Red Hat Streams for Apache Kafka 2.7

在带有 ZooKeeper 的 RHEL 上使用流 for Apache Kafka

在 Red Hat Enterprise Linux 中配置和管理 Apache Kafka 2.7 的流部署

Last Updated: 2024-05-29

Red Hat Streams for Apache Kafka 2.7 在带有 ZooKeeper 的 RHEL 上使用流 for Apache Kafka

在 Red Hat Enterprise Linux 中配置和管理 Apache Kafka 2.7 的流部署

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

配置使用 Apache Kafka 的 Streams 部署的 operator 和 Kafka 组件来构建大规模消息传递网络。

目录

前言 5

对红帽文档提供反馈 6

第 1 章 APACHE KAFKA 的 STREAMS 概述 7
- 1.1. 使用 KAFKA BRIDGE 与 KAFKA 集群连接 7
- 1.2. 文档惯例 7

第 2 章 FIPS 支持 9
- 2.1. 安装启用了 FIPS 模式的 APACHE KAFKA 的流 9

第 3 章 开始使用 10
- 3.1. 安装环境 10
- 3.2. 下载 APACHE KAFKA 的流 11
- 3.3. 安装 KAFKA 11
- 3.4. 运行单节点 KAFKA 集群 12
- 3.5. 从主题发送和接收信息 13
- 3.6. 停止 APACHE KAFKA 服务的流 14

第 4 章 运行多节点环境 16
- 4.1. 运行多节点 ZOOKEEPER 集群 16
- 4.2. 运行多节点 KAFKA 集群 17
- 4.3. 执行 KAFKA 代理的安全滚动重启 19

第 5 章 为 APACHE KAFKA 配置流 22
- 5.1. 使用标准 KAFKA 配置属性 22
- 5.2. 从环境变量加载配置值 22
- 5.3. 配置 ZOOKEEPER 23
- 5.4. 配置 KAFKA 30

第 6 章 保护对 KAFKA 的访问 33
- 6.1. 侦听器配置 33
- 6.2. TLS 加密 33
- 6.3. 身份验证 35
- 6.4. 授权 46
- 6.5. ZOOKEEPER 身份验证 50
- 6.6. ZOOKEEPER 授权 51
- 6.7. 使用基于 OAUTH 2.0 令牌的身份验证 53
- 6.8. 使用基于 OAUTH 2.0 令牌的授权 77
- 6.9. 使用基于 OPA 策略的授权 81

第 7 章 创建和管理主题 84
- 7.1. 分区和副本 84
- 7.2. 消息保留 84
- 7.3. TOPIC 自动创建 84
- 7.4. 主题删除 85
- 7.5. 主题配置 85
- 7.6. 内部主题 86
- 7.7. 创建主题 86
- 7.8. 列出和描述主题 87
- 7.9. 修改主题配置 88
- 7.10. 删除主题 89

第 8 章 在 KAFKA CONNECT 中使用 APACHE KAFKA 的 STREAMS 91

8.1. 在独立模式中使用 KAFKA 连接 91
8.2. 在分布式模式中使用 KAFKA CONNECT 92
8.3. 管理连接器 94
第 9 章 使用带有 MIRRORMAKER 2 的 APACHE KAFKA 的 STREAMS 99
9.1. 配置主动/主动或主动/被动模式 99
9.2. 配置 MIRRORMAKER 2 连接器 100
9.3. 连接器生成者和消费者配置 105
9.4. 指定最大任务数 106
9.5. ACL 规则同步 106
9.6. 在专用模式下运行 MIRRORMAKER 2 107
9.7. (DEPRECTAED)在旧模式中使用 MIRRORMAKER 2 110
第 10 章 为 KAFKA 组件配置日志记录 112
10.1. 配置 KAFKA 日志记录属性 112
10.2. 为 KAFKA 代理日志记录器动态更改日志记录级别 113
10.3. 动态更改 KAFKA CONNECT 和 MIRRORMAKER 2 的日志记录级别 114
第 11 章 使用 KAFKA 静态配额插件对代理设置限制 117
第 12 章 添加和删除 KAFKA 代理和 ZOOKEEPER 节点 119
12.1. 通过添加或删除代理来扩展集群 119
12.2. 在 ZOOKEEPER 集群中添加节点 119
12.3. 从 ZOOKEEPER 集群中删除节点 120
第 13 章 使用 CRUISE CONTROL 进行集群重新平衡 122
13.1. CRUISE CONTROL 组件和功能 122
13.2. 下载 CRUISE CONTROL 123
13.3. 部署 CRUISE CONTROL METRICS REPORTER 124
13.4. 配置并启动 CRUISE CONTROL 125
13.5. 优化目标概述 127
13.6. 优化提议概述 129
13.7. 重新平衡性能调优概述 133
13.8. CRUISE CONTROL 配置 135
13.9. 生成优化建议 137
13.10. 批准优化提议 141
13.11. 停止活跃的集群重新平衡 142
第 14 章 使用 CRUISE CONTROL 修改主题复制因素 144
第 15 章 使用分区重新分配工具 145
15.1. 分区重新分配工具概述 145
15.2. 添加代理后重新分配分区 148
15.3. 在删除代理前重新分配分区 150
15.4. 更改主题的复制因素 152
第 16 章 设置分布式追踪 155
16.1. 流程概述 155
16.2. 追踪选项 156
16.3. 用于追踪的环境变量 156
16.4. 为 KAFKA CONNECT 启用追踪 157
16.5. 为 MIRRORMAKER 2 启用追踪 158
16.6. 为 MIRRORMAKER 启用追踪 158
16.7. 初始化 KAFKA 客户端的追踪 159
16.8. 用于追踪的制作者和消费者的工具 161

16.9. 为追踪检测 KAFKA STREAMS 应用程序 162
16.10. 使用 OPENTELEMETRY 指定追踪系统 164
16.11. 为 OPENTELEMETRY 指定自定义 SPAN 名称 164

第 17 章 使用 KAFKA EXPORTER 166
17.1. 消费者滞后 166
17.2. KAFKA EXPORTER 警报规则示例 166
17.3. KAFKA EXPORTER 指标 167
17.4. 运行 KAFKA EXPORTER 168
17.5. 在 GRAFANA 中显示 KAFKA EXPORTER 指标 170

第 18 章 升级 APACHE KAFKA 和 KAFKA 的流 171
18.1. 升级先决条件 171
18.2. 更新 KAFKA 版本 171
18.3. 升级客户端的策略 172
18.4. 升级 KAFKA 代理和 ZOOKEEPER 172
18.5. 升级 KAFKA 组件 175

第 19 章 使用 JMX 监控集群 178
19.1. 启用 JMX 代理 178
19.2. 禁用 JMX 代理 178
19.3. 指标命名约定 179
19.4. 分析 KAFKA JMX 指标以进行故障排除 180

附录 A. 使用您的订阅 186
访问您的帐户 186
激活订阅 186
下载 Zip 和 Tar 文件 186
使用 DNF 安装软件包 186

前言

对红帽文档提供反馈

我们感谢您对我们文档的反馈。

要改进，创建一个 JIRA 问题并描述您推荐的更改。提供尽可能多的详细信息，以便我们快速解决您的请求。

前提条件

- 您有红帽客户门户网站帐户。此帐户可让您登录到 Red Hat Jira Software 实例。
  如果您没有帐户，系统会提示您创建一个帐户。

流程

1. 点以下内容： Create issue。
2. 在 **Summary** 文本框中输入问题的简短描述。
3. 在 **Description** 文本框中提供以下信息：
   - 找到此问题的页面的 URL。
   - 有关此问题的详细描述。
     您可以将信息保留在任何其他字段中的默认值。
4. 添加 reporter 名称。
5. 点 **Create** 将 JIRA 问题提交到文档团队。

感谢您花时间来提供反馈。

第 1 章 APACHE KAFKA 的 STREAMS 概述

AMQ 流支持基于 Apache Kafka 项目的高可扩展、分布式和高性能数据流。

主要组件包括：

Kafka Broker
: 负责将记录从生成客户端到使用客户端的消息传递代理。

Kafka Streams API
: 用于编写 *流处理器* 应用的 API。

Producer 和 Consumer APIs
: 基于 Java 的 API，用于向 Kafka 代理生成和使用信息。

Kafka Bridge
: Apache Kafka Bridge 的 Streams 提供了一个 RESTful 接口，它允许基于 HTTP 的客户端与 Kafka 集群交互。

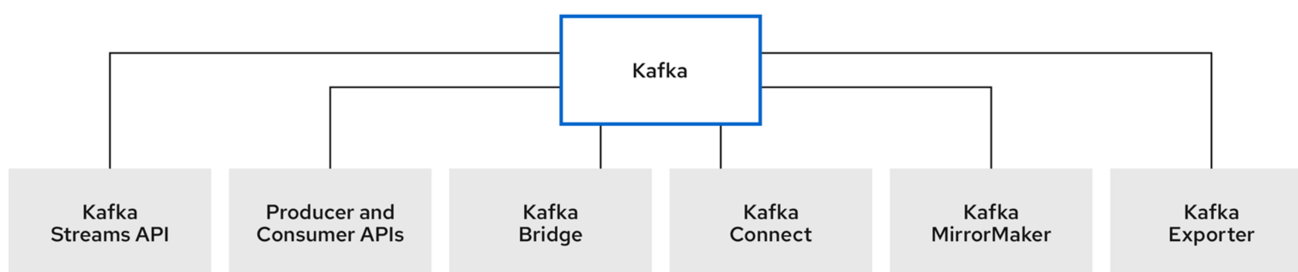
Kafka Connect
: 使用 *Connector* 插件在 Kafka 代理和其他系统间流传输数据的工具包。

Kafka MirrorMaker
: 在两个 Kafka 集群或数据中心之间复制数据。

Kafka Exporter
: 用于监控 Kafka 指标数据的导出器。

Kafka 代理的集群是 hub 连接所有这些组件。

图 1.1. Apache Kafka 架构流



1.1. 使用 KAFKA BRIDGE 与 KAFKA 集群连接

您可以使用 Apache Kafka Bridge API 的 Streams 来创建和管理消费者，并通过 HTTP 而不是原生 Kafka 协议发送和接收记录。

设置 Kafka Bridge 时，您可以配置对 Kafka 集群的 HTTP 访问。然后，您可以使用 Kafka Bridge 来生成和消费来自集群的消息，以及通过其 REST 接口执行其他操作。

其他资源

- 有关安装和使用 Kafka Bridge 的详情，请参考使用 Apache Kafka Bridge 的 Streams 。

1.2. 文档惯例

user-replaced 值

用户替换的值（也称为 *可替换值*）以尖括号(< >)一同显示。下划线(_)用于多词语值。如果值引用代码或命令，也使用 **monospace**。

例如，以下代码显示 < **bootstrap_address>**; 和 < **topic_name** > 必须替换为您自己的地址和主题名称：

```
bin/kafka-console-consumer.sh --bootstrap-server <broker_host>:<port> --topic <topic_name> --
from-beginning
```

第 2 章 FIPS 支持

联邦信息处理标准(FIPS)是计算机安全和互操作性的标准。要将 FIPS 与 Apache Kafka 搭配使用，您必须在系统中安装了 FIPS 兼容的 OpenJDK (Open Java Development Kit)。如果您的 RHEL 系统启用了 FIPS，OpenJDK 会在为 Apache Kafka 运行 Streams 时自动切换到 FIPS 模式。这样可确保 Apache Kafka 的 Streams 使用 OpenJDK 提供的 FIPS 兼容安全库。

最小密码长度

在 FIPS 模式下运行时，SCRAM-SHA-512 密码至少需要 32 个字符。如果您的 Kafka 集群带有使用小于 32 个字符的密码长度的自定义配置，则需要更新您的配置。如果您有任何密码少于 32 个字符的用户，则需要重新生成具有所需长度的密码。

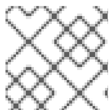
其他资源

- 什么是联邦信息处理标准(FIPS)

2.1. 安装启用了 FIPS 模式的 APACHE KAFKA 的流

在 RHEL 上安装 Apache Kafka 前启用 FIPS 模式。红帽建议安装启用了 FIPS 模式的 RHEL，而不是在以后启用 FIPS 模式。在安装过程中启用 FIPS 模式可确保系统使用 FIPS 批准的算法生成所有的密钥，并持续监控测试。

在 FIPS 模式下运行 RHEL 时，您必须确保 Apache Kafka 配置的 Streams 与 FIPS 兼容。另外，您的 Java 实现还必须与 FIPS 兼容。

