Red Hat build of Apache Camel K 1.10.5

使用 Camel K 开发和测试集成

Camel K 的开发人员指南

Last Updated: 2024-03-06

Red Hat build of Apache Camel K 1.10.5 使用 Camel K 开发和测试集成

Camel K 的开发人员指南

法律通告

摘要

开发、配置和管理红帽构建的 Apache Camel K 应用程序的基本知识。

目录

前言

使开源包含更多

红帽致力于替换我们的代码、文档和 Web 属性中存在问题的语言。我们从这四个术语开始：master、slave、黑名单和白名单。由于此项工作十分艰巨，这些更改将在即将推出的几个发行版本中逐步实施。有关更多详情，请参阅我们的首席技术官 Chris Wright 提供的消息。

第 1 章 管理 CAMEL K 集成

您可以使用 Camel K 命令行或使用开发工具管理 Red Hat Integration - Camel K 集成。本章介绍了如何在命令行中管理 Camel K 集成，并提供指向其他资源的链接来说明如何使用 VS Code 开发工具。

- 第 1.1 节 “管理 Camel K 集成”
- 第 1.2 节 “管理 Camel K 集成日志记录级别”
- 第 1.3 节 “扩展 Camel K 集成”

1.1. 管理 CAMEL K 集成

Camel K 在命令行中提供不同的选项来管理 OpenShift 集群上的 Camel K 集成。本节展示了使用以下命令的简单示例：

- **kamel get**
- **kamel describe**
- **kamel log**
- **kamel delete**

先决条件

- 设置 Camel K 开发环境
- 您必须已有一个使用 Java 或 YAML DSL 编写的 Camel 集成

流程

1. 确保 Camel K Operator 在 OpenShift 集群上运行，例如：

```
oc get pod
```

```
NAME                              READY   STATUS    RESTARTS   AGE
camel-k-operator-86b8d94b4-pk7d6   1/1     Running   0          6m28s
```

2. 输入 **kamel run** 命令，以在 OpenShift 上的云中运行集成。例如：

```
kamel run hello.camelk.yaml
```

```
integration "hello" created
```

3. 输入 **kamel get** 命令来检查集成的状态：

```
kamel get
```

```
NAME    PHASE          KIT
hello   Building Kit kit-bqatqib5t4kse5vukt40
```

4. 输入 **kamel describe** 命令，以打印有关集成的详细信息：

```
kamel describe integration hello
```

```
Name:               hello
Namespace:          myproject
Creation Timestamp:  Fri, 13 Aug 2021 16:23:21 +0200
Phase:              Building Kit
Runtime Version:    1.7.1.fuse-800025-redhat-00001
Kit:                myproject/kit-c4ci6mbe9hl5ph5c9sjg
Image:
Version:            1.6.6
Dependencies:
  camel:core
  camel:log
  camel:timer
  mvn:org.apache.camel.k:camel-k-runtime
  mvn:org.apache.camel.quarkus:camel-quarkus-yaml-dsl
Sources:
  Name                    Language  Compression  Ref  Ref Key
  camel-k-embedded-flow.yaml  yaml      false
Conditions:
  Type                    Status  Reason                    Message
  IntegrationPlatformAvailable  True    IntegrationPlatformAvailable  myproject/camel-k
  IntegrationKitAvailable       True    IntegrationKitAvailable       kit-c4ci6mbe9hl5ph5c9sjg
  CronJobAvailable              False   CronJobNotAvailableReason     different controller
strategy used (deployment)
  DeploymentAvailable           True    DeploymentAvailable           deployment name is hello
  KnativeServiceAvailable       False   KnativeServiceNotAvailable    different controller
strategy used (deployment)
  Ready                         True    ReplicaSetReady
```

5. 输入 **kamel log** 命令将日志输出到 **stdout** ：

```
kamel log hello
```

```
...
[1] 2021-08-13 14:37:15,860 INFO [info] (Camel (camel-1) thread #0 - timer://yaml)
Exchange[ExchangePattern: InOnly, BodyType: String, Body: Hello Camel K from yaml]
...
```

6. 按 **Ctrl-C** 终止终端中的日志记录。

7. 输入 **kamel delete** 以删除 OpenShift 上部署的集成：

```
kamel delete hello
```

```
Integration hello deleted
```

其他资源

- 有关日志记录的详情，请参阅管理 Camel K 集成日志记录级别
- 有关更快的部署周转时间，请参阅 在开发模式下运行 Camel K 集成

- 有关管理集成的开发工具的详情，请参阅 红帽 Apache Camel K 的 VS Code 工具

1.2. 管理 CAMEL K 集成日志记录级别

Camel K 使用 Quarkus Logging 机制作为集成的日志框架。您可以在运行时在命令行中配置各种日志记录器的日志记录级别，方法是将 **quarkus.log.category** 前缀指定为集成属性。例如：

示例

```
--property 'quarkus.log.category."org".level'=DEBUG
```

注意

使用单引号转义属性非常重要。

先决条件

- 设置 Camel K 开发环境

流程

1. 输入 **kamel run** 命令，并使用 **--property** 选项指定日志级别。例如：

```
kamel run --dev --property 'quarkus.log.category."org.apache.camel.support".level'=DEBUG
Basic.java

...
integration "basic" created
    Progress: integration "basic" in phase Initialization
    Progress: integration "basic" in phase Building Kit
    Progress: integration "basic" in phase Deploying
    Condition "IntegrationPlatformAvailable" is "True" for Integration basic: myproject/camel-k
    Integration basic in phase "Initialization"
    Integration basic in phase "Building Kit"
    Integration basic in phase "Deploying"
    Condition "IntegrationKitAvailable" is "True" for Integration basic: kit-
c4dn5l62v9g3aopkocag
    Condition "DeploymentAvailable" is "True" for Integration basic: deployment name is basic
    Condition "CronJobAvailable" is "False" for Integration basic: different controller strategy
used (deployment)
    Progress: integration "basic" in phase Running
    Condition "KnativeServiceAvailable" is "False" for Integration basic: different controller
strategy used (deployment)
    Integration basic in phase "Running"
    Condition "Ready" is "False" for Integration basic
    Condition "Ready" is "True" for Integration basic
    [1] Monitoring pod basic-575b97f64b-7l5rl
    [1] 2021-08-17 08:35:22,906 DEBUG [org.apa.cam.sup.LRUCacheFactory] (main)
Creating DefaultLRUCacheFactory
    [1] 2021-08-17 08:35:23,132 INFO  [org.apa.cam.k.Runtime] (main) Apache Camel K
Runtime 1.7.1.fuse-800025-redhat-00001
    [1] 2021-08-17 08:35:23,134 INFO  [org.apa.cam.qua.cor.CamelBootstrapRecorder] (main)
bootstrap runtime: org.apache.camel.quarkus.main.CamelMainRuntime
    [1] 2021-08-17 08:35:23,224 INFO  [org.apa.cam.k.lis.SourcesConfigurer] (main) Loading
```

```
routes from: SourceDefinition{name='Basic', language='java',
location='file:/etc/camel/sources/Basic.java', }
    [1] 2021-08-17 08:35:23,232 DEBUG [org.apa.cam.sup.ResolverHelper] (main) Found
RoutesBuilderLoader: org.apache.camel.dsl.java.joor.JavaRoutesBuilderLoader via: META-
INF/services/org/apache/camel/java
    [1] 2021-08-17 08:35:23,232 DEBUG [org.apa.cam.sup.ResolverHelper] (main) Detected
and using RoutesBuilderLoader:
org.apache.camel.dsl.java.joor.JavaRoutesBuilderLoader@68dc098b
    [1] 2021-08-17 08:35:23,236 DEBUG [org.apa.cam.sup.ResolverHelper] (main) Found
ResourceResolver: org.apache.camel.impl.engine.DefaultResourceResolvers$FileResolver
via: META-INF/services/org/apache/camel/file
    [1] 2021-08-17 08:35:23,237 DEBUG [org.apa.cam.sup.ResolverHelper] (main) Detected
and using ResourceResolver:
org.apache.camel.impl.engine.DefaultResourceResolvers$FileResolver@5b67bb7e
    [1] 2021-08-17 08:35:24,320 DEBUG [org.apa.cam.sup.ResolverHelper] (main) Lookup
Language with name simple in registry. Found:
org.apache.camel.language.simple.SimpleLanguage@74d7184a
    [1] 2021-08-17 08:35:24,328 DEBUG [org.apa.cam.sup.EventHelper] (main) Ignoring
notifying event Initializing CamelContext: camel-1. The EventNotifier has not been started
yet: org.apache.camel.quarkus.core.CamelManagementEventBridge@3301500b
    [1] 2021-08-17 08:35:24,336 DEBUG [org.apa.cam.sup.ResolverHelper] (main) Lookup
Component with name timer in registry. Found:
org.apache.camel.component.timer.TimerComponent@3ef41c66
    [1] 2021-08-17 08:35:24,342 DEBUG [org.apa.cam.sup.DefaultComponent] (main)
Creating endpoint uri=[timer://java?period=1000], path=[java]
    [1] 2021-08-17 08:35:24,350 DEBUG [org.apa.cam.sup.ResolverHelper] (main) Found
ProcessorFactory: org.apache.camel.processor.DefaultProcessorFactory via: META-
INF/services/org/apache/camel/processor-factory
    [1] 2021-08-17 08:35:24,351 DEBUG [org.apa.cam.sup.ResolverHelper] (main) Detected
and using ProcessorFactory:
org.apache.camel.processor.DefaultProcessorFactory@704b2127
    [1] 2021-08-17 08:35:24,369 DEBUG [org.apa.cam.sup.ResolverHelper] (main) Found
InternalProcessorFactory: org.apache.camel.processor.DefaultInternalProcessorFactory via:
META-INF/services/org/apache/camel/internal-processor-factory
    [1] 2021-08-17 08:35:24,369 DEBUG [org.apa.cam.sup.ResolverHelper] (main) Detected
and using InternalProcessorFactory:
org.apache.camel.processor.DefaultInternalProcessorFactory@4f8caaf3
    [1] 2021-08-17 08:35:24,442 DEBUG [org.apa.cam.sup.ResolverHelper] (main) Lookup
Component with name log in registry. Found:
org.apache.camel.component.log.LogComponent@46b695ec
    [1] 2021-08-17 08:35:24,444 DEBUG [org.apa.cam.sup.DefaultComponent] (main)
Creating endpoint uri=[log://info], path=[info]
    [1] 2021-08-17 08:35:24,461 DEBUG [org.apa.cam.sup.EventHelper] (main) Ignoring
notifying event Initialized CamelContext: camel-1. The EventNotifier has not been started yet:
org.apache.camel.quarkus.core.CamelManagementEventBridge@3301500b
    [1] 2021-08-17 08:35:24,467 DEBUG [org.apa.cam.sup.DefaultProducer] (main) Starting
producer: Producer[log://info]
    [1] 2021-08-17 08:35:24,469 DEBUG [org.apa.cam.sup.DefaultConsumer] (main) Build
consumer: Consumer[timer://java?period=1000]
    [1] 2021-08-17 08:35:24,475 DEBUG [org.apa.cam.sup.DefaultConsumer] (main) Starting
consumer: Consumer[timer://java?period=1000]
    [1] 2021-08-17 08:35:24,481 INFO  [org.apa.cam.imp.eng.AbstractCamelContext] (main)
Routes startup summary (total:1 started:1)
    [1] 2021-08-17 08:35:24,481 INFO  [org.apa.cam.imp.eng.AbstractCamelContext] (main)
Started java (timer://java)
    [1] 2021-08-17 08:35:24,482 INFO  [org.apa.cam.imp.eng.AbstractCamelContext] (main)
```

```
Apache Camel 3.10.0.fuse-800010-redhat-00001 (camel-1) started in 170ms (build:0ms
init:150ms start:20ms)
    [1] 2021-08-17 08:35:24,487 INFO  [io.quarkus] (main) camel-k-integration 1.6.6 on JVM
(powered by Quarkus 1.11.7.Final-redhat-00009) started in 2.192s.
    [1] 2021-08-17 08:35:24,488 INFO  [io.quarkus] (main) Profile prod activated.
    [1] 2021-08-17 08:35:24,488 INFO  [io.quarkus] (main) Installed features: [camel-bean,
camel-core, camel-java-joor-dsl, camel-k-core, camel-k-runtime, camel-log, camel-support-
common, camel-timer, cdi]
    [1] 2021-08-17 08:35:25,493 INFO  [info] (Camel (camel-1) thread #0 - timer://java)
Exchange[ExchangePattern: InOnly, BodyType: String, Body: Hello Camel K from java]
    [1] 2021-08-17 08:35:26,479 INFO  [info] (Camel (camel-1) thread #0 - timer://java)
Exchange[ExchangePattern: InOnly, BodyType: String, Body: Hello Camel K from java]
...
```

2. 按 **Ctrl-C** 终止终端中的日志记录。

其他资源

- 有关日志记录框架的详情，请参阅 配置日志记录格式
- 有关查看日志记录的开发工具的详情，请参阅 红帽 Apache Camel K 的 VS Code 工具

1.3. 扩展 CAMEL K 集成

您可以使用 **oc scale** 命令扩展集成。

流程

- 要扩展 Camel K 集成，请运行以下命令：

```
oc scale it <integration_name> --replicas <number_of_replicas>
```

- 您还可以直接编辑 Integration 资源来扩展集成。

```
oc patch it <integration_name> --type merge -p '{"spec":{"replicas":<number_of_replicas>}}'
```

要查看集成的副本数量，请使用以下命令：

```
oc get it <integration_name> -o jsonpath='{.status.replicas}'
```

第 2 章 监控 CAMEL K 集成

Red Hat Integration - Camel K 监控基于 OpenShift 监控系统。本章解释了如何使用可用选项来监控 Red Hat Integration - Camel K 集成。您可以使用已部署为 OpenShift Monitoring 一部分的 Prometheus Operator 来监控您自己的应用程序。

- 第 2.1 节 “在 OpenShift 中启用用户工作负载监控”
- 第 2.2 节 “配置 Camel K 集成指标”
- 第 2.3 节 “添加自定义 Camel K 集成指标”

2.1. 在 OPENSHIFT 中启用用户工作负载监控

OpenShift 4.3 或更高版本包括已作为 OpenShift Monitoring 的一部分部署的嵌入式 Prometheus Operator。本节介绍如何在 OpenShift Monitoring 中启用对您自己的应用程序服务的监控。这个选项避免了安装和管理独立 Prometheus 实例的额外开销。

先决条件

- 您必须具有安装 Camel K Operator 的 OpenShift 集群的集群管理员访问权限。请参阅安装 Camel K。

流程

1. 输入以下命令检查 **openshift-monitoring** 项目中是否存在 **cluster-monitoring-config** ConfigMap 对象：

```
$ oc -n openshift-monitoring get configmap cluster-monitoring-config
```

2. 如果不存在，请创建 **cluster-monitoring-config** ConfigMap：

```
$ oc -n openshift-monitoring create configmap cluster-monitoring-config
```

3. 编辑 **cluster-monitoring-config** ConfigMap：

```
$ oc -n openshift-monitoring edit configmap cluster-monitoring-config
```

