Red Hat OpenShift Serverless 1.28

Observability（可观察性）

可观察功能，包括管理员和开发人员指标、集群日志记录和追踪。

Last Updated: 2023-09-26

Red Hat OpenShift Serverless 1.28 Observability（可观察性）

可观察功能，包括管理员和开发人员指标、集群日志记录和追踪。

法律通告

摘要

本文档提供了有关如何监控 Knative 服务性能的详细信息。它还详细介绍了如何在 OpenShift Serverless 中使用 OpenShift Logging 和 OpenShift distributed tracing。

目录

第 1 章 管理员指标

1.1. 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 web 控制台 **Administrator** 视角中导航到 **Dashboards** 来查看 OpenShift Serverless 的不同指标。

1.1.1. 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。
- 您可以访问具有集群管理员访问权限的帐户（或 OpenShift Dedicated 或 Red Hat OpenShift Service on AWS 的专用管理员访问权限）。
- 您可以在 web 控制台中访问 **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。

1.2. SERVERLESS 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

指标名称	描述	类型	Tags	单位
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

1.3. 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
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）

指标名称	描述	类型	Tags	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

1.4. 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) 。

1.4.1. 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
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

1.4.2. 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
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

1.4.3. 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）

指标名称	描述	类型	Tags	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

1.4.4. 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
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）

1.5. KNATIVE SERVING 指标

集群管理员可查看 Knative Serving 组件的以下指标。

1.5.1. 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）
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

1.5.2. 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）

指标名称	描述	类型	Tags	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）
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）

指标名称	描述	类型	Tags	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）
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）

1.5.3. 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
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）
go_mspan_in_use	分配的 **mspan** 结构的字节数。	量表	**name**	整数（无单位）

指标名称	描述	类型	Tags	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）

第 2 章 开发人员指标

2.1. SERVERLESS 开发人员指标概述

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 web 控制台的 **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。

2.1.1. OpenShift Container Platform 的其他资源

- 监控概述
- 为用户定义的项目启用监控
- 指定如何监控服务

2.2. KNATIVE 服务指标默认公开

表 2.1. 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"

指标名称、单元和类型	描述	指标标签
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。	
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"

指标名称、单元和类型	描述	指标标签
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 2.2. 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"

指标名称、单元和类型	描述	指标标签
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"
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"

2.3. 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。

2.4. 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。

2.5. 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!
```

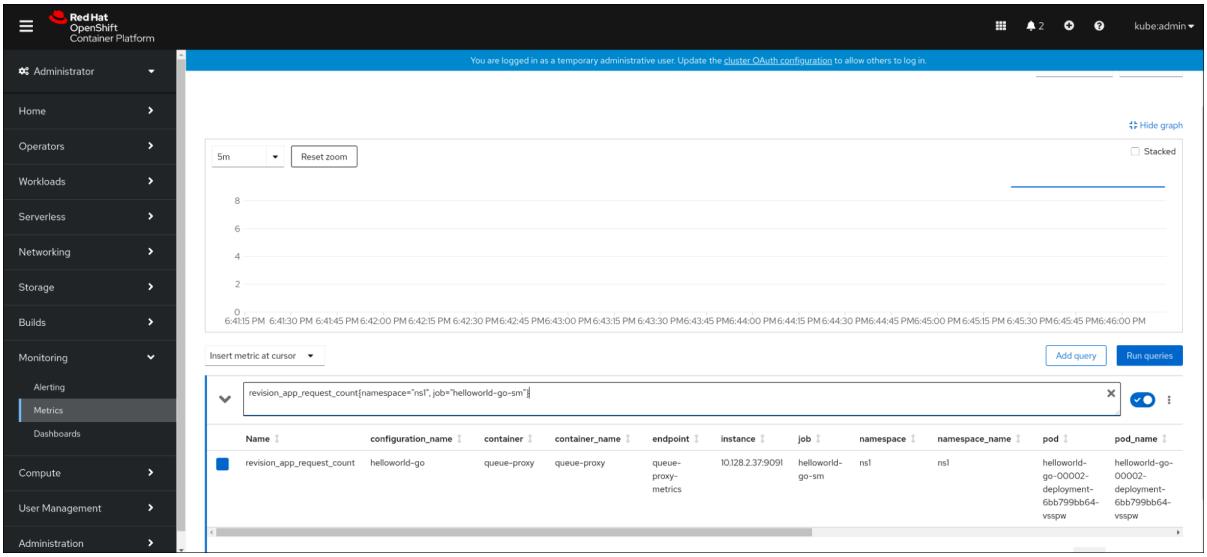
2. 在 Web 控制台中，进入 **Observe → Metrics** 界面。
3. 在输入字段中，输入您要观察到的指标的查询，例如：

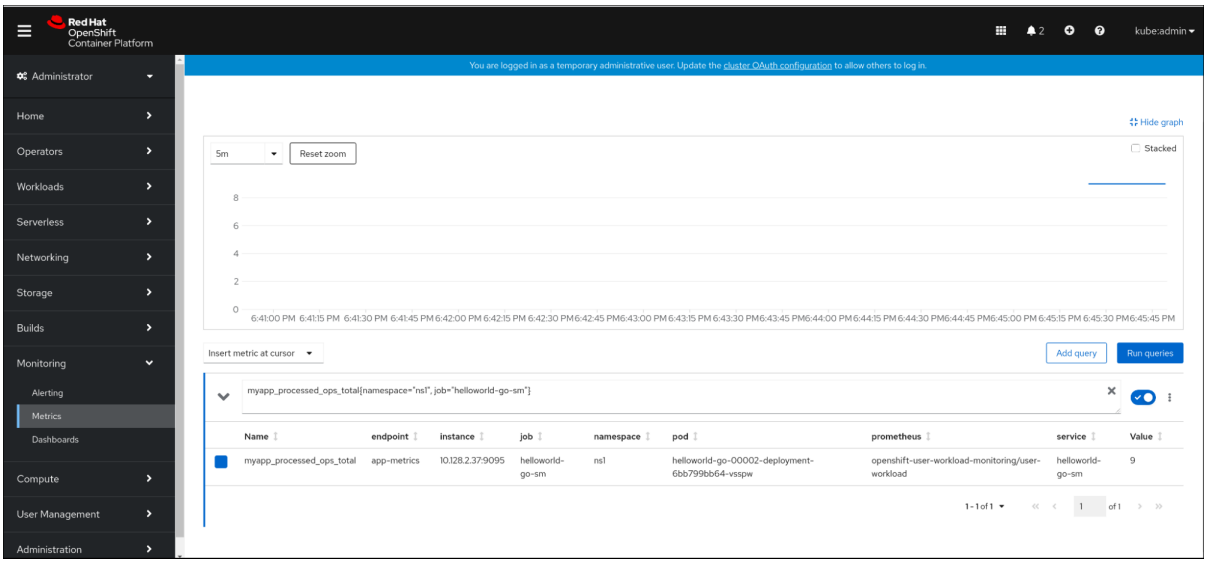
```
revision_app_request_count{namespace="ns1", job="helloworld-go-sm"}
```

另一个示例：