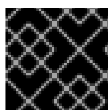


注意

在 FIPS 模式下在 RHEL 上运行 Apache Kafka 的 Streams 需要符合 FIPS 的 JDK。

流程

1. 在 FIPS 模式中安装 RHEL。
   如需更多信息，请参阅 RHEL 文档中的 安全强化的信息。
2. 继续安装 Apache Kafka 的 Streams。
3. 将 Apache Kafka 的流配置为使用 FIPS 兼容算法和协议。
   如果使用，请确保以下配置兼容：
   - JDK 框架必须支持 SSL 密码套件和 TLS 版本。
   - SCRAM-SHA-512 密码必须至少为 32 个字符。

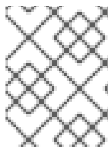


重要

请确定您的安装环境和 Apache Kafka 配置的 Streams 保持合规，因为 FIPS 要求改变。

第 3 章 开始使用

Apache Kafka 的流在 ZIP 文件中发布，其中包含 Kafka 组件的安装工件。



注意

Kafka Bridge 有单独的安装文件。有关安装和使用 Kafka Bridge 的详情，请参考使用 Apache Kafka Bridge 的 Streams。

3.1. 安装环境

Apache Kafka 的流在 Red Hat Enterprise Linux 上运行。主机（节点）可以是物理或虚拟虚拟机(VM)。使用流为 Apache Kafka 提供的安装文件来安装 Kafka 组件。您可以在单节点或多节点环境中安装 Kafka。

单节点环境

单节点 Kafka 集群在单个主机上运行 Kafka 组件实例。此配置不适用于生产环境。

多节点环境

多节点 Kafka 集群在多个主机上运行 Kafka 组件实例。

建议您在单独的主机上运行 Kafka 和其他 Kafka 组件，如 Kafka Connect。通过以这种方式运行组件，可以更轻松地维护和升级每个组件。

Kafka 客户端使用 **bootstrap.servers** 配置属性建立与 Kafka 集群的连接。如果使用 Kafka Connect，例如 Kafka Connect 配置属性必须包含 **bootstrap.servers** 值，用于指定运行 Kafka 代理的主机的主机名和端口。如果 Kafka 集群在多个带有多个 Kafka 代理的主机上运行，您可以为每个代理指定一个主机名和端口。每个 Kafka 代理都由一个 **node.id** 标识。

3.1.1. 数据存储注意事项

有效的数据存储基础架构对于 Apache Kafka Streams 的最佳性能至关重要。

块存储是必需的。文件存储（如 NFS）无法用于 Kafka。

为您的块存储选择以下选项之一：

- 基于云的块存储解决方案，如 Amazon Elastic Block Store (EBS)
- 本地存储
- 由 *光纤通道或 iSCSI*等协议访问的存储区域网络(SAN)卷

3.1.2. 文件系统

Kafka 使用文件系统来存储信息。Apache Kafka 的流与 XFS 和 ext4 文件系统兼容，它们通常与 Kafka 一起使用。在选择和设置文件系统时，请考虑部署的底层架构和要求。

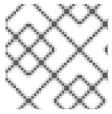
如需更多信息，请参阅 Kafka 文档中的 Filesystem Selection。

3.1.3. Apache Kafka 和 ZooKeeper 存储

为 Apache Kafka 和 ZooKeeper 使用单独的磁盘。

Kafka 支持 JBOD (Just a Bunch of Disks)存储，这是多个磁盘或卷的数据存储配置。JBOD 为 Kafka 代理提供更高的数据存储。它还可以提高性能。

虽然使用固态驱动器 (SSD) 并不是必须的，但它可以在大型集群中提高 Kafka 的性能，其中数据会异步发送到多个主题，并从多个主题接收。SSD 与 ZooKeeper 特别有效，这需要快速、低延迟数据访问。



注意

您不需要置备复制存储，因为 Kafka 和 ZooKeeper 都有内置数据复制。

3.2. 下载 APACHE KAFKA 的流

可以从红帽网站下载 Apache Kafka 的 ZIP 文件发布。您可以从 Apache Kafka 软件下载页面的 Streams for Apache Kafka 下载最新版本的 Red Hat Streams for Apache Kafka。

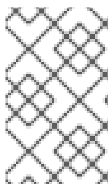
- 对于 Kafka 和其他 Kafka 组件，下载 **amq-streams-<version>-bin.zip** 文件
- 对于 Kafka Bridge，下载 **amq-streams-<version>-bridge-bin.zip** 文件。
  有关安装说明，请参阅使用 Apache Kafka Bridge 的流 。

3.3. 安装 KAFKA

使用 Apache Kafka ZIP 文件的 Streams 在 Red Hat Enterprise Linux 上安装 Kafka。您可以在单节点或多节点环境中安装 Kafka。在此过程中，单个 Kafka 代理和 ZooKeeper 实例安装在单一主机上（节点）。

Apache Kafka 安装文件流包括运行其他 Kafka 组件的二进制文件，如 Kafka Connect、Kafka MirrorMaker 2 和 Kafka Bridge。在单节点环境中，您可以从安装 Kafka 的同一主机上运行这些组件。但是，我们建议您添加安装文件并在单独的主机上运行其他 Kafka 组件。

Apache ZooKeeper 为高度可靠的分布式协调提供集群协调服务。Kafka 使用 ZooKeeper 存储配置数据和集群协调。在运行 Kafka 之前，ZooKee 集群必须就绪。



注意

如果您使用多节点环境，您可以在多个主机上安装 Kafka 代理和 ZooKeeper 实例。对每个主机重复安装步骤。要识别每个 ZooKeeper 实例和代理，您可以在配置中添加唯一 ID。更多信息请参阅 第 4 章 *运行多节点环境*。

先决条件

- 您已下载了 安装文件。
- 您已在 Red Hat Enterprise Linux 发行注记中查看了 Apache Kafka 2.7 的流中支持的配置。
- 以 admin (**root**)用户身份登录 Red Hat Enterprise Linux。

流程

在主机上安装带有 ZooKeeper 的 Kafka。

1. 添加新的 **kafka** 用户和组：

```
groupadd kafka
useradd -g kafka kafka
passwd kafka
```

2. 将 **amq-streams-<version>-bin.zip** 文件的内容提取到 **/opt/kafka** 目录中：

```
unzip amq-streams-<version>-bin.zip -d /opt
mv /opt/kafka*redhat* /opt/kafka
```

3. 将 **/opt/kafka** 目录的所有权更改为 **kafka** 用户：

```
chown -R kafka:kafka /opt/kafka
```

4. 创建用于存储 ZooKeeper 数据的 **/var/lib/zookeeper** 目录，并将其所有权设置为 **kafka** 用户：

```
mkdir /var/lib/zookeeper
chown -R kafka:kafka /var/lib/zookeeper
```

5. 创建用于存储 Kafka 数据的目录 **/var/lib/kafka**，并将其所有权设置为 **kafka** 用户：

```
mkdir /var/lib/kafka
chown -R kafka:kafka /var/lib/kafka
```

现在，您可以作为单节点集群运行 Kafka 的默认配置。

您还可以使用安装在同一主机上运行其他 Kafka 组件，如 Kafka Connect。

要运行其他组件，请使用组件配置中的 **bootstrap.servers** 属性指定要连接到 Kafka 代理的主机名和端口。

指向同一主机上单个 Kafka 代理的 bootstrap 服务器配置示例

```
bootstrap.servers=localhost:9092
```

但是，我们建议在单独的主机上安装并运行 Kafka 组件。

6. （可选）在单独的主机上安装 Kafka 组件。
   a. 重复这些步骤，将安装文件提取到每个主机上的 **/opt/kafka** 目录中。
   b. 添加 **bootstrap.servers** 配置，将组件连接到运行 Kafka 代理的主机（或多节点环境中的主机）。

   指向不同主机上 Kafka 代理的 bootstrap 服务器配置示例

```
bootstrap.servers=kafka0.<host_ip_address>:9092,kafka1.
<host_ip_address>:9092,kafka2.<host_ip_address>:9092
```

   您可以将此配置用于 Kafka Connect、MirrorMaker 2 和 Kafka Bridge。

3.4. 运行单节点 KAFKA 集群

此流程演示了如何对 Apache Kafka 集群运行基本流，它由单个 Apache ZooKeeper 节点和一个 Apache Kafka 节点组成，它们在同一主机上运行。默认配置文件用于 Kafka。

警告

单个用于 Apache Kafka 集群的节点流不提供可靠性和高可用性，且仅适用于开发目的。

先决条件

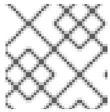
- 在主机上安装了 Apache Kafka 的流

运行集群

1. 为 Kafka 集群生成唯一 ID。
   您可以使用 **kafka-storage** 工具进行此操作：

   ```
   /opt/kafka/bin/kafka-storage.sh random-uuid
   ```

   该命令返回一个 ID。

   注意

   KRaft 模式需要一个集群 ID。

2. 编辑 Kafka 配置文件 **/opt/kafka/config/server.properties**。将 **log.dirs** 选项设置为 **/var/lib/kafka/** ：

   ```
   log.dirs=/var/lib/kafka/
   ```

3. 切换到 **kafka** 用户：

   ```
   su - kafka
   ```

4. 启动 Kafka：

   ```
   /opt/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/config/server.properties
   ```

5. 检查 Kafka 是否正在运行：

   ```
   jcmd | grep kafka
   ```

   返回：

   ```
   process ID kafka.Kafka /opt/kafka/config/server.properties
   ```

3.5. 从主题发送和接收信息

此流程描述了如何启动 Kafka 控制台生成者和消费者客户端，并使用它们发送和接收多个信息。

步骤 1 中会自动创建一个新主题。Topic auto-creation 使用 **auto.create.topics.enable** 配置属性（默认为 **true**）进行控制。另外，您可以在使用集群前配置和创建主题。如需更多信息，请参阅 主题。

先决条件

- 在主机上安装了 Apache Kafka 的流
- Kafka 正在运行

流程

1. 启动 Kafka 控制台制作者并将其配置为将信息发送到新主题：

```
/opt/kafka/bin/kafka-console-producer.sh --broker-list <bootstrap_address> --topic <topic-name>
```

例如：

```
/opt/kafka/bin/kafka-console-producer.sh --broker-list localhost:9092 --topic my-topic
```

2. 在控制台中输入多个信息。按 **Enter** 将每个消息发送到您的新主题：

```
>message 1
>message 2
>message 3
>message 4
```

当 Kafka 自动创建新主题时，您可能会收到主题不存在的警告：

```
WARN Error while fetching metadata with correlation id 39 :
{4-3-16-topic1=LEADER_NOT_AVAILABLE} (org.apache.kafka.clients.NetworkClient)
```

在发送进一步消息后，该警告不应重新显示。

3. 在新的终端窗口中，启动 Kafka 控制台消费者，并将其配置为从您的新主题开始读取消息。

```
/opt/kafka/bin/kafka-console-consumer.sh --bootstrap-server <bootstrap_address> --topic <topic-name> --from-beginning
```

例如：

```
/opt/kafka/bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic my-topic --from-beginning
```

传入的消息显示在消费者控制台中。

4. 切换到制作者控制台并发送其他消息。检查它们是否在使用者控制台中显示。

5. 停止 Kafka 控制台制作者，然后按 **Ctrl+C**。

3.6. 停止 APACHE KAFKA 服务的流

您可以通过运行脚本来停止 Kafka 和 ZooKeeper 服务。所有到 Kafka 和 ZooKeeper 服务的连接都将被终止。

先决条件

- 在主机上安装了 Apache Kafka 的流
- ZooKeeper 和 Kafka 已启动并正在运行

流程

1. 停止 Kafka 代理。

```
su - kafka
/opt/kafka/bin/kafka-server-stop.sh
```

2. 确认 Kafka 代理已停止。

```
jcmd | grep kafka
```

3. 停止 ZooKeeper。

```
su - kafka
/opt/kafka/bin/zookeeper-server-stop.sh
```

第 4 章 运行多节点环境

多节点环境由多个节点组成，它们作为集群运行。您可以有一个复制 ZooKeeper 节点和代理节点集群，并在代理间复制主题。

多节点环境提供稳定性和可用性。

4.1. 运行多节点 ZOOKEEPER 集群

将 ZooKeeper 配置为多节点集群。

先决条件

- Apache Kafka 的流安装在用作 ZooKeeper 集群节点的所有主机上。

运行集群

1. 在 **/var/lib/zookeeper/** 中创建 **myid** 文件。为第一个 ZooKeeper 节点输入 ID **1**，为第二个 ZooKeeper 节点输入 **2**，以此类推。

```
su - kafka
echo "<NodeID>" > /var/lib/zookeeper/myid
```