4. 在 **data:config.yaml:** 下，将 **enableUserWorkload** 设置为 **true** ：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: cluster-monitoring-config
  namespace: openshift-monitoring
data:
  config.yaml: |
    enableUserWorkload: true
```

其他资源

- 为用户定义的项目启用监控

2.2. 配置 CAMEL K 集成指标

您可以在运行时使用 Camel K Prometheus trait 配置 Camel K 集成监控。这会自动化配置依赖项和集成 Pod，以公开指标端点，然后被 Prometheus 发现并显示。Camel Quarkus MicroProfile Metrics 扩展 自动以 OpenMetrics 格式收集并公开默认的 Camel K 指标。

先决条件

- 您必须已在 OpenShift 中启用对您自己的服务的监控。请参阅在 OpenShift 中启用用户工作负载监控。

流程

1. 输入以下命令运行启用了 Prometheus 特征的 Camel K 集成：

```
kamel run myIntegration.java -t prometheus.enabled=true
```

另外，您可以通过更新集成平台来全局启用 Prometheus trait，如下所示：

```
$ oc patch ip camel-k --type=merge -p '{"spec":{"traits":{"prometheus":{"configuration":{"enabled":true}}}}}'
```

2. 查看 Prometheus 中 Camel K 集成指标的监控。例如，对于嵌入式 Prometheus，在 OpenShift 管理员或开发人员 Web 控制台中选择 **Monitoring** > **Metrics**。
3. 输入您要查看的 Camel K 指标。例如，在 **Administrator** 控制台中，在 **Insert Metric at Cursor** 下，输入 **application_camel_context_uptime_seconds**，然后单击 **Run Queries**。
4. 点 **Add Query** 查看额外的指标。

PROMETHEUS TRAIT 提供的默认 Camel 指标

一些特定于 Camel 的指标会开箱即用。

Name	类型	描述
application_camel_message_history_processing	timer	启用消息历史记录时，路由中每个节点的性能示例
application_camel_route_count	gauge	添加的路由数
application_camel_route_running_count	gauge	运行nning 的路由数
application_camel_[route 或 context]_exchanges_inflight_count	gauge	CamelContext 或路由的路由
application_camel_[route 或 context]_exchanges_total	计数	CamelContext 或路由的已处理交换总数

Name	类型	描述
application_camel_[route 或 context]_exchanges_completed_total	计数	CamelContext 或路由成功完成的交换数量
application_camel_[route 或 context]_exchanges_failed_total	计数	CamelContext 或路由的失败交换数
application_camel_[route 或 context]_failuresHandled_total	计数	CamelContext 或路由处理的失败数
application_camel_[route 或 context]_externalRedeliveries_total	计数	CamelContext 或路由的外部启动 redeliveries （如来自 JMS 代理）的数量
application_camel_context_status	gauge	Camel 上下文的状态
application_camel_context_uptime_seconds	gauge	Camel 上下文启动后的时间长度
application_camel_[route 或 exchange]*processing*[rate_per_second 或 one_min_rate_second 或 five_min_rate_per_second 或 fifteen_min_rate_per_second 或 min_seconds 或 max_seconds 或 mean_seconds 或 stddev_seconds]	gauge	使用多个选项交换消息或路由处理
application_camel_[route 或 exchange]_processing_seconds	summary	交换消息或路由处理指标

其他资源

- Prometheus Trait
- Camel Quarkus MicroProfile 指标

2.3. 添加自定义 CAMEL K 集成指标

您可以通过在 Java 代码中使用 Camel MicroProfile Metrics 组件和注解，将自定义指标添加到 Camel K 集成。然后，Prometheus 会自动发现并显示这些自定义指标。

本节介绍在 Camel K 集成和服务实施代码中添加 Camel MicroProfile Metrics 注释的示例。

先决条件

- 您必须已在 OpenShift 中启用对您自己的服务的监控。请参阅在 OpenShift 中启用用户工作负载监控。

流程

1. 使用 Camel MicroProfile Metrics 组件注解在 Camel 集成代码中注册自定义指标。以下示例显示了 **Metrics.java** 集成：

```
// camel-k: language=java trait=prometheus.enabled=true dependency=mvn:org.my/app:1.0 (1)

import org.apache.camel.Exchange;
import org.apache.camel.LoggingLevel;
import org.apache.camel.builder.RouteBuilder;
import org.apache.camel.component.microprofile.metrics.MicroProfileMetricsConstants;

import javax.enterprise.context.ApplicationScoped;

@ApplicationScoped
public class Metrics extends RouteBuilder {

  @Override
  public void configure() {
      onException()
          .handled(true)
          .maximumRedeliveries(2)
          .logStackTrace(false)
          .logExhausted(false)
          .log(LoggingLevel.ERROR, "Failed processing ${body}")
          // Register the 'redelivery' meter
          .to("microprofile-metrics:meter:redelivery?mark=2")
          // Register the 'error' meter
          .to("microprofile-metrics:meter:error"); (2)

      from("timer:stream?period=1000")
          .routeId("unreliable-service")
          .setBody(header(Exchange.TIMER_COUNTER).prepend("event #"))
          .log("Processing ${body}...")
          // Register the 'generated' meter
          .to("microprofile-metrics:meter:generated") (3)
          // Register the 'attempt' meter via @Metered in Service.java
          .bean("service") (4)
          .filter(header(Exchange.REDELIVERED))
              .log(LoggingLevel.WARN, "Processed ${body} after
${header.CamelRedeliveryCounter} retries")
              .setHeader(MicroProfileMetricsConstants.HEADER_METER_MARK,
header(Exchange.REDELIVERY_COUNTER))
              // Register the 'redelivery' meter
              .to("microprofile-metrics:meter:redelivery") (5)
          .end()
          .log("Successfully processed ${body}")
          // Register the 'success' meter
          .to("microprofile-metrics:meter:success"); (6)
  }
}
```

1. 使用 Camel K modeline 来自动配置 Prometheus trait 和 Maven 依赖项
2. 错误 ：指标与尚未处理的事件数对应的错误数
3. **generated** ：指标要处理的事件数量
4. **attempt**：指标为服务 bean 发出的调用数，以处理传入的事件
5. **redelivery** ：指标为处理事件的重试次数
6. **success**：指标成功处理的事件数

2. 根据需要，将 Camel MicroProfile 指标注释添加到任何实施文件中。以下示例显示了 Camel K 集成调用的服务 bean，该服务会生成随机故障：

```
package com.redhat.integration;

import java.util.Random;

import org.apache.camel.Exchange;
import org.apache.camel.RuntimeExchangeException;

import org.eclipse.microprofile.metrics.Meter;
import org.eclipse.microprofile.metrics.annotation.Metered;
import org.eclipse.microprofile.metrics.annotation.Metric;

import javax.inject.Named;
import javax.enterprise.context.ApplicationScoped;

@Named("service")
@ApplicationScoped
@io.quarkus.arc.Unremovable

public class Service {

  //Register the attempt meter
  @Metered(absolute = true)
  public void attempt(Exchange exchange) { 1
    Random rand = new Random();
      if (rand.nextDouble() < 0.5) {
        throw new RuntimeExchangeException("Random failure", exchange); 2
    }
  }
}
```

1 @Metered MicroProfile Metrics 注释声明量表，名称会根据指标方法名称自动生成，本例中为 attempt。

2

3. 按照 配置 Camel K 集成指标 中的步骤运行集成，并在 Prometheus 中查看自定义 Camel K 指标。

在这种情况下，示例已在 Metrics.java 中使用 Camel K 模式行来自动配置 Prometheus 和 Service.java 所需的 Maven 依赖项。

其他资源

- Camel MicroProfile Metrics 组件
- Camel Quarkus MicroProfile Metrics Extension

第 3 章 监控 CAMEL K OPERATOR

Red Hat Integration - Camel K 监控基于 OpenShift 监控系统。本章论述了如何使用可用选项来监控 Red Hat Integration - Camel K operator 在运行时。您可以使用已部署为 OpenShift Monitoring 一部分的 Prometheus Operator 来监控您自己的应用程序。

- 第 3.1 节 “Camel K Operator 指标”
- 第 3.2 节 “启用 Camel K Operator 监控”
- 第 3.3 节 “Camel K operator 警报”

3.1. CAMEL K OPERATOR 指标

Camel K operator 监控端点公开以下指标：

表 3.1. Camel K operator 指标

名称	类型	描述	Buckets	标签
camel_k_reconciliation_duration_seconds	HistogramVec	协调请求持续时间	0.25s, 0.5s, 1s, 5s	命名空间,group,version,kind,result:Reconciled|Errored|Requeued,tag:""|PlatformError|UserError
camel_k_build_duration_seconds	HistogramVec	构建持续时间	30s, 1m, 1.5m, 2m, 5m, 10m	结果：Succeeded|Error
camel_k_build_recovery_attempts	Histogram	构建恢复尝试	0, 1, 2, 3, 4, 5	结果：Succeeded|Error
camel_k_build_queue_duration_seconds	Histogram	构建队列持续时间	5s, 15s, 30s, 1m, 5m,	N/A
camel_k_integration_first_readiness_seconds	Histogram	首次集成就绪状态	5s, 10s, 30s, 1m, 2m	N/A

名称	类型	描述	Buckets	标签

3.2. 启用 CAMEL K OPERATOR 监控

OpenShift 4.3 或更高版本包括已作为 OpenShift Monitoring 的一部分部署的嵌入式 Prometheus Operator。本节介绍如何在 OpenShift Monitoring 中启用对您自己的应用程序服务的监控。

先决条件

- 您必须具有安装 Camel K Operator 的 OpenShift 集群的集群管理员访问权限。请参阅安装 Camel K。
- 您必须已在 OpenShift 中启用对您自己的服务的监控。请参阅在 OpenShift 中启用用户工作负载监控。

流程

1. 创建一个以 operator 指标端点为目标的 PodMonitor 资源，以便 Prometheus 服务器可以提取 Operator 公开的指标。

   operator-pod-monitor.yaml

```
apiVersion: monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: camel-k-operator
  labels:
    app: "camel-k"
    camel.apache.org/component: operator
spec:
  selector:
    matchLabels:
      app: "camel-k"
      camel.apache.org/component: operator
  podMetricsEndpoints:
    - port: metrics
```

2. 创建 PodMonitor 资源。

```
oc apply -f operator-pod-monitor.yaml
```

其它资源

- 如需有关发现机制和 Operator 资源之间的关系的更多信息，请参阅 Prometheus Operator 入门指南。
- 如果没有发现 Operator 指标，您可以在 Troubleshooting ServiceMonitor 更改 中找到更多信息，这也适用于 PodMonitor 资源故障排除。

3.3. CAMEL K OPERATOR 警报

您可以创建一个 PrometheusRule 资源，以便 OpenShift 监控堆栈中的 AlertManager 实例可以根据 Camel K operator 公开的指标触发警报。

示例

您可以根据公开的指标使用警报规则创建 PrometheusRule 资源，如下所示。

```
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
 name: camel-k-operator
spec:
 groups:
  - name: camel-k-operator
    rules:
     - alert: CamelKReconciliationDuration
       expr: |
        (
        1 - sum(rate(camel_k_reconciliation_duration_seconds_bucket{le="0.5"}[5m])) by (job)
        /
        sum(rate(camel_k_reconciliation_duration_seconds_count[5m])) by (job)
        )
        * 100
        > 10
       for: 1m
       labels:
        severity: warning
```

```
      annotations:
        message: |
          {{ printf "%0.0f" $value }}% of the reconciliation requests
          for {{ $labels.job }} have their duration above 0.5s.
    - alert: CamelKReconciliationFailure
      expr: |
        sum(rate(camel_k_reconciliation_duration_seconds_count{result="Errored"}[5m])) by
(job)
        /
        sum(rate(camel_k_reconciliation_duration_seconds_count[5m])) by (job)
        * 100
        > 1
      for: 10m
      labels:
        severity: warning
      annotations:
        message: |
          {{ printf "%0.0f" $value }}% of the reconciliation requests
          for {{ $labels.job }} have failed.
    - alert: CamelKSuccessBuildDuration2m
      expr: |
        (
        1 - sum(rate(camel_k_build_duration_seconds_bucket{le="120",result="Succeeded"}
[5m])) by (job)
        /
        sum(rate(camel_k_build_duration_seconds_count{result="Succeeded"}[5m])) by (job)
        )
        * 100
        > 10
      for: 1m
      labels:
        severity: warning
      annotations:
        message: |
          {{ printf "%0.0f" $value }}% of the successful builds
          for {{ $labels.job }} have their duration above 2m.
    - alert: CamelKSuccessBuildDuration5m
      expr: |
        (
        1 - sum(rate(camel_k_build_duration_seconds_bucket{le="300",result="Succeeded"}
[5m])) by (job)
        /
        sum(rate(camel_k_build_duration_seconds_count{result="Succeeded"}[5m])) by (job)
        )
        * 100
        > 1
      for: 1m
      labels:
        severity: critical
      annotations:
        message: |
          {{ printf "%0.0f" $value }}% of the successful builds
          for {{ $labels.job }} have their duration above 5m.
    - alert: CamelKBuildFailure
      expr: |
        sum(rate(camel_k_build_duration_seconds_count{result="Failed"}[5m])) by (job)
```

