OpenShift Container Platform 4.11

Red Hat build of OpenTelemetry

Red Hat build of OpenTelemetry

Last Updated: 2024-03-30

OpenShift Container Platform 4.11 Red Hat build of OpenTelemetry

Red Hat build of OpenTelemetry

法律通告

Copyright © 2024 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at
http://creativecommons.org/licenses/by-sa/3.0/
. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

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

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

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

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

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

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

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

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

All other trademarks are the property of their respective owners.

摘要

使用红帽构建的开源 OpenTelemetry 项目，为云原生软件收集统一、标准化和厂商中立的遥测数据收集。

目录

第 1 章 RED HAT BUILD OF OPENTELEMETRY 发行注记 ... 3
1.1. RED HAT BUILD OF OPENTELEMETRY 概述 3
1.2. 红帽构建的 OPENTELEMETRY 3.0 3
1.3. 获取支持 5
1.4. 使开源包含更多 5

第 2 章 安装红帽构建的 OPENTELEMETRY ... 6
2.1. 从 WEB 控制台安装红帽构建的 OPENTELEMETRY 6
2.2. 使用 CLI 安装红帽构建的 OPENTELEMETRY 7
2.3. 其他资源 10

第 3 章 配置和部署 OPENTELEMETRY 的红帽构建 ... 11
3.1. OPENTELEMETRY COLLECTOR 配置选项 11
3.2. 使用 OPENTELEMETRY COLLECTOR 从不同集群收集可观察性数据 35
3.3. 将指标发送到监控堆栈的配置 40
3.4. 为红帽构建的 OPENTELEMETRY 设置监控 41
3.5. 其他资源 42

第 4 章 配置和部署 OPENTELEMETRY 检测注入 ... 43
4.1. OPENTELEMETRY 检测配置选项 43

第 5 章 使用红帽构建的 OPENTELEMETRY ... 49
5.1. 使用 OPENTELEMETRY COLLECTOR 将 TRACE 转发到 TEMPOSTACK 49
5.2. 将 TRACE 和 METRICS 发送到 OPENTELEMETRY COLLECTOR 51

第 6 章 对红帽构建的 OPENTELEMETRY 进行故障排除 ... 57
6.1. 获取 OPENTELEMETRY COLLECTOR 日志 57
6.2. 公开指标 57
6.3. DEBUG EXPORTER 58

第 7 章 从分布式追踪平台 (JAEGER) 迁移到 OPENTELEMETRY 的红帽构建 ... 59
7.1. 从分布式追踪平台 (JAEGER) 迁移到带有 SIDECAR 的红帽构建的 OPENTELEMETRY 59
7.2. 从分布式追踪平台 (JAEGER) 迁移到没有 SIDECAR 的红帽构建的 OPENTELEMETRY 61

第 8 章 更新红帽构建的 OPENTELEMETRY ... 64
8.1. 其他资源 64

第 9 章 删除红帽构建的 OPENTELEMETRY ... 65
9.1. 使用 WEB 控制台删除 OPENTELEMETRY COLLECTOR 实例 65
9.2. 使用 CLI 删除 OPENTELEMETRY COLLECTOR 实例 65
9.3. 其他资源 66

第 1 章 RED HAT BUILD OF OPENTELEMETRY 发行注记

1.1. RED HAT BUILD OF OPENTELEMETRY 概述

红帽构建的 OpenTelemetry 基于开源 OpenTelemetry 项目，旨在为云原生软件提供统一、标准化和供应商中立的遥测数据收集。Red Hat build of OpenTelemetry 产品支持部署和管理 OpenTelemetry Collector 并简化工作负载检测。

OpenTelemetry Collector 可以接收、处理和转发多种格式的遥测数据，使其成为遥测系统之间的遥测处理和互操作性的理想组件。Collector 提供了一个统一解决方案，用于收集和处理指标、追踪和日志。

OpenTelemetry Collector 有多个功能，包括：

数据收集和处理 Hub

它充当一个中央组件，用于收集来自各种源的指标和追踪等遥测数据。可以从检测的应用程序和基础架构创建这些数据。

可自定义的遥测数据管道

OpenTelemetry Collector 设计为可进行自定义。它支持各种处理器、导出器和接收器。

自动检测功能

自动检测简化了向应用程序添加可观察性的过程。开发人员不需要为基本遥测数据手动检测其代码。

以下是 OpenTelemetry Collector 的一些用例：

集中数据收集

在微服务架构中，可以部署 Collector 来聚合来自多个服务的数据。

数据增强和处理

在将数据转发到分析工具之前，Collector 可以增强、过滤和处理这些数据。

多后端接收和导出

Collector 可以同时接收数据并将其发送到多个监控和分析平台。

1.2. 红帽构建的 OPENTELEMETRY 3.0

红帽构建的 OpenTelemetry 3.0 基于 OpenTelemetry 0.89.0。

1.2.1. 新功能及功能增强

这个版本引进了以下改进：

- **OpenShift distributed tracing data collection Operator** 被重命名为 **红帽构建的 OpenTelemetry Operator**。
- 支持 ARM 架构。
- 支持指标集合的 Prometheus 接收器。
- 支持 Kafka 接收器和导出器，将 trace 和 metrics 发送到 Kafka。
- 支持集群范围的代理环境。
- 如果启用了 Prometheus exporter，Red Hat build of OpenTelemetry Operator 会创建 Prometheus **ServiceMonitor** 自定义资源。

- Operator 启用 **Instrumentation** 自定义资源，允许注入上游 OpenTelemetry 自动检测库。

1.2.2. 删除通知

- 在红帽构建的 OpenTelemetry 3.0 中，Jaeger exporter 已被删除。程序错误修复和支持仅在 2.9 生命周期结束时提供。作为将数据发送到 Jaeger 收集器的 Jaeger exporter 的替代选择，您可以使用 OTLP exporter。

1.2.3. 程序错误修复

在这个版本中引进了以下程序错误修复：

- 修复了在使用 **oc adm catalog mirror** CLI 命令时对断开连接的环境的支持。

1.2.4. 已知问题

因此，因为一个程序错误(TRACING-3761)，红帽构建的 OpenTelemetry Operator 的集群监控会被禁用。这个程序错误可防止集群监控因为集群监控和服务监控对象缺少标签 **openshift.io/cluster-monitoring=true**，所以从红帽构建的 OpenTelemetry Operator 中提取指标。

临时解决方案

您可以启用集群监控，如下所示：

1. 在 Operator 命名空间中添加以下标签：**oc label namespace openshift-opentelemetry-operator openshift.io/cluster-monitoring=true**

2. 创建服务监控器、角色和角色绑定：

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: opentelemetry-operator-controller-manager-metrics-service
  namespace: openshift-opentelemetry-operator
spec:
  endpoints:
  - bearerTokenFile: /var/run/secrets/kubernetes.io/serviceaccount/token
    path: /metrics
    port: https
    scheme: https
    tlsConfig:
      insecureSkipVerify: true
  selector:
    matchLabels:
      app.kubernetes.io/name: opentelemetry-operator
      control-plane: controller-manager
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: otel-operator-prometheus
  namespace: openshift-opentelemetry-operator
  annotations:
    include.release.openshift.io/self-managed-high-availability: "true"
    include.release.openshift.io/single-node-developer: "true"
rules:
```

```yaml
- apiGroups:
  - ""
  resources:
  - services
  - endpoints
  - pods
  verbs:
  - get
  - list
  - watch
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: otel-operator-prometheus
  namespace: openshift-opentelemetry-operator
  annotations:
    include.release.openshift.io/self-managed-high-availability: "true"
    include.release.openshift.io/single-node-developer: "true"
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: otel-operator-prometheus
subjects:
- kind: ServiceAccount
  name: prometheus-k8s
  namespace: openshift-monitoring
```

1.3. 获取支持

如果您在执行本文档所述的某个流程或 OpenShift Container Platform 时遇到问题，请访问 红帽客户门户网站。通过红帽客户门户网站：

- 搜索或者浏览红帽知识库，了解与红帽产品相关的文章和解决方案。
- 提交问题单给红帽支持。
- 访问其他产品文档。

要识别集群中的问题，您可以在 OpenShift Cluster Manager Hybrid Cloud Console 中使用 Insights。Insights 提供了问题的详细信息，并在有可用的情况下，提供了如何解决问题的信息。

如果您对本文档有任何改进建议，或发现了任何错误，请为相关文档组件提交 JIRA 问题。请提供具体详情，如章节名称和 OpenShift Container Platform 版本。

1.4. 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。详情请查看 CTO Chris Wright 的信息。

第 2 章 安装红帽构建的 OPENTELEMETRY

安装红帽构建的 OpenTelemetry 涉及以下步骤：

1. 安装红帽构建的 OpenTelemetry Operator。
2. 为 OpenTelemetry Collector 实例创建命名空间。
3. 创建 **OpenTelemetryCollector** 自定义资源来部署 OpenTelemetry Collector 实例。

2.1. 从 WEB 控制台安装红帽构建的 OPENTELEMETRY

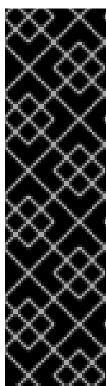
您可以从 web 控制台的 **Administrator** 视图安装红帽构建的 OpenTelemetry。

先决条件

- 以集群管理员身份使用 **cluster-admin** 角色登录到 web 控制台。
- 对于 Red Hat OpenShift Dedicated，您必须使用具有 **dedicated-admin** 角色的帐户登录。

流程

1. 安装红帽构建的 OpenTelemetry Operator：
   a. 进入 **Operators** → **OperatorHub**，搜索 **红帽构建的 OpenTelemetry Operator**。
   b. 选择 **Red Hat build of OpenTelemetry Operator**，**provided by Red Hat** → **Install** → **Install** → **View Operator**.



重要

这会使用默认预设置来安装 Operator：

- **Update channel** → **stable**
- **Installation mode** → **All namespaces on the cluster**
- **Installed Namespace** → **openshift-operators**
- **Update approval** → **Automatic**

   c. 在安装的 Operator 页面的 **Details** 选项卡中，在 **ClusterServiceVersion details** 下验证安装 **Status** 是否为 **Succeeded**。
2. 通过转至 **Home** → **Projects** → **Create Project**，为您在下一步中创建的 OpenTelemetry Collector 实例创建一个项目。
3. 创建 **OpenTelemetry Collector** 实例。
   a. 进入 **Operators** → **Installed Operators**。
   b. 选择 **OpenTelemetry Collector** → **Create OpenTelemetry Collector** → **YAML view**。
   c. 在 **YAML 视图中**，使用 OTLP、Jaeger、Zipkin receivers 和 debug exporter 自定义 **OpenTelemetryCollector** 自定义资源(CR)。

–

```
apiVersion: opentelemetry.io/v1alpha1
kind: OpenTelemetryCollector
metadata:
  name: otel
  namespace: <project_of_opentelemetry_collector_instance>
spec:
  mode: deployment
  config: |
    receivers:
      otlp:
        protocols:
          grpc:
          http:
      jaeger:
        protocols:
          grpc:
          thrift_binary:
          thrift_compact:
          thrift_http:
      zipkin:
    processors:
      batch:
      memory_limiter:
        check_interval: 1s
        limit_percentage: 50
        spike_limit_percentage: 30
    exporters:
      debug:
    service:
      pipelines:
        traces:
          receivers: [otlp,jaeger,zipkin]
          processors: [memory_limiter,batch]
          exporters: [debug]
```

d. 选择 **Create**。

验证

1. 使用 **Project:** 下拉列表选择 **OpenTelemetry Collector** 实例的项目。
2. 进入 **Operators** → **Installed Operators**，以验证 **OpenTelemetry Collector** 实例的 **Status** 是否为 **Condition: Ready**。
3. 进入 **Workloads** → **Pods**，以验证 **OpenTelemetry Collector** 实例的所有组件 pod 都在运行。

2.2. 使用 CLI 安装红帽构建的 OPENTELEMETRY

您可以从命令行安装红帽构建的 OpenTelemetry。

先决条件

- 集群管理员具有 **cluster-admin** 角色的活跃 OpenShift CLI (**oc**) 会话。

提示

- 确保您的 OpenShift CLI (**oc**) 版本为最新版本，并与您的 OpenShift Container Platform 版本匹配。
- 运行 **oc login**:

