时向上扩展。**target** 选项可以是大于 0.01 的浮点值。除非设置了 **options.resources.limits.concurrency**，否则默认为 **100**，在这种情况下，目标默认为其值。
- **utilization**：向上扩展前允许的并发请求利用率百分比.它可以是 1 到 100 之间的一个浮点值。默认值为 70。

- **资源**
  - **requests**
    - **cpu**：具有部署功能的容器的 CPU 资源请求。
    - **memory**：具有部署功能的容器的内存资源请求。
  - **limits**
    - **cpu**：具有部署功能的容器的 CPU 资源限值。
    - **memory**：具有部署功能的容器的内存资源限制。
    - **concurrency**：单个副本处理的并发请求的硬限制。它可以是大于或等于 0 的整数值，默认为 0 - 表示无限制。

这是 **scale** 选项配置示例：

```
name: test
namespace: ""
runtime: go
...
options:
  scale:
    min: 0
    max: 10
    metric: concurrency
    target: 75
    utilization: 75
  resources:
    requests:
      cpu: 100m
      memory: 128Mi
    limits:
      cpu: 1000m
      memory: 256Mi
      concurrency: 100
```

11.9.1.7. image

image 字段在构建后为您的功能设置镜像名称。您可以修改此字段。如果您这样做，在下次运行 **kn func build** 或 **kn func deploy** 时，功能镜像将使用新名称创建。

11.9.1.8. imageDigest

在部署函数时，**imageDigest** 字段包含镜像清单的 SHA256 哈希。不要修改这个值。

11.9.1.9. labels

labels 字段允许您在部署的功能中设置标签。

您可以直接从值设置标签。在以下示例中，带有 **role** 键的标签直接被分配了 **backend** 的值：

```
labels:
- key: role
  value: backend
```

您还可以从本地环境变量设置标签。在以下示例中，为带有 **author** 键的标签分配 **USER** 本地环境变量的值：

```
labels:
- key: author
  value: '{{ env:USER }}'
```

11.9.1.10. name

name 字段定义您的函数的名称。该值在部署时用作 Knative 服务的名称。您可以更改此字段来重命名后续部署中的函数。

11.9.1.11. namespace

namespace 字段指定部署您的功能的命名空间。

11.9.1.12. runtime

runtime 字段指定您的功能的语言运行时，如 **python**。

11.9.2. 从 func.yaml 字段引用本地环境变量

如果要避免在功能配置中存储敏感信息，如 API 密钥，您可以添加对本地环境中可用的环境变量的引用。您可以通过修改 **func.yaml** 文件中的 **envs** 字段来完成此操作。

先决条件

- 您需要创建 function 项目。
- 本地环境需要包含您要引用的变量。

流程

- 要引用本地环境变量，请使用以下语法：

  ```
  {{ env:ENV_VAR }}
  ```

  将 **ENV_VAR** 替换为您要用于本地环境中的变量名称。

  例如，您可能在本地环境中提供 **API_KEY** 变量。您可以将其值分配给 **MY_API_KEY** 变量，然后您可以在功能内直接使用该变量：

功能示例

```
name: test
namespace: ""
runtime: go
...
envs:
- name: MY_API_KEY
  value: '{{ env:API_KEY }}'
...
```

11.9.3. 其他资源

- 功能入门
- 从 Serverless 功能访问 secret 和配置映射
- 自动扩展的 Knative 文档
- 管理容器的资源的 Kubernetes 文档
- 有关配置并发的 Knative 文档

11.10. 从功能访问 SECRET 和配置映射

将功能部署到集群后，可以访问存储在 secret 和配置映射中的数据。此数据可以挂载为卷，或分配到环境变量。您可以使用 Knative CLI 以互动方式配置此访问，或者通过编辑功能配置 YAML 文件来手动配置。