例如：

```
su - kafka
echo "1" > /var/lib/zookeeper/myid
```

2. 编辑 ZooKeeper **/opt/kafka/config/zookeeper.properties** 配置文件：
   - 将选项 **dataDir** 设置为 **/var/lib/zookeeper/**。
   - 配置 **initLimit** 和 **syncLimit** 选项。
   - 配置 **reconfigEnabled** 和 **standaloneEnabled** 选项。
   - 添加所有 ZooKeeper 节点的列表。该列表还应包含当前的节点。

   具有五个成员的 ZooKeeper 集群节点配置示例

```
tickTime=2000
dataDir=/var/lib/zookeeper/
initLimit=5
syncLimit=2
reconfigEnabled=true
standaloneEnabled=false

server.1=172.17.0.1:2888:3888:participant;172.17.0.1:2181
server.2=172.17.0.2:2888:3888:participant;172.17.0.2:2181
server.3=172.17.0.3:2888:3888:participant;172.17.0.3:2181
server.4=172.17.0.4:2888:3888:participant;172.17.0.4:2181
server.5=172.17.0.5:2888:3888:participant;172.17.0.5:2181
```

3. 使用默认配置文件启动 ZooKeeper。

```
su - kafka
/opt/kafka/bin/zookeeper-server-start.sh -daemon /opt/kafka/config/zookeeper.properties
```

4. 验证 ZooKeeper 是否正在运行。

```
jcmd | grep zookeeper
```

返回：

```
number org.apache.zookeeper.server.quorum.QuorumPeerMain
/opt/kafka/config/zookeeper.properties
```

5. 在集群的所有节点上重复此步骤。
6. 使用 **ncat** 实用程序向每个节点发送 **stat** 命令，以验证所有节点是否都是集群的成员。

使用 ncat stat 检查节点状态

```
echo stat | ncat localhost 2181
```

要使用四个字母单词命令，如 **stat**，您需要在 **zookeeper.properties** 中指定 **4lw.commands.whitelist** prerequisites。

输出显示节点是 **领导机** 或后续 。

ncat 命令的输出示例

```
ZooKeeper version: 3.4.13-2d71af4dbe22557fda74f9a9b4309b15a7487f03, built on
06/29/2018 00:39 GMT
Clients:
 /0:0:0:0:0:0:0:1:59726[0](queued=0,recved=1,sent=0)

Latency min/avg/max: 0/0/0
Received: 2
Sent: 1
Connections: 1
Outstanding: 0
Zxid: 0x200000000
Mode: follower
Node count: 4
```

4.2. 运行多节点 KAFKA 集群

将 Kafka 配置为多节点集群。

先决条件

- 每个主机上安装了 Apache Kafka 的流，且配置文件可用。
- 一个 ZooKeeper 集群已被配置并运行。

运行集群

对于 Apache Kafka 集群的 Streams 中的每个 Kafka 代理：

1. 编辑 **/opt/kafka/config/server.properties** Kafka 配置文件，如下所示：
   - 在第一个代理中，将 **broker.id** 设置为 **0**，在第二个代理中，将其设置为 **1**，以此类推。
   - 在 **zookeeper.connect** 选项中配置连接到 ZooKeeper 的详情。
   - 配置 Kafka 侦听程序。
   - 设置提交日志应存储在 **logs.dir** 目录中的目录。
     在这里，我们看到 Kafka 代理的示例配置：

     ```
     broker.id=0
     zookeeper.connect=zoo1.my-domain.com:2181,zoo2.my-domain.com:2181,zoo3.my-domain.com:2181
     listeners=REPLICATION://:9091,PLAINTEXT://:9092
     listener.security.protocol.map=PLAINTEXT:PLAINTEXT,REPLICATION:PLAINTEXT
     inter.broker.listener.name=REPLICATION
     log.dirs=/var/lib/kafka
     ```

     在典型的安装中，每个 Kafka 代理在相同的硬件上运行，只有 **broker.id** 配置属性在每个代理配置间有所不同。

2. 使用默认配置文件启动 Kafka 代理。

   ```
   su - kafka
   /opt/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/config/server.properties
   ```

3. 验证 Kafka 代理是否正在运行。

   ```
   jcmd | grep Kafka
   ```

   返回：

   ```
   process ID kafka.Kafka /opt/kafka/config/server.properties
   ```

4. 使用 **ncat** 实用程序将 **dump** 命令发送到其中一个 ZooKeeper 节点，以验证所有节点是否为 Kafka 集群的成员。

   使用 ncat dump 检查 ZooKeeper 中注册的所有 Kafka 代理

   ```
   echo dump | ncat zoo1.my-domain.com 2181
   ```

   要使用包括四个字母的命令，如 **dump**，需要在 **zookeeper.properties** 中指定 **4lw.commands.whitelist=***。

   输出必须包含您刚才配置和启动的所有 Kafka 代理。

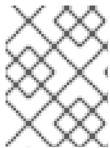
   具有 3 个节点的 Kafka 集群的 ncat 命令的输出示例

   ```
   SessionTracker dump:
   org.apache.zookeeper.server.quorum.LearnerSessionTracker@28848ab9
   ephemeral nodes dump:
   ```

```
Sessions with Ephemerals (3):
0x20000015dd00000:
	/brokers/ids/1
0x10000015dc70000:
	/controller
	/brokers/ids/0
0x10000015dc70001:
	/brokers/ids/2
```

4.3. 执行 KAFKA 代理的安全滚动重启

此流程演示了如何在多节点集群中安全滚动重启代理。在升级或更改 Kafka 集群配置属性后，通常需要滚动重启。



注意

有些代理配置不需要重启代理。如需更多信息，请参阅 Apache Kafka 文档中的 更新 代理配置。

执行代理重启后，检查复制主题分区，以确保副本分区已捕获。

要实现不丢失可用性的正常重启，请确保复制主题，并且至少复制主题(**min.insync.replicas**)副本是同步的。**min.insync.replicas** 配置决定了必须确认写入被视为成功的最小副本数。

对于多节点集群，标准方法是具有最少为 3 的主题复制因素，并将最小 in-sync 副本设置为复制因素减 1。对于数据的持续时间，如果您在生成者配置中使用了 **acks=all**，在重新启动下一个代理前需要检查您重启的代理是否与所有它所同步的分区保持同步。

单节点集群在重启后不可用，因为所有分区都在同一个代理中。

先决条件

- 一个 ZooKeeper 集群已被配置并运行。
- Kafka 集群按预期运行。
  检查有复制的分区或影响代理操作的任何其他问题。此流程中的步骤描述了如何检查有重复分区。

流程

在每个 Kafka 代理上执行以下步骤。在继续执行下一个代理前，在第一个代理上完成这些步骤。在最后一个活跃控制器的代理上执行步骤。否则，活动控制器需要在多个重启时更改。

1. 停止 Kafka 代理：

```
/opt/kafka/bin/kafka-server-stop.sh
```

2. 对在完成后需要重启的代理配置进行任何更改。
   如需更多信息，请参阅以下内容：
   - 配置 Kafka
   - 升级 Kafka 代理和 ZooKeeper
3. 重启 Kafka 代理：

```
/opt/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/config/server.properties
```

4. 检查 Kafka 是否正在运行：

```
jcmd | grep kafka
```

返回：

```
process ID kafka.Kafka /opt/kafka/config/server.properties
```

5. 使用 **ncat** 实用程序将 **dump** 命令发送到其中一个 ZooKeeper 节点，以验证所有节点是否为 Kafka 集群的成员。

使用 ncat dump 检查 ZooKeeper 中注册的所有 Kafka 代理

```
echo dump | ncat zoo1.my-domain.com 2181
```

要使用包括四个字母的命令，如 **dump**，需要在 **zookeeper.properties** 中指定 **4lw.commands.whitelist=***。

输出必须包含您启动的 Kafka 代理。

具有 3 个节点的 Kafka 集群的 ncat 命令的输出示例

```
SessionTracker dump:
org.apache.zookeeper.server.quorum.LearnerSessionTracker@28848ab9
ephemeral nodes dump:
Sessions with Ephemerals (3):
0x20000015dd00000:
        /brokers/ids/1
0x10000015dc70000:
        /controller
        /brokers/ids/0
0x10000015dc70001:
        /brokers/ids/2
```

6. 等待直到代理有 0 个重复分区。您可以从命令行检查或使用指标。

- 使用带有 **--under-replicated-partitions** 参数的 **kafka-topics.sh** 命令：

```
/opt/kafka/bin/kafka-topics.sh --bootstrap-server <bootstrap_address> --describe --under-replicated-partitions
```

例如：

```
/opt/kafka/bin/kafka-topics.sh --bootstrap-server localhost:9092 --describe --under-replicated-partitions
```

命令提供了集群中带有重复分区的主题列表。

带有复制分区的主题

```
Topic: topic3 Partition: 4 Leader: 2 Replicas: 2,3 Isr: 2
```

```
Topic: topic3 Partition: 5 Leader: 3 Replicas: 1,2 Isr: 1
Topic: topic1 Partition: 1 Leader: 3 Replicas: 1,3 Isr: 3
# …
```

如果 ISR（同步副本）计数小于副本数，则会列出 in-replicated 分区。如果没有返回列表，则没有复制的分区。

- 使用 **UnderReplicatedPartitions** 指标：

```
kafka.server:type=ReplicaManager,name=UnderReplicatedPartitions
```

指标提供副本未捕获的分区计数。您等到计数为零。

提示

当主题有一个或多个重复分区时，请使用 Kafka Exporter 创建警报。

重启时检查日志

如果代理无法启动，请检查应用程序日志中的信息。您还可以检查代理关闭的状态，并在 **/opt/kafka/logs/server.log** 应用程序日志中重启。

日志以成功关闭代理

```
# ...
[2022-06-08 14:32:29,885] INFO Terminating process due to signal SIGTERM
(org.apache.kafka.common.utils.LoggingSignalHandler)
[2022-06-08 14:32:29,886] INFO [KafkaServer id=0] shutting down (kafka.server.KafkaServer)
[2022-06-08 14:32:29,887] INFO [KafkaServer id=0] Starting controlled shutdown
(kafka.server.KafkaServer)
[2022-06-08 14:32:29,896] INFO [KafkaServer id=0] Controlled shutdown request returned
successfully after 6ms (kafka.server.KafkaServer)
# ...
```

成功重启代理的日志

```
# ...
[2022-06-08 14:39:35,245] INFO [KafkaServer id=0] started (kafka.server.KafkaServer)
# ...
```

其他资源

- 第 19.4 节 “分析 Kafka JMX 指标以进行故障排除”
- 第 10 章 *为 Kafka 组件配置日志记录*
- Kafka 配置调整

第 5 章 为 APACHE KAFKA 配置流

使用 Kafka 和 ZooKeeper 属性文件为 Apache Kafka 配置流。

ZooKeeper

/kafka/config/zookeeper.properties

Kafka

/kafka/config/server.properties

属性文件是 Java 格式，每个属性位于一个独立的行中，使用一些格式：

```
<option> = <value>
```

以 **#** 或 **!** 开头的行将被视为注释，并将由 Apache Kafka 组件的 Streams 忽略。

```
# This is a comment
```

可以使用 \ 在换行 / carriage 返回前直接将值分成多行。

```
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required \
    username="bob" \
    password="bobs-password";
```

在属性文件中保存更改后，您需要重启 Kafka 代理或 ZooKeeper。在多节点环境中，您需要在集群中的每个节点上重复该过程。

5.1. 使用标准 KAFKA 配置属性

使用标准 Kafka 配置属性来配置 Kafka 组件。

属性提供控制和调整以下 Kafka 组件配置的选项：

- 代理（Broker）
- topics
- 生产者、消费者和管理客户端
- Kafka Connect
- Kafka Streams

代理和客户端参数包括配置授权、身份验证和加密的选项。

如需有关 Kafka 配置属性以及如何使用属性调整部署的更多信息，请参阅以下指南：

- Kafka 配置属性
- Kafka 配置调整

5.2. 从环境变量加载配置值

使用 Environment Variables Configuration Provider 插件从环境变量加载配置数据。您可以使用 Environment Variables Configuration Provider，例如从环境变量加载证书或 JAAS 配置。

您可以使用供应商为所有 Kafka 组件加载配置数据，包括生成者和消费者。例如，使用供应商为 Kafka Connect 连接器配置提供凭证。

先决条件

- 每个主机上安装了 Apache Kafka 的流，且配置文件可用。
- 环境变量配置提供程序 JAR 文件。
  JAR 文件可从 Apache Kafka 归档 的 Streams 中提供。

流程

1. 将 Environment Variables Configuration Provider JAR 文件添加到 Kafka **libs** 目录。
2. 在 Kafka 组件的配置属性文件中初始化 Environment Variables Configuration Provider。例如，要初始化 Kafka 的供应商，请将配置添加到 **server.properties** 文件中。