```
$ oc login --username=<your_username>
```

流程

1. 安装红帽构建的 OpenTelemetry Operator：
   a. 运行以下命令，为红帽构建的 OpenTelemetry Operator 创建项目：

```
$ oc apply -f - << EOF
apiVersion: project.openshift.io/v1
kind: Project
metadata:
  labels:
    kubernetes.io/metadata.name: openshift-opentelemetry-operator
    openshift.io/cluster-monitoring: "true"
  name: openshift-opentelemetry-operator
EOF
```

   b. 运行以下命令来创建 Operator 组：

```
$ oc apply -f - << EOF
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: openshift-opentelemetry-operator
  namespace: openshift-opentelemetry-operator
spec:
  upgradeStrategy: Default
EOF
```

   c. 运行以下命令来创建订阅：

```
$ oc apply -f - << EOF
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: opentelemetry-product
  namespace: openshift-opentelemetry-operator
spec:
  channel: stable
  installPlanApproval: Automatic
  name: opentelemetry-product
  source: redhat-operators
  sourceNamespace: openshift-marketplace
EOF
```

   d. 运行以下命令检查 Operator 状态：

```
$ oc get csv -n openshift-opentelemetry-operator
```

2. 为您要在后续步骤中创建的 OpenTelemetry Collector 实例创建一个您选择的项目：

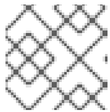
   - 要创建没有元数据的项目，请运行以下命令：

   ```
   $ oc new-project <project_of_opentelemetry_collector_instance>
   ```

   - 要使用元数据创建项目，请运行以下命令：

   ```
   $ oc apply -f - << EOF
   apiVersion: project.openshift.io/v1
   kind: Project
   metadata:
     name: <project_of_opentelemetry_collector_instance>
   EOF
   ```

3. 在为您创建的项目中创建一个 OpenTelemetry Collector 实例。



注意

您可以在同一集群中的独立项目中创建多个 OpenTelemetry Collector 实例。

   a. 使用 OTLP、Jaeger 和 Zipkin receivers 和 debug exporter 自定义 **OpenTelemetry Collector** 自定义资源 (CR)：

   ```
   apiVersion: opentelemetry.io/v1alpha1
   kind: OpenTelemetryCollector
   metadata:
     name: otel
     namespace: <project_of_opentelemetry_collector_instance>
   spec:
     mode: deployment
     config: |
       receivers:
         otlp:
           protocols:
             grpc:
             http:
         jaeger:
           protocols:
             grpc:
             thrift_binary:
             thrift_compact:
             thrift_http:
         zipkin:
       processors:
         batch:
         memory_limiter:
           check_interval: 1s
           limit_percentage: 50
           spike_limit_percentage: 30
       exporters:
         debug:
   ```

```
service:
  pipelines:
    traces:
      receivers: [otlp,jaeger,zipkin]
      processors: [memory_limiter,batch]
      exporters: [debug]
```

b. 运行以下命令来应用自定义 CR：

```
$ oc apply -f - << EOF
<OpenTelemetryCollector_custom_resource>
EOF
```

验证

1. 运行以下命令，验证 OpenTelemetry Collector pod 的 **status.phase** 是否为 **Running**，**条件**为 **type: Ready** ：

```
$ oc get pod -l app.kubernetes.io/managed-by=opentelemetry-operator,app.kubernetes.io/instance=<namespace>.<instance_name> -o yaml
```

2. 运行以下命令来获取 OpenTelemetry Collector 服务：

```
$ oc get service -l app.kubernetes.io/managed-by=opentelemetry-operator,app.kubernetes.io/instance=<namespace>.<instance_name>
```

2.3. 其他资源

- 创建集群管理员
- OperatorHub.io
- 访问Web控制台
- 使用 Web 控制台从 OperatorHub 安装
- 从已安装的 Operator 创建应用程序
- OpenShift CLI 入门

第 3 章 配置和部署 OPENTELEMETRY 的红帽构建

红帽构建的 OpenTelemetry Operator 使用自定义资源定义(CRD)文件来定义创建和部署分布式追踪平台 (Tempo) 资源时要使用的架构和配置设置。您可以安装默认配置或修改该文件。

3.1. OPENTELEMETRY COLLECTOR 配置选项

OpenTelemetry Collector 由访问遥测数据的一组组件组成：

Receivers

可以推送或拉取的接收器（基于推送或拉取）是数据如何进入 Collector 中。通常，接收器接受指定格式的数据，将其转换为内部格式，并将其传递给适用管道中定义的处理器和导出器。默认情况下，不会配置接收器。必须配置一个或多个接收器。接收器可以支持一个或多个数据源。

Processors

可选。处理器处理接收和导出的数据。默认情况下，不启用处理器。每个数据源都必须启用处理器。不是所有处理器都支持所有数据源。根据数据源，可能会启用多个处理器。请注意，处理器的顺序很重要。

Exporters

可以推送或拉取的导出器是如何将数据发送到一个或多个后端或目的地。默认情况下，不会配置导出器。必须配置一个或多个导出器。导出器可以支持一个或多个数据源。导出器可能会与其默认设置一起使用，但许多导出器需要配置来至少指定目标和安全设置。

连接器

连接器连接两个管道。它在一个管道的末尾将数据视为导出器，并在另一个管道开始时将数据作为接收器发送。它可以消耗和发送相同或不同数据类型的数据。它可以生成并发送数据以汇总已消耗的数据，或者可以完全复制或路由数据。

扩展

扩展为 Collector 添加了功能。例如，身份验证可以自动添加到接收器和导出器中。

您可以在自定义资源 YAML 文件中定义多个组件实例。配置后，必须通过 YAML 文件的 **spec.config.service** 部分中定义的管道启用这些组件。作为最佳实践，仅启用您需要的组件。

OpenTelemetry Collector 自定义资源文件示例

```
apiVersion: opentelemetry.io/v1alpha1
kind: OpenTelemetryCollector
metadata:
  name: cluster-collector
  namespace: tracing-system
spec:
  mode: deployment
  observability:
    metrics:
      enableMetrics: true
  config: |
    receivers:
      otlp:
        protocols:
          grpc:
          http:
    processors:
    exporters:
      otlp:
```

```
      endpoint: jaeger-production-collector-headless.tracing-system.svc:4317
      tls:
        ca_file: "/var/run/secrets/kubernetes.io/serviceaccount/service-ca.crt"
    prometheus:
      endpoint: 0.0.0.0:8889
      resource_to_telemetry_conversion:
        enabled: true # by default resource attributes are dropped
  service: (1)
    pipelines:
      traces:
        receivers: [otlp]
        processors: []
        exporters: [jaeger]
      metrics:
        receivers: [otlp]
        processors: []
        exporters: [prometheus]
```

1. 如果一个组件被配置但没有在 **service** 部分中定义，则组件不会被启用。

表 3.1. Operator 用来定义 OpenTelemetry Collector 的参数

参数	描述	值	default
receivers:	接收器用于控制数据如何进入 Collector。默认情况下，不会配置接收器。必须至少有一个启用的接收器才能使配置被视为有效。接收器通过添加到管道中来启用。	**otlp**, **jaeger**, **prometheus**, **zipkin**, **kafka**, **opencensus**	None
processors:	处理器通过数据接收和导出。默认情况下，不启用处理器。	**batch**, **memory_limiter**, **resourcedetection**, **attributes**, **span**, **k8sattributes**, **filter**, **routing**	None
exporters:	导出器将数据发送到一个或多个后端或目的地。默认情况下，不会配置导出器。必须至少启用了一个 exporter 时，配置才被视为有效。将导出器添加到管道中即可启用。导出器可能会与其默认设置一起使用，但很多需要配置至少指定目标和安全设置。	**otlp**, **otlphttp**, **debug**, **prometheus**, **kafka**	None

参数	描述	值	default
connectors:	连接器加入管道对，这通过使用数据作为管道导出器，并将数据作为管道接收器发送，并可用于总结、复制或路由消耗的数据。	**spanmetrics**	None
extensions:	不涉及处理遥测数据的任务的可选组件。	**bearertokenauth**, **oauth2client**, **jaegerremotesampling**, **pprof**, **health_check**, **memory_ballast**, **zpages**	None
service: pipelines:	组件通过将组件添加到 **services.pipeline** 下的管道中来启用。		
service: pipelines: traces: receivers:	您可以通过在 **service.pipelines.traces** 下添加用于追踪的接收器。		无
service: pipelines: traces: processors:	您可以通过在 **service.pipelines.traces** 下添加处理器来启用追踪的处理器。		无
service: pipelines: traces: exporters:	您可以通过在 **service.pipelines.traces** 下添加用于追踪的导出器。		None
service: pipelines: metrics: receivers:	您可以通过在 **service.pipelines.metrics** 下添加指标来启用指标接收器。		None

参数	描述	值	default

service:
  pipelines:
    metrics:
      processors:
	您可以通过在 **service.pipelines.metrics** 下添加 services.pipelines.metrics 来为 metircs 启用处理器。		None

service:
  pipelines:
    metrics:
      exporters:
	您可以通过在 **service.pipelines.metrics** 下添加指标启用导出器。		None

3.1.1. OpenTelemetry Collector 组件

3.1.1.1. Receivers

接收器将数据放入 Collector 中。

3.1.1.1.1. OTLP Receiver

OTLP 接收器使用 OpenTelemetry 协议 (OTLP) ingests trace 和 metrics。

OpenTelemetry Collector 自定义资源带有启用的 OTLP 接收器

```
config: |
  receivers:
    otlp:
      protocols:
        grpc:
          endpoint: 0.0.0.0:4317 (1)
          tls: (2)
            ca_file: ca.pem
            cert_file: cert.pem
            key_file: key.pem
            client_ca_file: client.pem (3)
            reload_interval: 1h (4)
        http:
          endpoint: 0.0.0.0:4318 (5)
          tls: (6)

  service:
    pipelines:
      traces:
        receivers: [otlp]
      metrics:
        receivers: [otlp]
```

1. OTLP gRPC 端点。如果省略，则使用默认的 **0.0.0.0:4317**。

2. 服务器端 TLS 配置。定义 TLS 证书的路径。如果省略，则禁用 TLS。
3. 服务器验证客户端证书的 TLS 证书的路径。这会将 **TLSConfig** 中的 **ClientCAs** 和 **ClientAuth** 的值设置为 **RequireAndVerifyClientCert**。如需更多信息，请参阅 Golang TLS 软件包的配置。
4. 指定重新载入证书的时间间隔。如果没有设置值，则证书永远不会重新加载。**reload_interval** 接受包含有效时间单位的字符串，如 **ns**、**us**（或 unmarshal**s**）、**ms**、**s**、**m**、**h**。
5. OTLP HTTP 端点。默认值为 **0.0.0.0:4318**。
6. 服务器端 TLS 配置。如需更多信息，请参阅 **grpc** 协议配置部分。

3.1.1.1.2. Jaeger Receiver

Jaeger 接收器以 Jaeger 格式计算 trace。

OpenTelemetry Collector 自定义资源，启用了 Jaeger 接收器

```
config: |
  receivers:
    jaeger:
      protocols:
        grpc:
          endpoint: 0.0.0.0:14250 # 1
        thrift_http:
          endpoint: 0.0.0.0:14268 # 2
        thrift_compact:
          endpoint: 0.0.0.0:6831 # 3
        thrift_binary:
          endpoint: 0.0.0.0:6832 # 4
        tls: # 5

  service:
    pipelines:
      traces:
        receivers: [jaeger]
```

1. Jaeger gRPC 端点。如果省略，则使用默认的 **0.0.0.0:14250**。
2. Jaeger Thrift HTTP 端点。如果省略，则使用默认的 **0.0.0.0:14268**。
3. Jaeger Thrift Compact 端点。如果省略，则使用默认的 **0.0.0.0:6831**。
4. Jaeger Thrift Binary 端点。如果省略，则使用默认的 **0.0.0.0:6832**。
5. 服务器端 TLS 配置。详情请查看 OTLP 接收器配置部分。

3.1.1.1.3. Prometheus Receiver

Prometheus 接收器目前只是一个技术预览功能。

Prometheus 接收器提取指标端点。

OpenTelemetry Collector 自定义资源带有启用 Prometheus 接收器