重要

要访问 secret 和配置映射，必须在集群中部署该功能。此功能不适用于本地运行的函数。

如果无法访问 secret 或配置映射值，则部署会失败，并显示一条错误消息，指定不可访问的值。

11.10.1. 以互动方式修改对 secret 和配置映射的功能访问

您可以使用 **kn func config** 互动程序来管理您的功能访问的 secret 和配置映射。可用的操作包括列表、添加和删除配置映射和 secret 中存储的值，以及列出、添加和删除卷。通过此功能，您可以管理集群中存储的数据，可以被您的功能访问。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 已安装 Knative (**kn**) CLI。
- 您已创建了一个功能。

流程

1. 在功能项目目录中运行以下命令：

```
$ kn func config
```

-

或者，您可以使用 **--path** 或 **-p** 选项指定功能项目目录。

2. 使用交互式界面执行必要的操作。例如，使用工具列出配置的卷会生成类似如下的输出：

```
$ kn func config
? What do you want to configure? Volumes
? What operation do you want to perform? List
Configured Volumes mounts:
- Secret "mysecret" mounted at path: "/workspace/secret"
- Secret "mysecret2" mounted at path: "/workspace/secret2"
```

这个方案显示互动工具中所有可用的操作以及如何导航到它们：

```
kn func config
  ├─> Environment variables
  │           ├─> Add
  │           │    ├─> ConfigMap: Add all key-value pairs from a config map
  │           │    ├─> ConfigMap: Add value from a key in a config map
  │           │    ├─> Secret: Add all key-value pairs from a secret
  │           │    └─> Secret: Add value from a key in a secret
  │           ├─> List: List all configured environment variables
  │           └─> Remove: Remove a configured environment variable
  └─> Volumes
        ├─> Add
        │    ├─> ConfigMap: Mount a config map as a volume
        │    └─> Secret: Mount a secret as a volume
        ├─> List: List all configured volumes
        └─> Remove: Remove a configured volume
```

3. 可选。部署该功能以使更改生效：

```
$ kn func deploy -p test
```

11.10.2. 使用专用命令以互动方式修改对 secret 和配置映射的功能访问

每次运行 **kn func config** 时，您需要浏览整个对话框来选择您需要的操作，如上一节中所示。要保存步骤，您可以通过运行更具体的 **kn func config** 命令来直接执行特定的操作：

- 列出配置的环境变量：

```
$ kn func config envs [-p <function-project-path>]
```

- 在功能配置中添加环境变量：

```
$ kn func config envs add [-p <function-project-path>]
```

- 从功能配置中删除环境变量：

```
$ kn func config envs remove [-p <function-project-path>]
```

- 列出配置的卷：

```
$ kn func config volumes [-p <function-project-path>]
```

- 在功能配置中添加卷：

```
$ kn func config volumes add [-p <function-project-path>]
```

- 从功能配置中删除卷：

```
$ kn func config volumes remove [-p <function-project-path>]
```

11.10.3. 手动添加对 secret 和配置映射的功能访问

您可以将用于访问 secret 和配置映射的配置手动添加到您的功能中。这可能最好使用 **kn func config** 交互式实用程序和命令，例如您已有配置片段时。

11.10.3.1. 将 secret 挂载为卷

您可以将 secret 挂载为卷。挂载 secret 后，您可以作为常规文件从函数访问它。这可让您存储在功能所需的集群数据中，例如，函数需要访问的 URI 列表。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 已安装 Knative (**kn**) CLI。
- 您已创建了一个功能。

流程

1. 为您的功能打开 **func.yaml** 文件。
2. 对于您要挂载为卷的每个 secret，将以下 YAML 添加到 **volumes** 部分：

```
name: test
namespace: ""
runtime: go
...
volumes:
- secret: mysecret
  path: /workspace/secret
```

   - 将 **mysecret** 替换为目标 secret 的名称。
   - 将 **/workspace/secret** 替换为您要挂载 secret 的路径。
     例如，要挂载 **addresses** secret，请使用以下 YAML：

```
name: test
namespace: ""
runtime: go
...
```

```
volumes:
- configMap: addresses
  path: /workspace/secret-addresses
```

3. 保存配置。

11.10.3.2. 将配置映射挂载为卷

您可以将配置映射挂载为卷。挂载配置映射后，您可以作为常规文件从函数访问它。这可让您存储在功能所需的集群数据中，例如，函数需要访问的 URI 列表。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 已安装 Knative (**kn**) CLI。
- 您已创建了一个功能。