   配置以启用 Environment Variables Configuration Provider

   ```
   config.providers.env.class=org.apache.kafka.common.config.provider.EnvVarConfigProvider
   ```

3. 添加配置到属性文件，以从环境变量加载数据。

   配置以从环境变量加载数据

   ```
   option=${env:<MY_ENV_VAR_NAME>}
   ```

   使用大写或大写环境变量命名约定，如 **MY_ENV_VAR_NAME**。

4. 保存更改。
5. 重启 Kafka 组件。
   有关在多节点集群中重启代理的详情，请参考 第 4.3 节 "执行 Kafka 代理的安全滚动重启"。

5.3. 配置 ZOOKEEPER

Kafka 使用 ZooKeeper 存储配置数据和集群协调。强烈建议您运行复制 ZooKeeper 实例的集群。

5.3.1. 基本配置

最重要的 ZooKeeper 配置选项是：

tickTime

ZooKeeper 的基本时间单位（以毫秒为单位）。它用于心跳和会话超时。例如，最小会话超时将是两个 ticks。

dataDir

ZooKeeper 存储其事务日志及其内存数据库的快照的目录。这应该设置为在安装过程中创建的 **/var/lib/zookeeper/** 目录。

clientPort

客户端可以连接的端口号。默认值为 **2181**。

名为 **config/zookeeper.properties** 的 ZooKeeper 配置文件示例位于 Apache Kafka 安装目录中的 Streams 中。建议您将 **dataDir** 目录放在单独的磁盘设备中，以最小化 ZooKeeper 中的延迟。

zookeeper 配置文件应该位于 **/opt/kafka/config/zookeeper.properties** 中。可在下面找到配置文件的基本示例。该配置文件必须可由 **kafka** 用户读取。

```
tickTime=2000
dataDir=/var/lib/zookeeper/
clientPort=2181
```

5.3.2. zookeeper 集群配置

在大多数生产环境中，建议您部署复制 ZooKeeper 实例的集群。稳定且高度可用的 ZooKeeper 集群对于运行可靠的 ZooKeeper 服务非常重要。ZooKeeper 集群也称为 *ensembles*。

ZooKeeper 集群通常由奇数个节点组成。ZooKeeper 要求集群中的大多数节点都已启动并在运行。例如：

- 在具有三个节点的集群中，至少有两个节点必须启动并在运行。这意味着它可以容许一个节点被停机。
- 在由五个节点组成的集群中，必须至少有三个节点可用。这意味着它可以容许两个节点处于 down 状态。
- 在由 7 个节点组成的集群中，必须至少有四个节点可用。这意味着它可以容许三个节点被停机。

在 ZooKeeper 集群中拥有更多节点可以提供更好的弹性和可靠性。

ZooKeeper 可以在带有偶数节点的集群中运行。但是，额外的节点不会增加集群的弹性。具有四个节点的集群至少需要三个节点可用，且只能容忍一个节点被停机。因此，它具有与只有三个节点的集群相同的弹性。

理想情况下，不同的 ZooKeeper 节点应该位于不同的数据中心或网络片段中。增加 ZooKeeper 节点数量会增加集群同步上消耗的工作负载。对于大多数 Kafka 用例，有 3、5 或 7 节点的 ZooKeeper 集群应该足够了。

警告

具有 3 个节点的 ZooKeeper 集群只能容忍 1 个不可用的节点。这意味着，如果在 ZooKeeper 集群的其他节点上进行维护，集群节点会崩溃。

重复的 ZooKeeper 配置支持独立配置支持所有配置选项。为集群配置添加附加选项：

initLimit

允许后续者连接到集群领导机的时间长度。将时间指定为多个 ticks（更多详情请参阅 **tickTime** 选项）。

syncLimit

跟随者可以位于领导机后的时间长度。将时间指定为多个 ticks （更多详情请参阅 **tickTime** 选项）。

reconfigEnabled

启用或禁用动态重新配置。必须启用才能向 ZooKeeper 集群添加或删除服务器。

standaloneEnabled

启用或禁用独立模式，其中 ZooKeeper 只使用一个服务器运行。

除了以上选项外，每个配置文件还应包含应该是 ZooKeeper 集群成员的服务器列表。服务器记录应以 **server.id=hostname:port1:port2** 格式指定，其中：

id

ZooKeeper 集群节点的 ID。

hostname

节点侦听连接的主机名或 IP 地址。

port1

用于集群内通信的端口号。

port2

用于领导选举机制的端口号。

以下是具有三个节点的 ZooKeeper 集群配置文件示例：

```
tickTime=2000
dataDir=/var/lib/zookeeper/
initLimit=5
syncLimit=2
reconfigEnabled=true
standaloneEnabled=false

server.1=172.17.0.1:2888:3888:participant;172.17.0.1:2181
server.2=172.17.0.2:2888:3888:participant;172.17.0.2:2181
server.3=172.17.0.3:2888:3888:participant;172.17.0.3:2181
```

提示

要使用四个字母单词命令，在 **zookeeper.properties** 中指定 **4lw.commands.whitelist=***。

myid 文件

ZooKeeper 集群中的每个节点都必须被分配一个唯一 **ID**。每个节点的 **ID** 必须在 **myid** 文件中配置，并存储在 **dataDir** 文件夹中，如 **/var/lib/zookeeper/**。**myid** 文件应当仅包含一个将写入 **ID** 用作文本的一行。**ID** 可以是从 1 到 255 的任何整数。您必须在每个集群节点上手动创建此文件。使用此文件，每个 ZooKeeper 实例将使用配置文件中的相应 **server.** 行的配置来配置其监听程序。它还将使用所有其他 **server.** 行来识别其他群集成员。

在上例中，有三个节点，因此每个节点都有一个不同的 **myid**，值分别为 **1**、**2**、**3**。

5.3.3. 身份验证

默认情况下，Zoo ZooKeeper 不使用任何类型的身份验证并允许匿名连接。但是，它支持 Java 认证和授权服务(JAAS)，可用于使用简单身份验证和安全层 (SASL) 设置身份验证。ZooKeeper 支持在本地存储的凭证中使用 DIGEST-MD5 SASL 机制进行身份验证。

5.3.3.1. 使用 SASL 进行身份验证

JAAS 使用单独的配置文件进行配置。建议将 JAAS 配置文件放在与 ZooKeeper 配置相同的目录中 (**/opt/kafka/config/**)。推荐的文件名是 **zookeeper-jaas.conf**。当将 ZooKeeper 集群与多个节点搭配使用时，必须在所有集群节点上创建 JAAS 配置文件。

JAAS 使用上下文进行配置。服务器和客户端等单独部分始终使用单独的 *上下文* 进行配置。上下文是一个 *配置选项*，其格式如下：

```
ContextName {
    param1
    param2;
};
```

SASL 身份验证为服务器到服务器通信（实例 ZooKeeper 实例之间的沟通）和客户端到服务器通信（Kafka 和 ZooKeeper 之间相互通信）单独配置。服务器到服务器身份验证只适用于带有多个节点的 ZooKeeper 集群。

服务器到服务器身份验证

对于服务器到服务器身份验证，JAAS 配置文件包含两个部分：

- 服务器配置
- 客户端配置

使用 DIGEST-MD5 SASL 机制时，Qorum **Server** 上下文用于配置身份验证服务器。它必须包含所有用户名，以允许以未加密的形式与其密码连接。第二个上下文 **QuorumLearner** 必须为内置在 ZooKeeper 中的客户端配置。它还包含未加密的形式的密码。以下是 DIGEST-MD5 机制的 JAAS 配置文件示例：

```
QuorumServer {
    org.apache.zookeeper.server.auth.DigestLoginModule required
    user_zookeeper="123456";
};

QuorumLearner {
    org.apache.zookeeper.server.auth.DigestLoginModule required
    username="zookeeper"
    password="123456";
};
```

除了 JAAS 配置文件外，还必须通过指定以下选项在常规 ZooKeeper 配置文件中启用 server-to-server 身份验证：

```
quorum.auth.enableSasl=true
quorum.auth.learnerRequireSasl=true
quorum.auth.serverRequireSasl=true
quorum.auth.learner.loginContext=QuorumLearner
quorum.auth.server.loginContext=QuorumServer
quorum.cnxn.threads.size=20
```

使用 **KAFKA_OPTS** 环境变量将 JAAS 配置文件作为 Java 属性传递给 ZooKeeper 服务器：

```
su - kafka
export KAFKA_OPTS="-Djava.security.auth.login.config=/opt/kafka/config/zookeeper-jaas.conf";
/opt/kafka/bin/zookeeper-server-start.sh -daemon /opt/kafka/config/zookeeper.properties
```

有关服务器到服务器身份验证的更多信息，请参阅 ZooKeeper wiki。

客户端到服务器身份验证

客户端到服务器身份验证配置在与服务器到服务器身份验证相同的 JAAS 文件中。但是，与服务器到服务器身份验证不同，它仅包含服务器配置。配置的客户端部分必须在客户端中进行。有关如何配置 Kafka 代理以使用身份验证连接到 ZooKeeper 的详情，请参考 Kafka 安装 部分。

将服务器上下文添加到 JAAS 配置文件中，以配置客户端到服务器身份验证。对于 DIGEST-MD5 机制，它会配置所有用户名和密码：

```
Server {
    org.apache.zookeeper.server.auth.DigestLoginModule required
    user_super="123456"
    user_kafka="123456"
    user_someoneelse="123456";
};
```

配置 JAAS 上下文后，通过添加以下行在 ZooKeeper 配置文件中启用客户端到服务器身份验证：

```
requireClientAuthScheme=sasl
authProvider.1=org.apache.zookeeper.server.auth.SASLAuthenticationProvider
authProvider.2=org.apache.zookeeper.server.auth.SASLAuthenticationProvider
authProvider.3=org.apache.zookeeper.server.auth.SASLAuthenticationProvider
```

您必须为属于 ZooKeeper 集群的每个服务器添加 **authProvider.<*ID*>** 属性。

使用 **KAFKA_OPTS** 环境变量将 JAAS 配置文件作为 Java 属性传递给 ZooKeeper 服务器：

```
su - kafka
export KAFKA_OPTS="-Djava.security.auth.login.config=/opt/kafka/config/zookeeper-jaas.conf";
/opt/kafka/bin/zookeeper-server-start.sh -daemon /opt/kafka/config/zookeeper.properties
```

有关在 Kafka 代理中配置 ZooKeeper 身份验证的更多信息，请参阅 第 6.5 节 “ZooKeeper 身份验证”。

5.3.3.2. 使用 DIGEST-MD5 启用服务器到服务器身份验证

此流程描述了如何在 ZooKeeper 集群节点间使用 SASL DIGEST-MD5 机制启用身份验证。

先决条件

- 在主机上安装了 Apache Kafka 的流
- ZooKeeper 集群 配置了多个 节点。

启用 SASL DIGEST-MD5 身份验证

1. 在所有 ZooKeeper 节点上，创建或编辑 **/opt/kafka/config/zookeeper-jaas.conf** JAAS 配置文件并添加以下上下文：

```
QuorumServer {
    org.apache.zookeeper.server.auth.DigestLoginModule required
    user_<Username>="<Password>";
};

QuorumLearner {
    org.apache.zookeeper.server.auth.DigestLoginModule required
    username="<Username>"
    password="<Password>";
};
```

在 JAAS 上下文中，用户名和密码都必须相同。例如：

```
QuorumServer {
    org.apache.zookeeper.server.auth.DigestLoginModule required
    user_zookeeper="123456";
};

QuorumLearner {
    org.apache.zookeeper.server.auth.DigestLoginModule required
    username="zookeeper"
    password="123456";
};
```

2. 在所有 ZooKeeper 节点上，编辑 **/opt/kafka/config/zookeeper.properties** ZooKeeper 配置文件并设置以下选项：

```
quorum.auth.enableSasl=true
quorum.auth.learnerRequireSasl=true
quorum.auth.serverRequireSasl=true
quorum.auth.learner.loginContext=QuorumLearner
quorum.auth.server.loginContext=QuorumServer
quorum.cnxn.threads.size=20
```

3. 逐一重启所有 ZooKeeper 节点。要将 JAAS 配置传递给 ZooKeeper，请使用 **KAFKA_OPTS** 环境变量。

```
su - kafka
export KAFKA_OPTS="-Djava.security.auth.login.config=/opt/kafka/config/zookeeper-jaas.conf"; /opt/kafka/bin/zookeeper-server-start.sh -daemon
/opt/kafka/config/zookeeper.properties
```