```
    /
    sum(rate(camel_k_build_duration_seconds_count[5m])) by (job)
    * 100
    > 1
  for: 10m
  labels:
    severity: warning
  annotations:
    message: |
      {{ printf "%0.0f" $value }}% of the builds for {{ $labels.job }} have failed.
- alert: CamelKBuildError
  expr: |
    sum(rate(camel_k_build_duration_seconds_count{result="Error"}[5m])) by (job)
    /
    sum(rate(camel_k_build_duration_seconds_count[5m])) by (job)
    * 100
    > 1
  for: 10m
  labels:
    severity: critical
  annotations:
    message: |
      {{ printf "%0.0f" $value }}% of the builds for {{ $labels.job }} have errored.
- alert: CamelKBuildQueueDuration1m
  expr: |
    (
    1 - sum(rate(camel_k_build_queue_duration_seconds_bucket{le="60"}[5m])) by (job)
    /
    sum(rate(camel_k_build_queue_duration_seconds_count[5m])) by (job)
    )
    * 100
    > 1
  for: 1m
  labels:
    severity: warning
  annotations:
    message: |
      {{ printf "%0.0f" $value }}% of the builds for {{ $labels.job }}
      have been queued for more than 1m.
- alert: CamelKBuildQueueDuration5m
  expr: |
    (
    1 - sum(rate(camel_k_build_queue_duration_seconds_bucket{le="300"}[5m])) by (job)
    /
    sum(rate(camel_k_build_queue_duration_seconds_count[5m])) by (job)
    )
    * 100
    > 1
  for: 1m
  labels:
    severity: critical
  annotations:
    message: |
      {{ printf "%0.0f" $value }}% of the builds for {{ $labels.job }}
      have been queued for more than 5m.
```

Camel K operator 警报

下表显示了 PrometheusRule 资源中定义的警报规则。

名称	重要性	描述
CamelKReconciliationDuration	warning	协调请求超过 10% 的持续时间至少为 1 分钟。
CamelKReconciliationFailure	warning	协调请求超过 1% 的失败时间至少超过 10 分钟。
CamelKSuccessBuildDuration2m	warning	成功构建的时间超过 10% 的 2 分钟超过至少 1 分钟。
CamelKSuccessBuildDuration5m	critical	成功构建中有超过 1% 的持续时间长于 5 分钟超过最少 1 分钟。
CamelKBuildError	critical	超过 1% 的构建错误至少为 10 分钟。
CamelKBuildQueueDuration1m	warning	超过 1% 的构建已排队超过 1 分钟超过至少 1 分钟以上。
CamelKBuildQueueDuration5m	critical	超过 1% 的构建已排队，至少 5 分钟以上超过 1 分钟。

您可以在 OpenShift 文档 创建警报规则 中找到有关警报的更多信息。

第 4 章 配置 CAMEL K 集成

Camel K 集成生命周期有两个配置阶段：

- 构建时间 - 当 Camel Quarkus 构建 Camel K 集成时，它会消耗 build-time 属性。
- Runtime - 当 Camel K 集成运行时，集成将使用本地文件、OpenShift ConfigMap 或 Secret 中的运行时属性或配置信息。

您可以在 kamel run 命令中使用以下选项来提供配置信息：

- 对于 build-time 配置，请使用 --build-property 选项，如 指定 build-time 配置属性中所述
- 对于运行时配置，请使用 --property、--config 或 --resource 选项，如 指定运行时配置选项中所述

例如，您可以使用 build-time 和 runtime 选项在 Camel K 中快速配置数据源，如链接： Connect Camel K with database sample configuration 所示。

- 第 4.1 节 “指定构建时配置属性”
- 第 4.2 节 “指定运行时配置选项”
- 第 4.3 节 “配置 Camel 集成组件”
- 第 4.4 节 “配置 Camel K 集成依赖项”

4.1. 指定构建时配置属性

您可能需要为 Camel Quarkus 运行时提供属性值，以便它可以构建 Camel K 集成。有关构建期间生效的 Quarkus 配置的更多信息，请参阅 Quarkus 构建时间配置文档。您可以在命令行中直接指定 build-

time 属性，或引用属性文件。如果两个位置上都定义了属性，则直接在命令行中指定的值优先于属性文件中的值。

先决条件

- 您必须有权访问安装了 Camel K Operator 和 OpenShift Serverless Operator 的 OpenShift 集群：
- 安装 Camel K
- 从 OperatorHub 安装 OpenShift Serverless
- 您知道您要应用到 Camel K 集成的 Camel Quarkus 配置选项。

流程

- 使用 Camel K kamel run 命令指定 --build-property 选项：

```
kamel run --build-property <quarkus-property>=<property-value> <camel-k-integration>
```

例如，以下 Camel K 集成（名为 my-simple-timer.yaml）使用 quarkus.application.name 配置选项：

```
- from:
    uri: "timer:tick"
    steps:
      - set-body:
          constant: "{{quarkus.application.name}}"
      - to: "log:info"
```

要覆盖默认应用程序名称，请在运行集成时为 quarkus.application.name 属性指定一个值。

例如，将名称从 my-simple-timer 改为 my-favorite-app ：

```
kamel run --build-property quarkus.application.name=my-favorite-app my-simple-timer.yaml
```

- 要提供多个 build-time 属性，请在 kamel run 命令中添加额外的 --build-property 选项：

```
kamel run --build-property <quarkus-property1>=<property-value1> -build-property=
<quarkus-property2>=<property-value12> <camel-k-integration>
```

另外，如果您需要指定多个属性，您可以创建一个属性文件，并使用 --build-property 文件选项指定属性文件：

```
kamel run --build-property file:<property-filename> <camel-k-integration>
```

例如，以下属性文件（名为 quarkus.properties）定义两个 Quarkus 属性：

```
quarkus.application.name = my-favorite-app
quarkus.banner.enabled = true
```

quarkus.banner.enabled 属性指定在集成启动时显示 Quarkus 横幅。

使用 Camel Kamel run 命令指定 quarkus.properties 文件：

```
kamel run --build-property file:quarkus.properties my-simple-timer.yaml
```

Quarkus 解析属性文件，并使用属性值来配置 Camel K 集成。

其他资源

有关 Camel Quarkus 作为 Camel K 集成运行时的详情，请参考 Quarkus Trait。

4.2. 指定运行时配置选项

您可以为 Camel K 集成指定以下运行时配置信息，以便在它运行时使用：

- 您在命令行或 .properties 文件中提供的运行时属性。
- 集成启动时您希望 Camel K 操作器处理并作为运行时属性解析的配置值。您可以在本地文本文件、OpenShift ConfigMap 或 OpenShift secret 中提供配置值。

- 集成启动时没有作为属性文件解析的资源信息。您可以在本地文本文件、二进制文件、OpenShift ConfigMap 或 OpenShift secret 中提供资源信息。

使用以下 kamel run 选项：

- --property

  使用 --property 选项直接在命令行中指定运行时属性，或者引用 Java lf properties 文件。Camel K operator 将属性文件的内容附加到正在运行的集成的 user.properties 文件中。

- --config

  使用 --config 选项提供您希望 Camel K 操作器在集成启动时处理并作为运行时属性的配置值。

  您可以提供一个本地文本文件(1 MiB 最大文件大小)、ConfigMap (3MB)或 Secret (3MB)。该文件必须是 UTF-8 资源。材料化文件（在您从您提供的文件集成启动时生成的）在类路径级别上提供，以便您可在集成代码中引用它，而无需提供精确的位置。

  注： 如果您需要提供非 UTF-8 资源（例如，二进制文件），请使用 --resource 选项。

- --resource

  使用 --resource 选项为集成提供在运行时访问的资源。您可以提供本地文本或二进制文件(1 MiB 最大文件大小)、ConfigMap （最大 3MB）或 Secret （最大 3MB）。另外，您可以指定为资源分类的文件目的地。例如，如果要设置 HTTPS 连接，请使用 --resource 选项提供指定位置中预期的 SSL 证书（二进制文件）。

  Camel K 操作器不会解析属性的资源，且不会将资源添加到 classpath 中。（如果您想要将资源添加到 classpath 中，您可以在集成中使用 JVM 特征 ）。

4.2.1. 提供运行时属性

您可以在命令行中直接指定运行时属性，或使用 kamel run 命令的 --property 选项直接指定 Java lf properties 文件。

当您运行与 --property 选项集成时，Camel K operator 会将属性附加到正在运行的集成的 user.properties 文件中。

4.2.1.1. 在命令行中提供运行时属性

您可以在运行时在命令行中配置 Camel K 集成的属性。当您使用属性占位符在集成中定义属性时，例如 {{my.message}}，您可以在命令行中指定属性值，例如 --property my.message=Hello。您可以在单个命令中指定多个属性。

先决条件

- 设置 Camel K 开发环境

流程

1. 开发使用属性的 Camel 集成。以下简单示例包含 {{my.message}} 属性占位符：

```
...
- from:
    uri: "timer:tick"
    steps:
      - set-body:
          constant: "{{my.message}}"
      - to: "log:info"
...
```

2. 使用以下语法运行集成，以在运行时设置属性值。

```
kamel run --property <property>=<value> <integration>
```

另外，您可以使用 --p 简写表示法（代替 --property）：

```
kamel run --property <property>=<value> <integration>
```

例如：

```
kamel run --property my.message="Hola Mundo" HelloCamelK.java --dev
```

或者

```
kamel run --p my.message="Hola Mundo" HelloCamelK.java --dev
```

以下是示例结果：

```
...
[1] 2020-04-13 15:39:59.213 INFO  [main] ApplicationRuntime - Listener
org.apache.camel.k.listener.RoutesDumper@6e0dec4a executed in phase Started
[1] 2020-04-13 15:40:00.237 INFO  [Camel (camel-k) thread #1 - timer://java] info -
Exchange[ExchangePattern: InOnly, BodyType: String, Body: Hola Mundo from java]
...
```

另请参阅

- 在属性文件中提供运行时属性

4.2.1.2. 在属性文件中提供运行时属性

您可以通过在运行时在命令行中指定属性文件 (*.properties) 来为 Camel K 集成配置多个属性。当您使用属性占位符在集成中定义属性时，例如 {{my.items}}，您可以使用属性文件在命令行中指定属性值，如 --p 文件 my-integration.properties。

前提条件

- 设置 Camel K 开发环境

流程

1. 创建集成属性文件。以下示例来自名为 **my.properties** 的文件：

```
my.key.1=hello
my.key.2=world
```

2. 开发使用属性文件中定义的属性的 Camel 集成。以下示例 **Routing.java** 集成使用 **{{my.key.1}} 和 {{my.key.2=world}} 属性占位符：**

```
import org.apache.camel.builder.RouteBuilder;

public class Routing extends RouteBuilder {
```

```
  @Override
  public void configure() throws Exception {

    from("timer:property-file")
      .routeId("property-file")
      .log("property file content is: {{my.key.1}} {{my.key.2}}");

  }
}
```

3. 使用以下语法运行集成来引用属性文件：

```
kamel run --property file:<my-file.properties> <integration>
```

另外，您可以使用 **--p** 简写表示法（代替 **--property**）：

```
kamel run --p file:<my-file.properties> <integration>
```

例如：

```
kamel run Routing.java --property:file=my.properties --dev
```

其他资源

- 部署基本 Camel K Java 集成
- 在命令行中提供运行时属性

4.2.2. 提供配置值

您可以使用 **kamel run** 命令的 **--config** 选项提供您希望 Camel K 操作器处理并解析为运行时属性的配置值。您可以在本地文本(UTF-8)文件、OpenShift ConfigMap 或 OpenShift secret 中提供配置值。

运行集成时，Camel K operator 会对提供的文件进行分类，并将其添加到 classpath 中，以便您可以引用集成代码中的配置值，而无需提供准确的位置。

4.2.2.1. 指定文本文件

如果您有包含配置值的 UTF-8 文本文件，您可以使用 **--config file:/path/to/file** 选项使文件在运行的

集成类路径上可用（具有相同文件名）。

先决条件

- 设置 Camel K 开发环境
- 您有一个或者多个（非二进制）文本文件，其中包含配置值。

  例如，创建一个名为 resources-data.txt 的文件，其中包含以下行：

```
the file body
```

流程

1. 创建一个 Camel K 集成来引用包含配置值的文本文件。

   例如，以下集成(ConfigFileRoute.java)要求 resources-data.txt 文件在运行时在类路径上可用：

```
import org.apache.camel.builder.RouteBuilder;

public class ConfigFileRoute extends RouteBuilder {
  @Override
  public void configure() throws Exception {

    from("timer:config-file")
        .setBody()
            .simple("resource:classpath:resources-data.txt")
        .log("resource file content is: ${body}");

  }
}
```

2. 运行集成并使用 --config 选项指定文本文件，使其可用于正在运行的集成。例如：

```
kamel run --config file:resources-data.txt ConfigFileRoute.java --dev
```

   另外，您可以通过重复添加 --config 选项来提供多个文件，例如：

```
kamel run --config file:resources-data1.txt --config file:resources-data2.txt
ConfigFileRoute.java --dev
```

4.2.2.2. 指定 ConfigMap

如果您有包含配置值的 OpenShift ConfigMap，且您需要对 ConfigMap 进行组织，以便 Camel K 集成可用，请使用 **--config configmap:<configmap-name>** 语法。

先决条件

- 设置 Camel K 开发环境
- 您已在 OpenShift 集群中存储了一个或多个 ConfigMap 文件。

  例如，您可以使用以下命令创建 ConfigMap：

```
oc create configmap my-cm --from-literal=my-configmap-key="configmap content"
```

流程

1. 创建引用 ConfigMap 的 Camel K 集成。

   例如，以下集成（名为 ConfigConfigmapRoute.java）引用了名为 my-cm 的 ConfigMap 中的名为 my-configmap-key 的配置值。

```
import org.apache.camel.builder.RouteBuilder;

public class ConfigConfigmapRoute extends RouteBuilder {
  @Override
  public void configure() throws Exception {

    from("timer:configmap")
        .setBody()
            .simple("resource:classpath:my-configmap-key")
        .log("configmap content is: ${body}");

  }
}
```

2. 运行集成并使用 **--config** 选项对 ConfigMap 文件进行定位，使其可用于正在运行的集成。例如：

```
kamel run --config configmap:my-cm ConfigConfigmapRoute.java --dev
```

集成启动时，Camel K operator 会使用 ConfigMap 的内容挂载 OpenShift 卷。

注：如果您指定了集群中还没有可用的 ConfigMap，则 Integration 只会在 ConfigMap 可用后等待并启动。

4.2.2.3. 指定 Secret

您可以使用 OpenShift Secret 来安全地包含配置信息。要对 secret 进行资料化，使其可用于 Camel K 集成，您可以使用 --config secret 语法。

先决条件

- 设置 Camel K 开发环境
- 您已在 OpenShift 集群中存储了一个或多个 Secret。

  例如，您可以使用以下命令创建 Secret：

```
oc create secret generic my-sec --from-literal=my-secret-key="very top secret"
```

流程

1. 创建引用 ConfigMap 的 Camel K 集成。

   例如，以下集成（名为 ConfigSecretRoute.java）引用了位于名为 my-sec 的 Secret 中的 my-secret 属性：

```
import org.apache.camel.builder.RouteBuilder;

public class ConfigSecretRoute extends RouteBuilder {
  @Override
  public void configure() throws Exception {

    from("timer:secret")
        .setBody()
            .simple("resource:classpath:my-secret")
        .log("secret content is: ${body}");

  }
}
```

2. 运行集成并使用 **--config** 选项来定位 Secret，使其可用于正在运行的集成。例如：

```
kamel run --config secret:my-sec ConfigSecretRoute.java --dev
```

集成启动时，Camel K operator 会使用 Secret 的内容挂载 OpenShift 卷。

4.2.2.4. 引用 ConfigMap 或 Secret 中包含的属性

当您运行集成并使用 **--config** 选项指定 ConfigMap 或 Secret 时，Camel K operator 会解析 ConfigMap 或 Secret 作为运行时属性文件。在集成中，您可以在引用任何其他运行时属性时引用属性。

前提条件

- 设置 Camel K 开发环境

流程

1. 创建包含属性的文本文件。

   例如，创建一个名为 **my.properties** 的文件，其中包含以下属性：

```
my.key.1=hello
my.key.2=world
```

2. 根据属性文件创建 ConfigMap 或 Secret。

   例如，使用以下命令从 **my.properties** 文件创建 secret：

```
oc create secret generic my-sec --from-file my.properties
```

3. 在集成中，请参阅 Secret 中定义的属性。

   例如，以下集成（名为 **ConfigSecretPropertyRoute.java**）引用了 **my.key.1** 和 **my.key.2** 属性：

```
import org.apache.camel.builder.RouteBuilder;
```

```
public class ConfigSecretPropertyRoute extends RouteBuilder {
  @Override
  public void configure() throws Exception {

    from("timer:secret")
        .routeId("secret")
        .log("{{my.key.1}} {{my.key.2}}");