流程

1. 为您的功能打开 **func.yaml** 文件。
2. 对于您要挂载为卷的每个配置映射，请将以下 YAML 添加到 **volumes** 部分：

```
name: test
namespace: ""
runtime: go
...
volumes:
- configMap: myconfigmap
  path: /workspace/configmap
```

   - 将 **myconfigmap** 替换为目标配置映射的名称。
   - 使用您要挂载配置映射的路径替换 **/workspace/configmap**。
     例如，要挂载 **addresses** 配置映射，请使用以下 YAML：

```
name: test
namespace: ""
runtime: go
...
volumes:
- configMap: addresses
  path: /workspace/configmap-addresses
```

3. 保存配置。

11.10.3.3. 从 secret 中定义的键值设置环境变量

您可以从定义为 secret 的键值设置环境变量。然后，之前存储在 secret 中的值可以被函数在运行时作为环境变量访问。这有助于获取存储在 secret 中的值，如用户的 ID。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 已安装 Knative (**kn**) CLI。
- 您已创建了一个功能。

流程

1. 为您的功能打开 **func.yaml** 文件。
2. 对于您要分配给环境变量的 secret 键值对的每个值，请将以下 YAML 添加到 **envs** 部分：

```
name: test
namespace: ""
runtime: go
...
envs:
- name: EXAMPLE
  value: '{{ secret:mysecret:key }}'
```

   - 将 **EXAMPLE** 替换为环境变量的名称。
   - 将 **mysecret** 替换为目标 secret 的名称。
   - 使用映射到目标值的键替换 **key**。
     例如，要访问存储在 **userdetailssecret** 中的用户 ID，请使用以下 YAML：

```
name: test
namespace: ""
runtime: go
...
envs:
- value: '{{ configMap:userdetailssecret:userid }}'
```

3. 保存配置。

11.10.3.4. 从配置映射中定义的键值设置环境变量

您可以从定义为配置映射的键值设置环境变量。然后，之前存储在配置映射中的值可以被函数在运行时作为环境变量访问。这对于获取配置映射中存储的值（如用户的 ID）非常有用。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 已安装 Knative (**kn**) CLI。
- 您已创建了一个功能。

流程

1. 为您的功能打开 **func.yaml** 文件。
2. 对于您要分配给环境变量的配置映射键值对中的每个值，请将以下 YAML 添加到 **envs** 部分：

```
name: test
namespace: ""
runtime: go
...
envs:
- name: EXAMPLE
  value: '{{ configMap:myconfigmap:key }}'
```

- 将 **EXAMPLE** 替换为环境变量的名称。
- 将 **myconfigmap** 替换为目标配置映射的名称。
- 使用映射到目标值的键替换 **key**。
  例如，要访问存储在 **userdetailsmap** 中的用户 ID，请使用以下 YAML：

```
name: test
namespace: ""
runtime: go
...
envs:
- value: '{{ configMap:userdetailsmap:userid }}'
```

3. 保存配置。

11.10.3.5. 从 secret 中定义的所有值设置环境变量

您可以从 secret 中定义的所有值设置环境变量。然后，之前存储在 secret 中的值可以被函数在运行时作为环境变量访问。这可用于同时访问存储在 secret 中的一组值，例如，一组与用户相关的数据。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 已安装 Knative (**kn**) CLI。
- 您已创建了一个功能。

流程

1. 为您的功能打开 **func.yaml** 文件。
2. 对于您要导入所有键值对作为环境变量的每个 secret，请将以下 YAML 添加到 **envs** 部分：

```
name: test
namespace: ""
runtime: go
...
envs:
- value: '{{ secret:mysecret }}' (1)
```

(1) 将 **mysecret** 替换为目标 secret 的名称。

例如，要访问存储在 **userdetailssecret** 中的所有用户数据，请使用以下 YAML：

-

```
name: test
namespace: ""
runtime: go
...
envs:
- value: '{{ configMap:userdetailssecret }}'
```