```
config: |
  receivers:
    prometheus:
      config:
        scrape_configs: (1)
          - job_name: 'my-app' (2)
            scrape_interval: 5s (3)
            static_configs:
              - targets: ['my-app.example.svc.cluster.local:8888'] (4)
  service:
    pipelines:
      metrics:
        receivers: [prometheus]
```

1. 使用 Prometheus 格式提取配置。
2. Prometheus 作业名称。
3. 提取指标数据的 Interval。接受时间单位。默认值为 **1m**。
4. 公开指标的目标。本例从 **example** 项目中的 **my-app** 应用程序中提取指标。

3.1.1.1.4. Zipkin Receiver

Zipkin 接收器使用 Zipkin v1 和 v2 格式的 ingests trace。

OpenTelemetry Collector 自定义资源带有启用 Zipkin 接收器

```
config: |
  receivers:
    zipkin:
      endpoint: 0.0.0.0:9411 (1)
      tls: (2)

  service:
    pipelines:
      traces:
        receivers: [zipkin]
```

1. Zipkin HTTP 端点。如果省略，则使用默认的 **0.0.0.0:9411**。
2. 服务器端 TLS 配置。详情请查看 OTLP 接收器配置部分。

3.1.1.1.5. Kafka Receiver

Kafka 接收器目前只是一个技术预览功能。

Kafka 接收器以 OTLP 格式接收 Kafka 中的 trace、metrics 和日志。

OpenTelemetry Collector 自定义资源带有启用 Kafka 接收器

```
config: |
  receivers:
    kafka:
      brokers: ["localhost:9092"] (1)
      protocol_version: 2.0.0 (2)
      topic: otlp_spans (3)
      auth:
        plain_text: (4)
          username: example
          password: example
        tls: (5)
          ca_file: ca.pem
          cert_file: cert.pem
          key_file: key.pem
          insecure: false (6)
          server_name_override: kafka.example.corp (7)
  service:
    pipelines:
      traces:
        receivers: [kafka]
```

1. Kafka 代理列表。默认值为 **localhost:9092**。
2. Kafka 协议版本。例如，**2.0.0**。这个为必填字段。
3. 要从中读取的 Kafka 主题的名称。默认值为 **otlp_spans**。
4. 纯文本身份验证配置。如果省略，则禁用纯文本身份验证。
5. 客户端 TLS 配置。定义 TLS 证书的路径。如果省略，则禁用 TLS 身份验证。
6. 禁用验证服务器的证书链和主机名。默认值为 **false**。
7. ServerName 表示客户端请求的服务器名称，以支持虚拟主机。

3.1.1.1.6. OpenCensus 接收器

OpenCensus 接收器提供与 OpenCensus 项目的向后兼容性，以便更轻松地迁移检测代码库。它通过 gRPC 或 HTTP 和 Json 以 OpenCensus 格式接收指标和跟踪。

OpenTelemetry Collector 自定义资源，启用了 OpenCensus 接收器

```
config: |
  receivers:
    opencensus:
      endpoint: 0.0.0.0:9411 (1)
      tls: (2)
      cors_allowed_origins: (3)
        - https://*.<example>.com
  service:
    pipelines:
```

```
        traces:
          receivers: [opencensus]
          ...
```

1. OpenCensus 端点。如果省略，则默认为 **0.0.0.0:55678**。
2. 服务器端 TLS 配置。详情请查看 OTLP 接收器配置部分。
3. 您还可以使用 HTTP JSON 端点选择性地配置 CORS，这通过在此字段中指定允许的 CORS 来源列表来启用。**cors_allowed_origins** 下接受带有 * 的通配符。要匹配任何源，请只输入 *。

3.1.1.2. Processors

处理器通过数据接收和导出。

3.1.1.2.1. 批处理处理器

批处理（Batch）处理器批处理跟踪和指标，以减少传输遥测信息所需的传出连接数量。

使用 Batch 处理器时 OpenTelemetry Collector 自定义资源示例

```
    config: |
      processor:
        batch:
          timeout: 5s
          send_batch_max_size: 10000
      service:
        pipelines:
          traces:
            processors: [batch]
          metrics:
            processors: [batch]
```

表 3.2. Batch 处理器使用的参数

参数	描述	default
timeout	将批处理发送到特定的持续时间，无论批处理大小如何。	**200ms**
send_batch_size	在指定数量的 span 或 metrics 后发送遥测数据的批处理。	**8192**
send_batch_max_size	批处理的最大允许大小。必须等于或大于 **send_batch_size**。	**0**
metadata_keys	激活后，会为在 **client.Metadata** 中找到的每个唯一值集创建一个批处理器实例。	**[]**

参数	描述	default
metadata_cardinality_limit	在填充 **metadata_keys** 时，此配置限制了在进程期间处理的不同元数据键值组合的数量。	**1000**

3.1.1.2.2. Memory Limiter 处理器

Memory Limiter 处理器定期检查 Collector 的内存用量，并在达到软内存限制时暂停数据处理。这个处理器支持 trace、metrics 和 logs。前面的组件（通常是接收器）应该重试发送同一数据，并可能对传入的数据应用回溯。当内存用量超过硬限制时，Memory Limiter 处理器会强制运行垃圾回收操作。

使用 Memory Limiter 处理器时 OpenTelemetry Collector 自定义资源示例

```
config: |
  processor:
    memory_limiter:
      check_interval: 1s
      limit_mib: 4000
      spike_limit_mib: 800
  service:
    pipelines:
      traces:
        processors: [batch]
      metrics:
        processors: [batch]
```

表 3.3. Memory Limiter 处理器使用的参数

参数	描述	default
check_interval	内存用量测量之间的时间。最佳值为 **1s**。对于 spiky 流量模式，您可以减少 **check_interval** 或增加 **spike_limit_mib**。	**0s**
limit_mib	硬限制，即堆上分配的最大内存量（以 MiB 为单位）。通常，OpenTelemetry Collector 的内存用量大约比这个值高 50 MiB。	**0**
spike_limit_mib	spike 限制，这是 MiB 中内存使用率最大激增。最佳值为 **limit_mib** 的 20%。要计算软限制，请从 **limit_mib** 中减去 **spike_limit_mib**。	**limit_mib** 的 20%

参数	描述	default
limit_percentage	与 **limit_mib** 相同，但以总可用内存的百分比表示。**limit_mib** 设置优先于此设置。	**0**
spike_limit_percentage	与 **spike_limit_mib** 相同，但以总可用内存的百分比表示。旨在与 **limit_percentage** 设置一起使用。	**0**

3.1.1.2.3. 资源检测处理器

资源检测处理器目前只是一个技术预览功能。

资源检测处理器识别主机资源详情与 OpenTelemetry 的资源语义标准保持一致。使用检测到的信息，它可以添加或替换遥测数据中的资源值。此处理器支持 trace、metrics，并可与多个检测器一起使用，如 Docket 元数据检测器或 **OTEL_RESOURCE_ATTRIBUTES** 环境变量检测器。

OpenShift Container Platform 权限用于资源检测处理器

```
kind: ClusterRole
metadata:
  name: otel-collector
rules:
- apiGroups: ["config.openshift.io"]
  resources: ["infrastructures", "infrastructures/status"]
  verbs: ["get", "watch", "list"]
```

OpenTelemetry Collector 使用资源检测处理器

```
config: |
  processor:
    resourcedetection:
      detectors: [openshift]
      override: true
  service:
    pipelines:
      traces:
        processors: [resourcedetection]
      metrics:
        processors: [resourcedetection]
```

OpenTelemetry Collector 使用带有环境变量检测器的资源检测器

```
config: |
  processors:
    resourcedetection/env:
```

```
      detectors: [env] (1)
      timeout: 2s
      override: false
```

(1) 指定要使用的检测器。在本例中，指定了环境检测器。

3.1.1.2.4. 属性处理器

属性处理器目前只是一个技术预览功能。

Attributes 处理器可以修改 span, log, 或 metric 的属性。您可以配置此处理器来过滤和匹配输入数据，并为特定操作包含或排除此类数据。

处理器对操作列表进行操作，按配置中指定的顺序执行它们。支持以下操作：

insert
: 当指定的键尚不存在时，将新属性插入到输入数据中。

Update（更新）
: 如果密钥已存在，更新输入数据中的属性。

Upsert
: 合并 insert 和 update 操作：如果键尚不存在，则插入新属性。如果密钥已存在，则更新 属性。

删除
: 从输入数据中删除属性。

Hash
: 将现有属性值哈希为 SHA1。

extract
: 通过使用输入键的正则表达式规则将值提取到规则中定义的目标键。如果目标键已存在，它将被像使用现有属性作为源的 Span 处理器 **to_attributes** 设置一样覆盖。

Convert
: 将现有属性转换为指定类型。

OpenTelemetry Collector 使用属性处理器

```
config: |
  processors:
    attributes/example:
      actions:
        - key: db.table
          action: delete
        - key: redacted_span
          value: true
          action: upsert
        - key: copy_key
          from_attribute: key_original
          action: update
        - key: account_id
          value: 2245
          action: insert
        - key: account_password
          action: delete
```

```
      - key: account_email
        action: hash
      - key: http.status_code
        action: convert
        converted_type: int
```

3.1.1.2.5. 资源处理器

资源处理器目前只是一个技术预览功能。

资源处理器对资源属性应用更改。这个处理器支持 trace、metrics 和 logs。

OpenTelemetry Collector 使用资源检测处理器

```
config: |
  processor:
    attributes:
    - key: cloud.availability_zone
      value: "zone-1"
      action: upsert
    - key: k8s.cluster.name
      from_attribute: k8s-cluster
      action: insert
    - key: redundant-attribute
      action: delete
```

属性代表应用到资源属性的操作，如删除属性、插入属性或 upsert 属性。

3.1.1.2.6. span 处理器

Span 处理器目前只是一个技术预览功能。

Span 处理器根据其属性修改 span 名称，或者从 span 名称中提取 span 属性。它还可以更改 span 状态。它还可以包含或排除 span。这个处理器支持 trace。

span rename 需要使用 **from_attributes** 配置为新名称指定属性。

OpenTelemetry Collector 使用 Span 处理器重命名范围

```
config: |
  processor:
    span:
      name:
        from_attributes: [<key1>, <key2>, ...] (1)
        separator: <value> (2)
```

(1) 定义组成新 span 名称的密钥。

(2) 可选分隔符。

您可以使用处理器从 span 名称中提取属性。

OpenTelemetry Collector 使用 Span 处理器从范围名称中提取属性

```
config: |
  processor:
    span/to_attributes:
      name:
        to_attributes:
          rules:
            - ^\/api\/v1\/document\/(?P<documentId>.*)\/update$ (1)
```

(1) 此规则定义如何执行提取。您可以定义更多规则：例如，如果正则表达式与名称匹配，则会创建一个 **documentID** 属性。在本例中，如果输入 span 名称是 **/api/v1/document/12345678/update**，则会产生 **/api/v1/document/{documentId}/update** 输出 span 名称，并且新的 **"documentId"="12345678"** 属性被添加到 span 中。

您可以修改 span 状态。

OpenTelemetry Collector 使用 Span Processor 进行状态更改