  }
}
```

4. 运行集成并使用 **--config** 选项指定包含 **my.key.1** 和 **my.key.2** 属性的 Secret。

   例如：

```
kamel run --config secret:my-sec ConfigSecretPropertyRoute.java --dev
```

4.2.2.5. 过滤从 ConfigMap 或 Secret 获取的配置值

ConfigMap 和 Secret 可以保存多个源。例如，以下命令从两个源创建一个 secret (my-sec-multi)：

```
oc create secret generic my-sec-multi --from-literal=my-secret-key="very top secret" --from-literal=my-secret-key-2="even more secret"
```

您可以在 **--config configmap** 或 **--config secret** 选项中使用 **/key** 标记来限制集成检索的信息数量。

先决条件

- 设置 Camel K 开发环境
- 您有一个 ConfigMap 或包含多个源的 Secret。

流程

1. 创建一个集成，它只使用来自 ConfigMap 或 Secret 中的一个源的配置值。

   例如，以下集成(ConfigSecretKeyRoute.java)仅使用来自 **my-sec-multi** secret 中的一个源的属性。

```
import org.apache.camel.builder.RouteBuilder;

public class ConfigSecretKeyRoute extends RouteBuilder {
  @Override
  public void configure() throws Exception {

    from("timer:secret")
      .setBody()
        .simple("resource:classpath:my-secret-key-2")
      .log("secret content is: ${body}");
  }
}
```

2. 使用 **--config secret** 选项和 **/key** 表示法运行集成。

   例如：

```
kamel run --config secret:my-sec-multi/my-secret-key-2 ConfigSecretKeyRoute.java --dev
```

3. 检查集成 pod，以验证是否挂载了指定的源（如 **my-secret-key-2**）。

   例如，运行以下命令列出 pod 的所有卷：

```
oc set volume pod/<pod-name> --all
```

4.2.3. 为正在运行的集成提供资源

您可以通过指定 **kamel run** 命令的 **--resource** 选项，为集成提供资源。您可以指定一个本地文本文件 (1 MiB 最大文件大小)、ConfigMap (3MB)或 Secret (3MB)。您可以选择指定资源分类的文件目的地。例如，如果要设置 HTTPS 连接，您可以使用 **--resource** 选项，因为您必须提供一个 SSL 证书，它是已知位置中预期的二进制文件。

当您使用 **--resource** 选项时，Camel K operator 不会解析查找运行时属性的资源，且不会将资源添加到 classpath 中。（如果您想要将资源添加到 classpath 中，您可以使用 JVM 特征。

4.2.3.1. 将文本或二进制文件指定为资源

如果您有包含配置值的文本或二进制文件，您可以使用 **--resource file:/path/to/file** 选项来定位该文件。默认情况下，Camel K 操作器将材料化文件复制到 **/etc/camel/resources/** 目录中。另外，您可以指定不同的目标目录，如 为资源指定目标路径 中所述。

先决条件

- 设置 Camel K 开发环境
- 您有一个或者多个包含配置属性的文本或二进制文件。

流程

1. 创建一个 Camel K 集成来读取您提供的文件内容。

   例如，以下集成(ResourceFileBinaryRoute.java)解压缩并读取 resources-data.zip 文件：

```
import org.apache.camel.builder.RouteBuilder;

public class ResourceFileBinaryRoute extends RouteBuilder {
  @Override
  public void configure() throws Exception {

    from("file:/etc/camel/resources/?fileName=resources-data.zip&noop=true&idempotent=false")
        .unmarshal().zipFile()
        .log("resource file unzipped content is: ${body}");

  }
}
```

2. 运行集成并使用 **--resource** 选项将文件复制到默认目标目录(/etc/camel/resources/)。例如：

```
kamel run --resource file:resources-data.zip ResourceFileBinaryRoute.java -d camel-zipfile --dev
```

   注： 如果您指定了二进制文件，则会在集成中透明地创建并解码文件内容。

   另外，您可以通过重复添加 **--resource** 选项来提供多个资源，例如：

```
kamel run --resource file:resources-data1.txt --resource file:resources-data2.txt ResourceFileBinaryRoute.java -d camel-zipfile --dev
```

4.2.3.2. 将 ConfigMap 指定为资源

如果您有包含配置值的 OpenShift ConfigMap，并且您需要将 ConfigMap 定位为集成的资源，请使用 --resource <configmap-file > 选项。

先决条件

- 设置 Camel K 开发环境
- 您已在 OpenShift 集群中存储了一个或多个 ConfigMap 文件。例如，您可以使用以下命令创建 ConfigMap：

```
oc create configmap my-cm --from-literal=my-configmap-key="configmap content"
```

流程

1. 创建一个 Camel K 集成来引用存储在 OpenShift 集群上的 ConfigMap。

   例如，以下集成（名为 ResourceConfigmapRoute.java）引用名为 my-cm 的 ConfigMap，其中包含 my-configmap-key。

```
import org.apache.camel.builder.RouteBuilder;

public class ResourceConfigmapRoute extends RouteBuilder {
  @Override
  public void configure() throws Exception {

    from("file:/etc/camel/resources/my-cm/?fileName=my-configmap-key&noop=true&idempotent=false")
        .log("resource file content is: ${body}");

  }
}
```

2. 运行集成并使用 --resource 选项将 ConfigMap 文件整理到默认的 /etc/camel/resources/ 目录中，以便其可用于正在运行的集成。

   例如：

```
kamel run --resource configmap:my-cm ResourceConfigmapRoute.java --dev
```

   集成启动时，Camel K 操作器使用 ConfigMap 的内容挂载卷（如 my-configmap-key）。

注：如果您指定了集群中还没有可用的 ConfigMap，则 Integration 只会在 ConfigMap 可用后等待并启动。

4.2.3.3. 将 Secret 指定为资源

如果您有包含配置信息的 OpenShift Secret，并且您需要将它作为可用于一个或多个集成的资源进行定位，请使用 --resource <secret> 语法。

先决条件

- 设置 Camel K 开发环境
- 在 OpenShift 集群中存储了一个或多个 Secret 文件。例如，您可以使用以下命令创建 Secret：

```
oc create secret generic my-sec --from-literal=my-secret-key="very top secret"
```

流程

1. 创建一个 Camel K 集成来引用存储在 OpenShift 集群中的 Secret。

   例如，以下集成（名为 ResourceSecretRoute.java）引用了 my-sec Secret：

```
import org.apache.camel.builder.RouteBuilder;

public class ResourceSecretRoute extends RouteBuilder {
  @Override
  public void configure() throws Exception {

     from("file:/etc/camel/resources/my-sec/?fileName=my-secret-
key&noop=true&idempotent=false")
        .log("resource file content is: ${body}");

  }
}
```

2. 运行集成并使用 --resource 选项将 Secret 整理到默认的 /etc/camel/resources/ 目录中，使其可用于正在运行的集成。

   例如：

```
kamel run --resource secret:my-sec ResourceSecretRoute.java --dev
```

当集成启动时，Camel K operator 会挂载带有 Secret 内容的卷（如 my-sec）。

注： 如果您指定了集群中还没有可用的 Secret，则 Integration 只会在 Secret 可用时等待并启动。

4.2.3.4. 为资源指定目标路径

/etc/camel/resources/ 目录是挂载您使用 --resource 选项指定的资源的默认位置。如果您需要指定挂载资源的不同目录，请使用 --resource @path 语法。

先决条件

- 设置 Camel K 开发环境
- 您有一个包含一个或多个配置属性的文件、ConfigMap 或 Secret。

流程

1. 创建一个引用包含配置属性的文件、ConfigMap 或 Secret 的 Camel K 集成。例如，以下集成（名为 ResourceFileLocationRoute.java）引用了 myprops 文件：

```
import org.apache.camel.builder.RouteBuilder;

public class ResourceFileLocationRoute extends RouteBuilder {
  @Override
  public void configure() throws Exception {

     from("file:/tmp/?fileName=input.txt&noop=true&idempotent=false")
       .log("resource file content is: ${body}");

  }
}
```

2. 运行集成并使用带有 @path 语法的 --resource 选项并指定在哪里挂载资源内容（文件、ConfigMap 或 Secret）：

例如，以下命令指定使用 /tmp 目录挂载 input.txt 文件：

```
kamel run --resource file:resources-data.txt@/tmp/input.txt ResourceFileLocationRoute.java --dev
```

3. 检查集成的 pod，以验证文件（如 input.txt）是否已挂载到正确的位置（例如，在 tmp 目录中）。例如，运行以下命令：

```
oc exec <pod-name> -- cat /tmp/input.txt
```

4.2.3.5. 过滤 ConfigMap 或 Secret 数据

在创建 ConfigMap 或 Secret 时，您可以指定多个信息源。例如，以下命令从两个源创建一个 ConfigMap （名为 my-cm-multi）：

```
oc create configmap my-cm-multi --from-literal=my-configmap-key="configmap content" --from-literal=my-configmap-key-2="another content"
```

当您运行与 --resource 选项集成时，会使用多个源创建的 ConfigMap 或 Secret，默认是两个源。

如果要限制从 ConfigMap 或 Secret 恢复的信息数量，您可以在 ConfigMap 或 Secret 名称后指定 --resource 选项的 /key 标记。例如，-- resource configmap:my-cm/my-key 或 --resource secret:my-secret/my-key。

您可以在 --resource configmap 或 --resource secret 选项后使用 /key 标记来限制集成检索的信息数量。

先决条件

- 设置 Camel K 开发环境
- 您有一个 ConfigMap 或 Secret，其中包含来自多个源的值。

流程

1. 创建一个集成，它使用来自 ConfigMap 或 Secret 中的其中一个资源的配置值。例如，以下集成（名为 ResourceConfigmapKeyLocationRoute.java）引用了 my-cm-multi ConfigMap：

```
import org.apache.camel.builder.RouteBuilder;
```

```
public class ResourceConfigmapKeyLocationRoute extends RouteBuilder {
  @Override
  public void configure() throws Exception {

    from("file:/tmp/app/data/?fileName=my-configmap-key-
2&noop=true&idempotent=false")
        .log("resource file content is: ${body} consumed from
         ${header.CamelFileName}");

  }
}
```

2. 运行集成并使用带有 @path 语法的 --resource 选项并指定在哪里挂载源内容（文件、ConfigMap 或 Secret）：

例如，以下命令指定只使用 ConfigMap 中包含的源之一(my-configmap-key-2@)，并使用 /tmp/app/data 目录挂载它：

```
kamel run --resource configmap:my-cm-multi/my-configmap-key-2@/tmp/app/data
ResourceConfigmapKeyLocationRoute.java --dev
```

3. 检查集成的 pod，以验证是否仅将一个文件（如 my-configmap-key-2）挂载到正确的位置（例如，/tmp/app/data 目录中）。例如，运行以下命令：

```
oc exec <pod-name> -- cat /tmp/app/data/my-configmap-key-2
```

4.3. 配置 CAMEL 集成组件

您可以在集成代码中以编程方式配置 Camel 组件，或在运行时在命令行中使用配置属性。您可以使用以下语法配置 Camel 组件：

```
camel.component.${scheme}.${property}=${value}
```

例如，要为暂存事件驱动的架构更改 Camel seda 组件的队列大小，您可以在命令行中配置以下属性：

```
camel.component.seda.queueSize=10
```

先决条件

- 设置 Camel K 开发环境

流程

- 输入 kamel run 命令，并使用 --property 选项指定 Camel 组件配置。例如：

```
kamel run --property camel.component.seda.queueSize=10 examples/Integration.java
```

其他资源

- 在命令行中提供运行时属性
- Apache Camel SEDA 组件

4.4. 配置 CAMEL K 集成依赖项

Camel K 会自动解决运行集成代码所需的各种依赖项。但是，您可以使用 kamel run --dependency 选项在运行时明确添加命令行中的依赖项。

以下示例集成使用 Camel K 自动依赖项解析：

```
...
  from("imap://admin@myserver.com")
    .to("seda:output")
...
```

由于此集成具有以 imap: 前缀开头的端点，所以 Camel K 可以自动将 camel-mail 组件添加到所需依赖项列表中。seda: 端点属于 camel-core，它会自动添加到所有集成，因此 Camel K 不会为这个组件添加其他依赖项。

Camel K 自动依赖项解析是用户在运行时透明的。这在开发模式中非常有用，因为您可以在不退出开发循环的情况下快速添加您需要的所有组件。

您可以使用 kamel run --dependency 或 -d 选项显式添加依赖项。您可能需要使用它来指定 Camel 目录中不包含的依赖项。您可以在命令行中指定多个依赖项。

先决条件

- 设置 Camel K 开发环境

流程

- **输入 kamel run 命令，并使用 -d 选项指定依赖项。例如：**

```
kamel run -d mvn:com.google.guava:guava:26.0-jre -d camel-mina2 Integration.java
```

注意

您可以通过禁用 dependencies trait: -trait dependencies.enabled=false 来禁用自动依赖项解析。但是，多数情况下不建议这样做。

依赖项类型

kamel run 命令的 -d 标志灵活，支持多个依赖项。

可以使用 -d 标志直接添加 Camel 依赖项，如下所示：

```
kamel run -d camel:http Integration.java
```

在这种情况下，依赖项会添加正确的版本。请注意，指定 Camel 依赖项的标准表示法为 camel:xxx，而 kamel 也接受 camel-xxx 的可用性。

您可以使用 -d 标志、mvn 前缀和 maven 协调添加 外部依赖项 ：

```
kamel run -d mvn:com.google.guava:guava:26.0-jre Integration.java
```

请注意，如果您的依赖项属于私有存储库，则必须定义此存储库。请参阅 Configure maven。

您可以使用 -d 标志和 file:// 前缀添加 Local 依赖项。

```
kamel run -d file://path/to/integration-dep.jar Integration.java
```

然后，可在您的集成中访问 integration-dep.jar 的内容供您使用。

您还可以指定要挂载到正在运行的容器中的数据文件：

```
kamel run -d file://path/to/data.csv:path/in/container/data.csv Integration.java
```

指定目录将递归工作。

请注意，这个功能依赖于 Image Registry 准确设置。

Jitpack Dependencies

如果您的依赖项没有包括在 maven 存储库中，您将找到 Jitpack 作为为运行时集成环境提供任何自定义依赖项的方法。在某些 occasion 中，您会发现只包括路由定义以及一些在定义集成行为时必须使用的帮助程序类或其他类很有用。使用 Jitpack，您将能够编译托管在远程存储库中的 java 项目，并使用生成的软件包作为集成的依赖项。

对于任何 maven 依赖项，用法与上面定义的相同。可以使用 -d 标志来添加它，但这一次您需要为您要使用的项目存储库（即 github）定义前缀。它必须以 repository-kind:user/repo/version 的形式提供。例如，您可以通过执行以下内容来提供 Apache Commons CSV 依赖项：

```
kamel run -d github:apache/commons-csv/1.1 Integration.java
```

我们支持最重要的公共代码存储库：