```
myapp_processed_ops_total{namespace="ns1", job="helloworld-go-sm"}
```

4. 观察视觉化的指标：





2.5.1. 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）

指标名称	描述	类型	Tags	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
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）

2.6. 服务指标的仪表板

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。

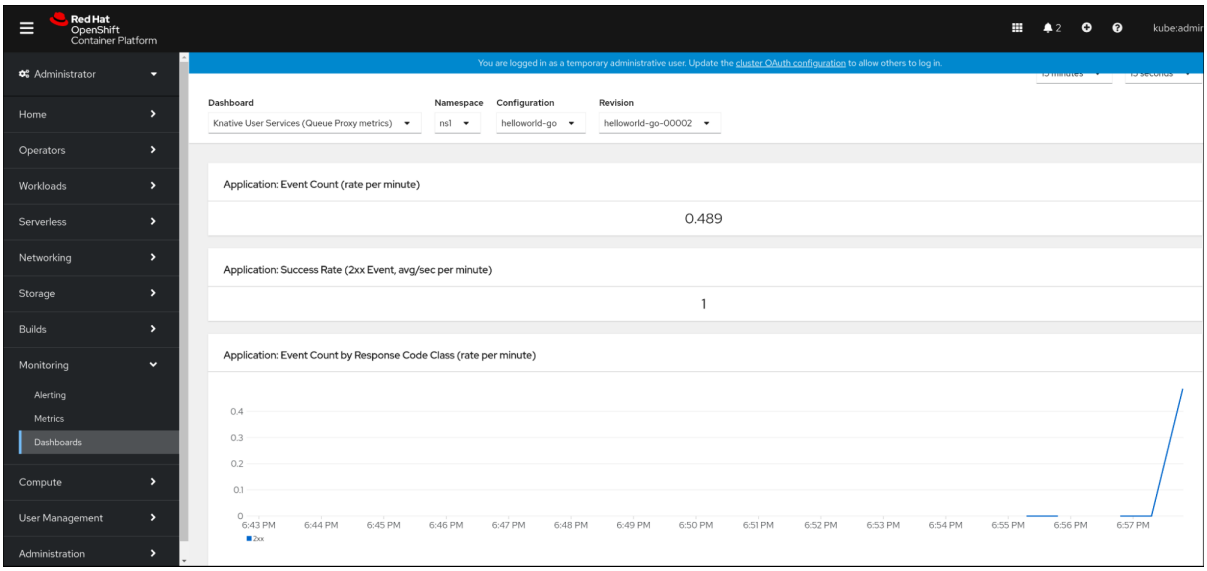
2.6.1. 在仪表板中检查服务的指标

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 **Observe → 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：



第 3 章 集群日志记录

3.1. 在 OPENSHIFT SERVERLESS 中使用 OPENSHIFT LOGGING

3.1.1. 关于为 Red Hat OpenShift 部署日志记录子系统

OpenShift Container Platform 集群管理员可以使用 OpenShift Container Platform Web 控制台或 CLI 部署 logging 子系统，以安装 OpenShift Elasticsearch Operator 和 Red Hat OpenShift Logging Operator。安装 Operator 后，您可以创建一个 **ClusterLogging** 自定义资源 (CR) 来调度 logging 子系统 pod 和支持 logging 子系统所需的其他资源。Operator 负责部署、升级和维护日志记录子系统。

ClusterLogging CR 定义包括日志记录堆栈的所有组件在内的完整日志记录子系统环境，以收集、存储和视觉化日志。Red Hat OpenShift Logging Operator 会监视 logging 子系统 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。

3.1.2. 关于为 Red Hat OpenShift 部署和配置日志记录子系统

Logging 子系统设计为与默认配置一起使用，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日志记录子系统实例并配置日志记录子系统环境。

如果要使用默认日志记录子系统安装，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。下面介绍了在安装 OpenShift Logging 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。

3.1.2.1. 配置和调优日志记录子系统

您可以通过修改 **openshift-logging** 项目中部署的 **ClusterLogging** 自定义资源来配置日志记录子系统。

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
```