3. 保存配置。

11.10.3.6. 从配置映射中定义的所有值设置环境变量

您可以从配置映射中定义的所有值设置环境变量。然后，之前存储在配置映射中的值可以被函数在运行时作为环境变量访问。这可用于同时访问配置映射中存储的值集合，例如，一组与用户相关的数据。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 已安装 Knative (**kn**) CLI。
- 您已创建了一个功能。

流程

1. 为您的功能打开 **func.yaml** 文件。
2. 对于您要导入所有键值对作为环境变量的每个配置映射，请将以下 YAML 添加到 **envs** 部分：

```
name: test
namespace: ""
runtime: go
...
envs:
- value: '{{ configMap:myconfigmap }}' (1)
```

(1) 将 **myconfigmap** 替换为目标配置映射的名称。

例如，要访问存储在 **userdetailsmap** 中的所有用户数据，请使用以下 YAML：

```
name: test
namespace: ""
runtime: go
...
envs:
- value: '{{ configMap:userdetailsmap }}'
```

3. 保存该文件。

11.11. 在功能中添加注解

您可以将 Kubernetes 注解添加到部署的 Serverless 功能中。注解可让您将任意元数据附加到函数，例如，关于功能目的的备注。注解添加到 **func.yaml** 配置文件的 **annotations** 部分。

功能注解功能有两个限制：

- 当功能注解传播到集群中的对应 Knative 服务后，无法通过从 **func.yaml** 文件中删除该服务来将其从服务中删除。您必须通过直接修改服务的 YAML 文件或使用 OpenShift Container Platform Web 控制台从 Knative 服务中删除注解。
- 您无法设置 Knative 设置的注解，例如 **autoscaling** 注解。

11.11.1. 在功能中添加注解

您可以在功能中添加注解。与标签类似，注解被定义为键值映射。例如，注解可用于提供与功能相关的元数据，如函数的作者。

先决条件

- 在集群中安装了 OpenShift Serverless Operator 和 Knative Serving。
- 已安装 Knative (**kn**) CLI。
- 您已创建了一个功能。

流程

1. 为您的功能打开 **func.yaml** 文件。
2. 对于您要添加的每个注解，将以下 YAML 添加到 **annotations** 部分：

```
name: test
namespace: ""
runtime: go
...
annotations:
  <annotation_name>: "<annotation_value>" 1
```

1 将 **<annotation_name>: "<annotation_value>"** 替换为您的注解。

例如，要指明某个函数由 Alice 编写，您可以包含以下注解：

```
name: test
namespace: ""
runtime: go
...
annotations:
  author: "alice@example.com"
```

3. 保存配置。

下次将功能部署到集群中时，注解会添加到对应的 Knative 服务中。

11.12. 功能开发参考指南

重要

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

有关红帽技术预览功能支持范围的详情，请参考 https://access.redhat.com/support/offerings/techpreview/。

OpenShift Serverless 功能提供模板，可用于创建基本功能。模板启动功能项目 boilerplate，并准备好将其用于 **kn func** 工具。每个功能模板是为特定的运行时量身定制的，并遵循其约定。使用模板，您可以自动启动功能项目。

可用的运行时模板如下：

- Node.js
- Python
- Go
- Quarkus
- TypeScript

11.12.1. Node.js 上下文对象引用

该 **上下文** 对象具有多个属性，可供函数开发人员访问。访问这些属性可提供有关 HTTP 请求的信息，并将输出写入集群日志。

11.12.1.1. log

提供一个日志记录对象，可用于将输出写入集群日志。日志遵循 Pino 日志记录 API。

日志示例

```
function handle(context) {
  context.log.info(“Processing customer”);
}
```

您可以使用 **kn func invoke** 命令访问功能：

示例命令

```
$ kn func invoke --target 'http://example.function.com'
```

输出示例

```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"Processing customer"}
```

您可以将日志级别更改为 **fatal**、**error**、**warn**、**info**、**debug**、**trace** 或 **silent** 之一。为此，请使用 **config** 命令将其中的一个值分配给环境变量 **FUNC_LOG_LEVEL**，以更改 **logLevel** 的值。