```
github:user/repo/version
gitlab:user/repo/version
bitbucket:user/repo/version
gitee:user/repo/version
azure:user/repo/version
```

当您想使用主分支时，可以省略 该版本。否则，它将代表项目存储库中使用的分支或标签。

动态 URI

Camel K 并不总是发现所有依赖项。当动态创建 URI 时，您必须指示 Camel K 哪个组件要加载（使用 -d 参数）。以下代码片段演示了这一点。

DynamicURI.java

```
String myTopic = "purchases"
from("kafka:" + myTopic + "? ... ")
    .to(...)
...
```

这里的 URI 由一些在运行时解析的变量动态创建。在这种情况下，您必须指定组件和相关依赖项来加载到 集成中。

其他资源

- 在开发模式下运行 Camel K 集成
- Camel K trait 和 profile 配置
- Apache Camel Mail 组件
- Apache Camel SEDA 组件

第 5 章 针对 KAFKA 验证 CAMEL K

您可以针对 Apache Kafka 验证 Camel K。

以下示例演示了如何设置 Kafka 主题，并在简单的 Producer/Consumer 模式集成中使用它。

5.1. 设置 KAFKA

要设置 Kafka，您必须：

1. 安装所需的 OpenShift operator
2. 创建 Kafka 实例
3. 创建 Kafka 主题

使用以下提到的红帽产品来设置 Kafka：

- Red Hat Advanced 消息队列(AMQ)流 - 自我管理的 Apache Kafka 产品。AMQ Streams 基于开源 Strimzi，并作为 Red Hat Integration 的一部分包含在内。AMQ Streams 是一个分布式、可扩展的流平台，它基于 Apache Kafka，其中包括发布/订阅消息传递代理。Kafka Connect 提供了一个框架，用于将基于 Kafka 的系统与外部系统集成。使用 Kafka Connect，您可以配置源和接收器连接器，将来自外部系统的数据流传输到 Kafka 代理。

5.1.1. 使用 AMQ 流设置 Kafka

AMQ Streams 简化了在 OpenShift 集群中运行 Apache Kafka 的过程。

5.1.1.1. 为 AMQ Streams 准备 OpenShift 集群

要使用 Camel K 或 Kamelets 和 Red Hat AMQ Streams，您必须安装以下 operator 和工具：

- Red Hat Integration - AMQ Streams operator - 管理 Openshift Cluster 和 AMQ

Streams for Apache Kafka 实例之间的通信。

- Red Hat Integration - Camel K operator - 安装和管理 Camel K - 在 OpenShift 上原生运行的轻量级集成框架。
- Camel K CLI 工具 - 允许您访问所有 Camel K 功能。

先决条件

- 熟悉 Apache Kafka 概念。
- 您可以使用正确访问级别访问 OpenShift 4.6 （或更新版本）集群、创建项目和安装操作器的功能，以及在本地系统上安装 OpenShift 和 Camel K CLI。
- 已安装 OpenShift CLI 工具(oc)，以便可以在命令行中与 OpenShift 集群交互。

流程

使用 AMQ Streams 设置 Kafka：

1. 登录您的 OpenShift 集群的 Web 控制台。
2. 创建或打开您要在其中创建集成的项目，如 my-camel-k-kafka。
3. 安装 Camel K operator 和 Camel K CLI，如 安装 Camel K 所述。
4. 安装 AMQ Streams Operator：
   a. 从任何项目中，选择 Operators > OperatorHub。
   b. 在 Filter by Keyword 字段中，键入 AMQ Streams。

c. 点 Red Hat Integration - AMQ Streams 卡，然后点 Install。

此时会打开 Install Operator 页面。

d. 接受默认值，然后点 Install。

5. 选择 Operators > Installed Operators 来验证是否安装了 Camel K 和 AMQ Streams operator。

后续步骤

使用 AMQ Streams 设置 Kafka 主题

5.1.1.2. 使用 AMQ Streams 设置 Kafka 主题

Kafka 主题提供在 Kafka 实例中存储数据的目标。在向它发送数据前，您必须设置 Kafka 主题。

先决条件

- 您可以访问 OpenShift 集群。
- 已安装 Red Hat Integration - Camel K 和 Red Hat Integration - AMQ Streams operator，如 准备 OpenShift 集群 中所述。
- 已安装 OpenShift CLI (oc)和 Camel K CLI (kamel)。

流程

使用 AMQ Streams 设置 Kafka 主题：

1. 登录您的 OpenShift 集群的 Web 控制台。
2. 选择 Projects，然后单击在其中安装 Red Hat Integration - AMQ Streams operator 的项目。例如，单击 my-camel-k-kafka 项目。

3. 选择 **Operators > Installed Operators，然后点 Red Hat Integration - AMQ Streams。**

4. 创建 **Kafka 集群：**

   a. **在 Kafka 下，点 Create instance。**

   b. 为**集群输入一个名称，如 kafka-test。**

   c. **接受其他默**认值，**然后**单击 **Create。**

   创建 **Kafka** 实例的过程可能需要几分钟时间来完成。

   当状态就绪时，继续**下一步。**

5. 创建 **Kafka 主题：**

   a. 选择 **Operators > Installed Operators，然后点 Red Hat Integration - AMQ Streams。**

   b. **在 Kafka 主题 下，点 Create Kafka Topic。**

   c. **输入主题的名称，如 test-topic。**

   d. **接受其他默**认值，**然后**单击 **Create。**

5.1.2. 使用 OpenShift 流设置 Kafka

要使用 OpenShift Streams for Apache Kafka，您必须登录到您的红帽帐户。

5.1.2.1. 为 OpenShift Streams 准备 OpenShift 集群

要使用受管云服务，您必须安装以下 operator 和工具：

- OpenShift Application Services (RHOAS) CLI - 允许您从终端管理应用程序服务。
- Red Hat Integration - Camel K operator 安装并管理 Camel K - 在 OpenShift 上原生运行的轻量级集成框架。
- Camel K CLI 工具 - 允许您访问所有 Camel K 功能。

先决条件

- 熟悉 Apache Kafka 概念。
- 您可以使用正确访问级别访问 OpenShift 4.6 （或更新版本）集群、创建项目和安装操作器的功能，以及在本地系统上安装 OpenShift 和 Apache Camel K CLI。
- 已安装 OpenShift CLI 工具(oc)，以便可以在命令行中与 OpenShift 集群交互。

流程

1. 使用集群管理员帐户登录 OpenShift Web 控制台。
2. 为您的 Camel K 或 Kamelets 应用程序创建 OpenShift 项目。
   a. 选择 Home > Projects。
   b. 点击 Create Project。
   c. 键入项目的名称，如 my-camel-k-kafka，然后单击 Create。
3. 下载并安装 RHOAS CLI，如 Getting started with the rhoas CLI 所述。

4. 安装 Camel K operator 和 Camel K CLI，如 安装 Camel K 所述。

5. 要验证是否安装了 Red Hat Integration - Camel K Operator，点 Operators > Installed Operators。

后续步骤

使用 RHOAS 设置 Kafka 主题

5.1.2.2. 使用 RHOAS 设置 Kafka 主题

Kafka 整理有关 *主题的消息*。每个主题都有一个名称。应用向主题发送消息并从主题检索消息。Kafka 主题提供在 Kafka 实例中存储数据的目标。在向它发送数据前，您必须设置 Kafka 主题。

先决条件

- 您可以使用正确访问级别访问 OpenShift 集群、创建项目和安装操作器，以及在本地系统上安装 OpenShift 和 Camel K CLI。
- 已安装 OpenShift CLI (oc)、Camel K CLI (kamel)和 RHOAS CLI (rhoas)工具，如 准备 OpenShift 集群 中所述。
- 已安装 Red Hat Integration - Camel K operator，如 准备 OpenShift 集群 中所述。
- 您已登录到 Red Hat Cloud 站点。

流程

设置 Kafka 主题：

1. 在命令行中登录到您的 OpenShift 集群。

2. 打开您的项目，例如：

```
oc project my-camel-k-kafka
```

3. 验证 Camel K Operator 是否已安装到项目中：

```
oc get csv
```

结果列出了 Red Hat Camel K operator，并表示它处于 Succeeded 阶段。

4. 准备 Kafka 实例并将其连接到 RHOAS：

   a. 使用以下命令登录到 RHOAS CLI：

   ```
   RHOAS 登录
   ```

   b. 创建一个 kafka 实例，如 kafka-test ：

   ```
   rhoas kafka create kafka-test
   ```

   创建 Kafka 实例的过程可能需要几分钟时间来完成。

5. 检查 Kafka 实例的状态：

```
RHOAS 状态
```

您还可以在 web 控制台中查看状态：

https://cloud.redhat.com/application-services/streams/kafkas/

当状态 就绪时，继续下一步。

6. 创建新的 Kafka 主题：

   rhoas kafka topic create --name test-topic

7. 将 Kafka 实例（集群）与 Openshift Application Services 实例连接：

   RHOAS 集群连接

8. 按照获取凭证令牌的脚本说明进行操作。

   您应该看到类似如下的输出：

```
Token Secret "rh-cloud-services-accesstoken-cli" created successfully
Service Account Secret "rh-cloud-services-service-account" created successfully
KafkaConnection resource "kafka-test" has been created
KafkaConnection successfully installed on your cluster.
```

后续步骤

- 获取 Kafka 凭证

5.1.2.3. 获取 Kafka 凭证

要将应用程序或服务连接到 Kafka 实例，您必须首先获取以下 Kafka 凭证：

- 获取 bootstrap URL。
- 使用凭证（用户名和密码）创建服务帐户。

对于 OpenShift Streams，身份验证协议是 SASL_SSL。

前提条件

- 您已创建了 Kafka 实例，它处于 ready 状态。

- 您已创建了 Kafka 主题。

流程

1. 获取 Kafka Broker URL (Bootstrap URL)：

   RHOAS 状态

   这个命令返回类似如下的输出：

```
Kafka
----------------------------------------------------------------
ID:                 1ptdfZRHmLKwqW6A3YKM2MawgDh
Name:                 my-kafka
Status:               ready
Bootstrap URL:        my-kafka--ptdfzrhmlkwqw-a-ykm-mawgdh.kafka.devshift.org:443
```

2. 要获取用户名和密码，请使用以下语法创建服务帐户：

   rhoas service-account create --name "<account-name>" --file-format json

   注意

   在创建服务帐户时，您可以选择文件格式和位置来保存凭证。如需更多信息，请键入 **rhoas service-account create --help**

   例如：

   rhoas service-account create --name "my-service-acct" --file-format json

   服务帐户已创建并保存到 JSON 文件中。

3. 要验证您的服务帐户凭证，请查看 **credentials.json** 文件：

cat credentials.json

这个命令返回类似如下的输出：

```
{"clientID":"srvc-acct-eb575691-b94a-41f1-ab97-50ade0cd1094", "password":"facf3df1-3c8d-4253-aa87-8c95ca5e1225"}
```

4. **授予向 Kakfa 主题或从 Kakfa 发送和接收消息的权限。使用以下命令，其中 clientID 是 credentials.json 文件中提供的值（从第 3 步中）。**

```
rhoas kafka acl grant-access --producer --consumer --service-account $CLIENT_ID --topic test-topic --group all
```

例如：

```
rhoas kafka acl grant-access --producer --consumer --service-account srvc-acct-eb575691-b94a-41f1-ab97-50ade0cd1094 --topic test-topic --group all
```

5.1.2.4. 使用 SASL/Plain 验证方法创建 secret

您可以使用您获取的凭证创建 secret (Kafka bootstrap URL、服务帐户 ID 和服务帐户 secret)。

流程

1. 编辑 **application.properties 文件并添加 Kafka 凭证。**

application.properties 文件

```
camel.component.kafka.brokers = <YOUR-KAFKA-BOOTSTRAP-URL-HERE>
camel.component.kafka.security-protocol = SASL_SSL
camel.component.kafka.sasl-mechanism = PLAIN
camel.component.kafka.sasl-jaas-config=org.apache.kafka.common.security.plain.PlainLoginModule required username='<YOUR-SERVICE-ACCOUNT-ID-HERE>' password='<YOUR-SERVICE-ACCOUNT-SECRET-HERE>';
consumer.topic=<TOPIC-NAME>
producer.topic=<TOPIC-NAME>
```

2. 运行以下命令，在 **application.properties** 文件中创建一个包含敏感属性的 **secret**：

```
oc create secret generic kafka-props --from-file application.properties
```

在运行 **Camel K** 集成时，您可以使用此 **secret**。

另请参阅

Camel K Kafka Basic Quickstart

5.1.2.5. 使用 SASL/OAUTHBearer 身份验证方法创建 secret

您可以使用您获取的凭证创建 **secret (Kafka bootstrap URL**、服务帐户 **ID** 和服务帐户 **secret)**。

流程

1. 编辑 **application-oauth.properties** 文件并添加 **Kafka** 凭证。

application-oauth.properties file

```
camel.component.kafka.brokers = <YOUR-KAFKA-BOOTSTRAP-URL-HERE>
camel.component.kafka.security-protocol = SASL_SSL
camel.component.kafka.sasl-mechanism = OAUTHBEARER
camel.component.kafka.sasl-jaas-config =
org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule required \
oauth.client.id='<YOUR-SERVICE-ACCOUNT-ID-HERE>' \
oauth.client.secret='<YOUR-SERVICE-ACCOUNT-SECRET-HERE>' \
oauth.token.endpoint.uri="https://identity.api.openshift.com/auth/realms/rhoas/protocol/openid-
connect/token" ;
camel.component.kafka.additional-
properties[sasl.login.callback.handler.class]=io.strimzi.kafka.oauth.client.JaasClientOauthLoginC
allbackHandler

consumer.topic=<TOPIC-NAME>
producer.topic=<TOPIC-NAME>
```

2. 运行以下命令，在 **application.properties** 文件中创建一个包含敏感属性的 **secret**：

```
oc create secret generic kafka-props --from-file application-oauth.properties
```

在运行 **Camel K** 集成时，您可以使用此 **secret**。

另请参阅

Camel K Kafka Basic Quickstart

5.2. 运行 KAFKA 集成

运行制作者集成

1. 创建示例制作者集成。这会每 **10** 秒填充一条消息。

Sample SaslSSLKafkaProducer.java

```
// kamel run --secret kafka-props SaslSSLKafkaProducer.java --dev
// camel-k: language=java dependency=mvn:org.apache.camel.quarkus:camel-quarkus-kafka dependency=mvn:io.strimzi:kafka-oauth-client:0.7.1.redhat-00003

import org.apache.camel.builder.RouteBuilder;
import org.apache.camel.component.kafka.KafkaConstants;

public class SaslSSLKafkaProducer extends RouteBuilder {
  @Override
  public void configure() throws Exception {
  log.info("About to start route: Timer -> Kafka ");
  from("timer:foo")
    .routeId("FromTimer2Kafka")
    .setBody()
      .simple("Message #${exchangeProperty.CamelTimerCounter}")
    .to("kafka:{{producer.topic}}")
    .log("Message correctly sent to the topic!");
  }
}
```

2. **然后，运行集成过程。**

```
kamel run --secret kafka-props SaslSSLKafkaProducer.java --dev
```

制作者将创建一个新消息并推送到主题，并记录一些信息。