5.3.3.3. 使用 DIGEST-MD5 启用客户端到服务器身份验证

此流程描述了如何使用 ZooKeeper 客户端和 ZooKeeper 之间的 SASL DIGEST-MD5 机制启用身份验证。

先决条件

- 在主机上安装了 Apache Kafka 的流
- zookeeper 集群 已配置并运行。

启用 SASL DIGEST-MD5 身份验证

1. 在所有 ZooKeeper 节点上，创建或编辑 **/opt/kafka/config/zookeeper-jaas.conf** JAAS 配置文件并添加以下上下文：

```
Server {
    org.apache.zookeeper.server.auth.DigestLoginModule required
    user_super="<SuperUserPassword>"
    user<Username1>_="<Password1>" user<USername2>_="<Password2>";
};
```

超级用户 自动具有 priviledges 管理员。该文件可以包含多个用户，但 Kafka 代理只需要一个额外的用户。Kafka 用户的建议名称为 **kafka**。

以下示例显示了 **客户端到服务器** 身份验证的服务器上下文：

```
Server {
    org.apache.zookeeper.server.auth.DigestLoginModule required
    user_super="123456"
    user_kafka="123456";
};
```

2. 在所有 ZooKeeper 节点上，编辑 **/opt/kafka/config/zookeeper.properties** ZooKeeper 配置文件并设置以下选项：

```
requireClientAuthScheme=sasl
authProvider.<IdOfBroker1>=org.apache.zookeeper.server.auth.SASLAuthenticationProvider
authProvider.<IdOfBroker2>=org.apache.zookeeper.server.auth.SASLAuthenticationProvider
authProvider.<IdOfBroker3>=org.apache.zookeeper.server.auth.SASLAuthenticationProvider
```

必须为作为 ZooKeeper 集群一部分的每个节点添加 **authProvider. *<ID***> 属性。一个三节点 ZooKeeper 集群配置示例必须类似如下：

```
requireClientAuthScheme=sasl
authProvider.1=org.apache.zookeeper.server.auth.SASLAuthenticationProvider
authProvider.2=org.apache.zookeeper.server.auth.SASLAuthenticationProvider
authProvider.3=org.apache.zookeeper.server.auth.SASLAuthenticationProvider
```

3. 逐一重启所有 ZooKeeper 节点。要将 JAAS 配置传递给 ZooKeeper，请使用 **KAFKA_OPTS** 环境变量。

```
su - kafka
export KAFKA_OPTS="-Djava.security.auth.login.config=/opt/kafka/config/zookeeper-jaas.conf"; /opt/kafka/bin/zookeeper-server-start.sh -daemon /opt/kafka/config/zookeeper.properties
```

5.3.4. 授权

ZooKeeper 支持访问控制列表(ACL)来保护存储在其中的数据。Kafka 代理可以自动为他们创建的所有 ZooKeeper 记录配置 ACL 权限，因此没有其他 ZooKeeper 用户都可以修改它们。

有关在 Kafka 代理中启用 ZooKeeper ACL 的详情，请参考 第 6.6 节 “ZooKeeper 授权”。

5.3.5. TLS

ZooKeeper 支持 TLS 进行加密或身份验证。

5.3.6. 其他配置选项

您可以根据您的用例设置以下额外 ZooKeeper 配置选项：

maxClientCnxns
: 到 ZooKeeper 集群的单个成员的最大并发客户端连接数。

autopurge.snapRetainCount
: ZooKeeper 的 in-memory 数据库的快照数量，该数据库将被保留。默认值为 **3**。

autopurge.purgeInterval
: 清除快照的时间间隔（以小时为单位）。默认值为 **0**，这个选项被禁用。

所有可用的配置选项都可在 ZooKeeper 文档 中找到。

5.4. 配置 KAFKA

Kafka 使用属性文件来存储静态配置。配置文件的建议位置为 **/opt/kafka/config/server.properties**。该配置文件必须可由 **kafka** 用户读取。

Apache Kafka 的流附带了一个示例配置文件，它突出显示了该产品的基本和高级功能。它可在 Streams for Apache Kafka 安装目录中的 **config/server.properties** 下找到。

本章解释了最重要的配置选项。

5.4.1. ZooKeeper

Kafka 代理需要 ZooKeeper 存储其配置的一些部分，并协调集群（例如，决定哪个节点是哪个分区的领导）。ZooKeeper 集群的连接详情保存在配置文件中。字段 **zookeeper.connect** 包含 zookeeper 集群成员的主机名和端口列表。

例如：

```
zookeeper.connect=zoo1.my-domain.com:2181,zoo2.my-domain.com:2181,zoo3.my-domain.com:2181
```

Kafka 将使用这些地址来连接到 ZooKeeper 集群。使用这个配置，所有 Kafka **znodes** 都会直接在 ZooKeeper 数据库的根目录中创建。因此，此类 ZooKeeper 集群只能用于单个 Kafka 集群。要将多个 Kafka 集群配置为使用单个 ZooKeeper 集群，在 Kafka 配置文件中 ZooKeeper 连接字符串末尾指定基本（前缀）路径：

```
zookeeper.connect=zoo1.my-domain.com:2181,zoo2.my-domain.com:2181,zoo3.my-domain.com:2181/my-cluster-1
```

5.4.2. 监听器

监听器用于连接到 Kafka 代理。每个 Kafka 代理都可以配置为使用多个监听程序。每个侦听器都需要不同的配置，以便它可以侦听不同的端口或网络接口。

要配置监听程序，请编辑 Kafka 配置属性文件中的 **listeners** 属性。以逗号分隔列表的形式将 **监听程序添加到监听程序** 属性。配置每个属性，如下所示：

```
<listener_name>://<hostname>:<port>
```

如果 **<hostname>** 为空，则 Kafka 将使用 **java.net.InetAddress.getCanonicalHostName()** 类作为主机名。

多个监听器的配置示例

```
listeners=internal-1://:9092,internal-2://:9093,replication://:9094
```

当 Kafka 客户端连接到 Kafka 集群时，它首先连接到 *bootstrap 服务器*，这是集群节点之一。bootstrap 服务器为客户端提供集群中所有代理的列表，客户端会单独连接到每个代理。代理列表基于配置 **的监听程序**。

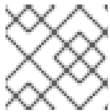
advertised 监听程序

另外，您可以使用 **advertised.listeners** 属性为客户端提供不同于监听程序属性中给出的一系列不同的**监听程序**地址。如果其他网络基础架构（如代理）在客户端和代理之间，或者会使用外部 DNS 名称而不是 IP 地址，这很有用。

advertised.listeners 属性的格式与 **listeners** 属性相同。

公告监听程序的配置示例

```
listeners=internal-1://:9092,internal-2://:9093
advertised.listeners=internal-1://my-broker-1.my-domain.com:1234,internal-2://my-broker-1.my-domain.com:1235
```



注意

公告的监听程序的名称必须与**监听程序**属性中列出的名称匹配。

inter-broker 监听程序

inter-broker 监听程序 用于 Kafka 代理之间的通信。需要代理间通信：

- 在不同代理间协调工作负载
- 在存储在不同代理中的分区间复制消息

inter-broker 侦听器可以分配给您选择的端口。当配置了多个监听程序时，您可以在代理配置的 **inter.broker.listener.name** 属性中定义 inter-broker 监听程序的名称。

在这里，inter-broker 侦听器被命名为 **REPLICATION** ：

```
listeners=REPLICATION://0.0.0.0:9091
inter.broker.listener.name=REPLICATION
```

控制器监听程序

控制器配置用于连接和与协调集群的控制器通信，并管理用于跟踪代理和分区状态的元数据。

默认情况下，控制器和代理之间的通信使用专用的控制器监听程序。控制器负责协调管理任务，如分区领导变化，因此需要一个或多个这些监听器。

使用 **controller.listener.names** 属性指定用于控制器的监听程序。您可以使用 **controller.quorum.voters** 属性指定控制器投票的仲裁。仲裁（仲裁）为管理任务启用领导后续结构，领导人主动管理操作，并将后续者作为热待机，确保内存中的元数据一致性并促进故障转移。

```
listeners=CONTROLLER://0.0.0.0:9090
controller.listener.names=CONTROLLER
controller.quorum.voters=1@localhost:9090
```

控制器 voters 的格式是 < **cluster_id>@<hostname>:<port>**。

5.4.3. 提交日志

Apache Kafka 将其从制作者接收的所有记录存储在提交日志中。提交日志包含 Kafka 需要提供的记录形式的实际数据。请注意，这些记录与应用程序日志文件不同，该文件详细说明了代理的活动。

日志目录

您可以使用 **log.dirs** 属性文件配置日志目录，将提交日志存储在一个或多个日志目录中。它应设置为在安装过程中创建的 **/var/lib/kafka** 目录：

```
log.dirs=/var/lib/kafka
```

出于性能考虑，您可以将 log.dirs 配置为多个目录，并将其每个目录放在不同的物理设备中，以提高磁盘 I/O 性能。例如：

```
log.dirs=/var/lib/kafka1,/var/lib/kafka2,/var/lib/kafka3
```

5.4.4. 代理 ID

代理 ID 是集群中每个代理的唯一标识符。您可以分配一个大于或等于 0 的整数作为代理 ID。代理 ID 用于在重启或崩溃后识别代理，因此 id 稳定且不会随时间变化。代理 ID 在代理属性文件中配置：

```
broker.id=1
```

第 6 章 保护对 KAFKA 的访问

通过管理客户端对 Kafka 代理的访问来保护 Kafka 集群。指定保护 Kafka 代理和客户端的配置选项

Kafka 代理和客户端之间的安全连接可以包括以下内容：

- 数据交换加密
- 证明身份的身份验证
- 允许或拒绝用户执行操作的授权

为客户端指定的身份验证和授权机制必须与为 Kafka 代理指定的匹配。

6.1. 侦听器配置

Kafka 代理中的加密和验证会针对每个监听程序进行配置。有关 Kafka 侦听器配置的更多信息，请参阅 第 5.4.2 节 "监听器"。

Kafka 代理中的每个监听程序都使用自己的安全协议配置。配置属性 **listener.security.protocol.map** 定义哪个监听程序使用哪个安全协议。它将每个侦听器名称映射到其安全协议。支持的安全协议有：

PLAINTEXT

无加密或身份验证的监听程序。

SSL

使用 TLS 加密（可选）使用 TLS 客户端证书进行身份验证的监听程序。

SASL_PLAINTEXT

没有加密的监听程序，但使用基于 SASL 的身份验证。

SASL_SSL

带有基于 TLS 的加密和基于 SASL 的验证的监听程序。

根据以下 **监听程序配置** ：

```
listeners=INT1://:9092,INT2://:9093,REPLICATION://:9094
```

listener.security.protocol.map 可能如下所示：

```
listener.security.protocol.map=INT1:SASL_PLAINTEXT,INT2:SASL_SSL,REPLICATION:SSL
```

这会将侦听器 **INT1** 配置为使用带有 SASL 身份验证的未加密的连接，侦听器 **INT2** 使用 SASL 身份验证的加密连接，以及 **REPLICATION** 接口以使用 TLS 加密（可能与 TLS 客户端身份验证一起使用）。相同的安全协议可以多次使用。以下示例也是有效的配置：

```
listener.security.protocol.map=INT1:SSL,INT2:SSL,REPLICATION:SSL
```

此类配置将 TLS 加密和 TLS 身份验证（可选）用于所有接口。

6.2. TLS 加密

Kafka 支持 TLS 来加密与 Kafka 客户端的通信。

要使用 TLS 加密和服务器身份验证，必须提供包含私钥和公钥的密钥存储。这通常使用 Java Keystore (JKS)格式的文件来完成。此文件的路径在 **ssl.keystore.location** 属性中设置。**ssl.keystore.password** 属性应该用于设置保护密钥存储的密码。例如：

```
ssl.keystore.location=/path/to/keystore/server-1.jks
ssl.keystore.password=123456
```

在某些情况下，使用额外的密码来保护私钥。可以使用 **ssl.key.password** 属性设置此类密码。

Kafka 可以使用由证书颁发机构和自签名密钥签名的密钥。使用由证书颁发机构签名的密钥应始终是首选的方法。为了允许客户端验证其正在连接的 Kafka 代理的身份，证书应始终包含公告的主机名 (CN) 或 Subject Alternative Name(SAN)。

可以将不同的 SSL 配置用于不同的监听程序。所有以 **ssl 开头的选项都可以以 listener. name. <NameOfTheListener>.** 前缀，其中监听程序的名称必须始终为小写。这将覆盖该特定监听器的默认 SSL 配置。以下示例演示了如何为不同的监听程序使用不同的 SSL 配置：

```
listeners=INT1://:9092,INT2://:9093,REPLICATION://:9094
listener.security.protocol.map=INT1:SSL,INT2:SSL,REPLICATION:SSL

# Default configuration - will be used for listeners INT1 and INT2
ssl.keystore.location=/path/to/keystore/server-1.jks
ssl.keystore.password=123456

# Different configuration for listener REPLICATION
listener.name.replication.ssl.keystore.location=/path/to/keystore/replication.jks
listener.name.replication.ssl.keystore.password=123456
```