11.12.1.2. query

返回请求的查询字符串（如果有），作为键值对。这些属性也可在上下文对象本身中找到。

查询**示例**

```
function handle(context) {
  // Log the 'name' query parameter
  context.log.info(context.query.name);
  // Query parameters are also attached to the context
  context.log.info(context.name);
}
```

您可以使用 **kn func invoke** 命令访问功能：

示例命令

```
$ kn func invoke --target 'http://example.com?name=tiger'
```

输**出示例**

```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"tiger"}
```

11.12.1.3. 正文（body）

如果有，返回请求正文。如果请求正文包含 JSON 代码，这将会进行解析，以便属性可以直接可用。

body 示例

```
function handle(context) {
  // log the incoming request body's 'hello' parameter
  context.log.info(context.body.hello);
}
```

您可以使用 **curl** 命令调用该函数：

示例命令

```
$ kn func invoke -d '{"Hello": "world"}'
```

输**出示例**

```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"world"}
```

11.12.1.4. 标头

将 HTTP 请求标头返回为对象。

标头**示例**

```
function handle(context) {
  context.log.info(context.headers["custom-header"]);
}
```

您可以使用 **kn func invoke** 命令访问功能：

示例命令

```
$ kn func invoke --target 'http://example.function.com'
```

输**出示例**

```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"some-value"}
```

11.12.1.5. HTTP 请求

方法

以字符串形式返回 HTTP 请求方法。

httpVersion

以字符串形式返回 HTTP 版本。

httpVersionMajor

将 HTTP 主版本号返回为字符串。

httpVersionMinor

以字符串形式返回 HTTP 次要版本号。

11.12.2. TypeScript 上下文对象引用

该 **上下文** 对象具有多个属性，可供函数开发人员访问。访问这些属性可提供有关传入 HTTP 请求的信息，并将输出写入集群日志。

11.12.2.1. log

提供一个日志记录对象，可用于将输出写入集群日志。日志遵循 Pino 日志记录 API。

日志示例

```
export function handle(context: Context): string {
    // log the incoming request body's 'hello' parameter
    if (context.body) {
      context.log.info((context.body as Record<string, string>).hello);
    } else {
      context.log.info('No data received');
    }
    return 'OK';
}
```

您可以使用 **kn func invoke** 命令访问功能：

示例命令

```
$ kn func invoke --target 'http://example.function.com'
```

输出示例

```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"Pr
ocessing customer"}
```

您可以将日志级别更改为 **fatal**、**error**、**warn**、**info**、**debug**、**trace** 或 **silent** 之一。为此，请使用 **config** 命令将其中的一个值分配给环境变量 **FUNC_LOG_LEVEL**，以更改 **logLevel** 的值。

11.12.2.2. query

返回请求的查询字符串（如果有），作为键值对。这些属性也可在上下文对象本身中找到。

查询示例

```
export function handle(context: Context): string {
    // log the 'name' query parameter
  if (context.query) {
    context.log.info((context.query as Record<string, string>).name);
  } else {
    context.log.info('No data received');
  }
  return 'OK';
}
```

您可以使用 **kn func invoke** 命令访问功能：

示例命令

```
$ kn func invoke --target 'http://example.function.com' --data '{"name": "tiger"}'
```

输出示例

```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"tig
er"}
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"tig
er"}
```

11.12.2.3. 正文（body）

返回请求正文（如果有）。如果请求正文包含 JSON 代码，这将会进行解析，以便属性可以直接可用。

body 示例

```
export function handle(context: Context): string {
  // log the incoming request body's 'hello' parameter
  if (context.body) {
    context.log.info((context.body as Record<string, string>).hello);
  } else {
    context.log.info('No data received');
```

```
    }
    return 'OK';
}
```

您可以使用 **kn func invoke** 命令访问功能：

示例命令

```
$ kn func invoke --target 'http://example.function.com' --data '{"hello": "world"}'
```