```
[2] 2021-05-06 08:48:11,854 INFO  [FromTimer2Kafka] (Camel (camel-1) thread #1 -
KafkaProducer[test]) Message correctly sent to the topic!
[2] 2021-05-06 08:48:11,854 INFO  [FromTimer2Kafka] (Camel (camel-1) thread #3 -
KafkaProducer[test]) Message correctly sent to the topic!
[2] 2021-05-06 08:48:11,973 INFO  [FromTimer2Kafka] (Camel (camel-1) thread #5 -
KafkaProducer[test]) Message correctly sent to the topic!
[2] 2021-05-06 08:48:12,970 INFO  [FromTimer2Kafka] (Camel (camel-1) thread #7 -
KafkaProducer[test]) Message correctly sent to the topic!
[2] 2021-05-06 08:48:13,970 INFO  [FromTimer2Kafka] (Camel (camel-1) thread #9 -
KafkaProducer[test]) Message correctly sent to the topic!
```

运行消费者集成

1. **创建消费者集成。**

Sample SaslSSLKafkaProducer.java

```
// kamel run --secret kafka-props SaslSSLKafkaConsumer.java --dev
// camel-k: language=java dependency=mvn:org.apache.camel.quarkus:camel-quarkus-
kafka dependency=mvn:io.strimzi:kafka-oauth-client:0.7.1.redhat-00003

import org.apache.camel.builder.RouteBuilder;

public class SaslSSLKafkaConsumer extends RouteBuilder {
  @Override
  public void configure() throws Exception {
 log.info("About to start route: Kafka -> Log ");
 from("kafka:{{consumer.topic}}")
    .routeId("FromKafka2Log")
    .log("${body}");
  }
}
```

2. **打开另一个 shell，并使用以下命令运行消费者集成：**

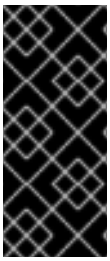
```
kamel run --secret kafka-props SaslSSLKafkaConsumer.java --dev
```

消费者将开始记录主题中找到的事件：

```
[1] 2021-05-06 08:51:08,991 INFO  [FromKafka2Log] (Camel (camel-1) thread #0 -
KafkaConsumer[test]) Message #8
[1] 2021-05-06 08:51:10,065 INFO  [FromKafka2Log] (Camel (camel-1) thread #0 -
KafkaConsumer[test]) Message #9
[1] 2021-05-06 08:51:10,991 INFO  [FromKafka2Log] (Camel (camel-1) thread #0 -
KafkaConsumer[test]) Message #10
[1] 2021-05-06 08:51:11,991 INFO  [FromKafka2Log] (Camel (camel-1) thread #0 -
KafkaConsumer[test]) Message #11
```

第 6 章 CAMEL K TRAIT 配置参考

本章提供有关您可以使用 *特征* 在运行时在命令行中配置的高级功能和核心功能的参考信息。Camel K 提供 *特征* 来配置特定功能和技术。Camel K 提供 *平台特征* 来配置内部 Camel K 核心功能。



重要

Red Hat Integration - Camel K 1.6 包括 OpenShift 和 Knative 配置集。Kubernetes 配置集仅支持社区支持。它还包括 Java 和 YAML DSL 支持集成。其他 XML、Groovy、JavaScript 和 Kotlin 等语言仅支持社区支持。

本章包括以下部分：

Camel K 功能特征

- 第 6.2.1 节 “Knative Trait” - 技术预览
- 第 6.2.2 节 “Knative Service Trait”- 技术预览
- 第 6.2.3 节 “Prometheus Trait”
- 第 6.2.4 节 “PDB Trait”
- 第 6.2.5 节 “Pull Secret Trait”
- 第 6.2.6 节 “路由 Trait”
- 第 6.2.7 节 “Service Trait”

Camel K 核心平台特征

- 第 6.3.1 节 “builder Trait”

6.1. CAMEL K TRAIT 和 PROFILE 配置

本节介绍 *特征* 和*配置文件* 的重要 Camel K 概念，它们在运行时用于配置高级 Camel K 功能。

Camel K traits

Camel K traits 是可在命令行中配置的高级功能和核心功能，以自定义 Camel K 集成。例如，这包括配置与 3scale API 管理、Quarkus、Knative 和 Prometheus 等技术交互 *的功能特征*。Camel K 还提供内部 *平台特征*，用于配置重要的核心平台功能，如 Camel 支持、容器、依赖项解析和 JVM 支持。

Camel K 配置集

Camel K 配置集定义了 Camel K 集成运行的目标云平台。支持的配置集有 OpenShift 和 Knative 配置集。

注意

当您在 OpenShift 上运行集成时，Camel K 会在集群中安装 OpenShift Serverless 时使用 Knative 配置集。如果没有安装 OpenShift Serverless，Camel K 使用 OpenShift 配置集。

您还可以使用 kamel run --profile 选项在运行时指定配置集。

Camel K 为所有特征提供有用的默认值，考虑集成运行的目标配置文件。但是，高级用户可以配置 Camel K traits 以进行自定义行为。有些特征只适用于特定的配置集，如 OpenShift 或 Knative。如需了解更多详细信息，请参阅每个特征描述中的可用配置集。

Camel K trait 配置

每个 Camel 特征都有一个唯一 ID，可用于在命令行上配置特征。例如，以下命令禁用为集成创建 OpenShift Service：

```
kamel run --trait service.enabled=false my-integration.yaml
```

您还可以使用 -t 选项指定特征。

Camel K trait 属性

您可以使用 enabled 属性启用或禁用每个特征。所有特征都有自己的内部逻辑，用于决定是否需要在

用户显式激活它们时启用它们。

警告

禁用平台特征可能会破坏平台功能。

有些特征有一个 auto 属性，您可以使用它来根据环境启用或禁用特征的自动配置。例如，这包括 3scale、Cron 和 Knative 等特征。当未明确设置 enabled 属性时，此自动配置可以启用或禁用特征，并可更改特征配置。

大多数特征都有额外的属性，您可以在命令行中配置。如需了解更多详细信息，请参阅后续小节中每个特征的描述。

6.2. CAMEL K 功能特征

6.2.1. Knative Trait

Knative trait 会自动发现 Knative 资源的地址并将其注入正在运行的集成。

完整的 Knative 配置以 JSON 格式以 CAMEL_KNATIVE_CONFIGURATION 注入。然后，Camel Knative 组件将使用完整配置来配置路由。

当 Knative 配置集处于活跃状态时，会默认启用特征。

这个特征包括在以下配置集：Knative。

6.2.1.1. 配置

运行与 CLI 的任何集成时可以指定特征属性：

```
$ kamel run --trait knative.[key]=[value] --trait knative.[key2]=[value2] integration.java
```

可用的配置选项如下：

属性	类型	描述
knative.enabled	bool	可用于启用或禁用特征。所有特征共享这个通用属性。
knative.configuration	字符串	可用于以 JSON 格式注入 Knative 完成配置。
knative.channel-sources	[]string	用作集成路由源的频道列表。可以包含简单的频道名称或完整的 Camel URI。
knative.channel-sinks	[]string	用作集成路由的频道列表。可以包含简单的频道名称或完整的 Camel URI。
knative.endpoint-sources	[]string	用作集成路由源的频道列表。
knative.endpoint-sinks	[]string	用作集成路由目标的端点列表。可以包含简单的端点名称或完整的 Camel URI。
knative.event-sources	[]string	集成将订阅的事件类型列表。可以包含简单事件类型或完整 Camel URI （使用与"默认"）不同的特定代理。
knative.event-sinks	[]string	集成生成的事件类型列表。可以包含简单的事件类型或完整 Camel URI （使用特定代理）。
knative.filter-source-channels	bool	启用根据标头 "ce-knativehistory" 过滤事件。因为这个标头已在较新版本的 Knative 中删除，所以默认禁用过滤。
knative.sink-binding	bool	允许通过 Knative SinkBinding 资源将集成绑定到接收器。当集成以单一接收器为目标时使用。当集成以单个 sink 为目标时（除集成由 Knative 源所有时），它会被默认启用。
knative.auto	bool	启用所有特征属性的自动发现。

6.2.2. Knative Service Trait

Knative Service trait 允许在作为 Knative 服务而不是标准 Kubernetes Deployment 运行集成时配置选项。

以 Knative Services 身份运行集成会添加自动扩展（和扩展到零）功能，但这些功能仅在路由使用 HTTP 端点消费者时才有意义。

这个特征包括在以下配置集：Knative。

6.2.2.1. 配置

运行与 CLI 的任何集成时可以指定特征属性：

```
$ kamel run --trait knative-service.[key]=[value] --trait knative-service.[key2]=[value2] Integration.java
```

可用的配置选项如下：

属性	类型	描述
knative-service.enabled	bool	可用于启用或禁用特征。所有特征共享这个通用属性。
knative-service.annotations	map[string]string	注解添加到路由中。这可用于设置 knative 服务特定注解。如需了解更多详细信息，请参阅 特定于路由的注解。 CLI usage example: -t "knative-service.annotations.'haproxy.router.openshift.io/balance'=roundrobin"
knative-service.autoscaling-class	字符串	配置 Knative autoscaling class 属性（例如，设置 **hpa.autoscaling.knative.dev** 或 **kpa.autoscaling.knative.dev** 自动扩展）。 如需更多信息，请参阅 Knative 文档。
knative-service.autoscaling-metric	字符串	配置 Knative 自动扩展指标属性（例如，设置基于 **并发** 或 **cpu** 的自动扩展）。 如需更多信息，请参阅 Knative 文档。
knative-service.autoscaling-target	int	为每个 Pod 设置允许的并发级别或 CPU 百分比（取决于自动扩展指标）。 如需更多信息，请参阅 Knative 文档。
knative-service.min-scale	int	集成时应随时运行的最小 Pod 数量。默认情况下 为**零**，这意味着当没有用于配置的时间时，集成将缩减为零。 如需更多信息，请参阅 Knative 文档。

属性	类型	描述
knative-service.max-scale	**int**	一个上限，用于可以并行针对集成运行的 Pod 数量。Knative 都有自己的 cap 值，它依赖于安装。 如需更多信息，请参阅 Knative 文档。
knative-service.auto	**bool**	当所有条件满足时，自动将集成部署为 Knative 服务： • 集成使用 Knative 配置集 • 所有路由都从基于 HTTP 的消费者或被动消费者开始（例如，**直接** 是被动消费者）

6.2.3. Prometheus Trait

Prometheus trait 配置一个与 Prometheus 兼容的端点。它还会创建一个 PodMonitor 资源，以便在使用 Prometheus operator 时自动提取端点。

指标通过 MicroProfile 指标来公开。

警告

创建 PodMonitor 资源需要安装 Prometheus Operator 自定义资源定义。您可以将 pod-monitor 设置为 false，以便 Prometheus 特征在没有 Prometheus Operator 的情况下正常工作。

Prometheus 特征默认为禁用。

此特征位于以下配置集： Kubernetes、Knative、OpenShift。

6.2.3.1. 配置

运行与 CLI 的任何集成时可以指定特征属性：

```
$ kamel run --trait prometheus.[key]=[value] --trait prometheus.[key2]=[value2]
Integration.java
```

可用的配置选项如下：

属性	类型	描述
prometheus.enabled	bool	可用于启用或禁用特征。所有特征共享这个通用属性。
prometheus.pod-monitor	bool	是否已创建 PodMonitor 资源（默认为 true）。
prometheus.pod-monitor-labels	[]string	PodMonitor 资源标签，适用于 pod-monitor 为 true。

6.2.4. PDB Trait

PDB 特征允许为集成 pod 配置 PodDisruptionBudget 资源。

此特征位于以下配置集： Kubernetes、 Knative、 OpenShift。

6.2.4.1. 配置

运行与 CLI 的任何集成时可以指定特征属性：

```
$ kamel run --trait pdb.[key]=[value] --trait pdb.[key2]=[value2] Integration.java
```

可用的配置选项如下：

属性	类型	描述
pdb.enabled	bool	可用于启用或禁用特征。所有特征共享这个通用属性。
pdb.min-available	字符串	集成的 pod 数量，在驱除后仍可用。它可以是绝对数字或一个百分比。只能指定 min-available 和 max-unavailable 之一。
pdb.max-unavailable	字符串	驱除后可以无法使用的集成 pod 数量。它可以是一个绝对数字或一个百分比（如果 min-available 也未设置，则默认为 1）。只能指定 max-unavailable 和 min-available 之一。

6.2.5. Pull Secret Trait

Pull Secret trait 在 pod 上设置 pull secret，以允许 Kubernetes 从外部 registry 检索容器镜像。

可以手动指定 pull secret，或者在 IntegrationPlatform 上为外部容器 registry 配置身份验证，则相同的 secret 用于拉取镜像。

每当您为外部容器 registry 配置身份验证时，它都会默认启用，因此它会假定外部 registry 是私有的。

如果您的 registry 不需要身份验证来拉取镜像，您可以禁用这个特征。

此特征位于以下配置集： Kubernetes、Knative、OpenShift。

6.2.5.1. 配置

运行与 CLI 的任何集成时可以指定特征属性：

```
$ kamel run --trait pull-secret.[key]=[value] --trait pull-secret.[key2]=[value2] Integration.java
```

可用的配置选项如下：

属性	类型	描述
pull-secret.enabled	bool	可用于启用或禁用特征。所有特征共享这个通用属性。
pull-secret.secret-name	字符串	在 Pod 上设置的 pull secret 名称。如果留空，则会自动从 **IntegrationPlatform** registry 配置获取。
pull-secret.image-puller-delegation	bool	将全局 Operator 与共享平台搭配使用时，这将启用将 operator 命名空间中的 **system:image-puller** 集群角色委派到集成服务帐户。
pull-secret.auto	bool	如果 pod 类型为 **kubernetes.io/dockerconfigjson**，则自动配置 pod 上的平台 registry secret。

6.2.6. 路由 Trait

Route 特征可用于配置用于集成的 OpenShift 路由的创建。

证书和密钥内容可以从本地文件系统或 Openshift secret 对象中提供。用户可以使用以 -secret 结尾的参数（例如： tls-certificate-secret）来引用存储在 secret 中的证书。以 -secret 结尾的参数具有更高的优先级，如果设置了相同的路由参数，例如： tls-key-secret 和 tls-key，然后使用 tls-key-secret。设置密钥和证书的建议方法是使用 secret 存储其内容，并使用以下参数引用它们： tls-certificate-secret,tls-ca-certificate-secret , tls- destination-ca-certificate-secret ,tls-destination-ca-certificate-secret 查看此页面末尾的 examples 部分，以查看设置选项。

此特征位于以下配置集： OpenShift。

6.2.6.1. 配置

运行与 CLI 的任何集成时可以指定特征属性：