```
config: |
  processor:
    span/set_status:
      status:
        code: Error
        description: "<error_description>"
```

3.1.1.2.7. Kubernetes 属性处理器

Kubernetes 属性处理器目前只是一个技术预览功能。

Kubernetes 属性处理器使用 Kubernetes 元数据启用 span、metrics 和 log 资源属性的自动配置。这个处理器支持 trace、metrics 和 logs。此处理器自动识别 Kubernetes 资源，从它们中提取元数据，并将此提取的元数据作为资源属性合并到相关的 span、metrics 和 logs 中。它使用 Kubernetes API 来发现在集群内运行的所有 pod，维护其 IP 地址、pod UID 和其他相关元数据的记录。

Kubernetes 属性处理器所需的最小 OpenShift Container Platform 权限

```
kind: ClusterRole
metadata:
  name: otel-collector
rules:
  - apiGroups: ['']
    resources: ['pods', 'namespaces']
    verbs: ['get', 'watch', 'list']
```

OpenTelemetry Collector 使用 Kubernetes 属性处理器

```
config: |
  processors:
      k8sattributes:
          filter:
              node_from_env_var: KUBE_NODE_NAME
```

3.1.1.3. 过滤处理器

Filter 处理器目前只是一个技术预览功能。

Filter 处理器利用 OpenTelemetry Transformation Language 来建立丢弃遥测数据的条件。如果满足这些条件，遥测数据将被丢弃。可以使用逻辑 OR 运算符来组合条件。这个处理器支持 trace、metrics 和 logs。

OpenTelemetry Collector 自定义资源，启用了 OTLP exporter

```
config: |
  processors:
    filter/ottl:
      error_mode: ignore (1)
      traces:
        span:
          - 'attributes["container.name"] == "app_container_1"' (2)
          - 'resource.attributes["host.name"] == "localhost"' (3)
```

1. 定义错误模式。当设置为 **ignore** 时，请忽略条件返回的错误。当设置为 **propagate** 时，会返回管道错误。错误会导致有效负载从 Collector 丢弃。
2. 过滤具有 **container.name == app_container_1** 属性的 span。
3. 过滤具有 **host.name == localhost** 资源属性的 span。

3.1.1.4. 路由处理器

路由处理器目前只是一个技术预览功能。

路由处理器将日志、指标或追踪路由到特定的导出器。此处理器可以从传入的 HTTP 请求 (gRPC 或普通 HTTP) 读取标头，或者可以读取资源属性，然后根据读值将 trace 信息定向到相关导出器。

OpenTelemetry Collector 自定义资源，启用了 OTLP exporter

```
config: |
  processors:
    routing:
      from_attribute: X-Tenant (1)
      default_exporters: (2)
      - jaeger
      table: (3)
      - value: acme
        exporters: [jaeger/acme]
  exporters:
    jaeger:
      endpoint: localhost:14250
    jaeger/acme:
      endpoint: localhost:24250
```

1. 执行路由时查找值的 HTTP 标头名称。
2. 下一节的表中不存在属性值时，默认导出器。
3. 定义将哪些值路由到哪个导出器的表。

您可以选择创建一个 **attribute_source** 配置，用于定义在 **from_attribute** 中查找属性的位置。允许的值是 **context**（搜索上下文），其中包括 HTTP 标头，或 **resource**（搜索资源属性）。

3.1.1.5. Exporters

导出器将数据发送到一个或多个后端或目的地。

3.1.1.5.1. OTLP exporter

OTLP gRPC 导出器使用 OpenTelemetry 协议 (OTLP) 导出追踪和指标。

OpenTelemetry Collector 自定义资源，启用了 OTLP exporter

```
config: |
  exporters:
    otlp:
      endpoint: tempo-ingester:4317 (1)
      tls: (2)
        ca_file: ca.pem
        cert_file: cert.pem
        key_file: key.pem
        insecure: false (3)
        insecure_skip_verify: false # (4)
        reload_interval: 1h (5)
        server_name_override: <name> (6)
      headers: (7)
        X-Scope-OrgID: "dev"
  service:
    pipelines:
      traces:
        exporters: [otlp]
      metrics:
        exporters: [otlp]
```

1. OTLP gRPC 端点。如果使用 **https://** 方案，则启用客户端传输安全性并覆盖 **tls** 中的 **不安全** 设置。
2. 客户端 TLS 配置。定义 TLS 证书的路径。
3. 当设置为 **true** 时禁用客户端传输安全性。默认值为 **false**。
4. 当设置为 **true** 时跳过验证证书。默认值为 **false**。
5. 指定重新载入证书的时间间隔。如果没有设置值，则证书永远不会重新加载。**reload_interval** 接受包含有效时间单位的字符串，如 **ns**、**us** （或 unmarshal**s** ）、**ms**、**s**、**m**、**h**。
6. 覆盖请求中的颁发机构的虚拟主机名，如授权标头字段。您可以使用此选项进行测试。
7. 为建立的连接期间执行的每个请求发送标头。

3.1.1.5.2. OTLP HTTP exporter

OTLP HTTP 导出器使用 OpenTelemetry 协议 (OTLP) 导出追踪和指标。

OpenTelemetry Collector 自定义资源，启用了 OTLP exporter

```
config: |
  exporters:
    otlphttp:
      endpoint: http://tempo-ingester:4318 (1)
      tls: (2)
      headers: (3)
        X-Scope-OrgID: "dev"
      disable_keep_alives: false (4)

  service:
    pipelines:
      traces:
        exporters: [otlphttp]
      metrics:
        exporters: [otlphttp]
```

1. OTLP HTTP 端点。如果使用 **https://** 方案，则启用客户端传输安全性并覆盖 **tls** 中的 **不安全** 设置。
2. 客户端 TLS 配置。定义 TLS 证书的路径。
3. 标头会在每个 HTTP 请求中发送。
4. 如果为 true，禁用 HTTP keep-alives。它将只对单个 HTTP 请求使用到服务器的连接。

3.1.1.5.3. Debug exporter

Debug exporter 会将 trace 和 metrics 打印到标准输出。

OpenTelemetry Collector 自定义资源启用了 Debug exporter

```
config: |
  exporters:
    debug:
      verbosity: detailed (1)
  service:
    pipelines:
      traces:
        exporters: [logging]
      metrics:
        exporters: [logging]
```

1. debug 导出的详细程度为：**detailed** 或 **normal** 或 **basic**。当设置为 **detailed** 时，管道数据会详细记录。默认为 **normal**。

3.1.1.5.4. Prometheus exporter

Prometheus exporter 当前只是一个技术预览功能。

Prometheus exporter 以 Prometheus 或 OpenMetrics 格式导出指标。

OpenTelemetry Collector 自定义资源，启用了 Prometheus exporter

```
ports:
- name: promexporter (1)
  port: 8889
  protocol: TCP
config: |
  exporters:
    prometheus:
      endpoint: 0.0.0.0:8889 (2)
      tls: (3)
        ca_file: ca.pem
        cert_file: cert.pem
        key_file: key.pem
      namespace: prefix (4)
      const_labels: (5)
        label1: value1
      enable_open_metrics: true (6)
      resource_to_telemetry_conversion: (7)
        enabled: true
      metric_expiration: 180m (8)
      add_metric_suffixes: false (9)
  service:
    pipelines:
      metrics:
        exporters: [prometheus]
```

1. 从 Collector pod 和服务公开 Prometheus 端口。您可以使用 **ServiceMonitor** 或 **PodMonitor** 自定义资源中的端口名称启用 Prometheus 提取指标。
2. 公开指标的网络端点。
3. 服务器端 TLS 配置。定义 TLS 证书的路径。
4. 如果设置，在提供的值下导出指标。无默认值。
5. 每个导出的指标应用的键值对标签。无默认值。
6. 如果为 **true**，则使用 OpenMetrics 格式导出指标。Exemplars 仅以 OpenMetrics 格式导出，仅适用于直方和 monotonic 摘要指标，如 **counter**。默认禁用此选项。
7. 如果 **enabled** 是 **true**，则默认情况下，所有资源属性都会转换为指标标签。默认禁用此选项。
8. 定义在没有更新的情况下公开指标的时间。默认值为 **5m**。
9. 添加指标类型和单元后缀。如果启用了 Jaeger 控制台中的 monitor 选项卡，则必须禁用。默认值是 **true**。

3.1.1.5.5. Kafka exporter

Kafka 导出器目前只是一个技术预览功能。

Kafka 导出器将日志、指标和追踪导出到 Kafka。此导出器使用同步制作者，用于阻止且不批处理消息。它必须与批处理和排队重试处理器一起使用，以获得更高的吞吐量和弹性。

OpenTelemetry Collector 自定义资源，启用了 Kafka exporter

```
config: |
  exporters:
    kafka:
      brokers: ["localhost:9092"] (1)
      protocol_version: 2.0.0 (2)
      topic: otlp_spans (3)
      auth:
        plain_text: (4)
          username: example
          password: example
        tls: (5)
          ca_file: ca.pem
          cert_file: cert.pem
          key_file: key.pem
          insecure: false (6)
          server_name_override: kafka.example.corp (7)
  service:
    pipelines:
      traces:
        exporters: [kafka]
```

1. Kafka 代理列表。默认值为 **localhost:9092**。
2. Kafka 协议版本。例如，**2.0.0**。这个为必填字段。
3. 要从中读取的 Kafka 主题的名称。以下是默认设置：**otlp_spans**（用于 traces），**otlp_metrics**（用于 metrics），**otlp_logs**（用于 logs）。
4. 纯文本身份验证配置。如果省略，则禁用纯文本身份验证。
5. 客户端 TLS 配置。定义 TLS 证书的路径。如果省略，则禁用 TLS 身份验证。
6. 禁用验证服务器的证书链和主机名。默认值为 **false**。
7. ServerName 表示客户端请求的服务器名称，以支持虚拟主机。

3.1.1.6. 连接器

连接器连接两个管道。

3.1.1.6.1. Spanmetrics 连接器

Spanmetrics 连接器目前只是一个技术预览功能。

Spanmetrics 连接器聚合了来自 span 数据的 Request, Error, 和 Duration (R.E.D) OpenTelemetry 指标。

OpenTelemetry Collector 自定义资源带有启用的 spanmetrics 连接器

```
config: |
  connectors:
    spanmetrics:
```

```
      metrics_flush_interval: 15s  (1)
    service:
      pipelines:
        traces:
          exporters: [spanmetrics]
        metrics:
          receivers: [spanmetrics]
```

1. 定义生成的指标的清除间隔。默认值为 **15s**。

3.1.1.7. 扩展

扩展为 Collector 添加功能。

3.1.1.7.1. BearerTokenAuth 扩展

BearerTokenAuth 扩展目前只是一个技术预览功能。

BearerTokenAuth 扩展是基于 HTTP 和 gRPC 协议的接收器和导出器的验证器。您可以使用 OpenTelemetry Collector 自定义资源为接收器和 exporter 端的 BearerTokenAuth 扩展配置客户端身份验证和服务器身份验证。此扩展支持 trace、metrics 和 logs。

OpenTelemetry Collector 自定义资源，为 BearerTokenAuth 扩展配置了客户端和服务器身份验证

```
  config: |
    extensions:
      bearertokenauth:
        scheme: "Bearer"  (1)
        token: "<token>"  (2)
        filename: "<token_file>"  (3)

    receivers:
      otlp:
        protocols:
          http:
            auth:
              authenticator: bearertokenauth  (4)
    exporters:
      otlp:
        auth:
          authenticator: bearertokenauth  (5)

    service:
      extensions: [bearertokenauth]
      pipelines:
        traces:
          receivers: [otlp]
          exporters: [otlp]
```

1. 您可以配置 BearerTokenAuth 扩展来发送自定义 **scheme**。默认值为 **Bearer**。
2. 您可以将 BearerTokenAuth 扩展令牌添加为元数据，以标识消息。

3. 包含随每个消息传输的授权令牌的文件路径。
4. 您可以将验证器配置分配给 OTLP 接收器。
5. 您可以将验证器配置分配给 OTLP 导出器。

3.1.1.7.2. OAuth2Client 扩展

OAuth2Client 扩展目前只是一个技术预览功能。

OAuth2Client 扩展是导出器的验证器，它基于 HTTP 和 gRPC 协议。OAuth2Client 扩展的客户端身份验证在 OpenTelemetry Collector 自定义资源中的单独部分中配置。此扩展支持 trace、metrics 和 logs。

OpenTelemetry Collector 自定义资源，为 OAuth2Client 扩展配置了客户端身份验证