其他 TLS 配置选项

除了上面描述的主要 TLS 配置选项外，Kafka 还支持很多选项来微调 TLS 配置。例如，启用或禁用 TLS / SSL 协议或密码套件：

ssl.cipher.suites

启用的密码套件列表。每个密码套件都是用于 TLS 连接的身份验证、加密、MAC 和密钥交换算法的组合。默认情况下启用所有可用的密码套件。

ssl.enabled.protocols

已启用的 TLS/SSL 协议列表。默认为 **TLSv1.2,TLSv1.1,TLSv1**。

6.2.1. 启用 TLS 加密

这个步骤描述了如何在 Kafka 代理中启用加密。

先决条件

- 每个主机上安装了 Apache Kafka 的流，且配置文件可用。

流程

1. 为集群中的所有 Kafka 代理生成 TLS 证书。证书应该在其 Common Name 或 Subject Alternative Name 中有其公告和 bootstrap 地址。
2. 编辑所有集群节点上的 Kafka 配置文件：
   - 更改 **listener.security.protocol.map** 字段，为您要使用 TLS 加密的监听程序指定 SSL 协

更改 **listener.security.protocol.map** 字段，为您要使用 TLS 加密的监听程序指定 **SSL** 协议。

- 使用代理证书将 **ssl.keystore.location** 选项设置为 JKS 密钥存储的路径。
- 将 **ssl.keystore.password** 选项设置为用来保护密钥存储的密码。
  例如：

```
listeners=UNENCRYPTED://:9092,ENCRYPTED://:9093,REPLICATION://:9094
listener.security.protocol.map=UNENCRYPTED:PLAINTEXT,ENCRYPTED:SSL,REPLICA
TION:PLAINTEXT
ssl.keystore.location=/path/to/keystore/server-1.jks
ssl.keystore.password=123456
```

3. （重新）启动 Kafka 代理

6.3. 身份验证

要验证到 Kafka 集群的客户端连接，可以使用以下选项：

TLS 客户端身份验证
: 在加密连接中使用 X.509 证书 TLS （传输层安全）

Kafka SASL
: 使用支持的身份验证机制的 Kafka SASL （简单身份验证和安全层）

OAuth 2.0
: 基于 OAuth 2.0 令牌的身份验证

SASL 身份验证支持普通未加密的连接和 TLS 连接的各种机制：

- **PLAIN** - 基于用户名和密码进行身份验证。
- **SCRAM-SHA-256** 和 **SCRAM-SHA-512** - 使用 Salted Challenge Response Authentication Mechanism (SCRAM)进行身份验证。
- **GSSAPI** - 对 Kerberos 服务器进行身份验证。

警告

PLAIN 机制通过网络发送用户名和密码，格式为未加密的格式。它应该只与 TLS 加密结合使用。

6.3.1. 启用 TLS 客户端身份验证

在 Kafka 代理中启用 TLS 客户端身份验证，以增强连接到已使用 TLS 加密的 Kafka 节点的安全性。

使用 **ssl.client.auth** 属性使用以下值之一设置 TLS 身份验证：

- **none** - TLS 客户端身份验证为 off （默认）

- **requested** - 可选 TLS 客户端身份验证
- **必需** - 客户端必须使用 TLS 客户端证书进行身份验证

当客户端使用 TLS 客户端身份验证进行身份验证时，经过身份验证的主体名称会派生自客户端证书中的可分辨名称。例如，具有可分辨名称 **CN=someuser** 证书的用户将使用主体 **CN=someuser,OU=Unknown,O=Unknown,L=Unknown,C=Unknown,C=Unknown,C=Unknown** 进行验证。这个主体名称为经过身份验证的用户或实体提供唯一标识符。如果没有使用 TLS 客户端身份验证，并且禁用 SASL，则主体名称默认为 **ANONYMOUS**。

先决条件

- 每个主机上安装了 Apache Kafka 的流，且配置文件可用。
- 启用 TLS 加密。

流程

1. 准备一个 JKS (Java Keystore)信任存储，其中包含用于签署用户证书的 CA （认证机构）的公钥。
2. 编辑所有集群节点上的 Kafka 配置文件，如下所示：
   - 使用 **ssl.truststore.location** 属性指定 JKS 信任存储的路径。
   - 如果信任存储受密码保护，请使用 **ssl.truststore.password** 属性设置密码。
   - 将 **ssl.client.auth** 属性设置为 **required**。

   TLS 客户端身份验证配置

   ```
   ssl.truststore.location=/path/to/truststore.jks
   ssl.truststore.password=123456
   ssl.client.auth=required
   ```
3. （重新）启动 Kafka 代理。

6.3.2. 启用 SASL PLAIN 客户端身份验证

在 Kafka 中启用 SASL PLAIN 身份验证，以增强连接到 Kafka 节点的安全性。

SASL 身份验证通过使用 **KafkaServer** JAAS 上下文的 Java 身份验证和授权服务(JAAS)启用。您可以在专用文件中或直接在 Kafka 配置中定义 JAAS 配置。

专用文件的建议位置为 **/opt/kafka/config/jaas.conf**。确保该文件可由 **kafka** 用户读取。保留 JAAS 配置文件在所有 Kafka 节点上同步。

先决条件

- 每个主机上安装了 Apache Kafka 的流，且配置文件可用。

流程

1. 编辑或创建 **/opt/kafka/config/jaas.conf** JAAS 配置文件以启用 **PlainLoginModule** 并指定允许的用户名和密码。

确保该文件在所有 Kafka 代理中都是相同的。

JAAS 配置

```
KafkaServer {
   org.apache.kafka.common.security.plain.PlainLoginModule required
   user_admin="123456"
   user_user1="123456"
   user_user2="123456";
};
```

2. 编辑所有集群节点上的 Kafka 配置文件，如下所示：

   - 使用 **listener.security.protocol.map** 属性在特定监听程序上启用 SASL PLAIN 身份验证。指定 **SASL_PLAINTEXT** 或 **SASL_SSL**。
   - 将 **sasl.enabled.mechanisms** 属性设置为 **PLAIN**。

     SASL 普通配置

```
listeners=INSECURE://:9092,AUTHENTICATED://:9093,REPLICATION://:9094
listener.security.protocol.map=INSECURE:PLAINTEXT,AUTHENTICATED:SASL_PLAINT
EXT,REPLICATION:PLAINTEXT
sasl.enabled.mechanisms=PLAIN
```

3. （重新）使用 **KAFKA_OPTS** 环境变量启动 Kafka 代理，将 JAAS 配置传递给 Kafka 代理：

```
su - kafka
export KAFKA_OPTS="-Djava.security.auth.login.config=/opt/kafka/config/jaas.conf";
/opt/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/config/server.properties
```

6.3.3. 启用 SASL SCRAM 客户端身份验证

在 Kafka 中启用 SASL SCRAM 身份验证，以增强连接到 Kafka 节点的安全性。

SASL 身份验证通过使用 **KafkaServer** JAAS 上下文的 Java 身份验证和授权服务(JAAS)启用。您可以在专用文件中或直接在 Kafka 配置中定义 JAAS 配置。

专用文件的建议位置为 **/opt/kafka/config/jaas.conf**。确保该文件可由 **kafka** 用户读取。保留 JAAS 配置文件在所有 Kafka 节点上同步。

先决条件

- 每个主机上安装了 Apache Kafka 的流，且配置文件可用。

流程

1. 编辑或创建 **/opt/kafka/config/jaas.conf** JAAS 配置文件以启用 **ScramLoginModule**。
   确保该文件在所有 Kafka 代理中都是相同的。

   JAAS 配置

```
KafkaServer {
   org.apache.kafka.common.security.scram.ScramLoginModule required;
};
```

2. 编辑所有集群节点上的 Kafka 配置文件，如下所示：

   - 使用 **listener.security.protocol.map** 属性在特定监听程序上启用 SASL SCRAM 身份验证。指定 **SASL_PLAINTEXT** 或 **SASL_SSL**。
   - 将 **sasl.enabled.mechanisms** 选项设置为 **SCRAM-SHA-256** 或 **SCRAM-SHA-512**。例如：

   ```
   listeners=INSECURE://:9092,AUTHENTICATED://:9093,REPLICATION://:9094
   listener.security.protocol.map=INSECURE:PLAINTEXT,AUTHENTICATED:SASL_PLAINT
   EXT,REPLICATION:PLAINTEXT
   sasl.enabled.mechanisms=SCRAM-SHA-512
   ```

3. （重新）使用 **KAFKA_OPTS** 环境变量启动 Kafka 代理，将 JAAS 配置传递给 Kafka 代理。

   ```
   su - kafka
   export KAFKA_OPTS="-Djava.security.auth.login.config=/opt/kafka/config/jaas.conf";
   /opt/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/config/server.properties
   ```

6.3.4. 启用多个 SASL 机制

使用 SASL 身份验证时，您可以启用多个机制。Kafka 可以同时使用多个 SASL 机制。启用多个机制后，您可以选择特定的客户端使用的机制。

要使用多个机制，您可以设置每个机制所需的配置。您可以将不同的 **KafkaServer** JAAS 配置添加到同一上下文中，并使用 **sasl.mechanism.inter.broker.protocol** 属性在 Kafka 配置中启用多个机制作为用逗号分开的列表。

JAAS 配置多个 SASL 机制

```
KafkaServer {
   org.apache.kafka.common.security.plain.PlainLoginModule required
   user_admin="123456"
   user_user1="123456"
   user_user2="123456";

   com.sun.security.auth.module.Krb5LoginModule required
   useKeyTab=true
   storeKey=true
   keyTab="/etc/security/keytabs/kafka_server.keytab"
   principal="kafka/kafka1.hostname.com@EXAMPLE.COM";

   org.apache.kafka.common.security.scram.ScramLoginModule required;
};
```

启用 SASL 机制

```
sasl.enabled.mechanisms=PLAIN,SCRAM-SHA-256,SCRAM-SHA-512
```

6.3.5. 为 inter-broker 身份验证启用 SASL

在 Kafka 节点间启用 SASL SCRAM 身份验证，以增强代理连接的安全性。除了将 SASL 身份验证用于客户端连接到 Kafka 集群外，您还可以使用 SASL 进行代理身份验证。与 SASL 用于客户端连接不同，您只能为代理通信选择一个机制。

先决条件

- ZooKeeper 安装在每个主机上，且配置文件可用。
- 如果您使用 SCRAM 机制，请在 Kafka 集群中注册 SCRAM 凭证。
  对于 Kafka 集群中的所有节点，将 inter-broker SASL SCRAM 用户添加到 ZooKeeper 中。这样可确保在 Kafka 集群运行前为 bootstrap 更新用于身份验证的凭证。

  注册 inter-broker SASL SCRAM 用户

  ```
  bin/kafka-configs.sh \
  --zookeeper localhost:2181 \
  --alter \
  --add-config 'SCRAM-SHA-512=[password=changeit]' \
  --entity-type users \
  --entity-name kafka
  ```

流程

1. 使用 **sasl.mechanism.inter.broker.protocol** 属性在 Kafka 配置中指定 inter-broker SASL 机制。

   inter-broker SASL 机制

   ```
   sasl.mechanism.inter.broker.protocol=SCRAM-SHA-512
   ```

2. （可选）如果您使用 SCRAM 机制，请通过 添加 SCRAM 用户在 Kafka 集群上注册 SCRAM 凭证。
   这样可确保在 Kafka 集群运行前为 bootstrap 更新用于身份验证的凭证。

3. 使用 **username** 和 password 字段在 **KafkaServer** JAAS 上下文中指定用于代理间通信的 **用户名和密码**。

   inter-broker JAAS 上下文

   ```
   KafkaServer {
      org.apache.kafka.common.security.plain.ScramLoginModule required
      username="admin"
      password="123456"
      # ...
   };
   ```

6.3.6. 添加 SASL SCRAM 用户

此流程概述了在 Kafka 中使用 SASL SCRAM 注册新用户进行身份验证的步骤。SASL SCRAM 身份验证增强了客户端连接的安全性。

先决条件

- 每个主机上安装了 Apache Kafka 的流，且配置文件可用。
- 启用 SASL SCRAM 身份验证。

流程

- 使用 **kafka-configs.sh** 工具添加新的 SASL SCRAM 用户。

```
/opt/kafka/kafka-configs.sh \
--bootstrap-server <broker_host>:<port> \
--alter \
--add-config 'SCRAM-SHA-512=[password=<password>]' \
--entity-type users --entity-name <username>
```