```
$ kamel run --trait route.[key]=[value] --trait route.[key2]=[value2] integration.java
```

可用的配置选项如下：

属性	类型	描述
route.enabled	bool	可用于启用或禁用特征。所有特征共享这个通用属性。
route.annotations	map[string]string	注解添加到路由中。这可用于设置路由特定注解。有关注解选项，请参阅特定于路由的注解。CLI usage example: -t "route.annotations.'haproxy.router.openshift.io/balance'=roundrobin
route.host	字符串	配置路由公开的主机。
route.tls-termination	字符串	TLS 终止类型，如 边缘、透传 或重新加密。 如需更多信息，请参阅 OpenShift 路由文档。
route.tls-certificate	字符串	TLS 证书内容。 如需更多信息，请参阅 OpenShift 路由文档。
route.tls-certificate-secret	字符串	对 TLS 证书的 secret 名称和密钥引用。如果 secret 中只有一个键要读取，则格式为 "secret-name[/key-name]"，则值代表 secret 名称，否则您可以设置使用 "/" 分隔的密钥名称。 如需更多信息，请参阅 OpenShift 路由文档。

属性	类型	描述
route.tls-key	**字符串**	TLS 证书密钥内容。 如需更多信息，请参阅 OpenShift 路由文档。
route.tls-key-secret	**字符串**	对 TLS 证书密钥的 secret 名称和密钥引用。如果 secret 中只有一个键要读取，则格式为 "secret-name[/key-name]"，则值代表 secret 名称，否则您可以设置使用 "/" 分隔的密钥名称。 如需更多信息，请参阅 OpenShift 路由文档。
route.tls-ca-certificate	**字符串**	TLS CA 证书内容。 如需更多信息，请参阅 OpenShift 路由文档。
route.tls-ca-certificate-secret	**字符串**	对 TLS CA 证书的 secret 名称和密钥引用。如果 secret 中只有一个键要读取，则格式为 "secret-name[/key-name]"，则值代表 secret 名称，否则您可以设置使用 "/" 分隔的密钥名称。 如需更多信息，请参阅 OpenShift 路由文档。
route.tls-destination-ca-certificate	**字符串**	目标 CA 证书提供最终目的地的 ca 证书的内容。使用重新加密终止时，应提供此文件，以便让路由器将它用于安全连接上的健康检查。如果没有指定此字段，路由器可能会提供自己的目标 CA，并使用短服务名称(service.namespace.svc)执行主机名验证，它允许基础架构生成的证书自动验证。 如需更多信息，请参阅 OpenShift 路由文档。
route.tls-destination-ca-certificate-secret	**字符串**	对目标 CA 证书的 secret 名称和密钥引用。如果 secret 中只有一个键要读取，则格式为 "secret-name[/key-name]"，则值代表 secret 名称，否则您可以设置使用 "/" 分隔的密钥名称。 如需更多信息，请参阅 OpenShift 路由文档。
route.tls-insecure-edge-termination-policy	**字符串**	要配置如何处理不安全的流量，例如 **允许**、**禁用** 或重定向流量。 **如需更多信息，请参阅 OpenShift 路由文档。**

6.2.6.2. 例子

这些示例使用 **secret** 来存储集成中引用的证书和密钥。有关路由的详细信息，请参阅 **Openshift** 路由文档。**PlatformHttpServer.java** 是集成示例。

作为运行这些示例的要求，您应该有一个带有密钥和证书的 secret。

6.2.6.2.1. 生成自签名证书并创建 secret

```
openssl genrsa -out tls.key
openssl req -new -key tls.key -out csr.csr -subj "/CN=my-server.com"
openssl x509 -req -in csr.csr -signkey tls.key -out tls.crt
oc create secret tls my-combined-certs --key=tls.key  --cert=tls.crt
```

6.2.6.2.2. 向路由发出 HTTP 请求

对于所有示例，您可以使用以下 curl 命令发出 HTTP 请求。如果您使用不支持这些内联脚本的 shell，它会使用内联脚本来检索 openshift 命名空间和集群基域，您应该将内联脚本替换为实际命名空间和基域的值。

```
curl -k https://platform-http-server-`oc config view --minify -o 'jsonpath={..namespace}'`.`oc get dnses/cluster -ojsonpath='{.spec.baseDomain}'`/hello?name=Camel-K
```

- 要使用 secret 添加 边缘路由，请使用以 -secret 结尾的参数来设置包含证书的 secret 名称。此路由示例特征引用名为 my-combined-certs 的 secret，它包含两个名为 tls.key 和 tls.crt 的键。

```
kamel run --dev PlatformHttpServer.java -t route.tls-termination=edge -t route.tls-certificate-secret=my-combined-certs/tls.crt -t route.tls-key-secret=my-combined-certs/tls.key
```

- 要使用 secret 添加 passthrough 路由，在集成 pod 中设置了 TLS，在运行集成 pod 中应该可以看到密钥和证书，以便我们使用 --resource kamel 参数在集成 pod 中挂载 secret，然后使用一些 camel quarkus 参数在运行的 pod 中引用这些证书文件，它们以 -p quarkus.http.ssl.certificate.此路由示例特征引用名为 my-combined-certs 的 secret，它包含两个名为 tls.key 和 tls.crt 的键。

```
kamel run --dev PlatformHttpServer.java --resource secret:my-combined-certs@/etc/ssl/my-combined-certs -p quarkus.http.ssl.certificate.file=/etc/ssl/my-combined-certs/tls.crt -p quarkus.http.ssl.certificate.key-file=/etc/ssl/my-combined-certs/tls.key -t route.tls-termination=passthrough -t container.port=8443
```

- 要使用 secret 添加 重新加密路由，在集成 pod 中设置 TLS，在运行集成 pod 中应该可以看到密钥和证书，以达到此目的，使用 --resource kamel 参数在集成 pod 中挂载 secret，然后使用一些 camel quarkus 参数在运行的 pod 中引用这些证书文件，它们以 -p quarkus.http.ssl.certificate.此路由示例特征引用名为 my-combined-certs 的 secret，它包含两个名为 tls.key 和 tls.crt 的键。

```
kamel run --dev PlatformHttpServer.java --resource secret:my-combined-certs@/etc/ssl/my-combined-certs  -p quarkus.http.ssl.certificate.file=/etc/ssl/my-combined-certs/tls.crt -p quarkus.http.ssl.certificate.key-file=/etc/ssl/my-combined-certs/tls.key -t route.tls-termination=reencrypt -t route.tls-destination-ca-certificate-secret=my-combined-certs/tls.crt -t route.tls-certificate-secret=my-combined-certs/tls.crt -t route.tls-key-secret=my-combined-certs/tls.key -t container.port=8443
```

- 使用来自路由和 Openshift 服务为集成端点的 secret 的特定证书 添加 重新加密路由。这样，Openshift 服务用证书只在集成 pod 中设置。密钥和证书应该在正在运行的集成 pod 中看到，为了达到此目的，我们使用 --resource kamel 参数在集成 pod 中挂载 secret，然后使用一些 camel quarkus 参数在运行的 pod 中引用这些证书文件，它们以 -p quarkus.http.ssl.certificate.此路由示例特征引用名为 my-combined-certs 的 secret，它包含两个名为 tls.key 和 tls.crt 的键。

```
kamel run --dev PlatformHttpServer.java --resource secret:cert-from-openshift@/etc/ssl/cert-from-openshift  -p quarkus.http.ssl.certificate.file=/etc/ssl/cert-from-openshift/tls.crt -p quarkus.http.ssl.certificate.key-file=/etc/ssl/cert-from-openshift/tls.key -t route.tls-termination=reencrypt -t route.tls-certificate-secret=my-combined-certs/tls.crt -t route.tls-key-secret=my-combined-certs/tls.key -t container.port=8443
```

然后，您应该注解集成服务来注入 Openshift 服务服务证书

```
oc annotate service platform-http-server service.beta.openshift.io/serving-cert-secret-name=cert-from-openshift
```

- 使用证书和私钥添加 边缘路由，请从本地文件系统提供的私钥。这个示例使用内联脚本读取证书和私钥文件内容，然后删除所有新行字符（这需要将证书设置为参数的值），因此这些值位于一行中。

```
kamel run PlatformHttpServer.java --dev -t route.tls-termination=edge -t route.tls-certificate="$(cat tls.crt|awk 'NF {sub(/\r/, ""); printf "%s\\n",$0;}')" -t route.tls-key="$(cat tls.key|awk 'NF {sub(/\r/, ""); printf "%s\\n",$0;}')"
```

6.2.7. Service Trait

Service trait 会公开与 Service 资源的集成，以便它可以被同一命名空间中的其他应用程序（或集成）访问。

如果集成依赖于可公开 HTTP 端点的 Camel 组件，则默认启用它。

此特征位于以下配置集： Kubernetes、OpenShift。

6.2.7.1. 配置

运行与 CLI 的任何集成时可以指定特征属性：

```
$ kamel run --trait service.[key]=[value] --trait service.[key2]=[value2] Integration.java
```

可用的配置选项如下：

属性	类型	描述
service.enabled	bool	可用于启用或禁用特征。所有特征共享这个通用属性。
service.auto	bool	如果需要创建服务，从代码自动检测。
service.node-port	bool	启用服务以 NodePort 的形式公开（默认为 **false**）。

6.3. CAMEL K 平台特征

6.3.1. builder Trait

builder trait 在内部用于决定构建和配置 IntegrationKit 的最佳策略。

此特征位于以下配置集： Kubernetes、 Knative、 OpenShift。

警告

builder trait 是一个 平台特征 ：禁用它可能会破坏平台功能。

6.3.1.1. 配置

运行与 CLI 的任何集成时可以指定特征属性：

```
$ kamel run --trait builder.[key]=[value] --trait builder.[key2]=[value2] Integration.java
```

可用的配置选项如下：

属性	类型	描述
builder.enabled	bool	可用于启用或禁用特征。所有特征共享这个通用属性。
builder.verbose	bool	对支持它的构建组件（如 OpenShift 构建 Pod）启用详细日志记录。不支持 Kaniko 和 Buildah。
builder.properties	[]string	要提供给构建任务的属性列表

6.3.2. 容器 Trait

Container trait 可用于配置运行集成的容器的属性。

它还为与容器关联的服务提供配置。

此特征位于以下配置集： Kubernetes、 Knative、 OpenShift。

警告

容器特征是一个 平台特征 ：禁用它可能会破坏平台功能。

6.3.2.1. 配置

运行与 CLI 的任何集成时可以指定特征属性：

```
$ kamel run --trait container.[key]=[value] --trait container.[key2]=[value2] Integration.java
```

可用的配置选项如下：

属性	类型	描述
container.enabled	**bool**	可用于启用或禁用特征。所有特征共享这个通用属性。
container.auto	**bool**	
container.request-cpu	**字符串**	需要的最小 CPU 量。
container.request-memory	**字符串**	需要的最小内存量。
container.limit-cpu	**字符串**	需要的最大 CPU 量。
container.limit-memory	**字符串**	需要的最大内存量。
container.expose	**bool**	用于通过 kubernetes Service 启用/禁用风险。
container.port	**int**	配置容器公开的不同端口（默认为 **8080**）。
container.port-name	**字符串**	为容器公开的端口配置不同的端口名称（默认 **http**）。
container.service-port	**int**	要配置要在哪个服务端口下公开容器端口（默认为 **80**）。
container.service-port-name	**字符串**	要配置要在哪个服务端口名称下公开容器端口（默认 **http**）。
container.name	**字符串**	主容器名称。默认名为 **integration**。
container.image	**字符串**	主容器镜像
container.probes-enabled	**bool**	容器中的 ProbesEnabled 启用/禁用探测（默认为 **false**）
container.liveness-initial-delay	**int32**	容器启动后的秒数，然后再启动存活度探测。
container.liveness-timeout	**int32**	探测超时的秒数。适用于存活度探测。
container.liveness-period	**int32**	执行探测的频率。适用于存活度探测。
container.liveness-success-threshold	**int32**	在失败后，探测被视为成功的最低连续成功。适用于存活度探测。

属性	类型	描述
container.liveness-failure-threshold	int32	在成功后，探测被视为失败的连续最小失败。适用于存活度探测。
container.readiness-initial-delay	int32	容器启动后的秒数，然后再启动就绪度探测。
container.readiness-timeout	int32	探测超时的秒数。适用于就绪度探测。
container.readiness-period	int32	执行探测的频率。适用于就绪度探测。
container.readiness-success-threshold	int32	在失败后，探测被视为成功的最低连续成功。适用于就绪度探测。
container.readiness-failure-threshold	int32	在成功后，探测被视为失败的连续最小失败。适用于就绪度探测。

6.3.3. Camel Trait

Camel trait 可用于配置 Apache Camel K 运行时和相关库的版本，无法禁用。

此特征位于以下配置集： Kubernetes、Knative、OpenShift。

警告

camel trait 是一个 平台特征 ：禁用它可能会破坏平台功能。

6.3.3.1. 配置

运行与 CLI 的任何集成时可以指定特征属性：

```
$ kamel run --trait camel.[key]=[value] --trait camel.[key2]=[value2] Integration.java
```

可用的配置选项如下：

属性	类型	描述
camel.enabled	bool	可用于启用或禁用特征。所有特征共享这个通用属性。

6.3.4. 依赖项 Trait

依赖项特征在内部用于根据用户希望运行的集成自动添加运行时依赖项。

此特征位于以下配置集： Kubernetes、 Knative、 OpenShift。

警告

依赖项特征是 一个平台特征 ：禁用它可能会破坏平台功能。

6.3.4.1. 配置

运行与 CLI 的任何集成时可以指定特征属性：

```
$ kamel run --trait dependencies.[key]=[value] Integration.java
```

可用的配置选项如下：

属性	类型	描述
dependencies.enabled	bool	可用于启用或禁用特征。所有特征共享这个通用属性。

6.3.5. deployer Trait

deployer trait 可用于显式选择要部署集成的高级别资源。

此特征位于以下配置集： Kubernetes、Knative、OpenShift。

警告

deployer trait 是一个 平台特征 ：禁用它可能会破坏平台功能。

6.3.5.1. 配置

运行与 CLI 的任何集成时可以指定特征属性：

```
$ kamel run --trait deployer.[key]=[value] --trait deployer.[key2]=[value2] Integration.java
```

可用的配置选项如下：

属性	类型	描述
deployer.enabled	bool	可用于启用或禁用特征。所有特征共享这个通用属性。
deployer.kind	字符串	在为运行的集成创建资源时，允许明确从 deployment, cron-job 或 knative-service 中选择需要的部署类型。

6.3.6. Deployment Trait

Deployment 特征负责生成 Kubernetes 部署，以确保集成在集群中运行。

此特征位于以下配置集： Kubernetes、Knative、OpenShift。

警告

部署特征是一个 平台特征 ：禁用它可能会破坏平台功能。

6.3.6.1. 配置

运行与 CLI 的任何集成时可以指定特征属性：

```
$ kamel run --trait deployment.[key]=[value] Integration.java
```

可用的配置选项如下：

属性	类型	描述
deployment.enabled	bool	可用于启用或禁用特征。所有特征共享这个通用属性。

6.3.7. 环境 Trait

环境特征在内部用于注入集成容器中的标准环境变量，如 NAMESPACE、POD_NAME 等等。

此特征位于以下配置集： Kubernetes、Knative、OpenShift。

警告

environment trait 是一个 平台特征 ：禁用它可能会破坏平台功能。

6.3.7.1. 配置

运行与 CLI 的任何集成时可以指定特征属性：