```
config: |
  extensions:
    oauth2client:
      client_id: <client_id> (1)
      client_secret: <client_secret> (2)
      endpoint_params: (3)
        audience: <audience>
      token_url: https://example.com/oauth2/default/v1/token (4)
      scopes: ["api.metrics"] (5)
      # tls settings for the token client
      tls: (6)
        insecure: true (7)
        ca_file: /var/lib/mycert.pem (8)
        cert_file: <cert_file> (9)
        key_file: <key_file> (10)
      timeout: 2s (11)

  receivers:
    otlp:
      protocols:
        http:

  exporters:
    otlp:
      auth:
        authenticator: oauth2client (12)

  service:
    extensions: [oauth2client]
    pipelines:
      traces:
        receivers: [otlp]
        exporters: [otlp]
```

1. 客户端标识符，由身份提供程序提供。
2. 用于向身份提供程序验证客户端的机密密钥。

3 其他元数据，采用键值对格式，在身份验证过程中传输。例如，**audience** 指定访问令牌的预期受众，指示令牌的接收者。

4 OAuth2 令牌端点的 URL，Collector 请求访问令牌。

5 范围定义客户端请求的特定权限或访问级别。

6 令牌客户端的传输层安全性 (TLS) 设置，用于在请求令牌时建立安全连接。

7 当设置为 **true** 时，将 Collector 配置为使用不安全或非验证的 TLS 连接来调用配置的令牌端点。

8 用于在 TLS 握手过程中验证服务器证书的证书颁发机构 (CA) 文件的路径。

9 如果需要，客户端必须用来向 OAuth2 服务器验证自己的客户端证书文件的路径。

10 身份验证所需的客户端私钥文件的路径。

11 为令牌客户端的请求设置超时。

12 您可以将验证器配置分配给 OTLP 导出器。

3.1.1.7.3. Jaeger Remote Sampling 扩展

Jaeger Remote Sampling 扩展目前只是一个技术预览功能。

Jaeger Remote Sampling 扩展允许在 Jaeger 的远程抽样 API 后提供抽样策略。您可以配置此扩展，将请求代理到后备远程抽样服务器，如 Jaeger 收集器关闭管道或从本地文件系统到静态 JSON 文件。

OpenTelemetry Collector 自定义资源带有配置的 Jaeger Remote Sampling 扩展

```
config: |
  extensions:
    jaegerremotesampling:
      source:
        reload_interval: 30s 1
        remote:
          endpoint: jaeger-collector:14250 2
        file: /etc/otelcol/sampling_strategies.json 3

  receivers:
    otlp:
      protocols:
        http:

  exporters:
    otlp:

  service:
    extensions: [jaegerremotesampling]
    pipelines:
      traces:
        receivers: [otlp]
        exporters: [otlp]
```

1. 抽样配置更新的时间间隔。
2. 用于访问 Jaeger 远程抽样策略供应商的端点。
3. JSON 格式包含抽样策略配置的本地文件的路径。

Jaeger Remote Sampling 策略文件示例

```
{
  "service_strategies": [
    {
      "service": "foo",
      "type": "probabilistic",
      "param": 0.8,
      "operation_strategies": [
        {
          "operation": "op1",
          "type": "probabilistic",
          "param": 0.2
        },
        {
          "operation": "op2",
          "type": "probabilistic",
          "param": 0.4
        }
      ]
    },
    {
      "service": "bar",
      "type": "ratelimiting",
      "param": 5
    }
  ],
  "default_strategy": {
    "type": "probabilistic",
    "param": 0.5,
    "operation_strategies": [
      {
        "operation": "/health",
        "type": "probabilistic",
        "param": 0.0
      },
      {
        "operation": "/metrics",
        "type": "probabilistic",
        "param": 0.0
      }
    ]
  }
}
```

3.1.1.7.4. Performance Profiler 扩展

Performance Profiler 扩展目前只是一个技术预览功能。

Performance Profiler 扩展启用 Go **net/http/pprof** 端点。这通常供开发人员用来收集性能配置集，并调查服务的问题。

OpenTelemetry Collector 自定义资源带有配置的 Performance Profiler 扩展

```
config: |
  extensions:
    pprof:
      endpoint: localhost:1777 (1)
      block_profile_fraction: 0 (2)
      mutex_profile_fraction: 0 (3)
      save_to_file: test.pprof (4)

  receivers:
    otlp:
      protocols:
        http:

  exporters:
    otlp:

  service:
    extensions: [pprof]
    pipelines:
      traces:
        receivers: [otlp]
        exporters: [otlp]
```

1. 此扩展侦听的端点。使用 **localhost:** 使其仅在本地可用； **":"** 使其在所有网络接口上可用。默认值为 **localhost:1777**。
2. 设置要配置集的一小部分阻塞事件。要禁用性能分析，请将其设置为 **0** 或负整数。请参阅 **runtime** 软件包 的文档。默认值为 **0**。
3. 设置要配置集的几部分 mutex 争用事件。要禁用性能分析，请将其设置为 **0** 或负整数。请参阅 **runtime** 软件包的文档。默认值为 **0**。
4. 要保存 CPU 配置集的文件的名称。分析会在 Collector 启动时启动。在 Collector 终止时，配置集被保存到 文件中。

3.1.1.7.5. 健康检查扩展

Health Check 扩展目前只是一个技术预览功能。

Health Check 扩展提供了一个 HTTP URL，用于检查 OpenTelemetry Collector 的状态。您可以将此扩展用作 OpenShift 上的存活度和就绪度探测。

OpenTelemetry Collector 自定义资源带有配置的 Health Check 扩展

```
config: |
  extensions:
    health_check:
      endpoint: "0.0.0.0:13133" (1)
      tls: (2)
```

```
        ca_file: "/path/to/ca.crt"
        cert_file: "/path/to/cert.crt"
        key_file: "/path/to/key.key"
      path: "/health/status" (3)
      check_collector_pipeline: (4)
        enabled: true (5)
        interval: "5m" (6)
        exporter_failure_threshold: 5 (7)

  receivers:
    otlp:
      protocols:
        http:

  exporters:
    otlp:

  service:
    extensions: [health_check]
    pipelines:
      traces:
        receivers: [otlp]
        exporters: [otlp]
```

1. 发布健康检查状态的目标 IP 地址。默认值为 **0.0.0.0:13133**。
2. TLS 服务器端配置。定义 TLS 证书的路径。如果省略，则禁用 TLS。
3. 健康检查服务器的路径。默认值为 /。
4. Collector 管道健康检查的设置。
5. 启用 Collector 管道健康检查。默认值为 **false**。
6. 检查失败次数的时间间隔。默认值为 **5m**。
7. 在容器仍标记为健康前，失败次数的阈值。默认值为 **5**。

3.1.1.7.6. 内存 Ballast 扩展

Memory Ballast 扩展目前只是一个技术预览功能。

Memory Ballast 扩展可让应用程序为进程配置内存 ballast。

OpenTelemetry Collector 自定义资源带有配置的 Memory Ballast 扩展

```
config: |
  extensions:
    memory_ballast:
      size_mib: 64 (1)
      size_in_percentage: 20 (2)

  receivers:
    otlp:
```

```
        protocols:
          http:

    exporters:
      otlp:

    service:
      extensions: [memory_ballast]
      pipelines:
        traces:
          receivers: [otlp]
          exporters: [otlp]
```

1. 以 MiB 为单位设置内存 ballast 大小。如果指定了这两个值，则优先于 **size_in_percentage**。
2. 将内存 ballast 设置为总内存的百分比 **1-100**。支持容器化和物理主机环境。

3.1.1.7.7. zPages 扩展

zPages 扩展目前只是一个技术预览功能。

zPages 扩展为提供 zPages 的扩展提供了一个 HTTP 端点。在端点，此扩展为调试检测组件提供实时数据。所有核心导出器和接收器提供一些 zPages 检测。

zPages 可用于进程内诊断，而无需依赖后端来检查 trace 或指标。

OpenTelemetry Collector 自定义资源带有配置的 zPages 扩展

```
  config: |
    extensions:
      zpages:
        endpoint: "localhost:55679" # (1)

    receivers:
      otlp:
        protocols:
          http:
    exporters:
      otlp:

    service:
      extensions: [zpages]
      pipelines:
        traces:
          receivers: [otlp]
          exporters: [otlp]
```

1. 指定提供 zPages 的 HTTP 端点。使用 **localhost:** 使其仅在本地可用，或 **":"** 使其在所有网络接口上可用。默认值为 **localhost:55679**。

3.2. 使用 OPENTELEMETRY COLLECTOR 从不同集群收集可观察性数据

对于多集群配置，您可以在每个远程集群中创建一个 OpenTelemetry Collector 实例，并将所有遥测数据转发到一个 OpenTelemetry Collector 实例。

先决条件

- 已安装红帽构建的 OpenTelemetry Operator。
- 已安装 Tempo Operator。
- 在集群中部署了 TempoStack 实例。
- 以下挂载的证书：签发者、自签名证书、CA 签发者、客户端和服务器证书。要创建这些证书，请参阅第 1 步。

流程

1. 在 OpenTelemetry Collector 实例中挂载以下证书，跳过已挂载的证书。
   a. 使用 cert-manager Operator for Red Hat OpenShift 生成这些证书的签发者。

```
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: selfsigned-issuer
spec:
  selfSigned: {}
```

   b. 一个自签名证书。

```
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: ca
spec:
  isCA: true
  commonName: ca
  subject:
    organizations:
      - Organization # <your_organization_name>
    organizationalUnits:
      - Widgets
  secretName: ca-secret
  privateKey:
    algorithm: ECDSA
    size: 256
  issuerRef:
    name: selfsigned-issuer
    kind: Issuer
    group: cert-manager.io
```

   c. 一个 CA 签发者。

```
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
```

```
  name: test-ca-issuer
spec:
  ca:
    secretName: ca-secret
```

d. 客户端和服务器证书。

```
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: server
spec:
  secretName: server-tls
  isCA: false
  usages:
    - server auth
    - client auth
  dnsNames:
  - "otel.observability.svc.cluster.local" (1)
  issuerRef:
    name: ca-issuer
---
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: client
spec:
  secretName: client-tls
  isCA: false
  usages:
    - server auth
    - client auth
  dnsNames:
  - "otel.observability.svc.cluster.local" (2)
  issuerRef:
    name: ca-issuer
```

1. 在 server OpenTelemetry Collector 实例中映射到 solver 的确切 DNS 名称列表。
2. 在客户端 OpenTelemetry Collector 实例中映射到 solver 的确切 DNS 名称列表。

2. 为 OpenTelemetry Collector 实例创建服务帐户。

ServiceAccount 示例

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: otel-collector-deployment
```

3. 为服务帐户创建集群角色。

ClusterRole 示例

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: otel-collector
rules:
  # (1)
  # (2)
- apiGroups: ["", "config.openshift.io"]
  resources: ["pods", "namespaces", "infrastructures", "infrastructures/status"]
  verbs: ["get", "watch", "list"]
```

1. **k8sattributesprocessor** 需要 pod 和命名空间资源的权限。

2. **resourcedetectionprocessor** 需要基础架构和状态的权限。

4. 将集群角色绑定到服务帐户。

ClusterRoleBinding 示例

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: otel-collector
subjects:
- kind: ServiceAccount
  name: otel-collector-deployment
  namespace: otel-collector-<example>
roleRef:
  kind: ClusterRole
  name: otel-collector
  apiGroup: rbac.authorization.k8s.io
```