例如：

```
/opt/kafka/kafka-configs.sh \
--bootstrap-server localhost:9092 \
--alter \
--add-config 'SCRAM-SHA-512=[password=123456]' \
--entity-type users \
--entity-name user1
```

6.3.7. 删除 SASL SCRAM 用户

此流程概述了在 Kafka 中使用 SASL SCRAM 删除注册的用户的步骤。

先决条件

- 每个主机上安装了 Apache Kafka 的流，且配置文件可用。
- 启用 SASL SCRAM 身份验证。

流程

- 使用 **kafka-configs.sh** 工具删除 SASL SCRAM 用户。

```
/opt/kafka/bin/kafka-configs.sh \
--bootstrap-server <broker_host>:<port> \
--alter \
--delete-config 'SCRAM-SHA-512' \
--entity-type users \
--entity-name <username>
```

例如：

```
/opt/kafka/bin/kafka-configs.sh \
--bootstrap-server localhost:9092 \
--alter \
--delete-config 'SCRAM-SHA-512' \
--entity-type users \
--entity-name user1
```

6.3.8. 启用 Kerberos (GSSAPI)身份验证

Apache Kafka 的流支持使用 Kerberos (GSSAPI)身份验证协议来保护对 Kafka 集群的单点登录访问。GSSAPI 是 Kerberos 功能的 API 包装程序，从底层实现更改中模拟应用程序。

Kerberos 是一种网络身份验证系统，其允许客户端和服务器使用对称加密和信任的第三方 KDC 来互相进行身份验证。

此流程演示了如何为 Apache Kafka 配置流，以便 Kafka 客户端可以使用 Kerberos (GSSAPI)身份验证访问 Kafka 和 ZooKeeper。

该流程假设已在 Red Hat Enterprise Linux 主机上设置了 Kerberos *krb5* 资源服务器。

该流程显示，带有如何配置的示例：

1. 服务主体
2. 使用 Kerberos 登录的 Kafka 代理
3. zookeeper 使用 Kerberos 登录
4. 生产者和消费者客户端使用 Kerberos 身份验证访问 Kafka

这些说明描述了在单个主机上为单个 ZooKeeper 和 Kafka 安装设置的 Kerberos，为生成者和消费者客户端提供额外的配置。

先决条件

要配置 Kafka 和 ZooKeeper 来验证和授权 Kerberos 凭证，您需要：

- 访问 Kerberos 服务器
- 每个 Kafka 代理主机上的 Kerberos 客户端

有关在代理主机上设置 Kerberos 服务器和客户端的详情请参考 RHEL 设置配置中的 Kerberos 示例。

为身份验证添加服务主体
在 Kerberos 服务器中，为 ZooKeeper、Kafka 代理和 Kafka 生成者和消费者客户端创建服务主体（用户）。

服务主体必须采用 *SERVICE-NAME/FULLY-QUALIFIED-HOST-NAME@DOMAIN-REALM* 的形式。

1. 创建通过 Kerberos KDC 存储主体密钥的服务主体和 keytab。
   确保 Kerberos 主体中的域名采用大写。

   例如：

   - **zookeeper/node1.example.redhat.com@EXAMPLE.REDHAT.COM**
   - **kafka/node1.example.redhat.com@EXAMPLE.REDHAT.COM**
   - **producer1/node1.example.redhat.com@EXAMPLE.REDHAT.COM**
   - **consumer1/node1.example.redhat.com@EXAMPLE.REDHAT.COM**
     ZooKeeper 服务主体必须与 Kafka **config/server.properties** 文件中的 **zookeeper.connect** 配置具有相同的主机名：

     ```
     zookeeper.connect=node1.example.redhat.com:2181
     ```

- 如果主机名不相同，则使用 *localhost*，身份验证将失败。

2. 在主机上创建一个目录并添加 keytab 文件：
   例如：

```
/opt/kafka/krb5/zookeeper-node1.keytab
/opt/kafka/krb5/kafka-node1.keytab
/opt/kafka/krb5/kafka-producer1.keytab
/opt/kafka/krb5/kafka-consumer1.keytab
```

3. 确保 **kafka** 用户可以访问目录：

```
chown kafka:kafka -R /opt/kafka/krb5
```

将 ZooKeeper 配置为使用 Kerberos 登录
配置 ZooKeeper 以使用 Kerberos 密钥分发中心(KDC)使用之前为 **zookeeper** 创建的用户主体和 keytabs 进行身份验证。

1. 创建或修改 **opt/kafka/config/jaas.conf** 文件来支持 ZooKeeper 客户端和服务器操作：

```
Client {
  com.sun.security.auth.module.Krb5LoginModule required debug=true
  useKeyTab=true (1)
  storeKey=true (2)
  useTicketCache=false (3)
  keyTab="/opt/kafka/krb5/zookeeper-node1.keytab" (4)
  principal="zookeeper/node1.example.redhat.com@EXAMPLE.REDHAT.COM"; (5)
};

Server {
  com.sun.security.auth.module.Krb5LoginModule required debug=true
  useKeyTab=true
  storeKey=true
  useTicketCache=false
  keyTab="/opt/kafka/krb5/zookeeper-node1.keytab"
  principal="zookeeper/node1.example.redhat.com@EXAMPLE.REDHAT.COM";
};

QuorumServer {
  com.sun.security.auth.module.Krb5LoginModule required debug=true
  useKeyTab=true
  storeKey=true
  keyTab="/opt/kafka/krb5/zookeeper-node1.keytab"
  principal="zookeeper/node1.example.redhat.com@EXAMPLE.REDHAT.COM";
};

QuorumLearner {
  com.sun.security.auth.module.Krb5LoginModule required debug=true
  useKeyTab=true
  storeKey=true
  keyTab="/opt/kafka/krb5/zookeeper-node1.keytab"
  principal="zookeeper/node1.example.redhat.com@EXAMPLE.REDHAT.COM";
};
```

1. 设置为 **true**，以从 keytab 获取主体密钥。
2. 设置为 **true** 以存储主体密钥。
3. 设置为 **true**，以从票据缓存获取 Ticket Granting Ticket (TGT)。
4. **keyTab** 属性指向从 Kerberos KDC 复制的 keytab 文件的位置。位置和文件必须可由 **kafka** 用户读取。
5. **principal** 属性配置为与 KDC 主机上创建的完全限定域名匹配，其格式是 ***SERVICE-NAME/FULLY-QUALIFIED-HOST@DOMAIN-NAME***。

2. 编辑 **opt/kafka/config/zookeeper.properties** 以使用更新的 JAAS 配置：

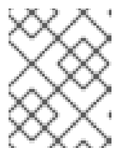
```
# ...

requireClientAuthScheme=sasl
jaasLoginRenew=3600000 (1)
kerberos.removeHostFromPrincipal=false (2)
kerberos.removeRealmFromPrincipal=false (3)
quorum.auth.enableSasl=true (4)
quorum.auth.learnerRequireSasl=true (5)
quorum.auth.serverRequireSasl=true
quorum.auth.learner.loginContext=QuorumLearner (6)
quorum.auth.server.loginContext=QuorumServer
quorum.auth.kerberos.servicePrincipal=zookeeper/_HOST (7)
quorum.cnxn.threads.size=20
```

1. 控制登录续订的频率（以毫秒为单位），可针对票据续订间隔进行调整。默认值为一小时。
2. 指定主机名是否用作登录主体名称的一部分。如果将单个 keytab 用于集群中的所有节点，则会将其设置为 **true**。但是，建议为每个代理主机生成单独的 keytab 和完全限定主体以进行故障排除。
3. 控制域名称是否从 Kerberos 协商的主体名称中分离。建议将此设置设置为 **false**。
4. 为 ZooKeeper 服务器和客户端启用 SASL 身份验证机制。
5. **RequireSasl** 属性控制仲裁事件是否需要 SASL 身份验证，如 master 选举机制。
6. **loginContext** 属性标识用于对指定组件进行身份验证的 JAAS 配置中登录上下文的名称。loginContext 名称与 **opt/kafka/config/jaas.conf** 文件中相关部分的名称对应。
7. 控制用于组成用于识别的主体名称的命名惯例。占位符 **_HOST** 会自动解析为运行时由 **server.1** 属性定义的主机名。

3. 使用 JVM 参数启动 ZooKeeper，以指定 Kerberos 登录配置：

```
su - kafka
export EXTRA_ARGS="-Djava.security.krb5.conf=/etc/krb5.conf -Djava.security.auth.login.config=/opt/kafka/config/jaas.conf"; /opt/kafka/bin/zookeeper-server-start.sh -daemon /opt/kafka/config/zookeeper.properties
```

如果您不使用默认服务名称(**zookeeper**)，请使用 **-Dzookeeper.sasl.client.username=*NAME*** 参数添加名称。



注意

如果您使用 **/etc/krb5.conf** 位置，则不需要在启动 ZooKeeper, Kafka, 或 Kafka 生成者和消费者时不需要指定 **-Djava.security.krb5.conf=/etc/krb5.conf**。

将 Kafka 代理服务器配置为使用 Kerberos 登录

使用之前为 **kafka** 创建的 user principals 和 keytabs，将 Kafka 配置为使用 Kerberos 密钥分发中心(KDC)进行身份验证。

1. 使用以下元素修改 **opt/kafka/config/jaas.conf** 文件：

```
KafkaServer {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    storeKey=true
    keyTab="/opt/kafka/krb5/kafka-node1.keytab"
    principal="kafka/node1.example.redhat.com@EXAMPLE.REDHAT.COM";
};
KafkaClient {
    com.sun.security.auth.module.Krb5LoginModule required debug=true
    useKeyTab=true
    storeKey=true
    useTicketCache=false
    keyTab="/opt/kafka/krb5/kafka-node1.keytab"
    principal="kafka/node1.example.redhat.com@EXAMPLE.REDHAT.COM";
};
```

2. 通过修改 **config/server.properties** 文件中的监听程序配置来配置 Kafka 集群中的每个代理，以便监听程序使用 SASL/GSSAPI 登录。
将 SASL 协议添加到监听器的安全协议映射中，并删除所有不需要的协议。

例如：

```
# ...
broker.id=0
# ...
listeners=SECURE://:9092,REPLICATION://:9094 (1)
inter.broker.listener.name=REPLICATION
# ...
listener.security.protocol.map=SECURE:SASL_PLAINTEXT,REPLICATION:SASL_PLAINTEX
T (2)
# ..
sasl.enabled.mechanisms=GSSAPI (3)
sasl.mechanism.inter.broker.protocol=GSSAPI (4)
sasl.kerberos.service.name=kafka (5)
...
```

(1) 配置了两个监听器：一个安全监听程序用于与客户端的通用通信（支持 TLS 用于通信），以及用于代理间通信的复制监听程序。

(2) 对于启用了 TLS 的监听程序，协议名称为 SASL_PLAINTEXT。对于启用了 TLS 的连接器，协议名称为 SASL_PLAINTEXT。如果不需要 SSL，您可以删除 **ssl.*** 属性。

协议名称为 SASL_PLAINTEXT。如果不需要 SSL，您可以删除 **ssl.** 属性。

3. Kerberos 验证的 SASL 机制是 **GSSAPI**。

4. 用于代理间通信的 Kerberos 身份验证。

5. 用于指定用于身份验证请求的服务名称，以将其与可能使用相同的 Kerberos 配置的其他服务区分开来。

3. 使用 JVM 参数启动 Kafka 代理，以指定 Kerberos 登录配置：

```
su - kafka
export KAFKA_OPTS="-Djava.security.krb5.conf=/etc/krb5.conf -
Djava.security.auth.login.config=/opt/kafka/config/jaas.conf"; /opt/kafka/bin/kafka-server-
start.sh -daemon /opt/kafka/config/server.properties
```

如果代理和 ZooKeeper 集群之前已经配置并使用基于非 Kerberos 的身份验证系统，则可以启动 ZooKeeper 和 broker 集群，并检查日志中是否存在配置错误。

启动代理和 Zookeeper 实例后，集群现在被配置为 Kerberos 身份验证。

配置 Kafka producer 和消费者客户端以使用 Kerberos 身份验证

使用之前为 **producer1** 和 **consumer1** 创建的用户主体和 keytabs 配置 Kafka producer 和 使用者客户端，以使用 Kerberos 密钥分发中心(KDC)进行身份验证。

1. 将 Kerberos 配置添加到制作者或消费者配置文件中。
例如：

/opt/kafka/config/producer.properties

```
# ...
sasl.mechanism=GSSAPI (1)
security.protocol=SASL_PLAINTEXT (2)
sasl.kerberos.service.name=kafka (3)
sasl.jaas.config=com.sun.security.auth.module.Krb5LoginModule required \ (4)
    useKeyTab=true  \
    useTicketCache=false \
    storeKey=true  \
    keyTab="/opt/kafka/krb5/producer1.keytab" \
    principal="producer1/node1.example.redhat.com@EXAMPLE.REDHAT.COM";
# ...
```