-

```
$ kamel run --trait environment.[key]=[value] --trait environment.[key2]=[value2]
Integration.java
```

可用的配置选项如下：

属性	类型	描述
environment.enabled	bool	可用于启用或禁用特征。所有特征共享这个通用属性。
environment.container-meta	bool	启用 NAMESPACE 和 POD_NAME 环境变量注入（默认为 true）

6.3.8. 错误处理程序 Trait

error-handler 是一个平台特征，用于将 Error Handler 源注入集成运行时。

此特征位于以下配置集： Kubernetes、Knative、OpenShift。

警告

error-handler trait 是一个 platform trait ：禁用它可能会破坏平台功能。

6.3.8.1. 配置

运行与 CLI 的任何集成时可以指定特征属性：

```
$ kamel run --trait error-handler.[key]=[value] --trait error-handler.[key2]=[value2]
Integration.java
```

可用的配置选项如下：

属性	类型	描述
error-handler.enabled	bool	可用于启用或禁用特征。所有特征共享这个通用属性。
error-handler.ref	字符串	在应用程序属性中提供或找到的错误处理器 ref 名称

6.3.9. JVM Trait

JVM 特征用于配置运行集成的 JVM。

此特征位于以下配置集： Kubernetes、 Knative、 OpenShift。

警告

jvm trait 是一个 平台特征 ：禁用它可能会破坏平台功能。

6.3.9.1. 配置

运行与 CLI 的任何集成时可以指定特征属性：

```
$ kamel run --trait jvm.[key]=[value] --trait jvm.[key2]=[value2] Integration.java
```

可用的配置选项如下：

属性	类型	描述
jvm.enabled	bool	可用于启用或禁用特征。所有特征共享这个通用属性。
jvm.debug	bool	激活远程调试，以便调试器可以附加到 JVM，例如，使用端口转发
jvm.debug-suspend	bool	在加载主类前立即挂起目标 JVM
jvm.print-command	bool	打印命令在容器日志中使用了启动 JVM （默认为 **true**）

属性	类型	描述
jvm.debug-address	**字符串**	要侦听新启动的 JVM 的传输地址（默认为 ***:5005**）
jvm.options	**[]string**	JVM 选项列表
jvm.classpath	**字符串**	其他 JVM 类路径（使用 **Linux** 类路径分隔符）

6.3.9.2. 例子

- **包括一个到 Integration 的额外类路径：**

```
$ kamel run -t jvm.classpath=/path/to/my-dependency.jar:/path/to/another-dependency.jar ...
```

6.3.10. kamelets Trait

kamelets trait 是一个平台特征，用于将 Kamelets 注入集成运行时。

此特征位于以下配置集： Kubernetes、Knative、OpenShift。

警告

kamelets trait 是一个 平台特征 ：禁用它可能会破坏平台功能。

6.3.10.1. 配置

运行与 CLI 的任何集成时可以指定特征属性：

```
$ kamel run --trait kamelets.[key]=[value] --trait kamelets.[key2]=[value2] Integration.java
```

可用的配置选项如下：

属性	类型	描述
kamelets.enabled	bool	可用于启用或禁用特征。所有特征共享这个通用属性。
kamelets.auto	bool	自动注入所有引用的 Kamelets 及其默认配置（默认启用）
kamelets.list	字符串	要加载到当前集成的 Kamelet 名称的逗号分隔列表

6.3.11. nodeAffinity Trait

NodeAffinity 特征允许您通过以下路径限制集成 pod 有资格在其上调度的节点：

- 基于节点上的标签，或具有 pod 间的关联性和反关联性。
- 基于已在节点上运行的 pod 的标签。

这个特征默认为禁用。

此特征位于以下配置集： Kubernetes、 Knative、 OpenShift。

6.3.11.1. 配置

运行与 CLI 的任何集成时可以指定特征属性：

```
$ kamel run --trait affinity.[key]=[value] --trait affinity.[key2]=[value2] Integration.java
```

可用的配置选项如下：

属性	类型	描述
affinity.enabled	bool	可用于启用或禁用特征。所有特征共享这个通用属性。
affinity.pod-affinity	bool	始终在同一节点上并置集成的多个副本（默认为 false）。
affinity.pod-anti-affinity	bool	不得在同一节点上并置集成的多个副本（默认为 false）。

属性	类型	描述
affinity.node-affinity-labels	[]string	根据节点上的标签，定义集成 pod 有资格调度到的一组节点。
affinity.pod-affinity-labels	[]string	定义一组与标签选择器匹配的 pod （相对于给定命名空间）应该与集成 pod 共存。
affinity.pod-anti-affinity-labels	[]string	定义一组与标签选择器匹配的 pod （相对于给定命名空间），该 pod 不应与它们共存。

6.3.11.2. 例子

- 使用 内置节点标签 kubernetes.io/hostname 将集成 pod 调度到特定的节点上：

```
$ kamel run -t affinity.node-affinity-labels="kubernetes.io/hostname in(node-66-50.hosted.k8s.tld)" ...
```

- 要调度每个节点的单一集成 pod （使用 Exists 运算符）：

```
$ kamel run -t affinity.pod-anti-affinity-labels="camel.apache.org/integration" ...
```

- 将集成 pod 与其他集成 pod 并置：

```
$ kamel run -t affinity.pod-affinity-labels="camel.apache.org/integration in(it1, it2)" ...
```

*-labels 选项遵循 标签选择器 中的 要求。它们可以是多评估，然后要求列表是 AND，例如，调度每个节点的一个集成 pod，而与 Camel K 操作器 pod 不共存：

```
$ kamel run -t affinity.pod-anti-affinity-labels="camel.apache.org/integration" -t affinity.pod-anti-affinity-labels="camel.apache.org/component=operator" ...
```

如需更多信息，请参阅有关将 Pod 分配给节点 的官方 Kubernetes 文档。

6.3.12. OpenAPI Trait

OpenAPI DSL 特征在内部用于允许从 OpenAPI specs 创建集成。

此特征位于以下配置集： Kubernetes、 Knative、 OpenShift。

警告

openapi trait 是一个 平台特征 ：禁用它可能会破坏平台功能。

6.3.12.1. 配置

运行与 CLI 的任何集成时可以指定特征属性：

```
$ kamel run --trait openapi.[key]=[value] Integration.java
```

可用的配置选项如下：

属性	类型	描述
openapi.enabled	bool	可用于启用或禁用特征。所有特征共享这个通用属性。

6.3.13. 所有者 Trait

Owner trait 确保所有创建的资源都属于创建的集成，并将集成上的注解和标签传输到这些拥有的资源。

此特征位于以下配置集： Kubernetes、 Knative、 OpenShift。

警告

owner trait 是一个 平台特征 ：禁用它可能会破坏平台功能。

6.3.13.1. 配置

运行与 CLI 的任何集成时可以指定特征属性：

```
$ kamel run --trait owner.[key]=[value] --trait owner.[key2]=[value2] Integration.java
```

可用的配置选项如下：

属性	类型	描述
owner.enabled	bool	可用于启用或禁用特征。所有特征共享这个通用属性。
owner.target-annotations	[]string	要传输的注解集合
owner.target-labels	[]string	要传输的标签集合

6.3.14. 平台 Trait

platform trait 是一个基础特征，用于为集成分配集成平台。

如果缺少平台，则允许特征创建默认平台。此功能在不需要为平台提供自定义配置的情况下特别有用（例如，在 OpenShift 中，默认设置可以正常工作，因为嵌入的容器镜像 registry）。

此特征位于以下配置集： Kubernetes、Knative、OpenShift。

警告

platform trait 是一个 平台特征 ：禁用它可能会破坏平台功能。

6.3.14.1. 配置

运行与 CLI 的任何集成时可以指定特征属性：

-

```
$ kamel run --trait platform.[key]=[value] --trait platform.[key2]=[value2] Integration.java
```

可用的配置选项如下：

属性	类型	描述
platform.enabled	bool	可用于启用或禁用特征。所有特征共享这个通用属性。
platform.create-default	bool	在缺少平台时创建默认（空）平台。
platform.global	bool	指明在全局 operator （默认值 true）中是否应全局创建平台。
platform.auto	bool	如果可以创建默认平台（仅在 OpenShift 上创建它），以自动检测环境。

6.3.15. Quarkus Trait

Quarkus trait 激活 Quarkus 运行时。

它默认启用。

注意

编译到原生可执行文件，即在使用 package-type=native 时，仅支持 kamelets 和 YAML 集成。它还需要至少 4GiB 内存，因此运行原生构建的 Pod 是 Operator Pod，或构建 Pod （取决于为平台配置的构建策略）必须有足够的内存可用。

此特征位于以下配置集： Kubernetes、Knative、OpenShift。

警告

quarkus trait 是一个 平台特征 ：禁用它可能会破坏平台功能。

6.3.15.1. 配置

运行与 CLI 的任何集成时可以指定特征属性：

```
$ kamel run --trait quarkus.[key]=[value] --trait quarkus.[key2]=[value2] integration.java
```

可用的配置选项如下：

属性	类型	描述
quarkus.enabled	bool	可用于启用或禁用特征。所有特征共享这个通用属性。
quarkus.package-type	[]github.com/apache/camel-k/pkg/trait.quarkusPackageType	Quarkus 软件包类型，可以是 fast-jar 或 native （默认 fast-jar）。如果同时指定了 fast-jar 和 native，则创建两个 IntegrationKit 资源，并且 原生 工具包优先于 fast-jar。顺序会影响当前集成工具包的解析。如果不存在与集成匹配的现有工具包，则与第一个软件包类型对应的 kit 将分配给集成。

6.3.15.2. 支持的 Camel 组件

Camel K 只支持作为 Camel Quarkus 扩展开箱即用的 Camel 组件。

6.3.15.3. 例子

6.3.15.3.1. 自动推出到原生集成的部署

虽然编译到原生可执行文件会生成集成，这些集成在运行时启动速度并消耗较少的内存，但构建过程会消耗大量时间，而打包到传统 Java 应用程序的时间要长。

为了组合两个领域的最佳选择，可以将 Quarkus 特征配置为在运行集成时并行运行传统和原生构建，例如：

```
$ kamel run -t quarkus.package-type=fast-jar -t quarkus.package-type=native ...
```

集成 Pod 将在 fast-jar 构建完成后立即运行，并在 原生 构建完成后立即触发对 原生 镜像的推出部署部署，而不会中断服务。

第 7 章 CAMEL K 命令参考

本章提供了 Camel K 命令行界面(CLI)的参考详情，并提供了使用 kamel 命令的示例。本章还提供有关您可以在 Camel K 集成源文件中指定的 Camel K 模式线选项的参考详情，该文件在运行时执行。

本章包括以下部分：

- 第 7.1 节 “Camel K 命令行”
- 第 7.2 节 “Camel K 模式行选项”

7.1. CAMEL K 命令行

Camel K CLI 提供了 kamel 命令，作为在 OpenShift 上运行 Camel K 集成的主要入口点。

7.1.1. 支持的命令

请注意以下键：

符号	描述
✔	支持
▮	不支持或还不支持

表 7.1. kamel 命令

名称	支持	描述	示例
bind	✔	将集成流中的 Kamelets 等 Kubernetes 资源绑定到 Knative 频道、Kafka 主题或其他端点。	**kamel bind telegram-source -p "source.authorizationToken=The Token" channel:mychannel**
completion	▮	生成完成脚本。	**kamel completion bash**

名称	支持	描述	示例
debug	▮	使用本地调试器调试远程集成。	**kamel debug my-integration**
delete	✔	删除 OpenShift 上部署的集成。	**kamel 删除 my-integration**
describe	✔	获取 Camel K 资源的详细信息。这包括 **集成**、**kit** 或 **平台**。	**kamel describe 集成 my-integration**
get	✔	获取部署在 OpenShift 上的集成状态。	**kamel get**
帮助	✔	获取可用命令的完整列表。您可以输入 **--help** 作为每个命令的参数以获取更多详细信息。	• **kamel help** • **kamel run --help**
init	✔	初始化在 Java 或 YAML 中实施的空 Camel K 文件。	**kamel init MyIntegration.java**
install	▮	在 OpenShift 集群上安装 Camel K。 **注：** 建议您使用 OpenShift Camel K Operator 安装和卸载 Camel K。	**kamel install**
kit	▮	配置集成工具包。	**kamel kit create my-integration --secret**
local	▮	在本地执行集成操作，指定一组输入集成文件。	**kamel local run**
log	✔	打印正在运行的集成的日志。	**kamel log my-integration**
promote	✔	您可以将集成从一个命名空间移到另一个命名空间。	**kamel promote**
重新构建	✔	清除一个或多个集成的状态会导致重建。	**kamel 重建 my-integration**

名称	支持	描述	示例
reset	✔	重置当前的 Camel K 安装。	**kamel reset**
run	✔	在 OpenShift 上运行集成。	**kamel run MyIntegration.java**
uninstall	▮	从 OpenShift 集群卸载 Camel K。 **注：** 建议您使用 OpenShift Camel K Operator 安装和卸载 Camel K。	**kamel uninstall**
version	✔	显示 Camel-K 客户端版本。	**kamel 版本**

其他资源

- 请参阅 安装 Camel K

7.2. CAMEL K 模式行选项

您可以使用 Camel K modeline 在 Camel K 集成源文件中输入配置选项，该文件在运行时执行，例如使用 kamel run MyIntegration.java。如需了解更多详细信息，请参阅使用模式行运行 Camel K 集成。

所有可用于 kamel run 命令的选项都可以指定为 modeline 选项。

下表描述了一些最常用的模式行选项。

表 7.2. Camel K 模式行选项

选项	描述
build-property	添加 build-time 属性或 build-time 属性文件。 syntax: **[my-key=my-value|file:/path/to/my-conf.properties]**

选项	描述
config	从 Configmap、Secret 或文件添加运行时配置 Syntax: **[configmap|secret|file]:name[/key]** - **name** 代表本地文件路径或 ConfigMap/Secret 名称。 - **键** （可选）代表要过滤的 ConfigMap/Secret 键。
依赖项	包括一个外部库（例如 Maven 依赖项） 示例： **dependency=mvn:org.my:app:1.0**
env	在集成容器中设置环境变量。例如，**env=MY_ENV_VAR=my-value**。
label	为集成添加标签。例如，**label=my.company=hello**。
name	添加集成名称。例如，**name=my-integration**。
open-api	添加 OpenAPI v2 规格。例如，**open-api=path/to/my-hello-api.json**。
配置集	设置用于部署的 Camel K trait 配置集。例如： **openshift**。
属性	添加 runtime 属性或运行时属性文件。 syntax: **[my-key=my-value|file:/path/to/my-conf.properties]**）
resource	从 ConfigMap、Secret 或文件添加运行时资源 syntax: **[configmap|secret|file]:name[/key][@path]** - **name** 代表本地文件路径或 ConfigMap/Secret 名称 - **键** （可选）代表要过滤的 ConfigMap 或 Secret 键 -**路径** （可选）代表目标路径
遍历	在特征中配置 Camel K 功能或核心功能。例如, **trait=service.enabled=false**。