输**出示例**

```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"world"}
```

11.12.2.4. 标头

将 HTTP 请求标头返回为对象。

标头**示例**

```
export function handle(context: Context): string {
    // log the incoming request body's 'hello' parameter
    if (context.body) {
      context.log.info((context.headers as Record<string, string>)['custom-header']);
    } else {
      context.log.info('No data received');
    }
    return 'OK';
}
```

您可以使用 **curl** 命令调用该函数：

示例命令

```
$ curl -H'x-custom-header: some-value" http://example.function.com
```

输**出示例**

```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"some-value"}
```

11.12.2.5. HTTP 请求

方法

以字符串形式返回 HTTP 请求方法。

httpVersion

以字符串形式返回 HTTP 版本。

httpVersionMajor

将 HTTP 主版本号返回为字符串。

httpVersionMinor

以字符串形式返回 HTTP 次要版本号。

第 12 章 集成

12.1. 将 SERVERLESS 与成本管理服务集成

Cost management 是一种 OpenShift Container Platform 服务，可让您更好地了解和跟踪云和容器的成本。它基于开源 Koku 项目。

12.1.1. 先决条件

- 有集群管理员权限。
- 您已设置成本管理，并添加了 OpenShift Container Platform 源。

12.1.2. 使用标签进行成本管理查询

标签（label）（在成本管理中也称为 *tag* ）可用于节点、命名空间或 pod。每个标签都是键和值对。您可以使用多个标签的组合来生成报告。您可以使用红帽混合控制台访问成本的相关报告。

标签从节点继承到命名空间，并从命名空间继承到 pod。但是，如果标签已在资源中已存在，则标签不会被覆盖。例如，Knative 服务具有默认的 **app=<revision_name>** 标签：

Knative 服务默认标签示例

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: example-service
spec:
...
      labels:
        app: <revision_name>
...
```

如果您为命名空间定义标签，如 **app=my-domain**，在查询使用 **app=my-domain** 标签的应用程序时，成本管理服务不会考虑带有 **app=<revision_name>** 标签的 Knative 服务的成本。具有此标签的 Knative 服务的成本必须在 **app=<revision_name>** 标签下查询。

12.1.3. 其他资源

- 为您的源配置标签
- 使用 Cost Explorer 来视觉化您的成本

12.2. 使用无服务器应用程序的 NVIDIA GPU 资源

NVIDIA 支持在 OpenShift Container Platform 上试验性地使用 GPU 资源。如需在OpenShift Container Platform 上设置 GPU 资源的更多信息，请参阅 NVIDIA GPU 加速集群上的 OpenShift Container Platform 。

12.2.1. 为服务指定 GPU 要求

为 OpenShift Container Platform 集群启用 GPU 资源后，您可以使用 Knative (**kn**) CLI 为 Knative 服务指定 GPU 要求。

先决条件

- 在集群中安装了 OpenShift Serverless Operator、Knative Serving 和 Knative Eventing。
- 已安装 Knative (**kn**) CLI。
- 为 OpenShift Container Platform 集群启用 GPU 资源。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。

注意

IBM Z 和 IBM Power Systems 不支持使用 NVIDIA GPU 资源。

流程

1. 创建 Knative 服务并使用 **--limit nvidia.com/gpu=1** 标志将 GPU 资源要求限制设置为 **1**：

```
$ kn service create hello --image <service-image> --limit nvidia.com/gpu=1
```

GPU 资源要求限制为 **1** 表示该服务有 1 个专用的 GPU 资源。服务不共享 GPU 资源。所有需要 GPU 资源的其他服务都必须等待 GPU 资源不再被使用为止。

限值为 1 个 GPU 意味着超过使用 1 个 GPU 资源的应用程序会受到限制。如果服务请求超过 1 个 GPU 资源，它将部署到可以满足 GPU 资源要求的节点。

2. 可选。对于现有服务，您可以使用 **--limit nvidia.com/gpu=3** 标志将 GPU 资源要求限制改为 **3**：

```
$ kn service update hello --limit nvidia.com/gpu=3
```

12.2.2. 其他资源

- 为扩展资源设置资源配额