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。Red Hat OpenShift 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。

3.1.2.2. 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:
          cpu: 200m
          memory: 16Gi
        requests:
          cpu: 200m
          memory: 16Gi
        storage:
          storageClassName: "gp2"
          size: "200G"
      redundancyPolicy: "SingleRedundancy"
  visualization:
    type: "kibana"
    kibana:
      resources:
        limits:
          memory: 1Gi
        requests:
          cpu: 500m
          memory: 1Gi
      replicas: 1
  collection:
    logs:
      type: "fluentd"
      fluentd:
```

```
    resources:
      limits:
        memory: 1Gi
      requests:
        cpu: 200m
        memory: 1Gi
```

3.2. 查找 KNATIVE SERVING 组件的日志

您可以按照以下流程查找 Knative Serving 组件的日志。

3.2.1. 使用 OpenShift Logging 查找 Knative Serving 组件的日志

先决条件

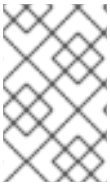
- 安装 OpenShift CLI (**oc**) 。

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 OpenShift Logging 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。

3.3. 查找 KNATIVE SERVING 服务的日志

您可以按照以下流程查找 Knative Serving 服务的日志。

3.3.1. 使用 OpenShift Logging 查找通过 Knative Serving 部署的服务的日志

使用 OpenShift 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。

第 4 章 TRACING

4.1. 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。

4.1.1. 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。

4.1.2. OpenShift Container Platform 的其他资源

- Red Hat OpenShift distributed tracing 架构
- 安装分布式追踪

4.2. 使用 RED HAT OPENSHIFT DISTRIBUTED TRACING

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。

4.2.1. 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。

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、Knative Serving 和 Knative Eventing。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、Knative Serving 和 Knative Eventing。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/v1beta1
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: "false"
      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. 通过创建以下 KnativeEventing CR 来安装 **Knative Eventing** ：

Example KnativeEventing CR

```
apiVersion: operator.knative.dev/v1beta1
kind: KnativeEventing
metadata:
    name: knative-eventing
    namespace: knative-eventing
spec:
  config:
    tracing:
      backend: "zipkin"
      zipkin-endpoint: "http://cluster-collector-collector-headless.tracing-
system.svc:9411/api/v2/spans"
      debug: "false"
      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。

7. 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"
```

8. 向服务发出一些请求：

HTTPS 请求示例

```
$ curl https://helloworld-go.example.com
```

9. 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。

4.3. 使用 JAEGER 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。

4.3.1. 配置 Jaeger 以启用分布式追踪

要使用 Jaeger 启用分布式追踪，您必须安装并配置 Jaeger 作为独立集成。

先决条件

- 在 OpenShift Container Platform 上具有集群管理员权限，或在 Red Hat OpenShift Service on AWS 或 OpenShift Dedicated 上具有集群或专用管理员权限。
- 已安装 OpenShift Serverless Operator、Knative Serving 和 Knative Eventing。
- 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：

Serving 的追踪 YAML 示例

```
apiVersion: operator.knative.dev/v1beta1
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。

3. 通过编辑 KnativeEventing CR 来启用 **Knative Eventing** 的追踪：

Eventing 的追踪 YAML 示例

```
apiVersion: operator.knative.dev/v1beta1
kind: KnativeEventing
metadata:
  name: knative-eventing
  namespace: knative-eventing
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. 将 **backend** 设置为 **zipkin**。
3. 将 **zipkin-endpoint** 指向 **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. 输入以下命令来获取 **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。