5. 创建 YAML 文件，在边缘集群中定义 **OpenTelemetryCollector** 自定义资源 (CR)。

边缘集群的 OpenTelemetryCollector 自定义资源示例

```
apiVersion: opentelemetry.io/v1alpha1
kind: OpenTelemetryCollector
metadata:
  name: otel
  namespace: otel-collector-<example>
spec:
  mode: daemonset
  serviceAccount: otel-collector-deployment
  config: |
    receivers:
      jaeger:
        protocols:
          grpc:
          thrift_binary:
          thrift_compact:
          thrift_http:
      opencensus:
      otlp:
```

```
        protocols:
          grpc:
          http:
      zipkin:
    processors:
      batch:
      k8sattributes:
      memory_limiter:
        check_interval: 1s
        limit_percentage: 50
        spike_limit_percentage: 30
      resourcedetection:
        detectors: [openshift]
    exporters:
      otlphttp:
        endpoint: https://observability-cluster.com:443 (1)
        tls:
          insecure: false
          cert_file: /certs/server.crt
          key_file: /certs/server.key
          ca_file: /certs/ca.crt
    service:
      pipelines:
        traces:
          receivers: [jaeger, opencensus, otlp, zipkin]
          processors: [memory_limiter, k8sattributes, resourcedetection, batch]
          exporters: [otlp]
  volumes:
    - name: otel-certs
      secret:
        name: otel-certs
  volumeMounts:
    - name: otel-certs
      mountPath: /certs
```

(1) Collector exporter 配置为导出 OTLP HTTP，并指向来自中央集群的 OpenTelemetry Collector。

6. 创建 YAML 文件，在中央集群中定义 **OpenTelemetryCollector** 自定义资源 (CR)。

Central 集群的 OpenTelemetryCollector 自定义资源示例

```
apiVersion: opentelemetry.io/v1alpha1
kind: OpenTelemetryCollector
metadata:
  name: otlp-receiver
  namespace: observability
spec:
  mode: "deployment"
  ingress:
    type: route
    route:
      termination: "passthrough"
  config: |
    receivers:
```

```
      otlp:
        protocols:
          http:
            tls: (1)
              cert_file: /certs/server.crt
              key_file: /certs/server.key
              client_ca_file: /certs/ca.crt
    exporters:
      logging:
      otlp:
        endpoint: "tempo-<simplest>-distributor:4317" (2)
        tls:
          insecure: true
    service:
      pipelines:
        traces:
          receivers: [otlp]
          processors: []
          exporters: [otlp]
  volumes:
    - name: otel-certs
      secret:
        name: otel-certs
  volumeMounts:
    - name: otel-certs
      mountPath: /certs
```

1. Collector 接收器需要第一步中列出的证书。
2. Collector exporter 配置为导出 OTLP 并指向 Tempo 经销商端点，本例中为 **"tempo-simplest-distributor:4317"** 并已创建。

3.3. 将指标发送到监控堆栈的配置

OpenTelemetry Collector 自定义资源 (CR) 可以配置为创建一个 Prometheus **ServiceMonitor** CR，以提取 Collector 的管道指标并启用 Prometheus exporter。

带有 Prometheus exporter 的 OpenTelemetry Collector 自定义资源示例

```
spec:
  mode: deployment
  observability:
    metrics:
      enableMetrics: true (1)
  config: |
    exporters:
      prometheus:
        endpoint: 0.0.0.0:8889
        resource_to_telemetry_conversion:
          enabled: true # by default resource attributes are dropped
    service:
      telemetry:
        metrics:
          address: ":8888"
```

```
      pipelines:
        metrics:
          receivers: [otlp]
          exporters: [prometheus]
```

1. 配置 Operator，以创建 Prometheus **ServiceMonitor** CR，以提取收集器的内部指标端点和 Prometheus exporter 指标端点。指标将存储在 OpenShift 监控堆栈中。

另外，手动创建 Prometheus **PodMonitor** 可以提供精细的控制，例如删除 Prometheus 提取过程中添加的重复标签。

配置监控堆栈以提取 Collector 指标的 PodMonitor 自定义资源示例

```
apiVersion: monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: otel-collector
spec:
  selector:
    matchLabels:
      app.kubernetes.io/name: `<cr_name>-collector` # (1)
  podMetricsEndpoints:
  - port: metrics # (2)
  - port: promexporter # (3)
    relabelings:
    - action: labeldrop
      regex: pod
    - action: labeldrop
      regex: container
    - action: labeldrop
      regex: endpoint
    metricRelabelings:
    - action: labeldrop
      regex: instance
    - action: labeldrop
      regex: job
```

1. OpenTelemetry Collector 自定义资源的名称。
2. OpenTelemetry Collector 的内部指标端口的名称。此端口名称始终是 **metrics**。
3. OpenTelemetry Collector 的 Prometheus exporter 端口的名称。

3.4. 为红帽构建的 OPENTELEMETRY 设置监控

Red Hat build of OpenTelemetry Operator 支持每个 OpenTelemtry Collector 实例的监控和警报，并公开有关 Operator 本身的升级和操作指标。

3.4.1. 配置 OpenTelemetry Collector 指标

您可以启用 OpenTelemetry Collector 实例的指标和警报。

先决条件

- 在集群中启用对用户定义的项目的监控。

流程

- 要启用 OpenTelemetry Collector 实例的指标，请将 **spec.observability.metrics.enableMetrics** 字段设置为 **true** ：

```
apiVersion: opentelemetry.io/v1alpha1
kind: OpenTelemetryCollector
metadata:
  name: <name>
spec:
  observability:
    metrics:
      enableMetrics: true
```

验证

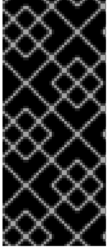
您可以使用 Web 控制台的 **Administrator** 视图来验证配置是否成功：

- 进入 **Observe** → **Targets**，按 **Source: User** 过滤，并检查 **opentelemetry-collector-<instance_name>** 格式的 ServiceMonitors 是否具有 **Up** 状态。

3.5. 其他资源

- 为用户定义的项目启用监控

第 4 章 配置和部署 OPENTELEMETRY 检测注入



重要

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

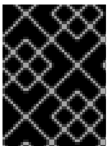
有关红帽技术预览功能支持范围的更多信息，请参阅技术预览功能支持范围。

Red Hat build of OpenTelemetry Operator 使用定义检测配置的自定义资源定义 (CRD) 文件。

4.1. OPENTELEMETRY 检测配置选项

红帽构建的 OpenTelemetry 可以注入并配置 OpenTelemetry 自动检测库到您的工作负载。目前，项目支持注入来自 Go、Java、Node.js、Python、.NET 和 Apache HTTP 服务器 (**httpd**) 的检测库。

OpenTelemetry 中的自动检测是指框架在没有手动代码更改的情况下自动检测应用程序的功能。这可让开发人员和管理员以最少的努力和更改现有代码库来观察到其应用程序中。



重要

红帽构建的 OpenTelemetry Operator 仅支持工具库的注入机制，但不支持检测库或上游镜像。客户可以构建自己的检测镜像，或使用社区镜像。

4.1.1. 检测选项

检测选项在 **OpenTelemetryCollector** 自定义资源中指定。

OpenTelemetryCollector 自定义资源文件示例

```
apiVersion: opentelemetry.io/v1alpha1
kind: Instrumentation
metadata:
  name: java-instrumentation
spec:
  env:
    - name: OTEL_EXPORTER_OTLP_TIMEOUT
      value: "20"
  exporter:
    endpoint: http://production-collector.observability.svc.cluster.local:4317
  propagators:
    - w3c
  sampler:
    type: parentbased_traceidratio
    argument: "0.25"
  java:
    env:
    - name: OTEL_JAVAAGENT_DEBUG
      value: "true"
```

表 4.1. Operator 用来定义调用的参数

参数	描述	值
env	在所有检测中要定义的通用环境变量。	
exporter	导出器配置。	
propagators	Propagators 定义进程上下文传播配置。	**tracecontext**, **baggage**, **b3**, **b3multi**, **jaeger**, **ottrace**, **none**
resource	资源属性配置。	
sampler	抽样配置。	
apacheHttpd	Apache HTTP 服务器检测的配置。	
dotnet	配置 .NET 检测。	
go	配置 Go 检测。	
java	Java 检测配置。	
nodejs	配置 Node.js 检测。	
python	Python 检测配置。	

4.1.2. 使用带有 Service Mesh 的检测 CR

当在 Red Hat OpenShift Service Mesh 中使用检测自定义资源 (CR) 时，您必须使用 **b3multi** propagator。

4.1.2.1. 配置 Apache HTTP 服务器自动检测

表 4.2. .spec.apacheHttpd 字段的 Prameters

Name	描述	default
attrs	特定于 Apache HTTP 服务器的属性。	
configPath	Apache HTTP 服务器配置的位置。	/usr/local/apache2/conf
env	特定于 Apache HTTP 服务器的环境变量。	
image	使用 Apache SDK 和自动检测的容器镜像。	
resourceRequirements	计算资源要求。	
version	Apache HTTP 服务器版本。	2.4

启用注入的 PodSpec 注解

```
instrumentation.opentelemetry.io/inject-apache-httpd: "true"
```

4.1.2.2. 配置 .NET 自动检测

Name	描述
env	特定于 .NET 的环境变量。
image	带有 .NET SDK 和自动检测的容器镜像。
resourceRequirements	计算资源要求。

对于 .NET 自动检测，如果需要的 **OTEL_EXPORTER_OTLP_ENDPOINT** 环境变量，如果导出器的端点被设置为 **4317**，则必须设置所需的 OTEL_EXPORTER_OTLP_ENDPOINT 环境变量。默认情况下，.NET autoinstrumentation 使用 **http/proto**，遥测数据必须设置为 **4318** 端口。

启用注入的 PodSpec 注解

```
instrumentation.opentelemetry.io/inject-dotnet: "true"
```

–

4.1.2.3. 配置 Go 自动检测

Name	描述
env	特定于 Go 的环境变量。
image	带有 Go SDK 和自动检测的容器镜像。
resourceRequirements	计算资源要求。

启用注入的 PodSpec 注解

```
instrumentation.opentelemetry.io/inject-go: "true"
```

OpenShift 集群中 Go 自动检测所需的额外权限

```
apiVersion: security.openshift.io/v1
kind: SecurityContextConstraints
metadata:
  name: otel-go-instrumentation-scc
allowHostDirVolumePlugin: true
allowPrivilegeEscalation: true
allowPrivilegedContainer: true
allowedCapabilities:
- "SYS_PTRACE"
fsGroup:
  type: RunAsAny
runAsUser:
  type: RunAsAny
seLinuxContext:
  type: RunAsAny
seccompProfiles:
- '*'
supplementalGroups:
  type: RunAsAny
```

提示

为 OpenShift 集群中的 Go auto-instrumentation 应用权限的 CLI 命令如下：

```
$ oc adm policy add-scc-to-user otel-go-instrumentation-scc -z <service_account>
```

4.1.2.4. 配置 Java 自动检测

Name	描述
env	特定于 Java 的环境变量。
image	使用 Java SDK 和自动检测的容器镜像。
resourceRequirements	计算资源要求。

启用注入的 PodSpec 注解

```
instrumentation.opentelemetry.io/inject-java: "true"
```

4.1.2.5. 配置 Node.js 自动检测

Name	描述
env	特定于 Node.js 的环境变量。
image	使用 Node.js SDK 和自动检测的容器镜像。
resourceRequirements	计算资源要求。

用于启用注入的 PodSpec 注解

```
instrumentation.opentelemetry.io/inject-nodejs: "true"
instrumentation.opentelemetry.io/otel-go-auto-target-exe: "/path/to/container/executable"
```

instrumentation.opentelemetry.io/otel-go-auto-target-exe 注解设置所需的 **OTEL_GO_AUTO_TARGET_EXE** 环境变量的值。

4.1.2.6. 配置 Python 自动检测

Name	描述
env	特定于 Python 的环境变量。

Name	描述
image	使用 Python SDK 和自动检测的容器镜像。
resourceRequirements	计算资源要求。

对于 Python 自动检测，如果导出器的端点被设置为 **4317**，则必须设置 **OTEL_EXPORTER_OTLP_ENDPOINT** 环境变量。Python 自动检测默认使用 **http/proto**，并且遥测数据必须设置为 **4318** 端口。

启用注入的 PodSpec 注解

```
instrumentation.opentelemetry.io/inject-python: "true"
```

4.1.2.7. 配置 OpenTelemetry SDK 变量

pod 中的 OpenTelemetry SDK 变量可通过以下注解进行配置：

```
instrumentation.opentelemetry.io/inject-sdk: "true"
```

请注意，所有注解都接受以下值：

true
: 从命名空间中注入 **Instrumentation** 资源。

false
: 不注入任何检测。

instrumentation-name
: 从当前命名空间注入的检测资源的名称。

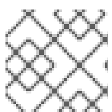
other-namespace/instrumentation-name
: 从另一个命名空间注入的检测资源的名称。

4.1.2.8. 多容器 pod

检测会根据 pod 规格在默认可用的第一个容器上运行。在某些情况下，您还可以为注入指定目标容器。

Pod 注解

```
instrumentation.opentelemetry.io/container-names: "<container_1>,<container_2>"
```



注意

Go 自动检测不支持多容器自动检测注入。

第 5 章 使用红帽构建的 OPENTELEMETRY

您可以设置并使用红帽构建的 OpenTelemetry 将 trace 发送到 OpenTelemetry Collector 或 TempoStack。

5.1. 使用 OPENTELEMETRY COLLECTOR 将 TRACE 转发到 TEMPOSTACK

要将转发追踪配置为 TempoStack，您可以部署和配置 OpenTelemetry Collector。您可以使用指定的处理器、接收器和导出器在部署模式中部署 OpenTelemetry Collector。有关其他模式，请参阅*附加资源*中的 OpenTelemetry Collector 文档链接。

先决条件

- 已安装红帽构建的 OpenTelemetry Operator。
- 已安装 Tempo Operator。
- 在集群中部署了 TempoStack。

流程

1. 为 OpenTelemetry Collector 创建服务帐户。

   ServiceAccount 示例

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: otel-collector-deployment
```

2. 为服务帐户创建集群角色。

   ClusterRole 示例

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: otel-collector
rules:
  1
  2
- apiGroups: ["", "config.openshift.io"]
  resources: ["pods", "namespaces", "infrastructures", "infrastructures/status"]
  verbs: ["get", "watch", "list"]
```

   1 **k8sattributesprocessor** 需要 pod 和命名空间资源的权限。

   2 **resourcedetectionprocessor** 需要基础架构和状态的权限。

3. 将集群角色绑定到服务帐户。

ClusterRoleBinding 示例

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: otel-collector
subjects:
- kind: ServiceAccount
  name: otel-collector-deployment
  namespace: otel-collector-example
roleRef:
  kind: ClusterRole
  name: otel-collector
  apiGroup: rbac.authorization.k8s.io
```

4. 创建 YAML 文件以定义 **OpenTelemetryCollector** 自定义资源(CR)。

OpenTelemetryCollector 示例

```
apiVersion: opentelemetry.io/v1alpha1
kind: OpenTelemetryCollector
metadata:
  name: otel
spec:
  mode: deployment
  serviceAccount: otel-collector-deployment
  config: |
    receivers:
      jaeger:
        protocols:
          grpc:
          thrift_binary:
          thrift_compact:
          thrift_http:
      opencensus:
      otlp:
        protocols:
          grpc:
          http:
      zipkin:
    processors:
      batch:
      k8sattributes:
      memory_limiter:
        check_interval: 1s
        limit_percentage: 50
        spike_limit_percentage: 30
      resourcedetection:
        detectors: [openshift]
    exporters:
      otlp:
        endpoint: "tempo-simplest-distributor:4317" (1)
        tls:
          insecure: true
    service:
```

```
  pipelines:
    traces:
      receivers: [jaeger, opencensus, otlp, zipkin] (2)
      processors: [memory_limiter, k8sattributes, resourcedetection, batch]
      exporters: [otlp]
```

1. Collector exporter 配置为导出 OTLP 并指向 Tempo 经销商端点 **"tempo-simplest-distributor:4317"**（在这个示例中已创建）。

2. Collector 配置了 Jaeger trace 的接收器，OpenCensus trace over the OpenCensus 协议, Zipkin trace over the Zipkin protocol, 和 OTLP trace over the GRPC 协议。

提示

您可以将 **tracegen** 部署为测试：

```
apiVersion: batch/v1
kind: Job
metadata:
  name: tracegen
spec:
  template:
    spec:
      containers:
        - name: tracegen
          image: ghcr.io/open-telemetry/opentelemetry-collector-contrib/tracegen:latest
          command:
            - "./tracegen"
          args:
            - -otlp-endpoint=otel-collector:4317
            - -otlp-insecure
            - -duration=30s
            - -workers=1
      restartPolicy: Never
  backoffLimit: 4
```

其他资源

- OpenTelemetry Collector 文档
- GitHub 上的部署示例

5.2. 将 TRACE 和 METRICS 发送到 OPENTELEMETRY COLLECTOR

使用或不进行 sidecar 注入功能，可以将 trace 和 metrics 发送到 OpenTelemetry Collector。

5.2.1. 使用 sidecar 注入向 OpenTelemetry Collector 发送 trace 和 metrics

您可以将遥测数据发送到带有 sidecar 注入的 OpenTelemetry Collector 实例。

Red Hat build of OpenTelemetry Operator 允许 sidecar 注入部署工作负载，并自动配置您的检测向 OpenTelemetry Collector 发送遥测数据。

先决条件

- 安装了 Red Hat OpenShift distributed tracing Platform (Tempo)，并部署了 TempoStack 实例。
- 您可以通过 Web 控制台或 OpenShift CLI (**oc**)访问集群：
  - 以集群管理员身份使用 **cluster-admin** 角色登录到 web 控制台。
  - 集群管理员具有 **cluster-admin** 角色的活跃 OpenShift CLI (**oc**) 会话。
  - 对于 Red Hat OpenShift Dedicated，您必须有一个具有 **dedicated-admin** 角色的帐户。

流程

1. 为 OpenTelemetry Collector 实例创建项目。

```
apiVersion: project.openshift.io/v1
kind: Project
metadata:
  name: observability
```

2. 创建一个服务帐户。

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: otel-collector-sidecar
  namespace: observability
```

3. 为 **k8sattributes** 和 **resourcedetection** 处理器的服务帐户授予权限。

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: otel-collector
rules:
- apiGroups: ["", "config.openshift.io"]
  resources: ["pods", "namespaces", "infrastructures", "infrastructures/status"]
  verbs: ["get", "watch", "list"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: otel-collector
subjects:
- kind: ServiceAccount
  name: otel-collector-sidecar
  namespace: observability
roleRef:
  kind: ClusterRole
  name: otel-collector
  apiGroup: rbac.authorization.k8s.io
```

4. 将 OpenTelemetry Collector 部署为 sidecar。

```
apiVersion: opentelemetry.io/v1alpha1
kind: OpenTelemetryCollector
metadata:
  name: otel
  namespace: observability
spec:
  serviceAccount: otel-collector-sidecar
  mode: sidecar
  config: |
    serviceAccount: otel-collector-sidecar
    receivers:
      otlp:
        protocols:
          grpc:
          http:
    processors:
      batch:
      memory_limiter:
        check_interval: 1s
        limit_percentage: 50
        spike_limit_percentage: 30
      resourcedetection:
        detectors: [openshift]
        timeout: 2s
    exporters:
      otlp:
        endpoint: "tempo-<example>-gateway:8090" (1)
        tls:
          insecure: true
    service:
      pipelines:
        traces:
          receivers: [jaeger]
          processors: [memory_limiter, resourcedetection, batch]
          exporters: [otlp]
```

(1) 这指向使用 Tempo Operator 部署的 **<example>** TempoStack 实例的网关。

5. 使用 **otel-collector-sidecar** 服务帐户创建部署。

6. 在您的 **Deployment** 对象中添加 **sidecar.opentelemetry.io/inject: "true"** 注解。这将注入所有需要的环境变量，将工作负载中的数据发送到 OpenTelemetry Collector 实例。

5.2.2. 在没有 sidecar 注入的情况下向 OpenTelemetry Collector 发送 trace 和 metrics

您可以在不进行 sidecar 注入的情况下将遥测数据发送到 OpenTelemetry Collector 实例，这涉及手动设置几个环境变量。

先决条件

- 安装了 Red Hat OpenShift distributed tracing Platform (Tempo)，并部署了 TempoStack 实例。
- 您可以通过 Web 控制台或 OpenShift CLI (**oc**)访问集群：

- 以集群管理员身份使用 **cluster-admin** 角色登录到 web 控制台。
- 集群管理员具有 **cluster-admin** 角色的活跃 OpenShift CLI (**oc**) 会话。
- 对于 Red Hat OpenShift Dedicated，您必须有一个具有 **dedicated-admin** 角色的帐户。

流程

1. 为 OpenTelemetry Collector 实例创建项目。

```
apiVersion: project.openshift.io/v1
kind: Project
metadata:
  name: observability
```

2. 创建一个服务帐户。

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: otel-collector-deployment
  namespace: observability
```

3. 为 **k8sattributes** 和 **resourcedetection** 处理器的服务帐户授予权限。

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: otel-collector
rules:
- apiGroups: ["", "config.openshift.io"]
  resources: ["pods", "namespaces", "infrastructures", "infrastructures/status"]
  verbs: ["get", "watch", "list"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: otel-collector
subjects:
- kind: ServiceAccount
  name: otel-collector
  namespace: observability
roleRef:
  kind: ClusterRole
  name: otel-collector
  apiGroup: rbac.authorization.k8s.io
```

4. 使用 OpenTelemetryCollector 自定义资源部署 **OpenTelemetry Collector** 实例。

```
apiVersion: opentelemetry.io/v1alpha1
kind: OpenTelemetryCollector
metadata:
  name: otel
  namespace: observability
spec:
```

```
mode: deployment
serviceAccount: otel-collector-deployment
config: |
  receivers:
    jaeger:
      protocols:
        grpc:
        thrift_binary:
        thrift_compact:
        thrift_http:
    opencensus:
    otlp:
      protocols:
        grpc:
        http:
    zipkin:
  processors:
    batch:
    k8sattributes:
    memory_limiter:
      check_interval: 1s
      limit_percentage: 50
      spike_limit_percentage: 30
    resourcedetection:
      detectors: [openshift]
  exporters:
    otlp:
      endpoint: "tempo-<example>-distributor:4317" (1)
      tls:
        insecure: true
  service:
    pipelines:
      traces:
        receivers: [jaeger, opencensus, otlp, zipkin]
        processors: [memory_limiter, k8sattributes, resourcedetection, batch]
        exporters: [otlp]
```

(1) 这指向使用 Tempo Operator 部署的 **<example>** TempoStack 实例的网关。

5. 使用您的检测应用程序设置容器中的环境变量。

Name	描述	默认值
OTEL_SERVICE_NAME	设置 **service.name** 资源属性的值。	""
OTEL_EXPORTER_OTLP_ENDPOINT	带有可选指定端口号的任何信号类型的基本端点 URL。	**https://localhost:4317**

Name	描述	默认值
OTEL_EXPORTER_OTLP_CERTIFICATE	gRPC 客户端的 TLS 凭证的证书文件的路径。	**https://localhost:4317**
OTEL_TRACES_SAMPLER	用于 trace 的 sampler。	**parentbased_always_on**
OTEL_EXPORTER_OTLP_PROTOCOL	OTLP 导出器的传输协议。	**grpc**
OTEL_EXPORTER_OTLP_TIMEOUT	OTLP 导出器等待每个批处理导出的最大时间间隔。	**10s**
OTEL_EXPORTER_OTLP_INSECURE	为 gRPC 请求禁用客户端传输安全性。HTTPS 模式会覆盖它。	**False**

第 6 章 对红帽构建的 OPENTELEMETRY 进行故障排除

OpenTelemetry Collector 提供了多种方法来测量其健康状况，并调查数据监控问题。

6.1. 获取 OPENTELEMETRY COLLECTOR 日志

您可以按照如下所示，获取 OpenTelemetry Collector 的日志。

流程

1. 在 **OpenTelemetryCollector** 自定义资源(CR) 中设置相关的日志级别：

```
config: |
  service:
    telemetry:
      logs:
        level: debug 1
```

1 收集器的日志级别。支持的值包括 **info**、**warn**、**error** 或 **debug**。默认为 **info**。

2. 使用 **oc logs** 命令或 Web 控制台来检索日志。

6.2. 公开指标

OpenTelemetry Collector 会公开有关它已处理的数据卷的指标。以下指标可用于 span，但为指标和日志信号公开类似的指标：

otelcol_receiver_accepted_spans
: 成功推送到管道中的 span 数量。

otelcol_receiver_refused_spans
: 无法推送到管道中的 span 数量。

otelcol_exporter_sent_spans
: 成功发送到目的地的 span 数量。

otelcol_exporter_enqueue_failed_spans
: 无法添加到发送队列的 span 数量。

Operator 会创建一个 **<cr_name>-collector-monitoring** 遥测服务，可用于提取指标端点。

流程

1. 通过在 **OpenTelemetryCollector** 自定义资源中添加以下行来启用 telemetry 服务：

```
config: |
  service:
    telemetry:
      metrics:
        address: ":8888" 1
```

1 公开内部收集器指标的地址。默认值为 **:8888**。

1. 运行以下命令来检索指标，该命令使用端口转发 Collector pod：

```
$ oc port-forward <collector_pod>
```

2. 访问位于 **http://localhost:8888/metrics** 的指标端点。

6.3. DEBUG EXPORTER

您可以配置 debug exporter，将收集的数据导出到标准输出。

流程

1. 配置 **OpenTelemetryCollector** 自定义资源，如下所示：

```
config: |
  exporters:
    debug:
      verbosity: detailed
  service:
    pipelines:
      traces:
        exporters: [debug]
      metrics:
        exporters: [debug]
      logs:
        exporters: [debug]
```

2. 使用 **oc logs** 命令或 Web 控制台将日志导出到标准输出。

第 7 章 从分布式追踪平台 (JAEGER) 迁移到 OPENTELEMETRY 的红帽构建

如果您已将 Red Hat OpenShift distributed tracing 平台(Jaeger)用于应用程序，您可以迁移到 OpenTelemetry 的红帽构建，它基于 OpenTelemetry 开源项目。

Red Hat build of OpenTelemetry 提供了一组 API、库、代理和工具，以便在分布式系统中促进可观察性。Red Hat build of OpenTelemetry 中的 OpenTelemetry Collector 可以影响 Jaeger 协议，因此您不需要在应用程序中更改 SDK。

从分布式追踪平台 (Jaeger) 迁移到红帽构建的 OpenTelemetry 需要配置 OpenTelemetry Collector 和应用程序来无缝报告 trace。您可以迁移 sidecar 和 sidecar 部署。

7.1. 从分布式追踪平台 (JAEGER) 迁移到带有 SIDECAR 的红帽构建的 OPENTELEMETRY

Red Hat build of OpenTelemetry Operator 支持 sidecar 注入部署工作负载，以便您可以从分布式追踪平台(Jaeger) sidecar 迁移到红帽构建的 OpenTelemetry sidecar。

先决条件

- 在集群中使用 Red Hat OpenShift distributed tracing Platform (Jaeger)。
- 已安装红帽构建的 OpenTelemetry。

流程

1. 将 OpenTelemetry Collector 配置为 sidecar。

```
apiVersion: opentelemetry.io/v1alpha1
kind: OpenTelemetryCollector
metadata:
  name: otel
  namespace: <otel-collector-namespace>
spec:
  mode: sidecar
  config: |
    receivers:
      jaeger:
        protocols:
          grpc:
          thrift_binary:
          thrift_compact:
          thrift_http:
    processors:
      batch:
      memory_limiter:
        check_interval: 1s
        limit_percentage: 50
        spike_limit_percentage: 30
      resourcedetection:
        detectors: [openshift]
        timeout: 2s
    exporters:
```

```
      otlp:
        endpoint: "tempo-<example>-gateway:8090" (1)
        tls:
          insecure: true
    service:
      pipelines:
        traces:
          receivers: [jaeger]
          processors: [memory_limiter, resourcedetection, batch]
          exporters: [otlp]
```

1 此端点指向使用 Tempo Operator 部署的 **<example>** TempoStack 实例的网关。

2. 创建用于运行应用程序的服务帐户。

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: otel-collector-sidecar
```

3. 为某些处理器所需的权限创建集群角色。

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: otel-collector-sidecar
rules:
  (1)
- apiGroups: ["config.openshift.io"]
  resources: ["infrastructures", "infrastructures/status"]
  verbs: ["get", "watch", "list"]
```

1 **resourcedetectionprocessor** 需要基础架构和基础架构/状态的权限。

4. 创建 **ClusterRoleBinding** 来为服务帐户设置权限。

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: otel-collector-sidecar
subjects:
- kind: ServiceAccount
  name: otel-collector-deployment
  namespace: otel-collector-example
roleRef:
  kind: ClusterRole
  name: otel-collector
  apiGroup: rbac.authorization.k8s.io
```

5. 将 OpenTelemetry Collector 部署为 sidecar。

6. 通过从 **Deployment** 对象中删除 **"sidecar.jaegertracing.io/inject": "true"** 注解，从应用程序中删除注入的 Jaeger Agent。

7. 通过将 **sidecar.opentelemetry.io/inject: "true"** 注解添加到 **Deployment** 对象的 **.spec.template.metadata.annotations** 字段来启用 OpenTelemetry sidecar 自动注入。

8. 使用为应用程序部署创建的服务帐户，以允许处理器获取正确的信息并将其添加到您的追踪中。

7.2. 从分布式追踪平台 (JAEGER) 迁移到没有 SIDECAR 的红帽构建的 OPENTELEMETRY

您可以从分布式追踪平台(Jaeger)迁移到没有 sidecar 部署的红帽构建的 OpenTelemetry。

先决条件

- 在集群中使用 Red Hat OpenShift distributed tracing Platform (Jaeger)。
- 已安装红帽构建的 OpenTelemetry。

流程

1. 配置 OpenTelemetry Collector 部署。

2. 创建部署 OpenTelemetry Collector 的项目。

```
apiVersion: project.openshift.io/v1
kind: Project
metadata:
  name: observability
```

3. 创建用于运行 OpenTelemetry Collector 实例的服务帐户。

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: otel-collector-deployment
  namespace: observability
```

4. 创建集群角色，以设置处理器所需的权限。

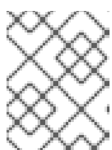
```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: otel-collector
rules:
  1
  2
- apiGroups: ["", "config.openshift.io"]
  resources: ["pods", "namespaces", "infrastructures", "infrastructures/status"]
  verbs: ["get", "watch", "list"]
```

1. **k8sattributesprocessor** 需要 **pods** 和 **namespaces** 资源的权限。
2. **resourcedetectionprocessor** 需要 **infrastructures** 和 **infrastructures/status** 的权限。

5. 创建 ClusterRoleBinding 来为服务帐户设置权限。

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: otel-collector
subjects:
- kind: ServiceAccount
  name: otel-collector-deployment
  namespace: observability
roleRef:
  kind: ClusterRole
  name: otel-collector
  apiGroup: rbac.authorization.k8s.io
```

6. 创建 OpenTelemetry Collector 实例。



注意

此收集器会将 trace 导出至 TempoStack 实例。您必须使用 Red Hat Tempo Operator 创建 TempoStack 实例，并放在正确的端点中。

```
apiVersion: opentelemetry.io/v1alpha1
kind: OpenTelemetryCollector
metadata:
  name: otel
  namespace: observability
spec:
  mode: deployment
  serviceAccount: otel-collector-deployment
  config: |
    receivers:
      jaeger:
        protocols:
          grpc:
          thrift_binary:
          thrift_compact:
          thrift_http:
    processors:
      batch:
      k8sattributes:
      memory_limiter:
        check_interval: 1s
        limit_percentage: 50
        spike_limit_percentage: 30
      resourcedetection:
        detectors: [openshift]
    exporters:
      otlp:
        endpoint: "tempo-example-gateway:8090"
        tls:
          insecure: true
    service:
      pipelines:
        traces:
```

```
      receivers: [jaeger]
      processors: [memory_limiter, k8sattributes, resourcedetection, batch]
      exporters: [otlp]
```

7. 将追踪端点指向 OpenTelemetry Operator。

8. 如果您要将 trace 直接从应用程序导出到 Jaeger，请将 API 端点从 Jaeger 端点改为 OpenTelemetry Collector 端点。

使用带有 Golang 的 jaegerexporter 导出 trace 的示例

```
exp, err := jaeger.New(jaeger.WithCollectorEndpoint(jaeger.WithEndpoint(url))) (1)
```

(1) URL 指向 OpenTelemetry Collector API 端点。

第 8 章 更新红帽构建的 OPENTELEMETRY

对于版本升级，Red Hat build of OpenTelemetry Operator 使用 Operator Lifecycle Manager (OLM)，它控制集群中的 Operator 的安装、升级和基于角色的访问控制(RBAC)。

OLM 默认在 OpenShift Container Platform 中运行。OLM 可以查询可用的 Operator 以及已安装的 Operator 的升级。

当红帽构建的 OpenTelemetry Operator 升级到新版本时，它会扫描运行 OpenTelemetry Collector 实例，并将其升级到与 Operator 新版本对应的版本。

8.1. 其他资源

- Operator Lifecycle Manager 概念和资源
- 更新安装的 Operator

第 9 章 删除红帽构建的 OPENTELEMETRY

从 OpenShift Container Platform 集群中删除红帽构建的 OpenTelemetry 步骤如下：

1. 关闭红帽构建的 OpenTelemetry pod。
2. 删除任何 OpenTelemetryCollector 实例。
3. 删除 OpenTelemetry Operator 的红帽构建。

9.1. 使用 WEB 控制台删除 OPENTELEMETRY COLLECTOR 实例

您可以在 web 控制台的 **Administrator** 视图中删除 OpenTelemetry Collector 实例。

先决条件

- 以集群管理员身份使用 **cluster-admin** 角色登录到 web 控制台。
- 对于 Red Hat OpenShift Dedicated，您必须使用具有 **dedicated-admin** 角色的帐户登录。

流程

1. 进入 **Operators** → **Installed Operators** → **Red Hat build of OpenTelemetry Operator**→ **OpenTelemetryInstrumentation** 或 **OpenTelemetryCollector**。

2. 要删除相关实例，请选择 → **Delete …** → **Delete**。
3. 可选：删除红帽构建的 OpenTelemetry Operator。

9.2. 使用 CLI 删除 OPENTELEMETRY COLLECTOR 实例

您可以在命令行中删除 OpenTelemetry Collector 实例。

先决条件

- 集群管理员具有 **cluster-admin** 角色的活跃 OpenShift CLI (**oc**) 会话。

  提示

  - 确保您的 OpenShift CLI (**oc**) 版本为最新版本，并与您的 OpenShift Container Platform 版本匹配。
  - 运行 **oc login**:

    ```
    $ oc login --username=<your_username>
    ```

流程

1. 运行以下命令，获取 OpenTelemetry Collector 实例的名称：

   ```
   $ oc get deployments -n <project_of_opentelemetry_instance>
   ```

2. 运行以下命令来删除 OpenTelemetry Collector 实例：

```
$ oc delete opentelemetrycollectors <opentelemetry_instance_name> -n
<project_of_opentelemetry_instance>
```

3. 可选：删除红帽构建的 OpenTelemetry Operator。

验证

- 要验证成功删除 OpenTelemetry Collector 实例，请再次运行 **oc get deployments** ：

```
$ oc get deployments -n <project_of_opentelemetry_instance>
```

9.3. 其他资源

- 从集群中删除 Operator
- OpenShift CLI 入门