1. 配置 Kerberos (GSSAPI)身份验证。

2. Kerberos 使用 SASL 纯文本（用户名/密码）安全协议。

3. 在 Kerberos KDC 中配置的 Kafka 的服务主体(user)。

4. 使用 **jaas.conf** 中定义的相同属性，为 JAAS 配置。

/opt/kafka/config/consumer.properties

```
# ...
sasl.mechanism=GSSAPI
```

```
security.protocol=SASL_PLAINTEXT
sasl.kerberos.service.name=kafka
sasl.jaas.config=com.sun.security.auth.module.Krb5LoginModule required \
   useKeyTab=true  \
   useTicketCache=false \
   storeKey=true  \
   keyTab="/opt/kafka/krb5/consumer1.keytab" \
   principal="consumer1/node1.example.redhat.com@EXAMPLE.REDHAT.COM";
# ...
```

2. 运行客户端，验证您可以从 Kafka 代理发送和接收信息。
   制作者客户端：

```
export KAFKA_HEAP_OPTS="-Djava.security.krb5.conf=/etc/krb5.conf -
Dsun.security.krb5.debug=true"; /opt/kafka/bin/kafka-console-producer.sh --producer.config
/opt/kafka/config/producer.properties  --topic topic1 --bootstrap-server
node1.example.redhat.com:9094
```

   消费者客户端：

```
export KAFKA_HEAP_OPTS="-Djava.security.krb5.conf=/etc/krb5.conf -
Dsun.security.krb5.debug=true"; /opt/kafka/bin/kafka-console-consumer.sh --
consumer.config /opt/kafka/config/consumer.properties  --topic topic1 --bootstrap-server
node1.example.redhat.com:9094
```

其他资源

- Kerberos man pages: krb5.conf(5), kinit(1), klist(1), 和 kdestroy(1)
- RHEL 设置配置中的 Kerberos 服务器示例
- 使用 Kerberos 票据进行 Kafka 集群验证的客户端应用程序示例

6.4. 授权

Kafka 代理中的授权使用授权器插件实现。

在本节中，我们描述了如何使用 Kafka 提供的 **AclAuthorizer** 插件。

或者，您可以使用自己的授权插件。例如，如果您使用 基于 OAuth 2.0 令牌的身份验证，您可以使用 OAuth 2.0 授权。

6.4.1. 启用 ACL 授权器

编辑 **/opt/kafka/config/server.properties** 文件以添加 ACL 授权器。通过在 **authorizer.class.name** 属性中指定完全限定名称来启用授权器：

启用授权器

```
authorizer.class.name=kafka.security.authorizer.AclAuthorizer
```

对于 **AclAuthorizer**，完全限定名称为 **kafka.security.authorizer.AclAuthorizer**。

6.4.1.1. ACL 规则

ACL 授权器使用 ACL 规则来管理 Kafka 代理的访问。

ACL 规则以以下格式定义：

主体 P 被允许/拒绝来自主机 H的 <kafka_resource> R on <kafka_resource> R

例如，可以设置一个规则，以使用户 John 可以查看 主机 127.0.0.1 中的主题 注释。Host 是 John 进行连接的机器的 IP 地址。

在大多数情况下，用户是生成者或消费者应用程序：

Consumer01 可以对来自主机 127.0.0.1 的消费者组账户有写入权限。

如果给定资源没有 ACL 规则，则所有操作都会被拒绝。通过在 Kafka 配置文件 **/opt/kafka/config/server.properties** 中将属性 **allow.everyone.if.no.acl.found** 设置为 **true** 来更改此行为。

6.4.1.2. 主体

*principal（主体）*代表用户的身份。ID 格式取决于客户端用来连接到 Kafka 的身份验证机制：

- 在没有身份验证的情况下连接时，user :**ANONYMOUS**.
- 使用简单身份验证机制（如 PLAIN 或 SCRAM）连接时，user:< **username** >。例如 **User:admin** or **User:user1**。
- 使用 TLS 客户端身份验证 连接时，**user:<DistinguishedName >**。例如 **User:CN=user1,O=MyCompany,L=Prague,C=CZ**。
- 使用 Kerberos 连接时，**user:<Kerberos username >**。

DistinguishedName 是与客户端证书区分的名称。

Kerberos 用户名是 Kerberos 主体的主要部分，在使用 Kerberos 连接时默认使用它。您可以使用 **sasl.kerberos.principal.to.local.rules** 属性来配置 Kafka 主体如何从 Kerberos 主体构建。

6.4.1.3. 用户身份验证

要使用授权，您需要启用身份验证并供您的客户端使用。否则，所有连接都将具有主体 **User:ANONYMOUS**。

有关验证方法的详情请参考 第 6.3 节 “身份验证”。

6.4.1.4. 超级用户

超级用户被允许执行所有操作，无论 ACL 规则是什么。

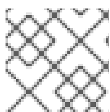
超级用户使用属性 **super.users** 在 Kafka 配置文件中定义。

例如：

```
super.users=User:admin,User:operator
```

6.4.1.5. 副本代理身份验证

启用授权后，它将适用于所有监听程序和所有连接。这包括用于在代理间复制数据的 inter-broker 连接。如果启用授权，请确保使用身份验证进行代理连接，并授予代理使用足够权利的用户。例如，如果代理之间的身份验证使用 **kafka-broker** 用户，则超级用户配置必须包含用户名 **super.users=User:kafka-broker**。



注意

有关您可以使用 ACL 控制的 Kafka 资源操作的更多信息，请参阅 Apache Kafka 文档。

6.4.2. 添加 ACL 规则

当使用 ACL 授权器根据访问控制列表(ACL)控制对 Kafka 的访问时，您可以使用 **kafka-acls.sh** 工具添加新的 ACL 规则。

使用 **kafka-acls.sh** 参数选项来添加、列出和删除 ACL 规则，并执行其他功能。参数需要双假设惯例，如 **--add**。

先决条件

- 已创建并授予了访问 Kafka 资源的适当权限。
- 每个主机上安装了 Apache Kafka 的流，且配置文件可用。
- 在 Kafka 代理中启用授权。

流程

- 使用 **--add** 选项运行 **kafka-acls.sh**。
  示例：
- 允许使用 **MyConsumerGroup** 消费者组从 **myTopic** 读取 **user1** 和 **user2** 访问权限。

```
opt/kafka/bin/kafka-acls.sh --bootstrap-server localhost:9092 --add --operation Read --topic myTopic --allow-principal User:user1 --allow-principal User:user2

opt/kafka/bin/kafka-acls.sh --bootstrap-server localhost:9092 --add --operation Describe --topic myTopic --allow-principal User:user1 --allow-principal User:user2

opt/kafka/bin/kafka-acls.sh --bootstrap-server localhost:9092 --add --operation Read --operation Describe --group MyConsumerGroup --allow-principal User:user1 --allow-principal User:user2
```

- 拒绝 **user1** 访问从 IP 地址主机 **127.0.0.1** 读取 **myTopic**。

```
opt/kafka/bin/kafka-acls.sh --bootstrap-server localhost:9092 --add --operation Describe --operation Read --topic myTopic --group MyConsumerGroup --deny-principal User:user1 --deny-host 127.0.0.1
```

- 添加 **user1** 作为带有 **MyConsumerGroup** 的 **myTopic** 的消费者。

```
opt/kafka/bin/kafka-acls.sh --bootstrap-server localhost:9092 --add --consumer --topic myTopic --group MyConsumerGroup --allow-principal User:user1
```

6.4.3. 列出 ACL 规则

当使用 ACL 授权器根据访问控制列表(ACL)控制对 Kafka 的访问时，您可以使用 **kafka-acls.sh** 工具来列出现有的 ACL 规则。

先决条件

- 添加了 ACL。

流程

- 使用 **--list** 选项运行 **kafka-acls.sh**。
例如：

```
opt/kafka/bin/kafka-acls.sh --bootstrap-server localhost:9092 --list --topic myTopic

Current ACLs for resource `Topic:myTopic`:

User:user1 has Allow permission for operations: Read from hosts: *
User:user2 has Allow permission for operations: Read from hosts: *
User:user2 has Deny permission for operations: Read from hosts: 127.0.0.1
User:user1 has Allow permission for operations: Describe from hosts: *
User:user2 has Allow permission for operations: Describe from hosts: *
User:user2 has Deny permission for operations: Describe from hosts: 127.0.0.1
```

6.4.4. 删除 ACL 规则

当使用 ACL 授权器根据访问控制列表(ACL)控制对 Kafka 的访问时，您可以使用 **kafka-acls.sh** 工具删除现有 ACL 规则。

先决条件

- 添加了 ACL。

流程

- 使用 **--remove** 选项运行 **kafka-acls.sh**。
示例：

- 删除 ACL，允许 user1 和 user2 使用 MyConsumerGroup 消费者组从 myTopic 读取。

```
opt/kafka/bin/kafka-acls.sh --bootstrap-server localhost:9092 --remove --operation
Read --topic myTopic --allow-principal User:user1 --allow-principal User:user2

opt/kafka/bin/kafka-acls.sh --bootstrap-server localhost:9092 --remove --operation
Describe --topic myTopic --allow-principal User:user1 --allow-principal User:user2

opt/kafka/bin/kafka-acls.sh --bootstrap-server localhost:9092 --remove --operation
Read --operation Describe --group MyConsumerGroup --allow-principal User:user1 --
allow-principal User:user2
```

- 删除 ACL 添加 user1 作为带有 MyConsumerGroup 的 myTopic 的消费者。

```
opt/kafka/bin/kafka-acls.sh --bootstrap-server localhost:9092 --remove --consumer --topic myTopic --group MyConsumerGroup --allow-principal User:user1
```

- 删除 ACL 拒绝 user1 从 IP 地址主机 127.0.0.1 读取 myTopic 的访问。

```
opt/kafka/bin/kafka-acls.sh --bootstrap-server localhost:9092 --remove --operation Describe --operation Read --topic myTopic --group MyConsumerGroup --deny-principal User:user1 --deny-host 127.0.0.1
```

6.5. ZOOKEEPER 身份验证

默认情况下，ZooZ 和 Kafka 之间的连接不会被身份验证。但是，Kafka 和 ZooKeeper 支持 Java 认证和授权服务(JAAS)，可用于使用简单身份验证和安全层(SASL)设置身份验证。ZooKeeper 支持在本地存储的凭证中使用 DIGEST-MD5 SASL 机制进行身份验证。

6.5.1. JAAS 配置

ZooKeeper 连接的 SASL 身份验证必须在 JAAS 配置文件中配置。默认情况下，Kafka 将使用名为 **Client** 的 JAAS 上下文连接到 ZooKeeper。**Client** 上下文应该在 **/opt/kafka/config/jass.conf** 文件中配置。上下文必须启用 **PLAIN** SASL 身份验证，如下例所示：

```
Client {
    org.apache.kafka.common.security.plain.PlainLoginModule required
    username="kafka"
    password="123456";
};
```

6.5.2. 启用 ZooKeeper 身份验证

这个流程描述了如何在连接到 ZooKeeper 时使用 SASL DIGEST-MD5 机制启用身份验证。

先决条件

- ZooKeeper 中启用了 客户端到服务器身份验证

启用 SASL DIGEST-MD5 身份验证

1. 在所有 Kafka 代理节点上，创建或编辑 **/opt/kafka/config/jaas.conf** JAAS 配置文件并添加以下上下文：

```
Client {
    org.apache.kafka.common.security.plain.PlainLoginModule required
    username="<Username>"
    password="<Password>";
};
```

   用户名和密码应该与 ZooKeeper 中配置相同。

   以下示例显示了 **Client** 上下文：

```
Client {
    org.apache.kafka.common.security.plain.PlainLoginModule required
```

```
    username="kafka"
    password="123456";
};
```

2. 逐一重启所有 Kafka 代理节点。要将 JAAS 配置传递给 Kafka 代理，请使用 **KAFKA_OPTS** 环境变量。

```
su - kafka
export KAFKA_OPTS="-Djava.security.auth.login.config=/opt/kafka/config/jaas.conf";
/opt/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/config/server.properties
```

有关在多节点集群中重启代理的详情，请参考 第 4.3 节 “执行 Kafka 代理的安全滚动重启”。

其他资源

- 身份验证

6.6. ZOOKEEPER 授权

当在 Kafka 和 ZooKeeper 之间启用身份验证时，您可以使用 ZooKeeper 访问控制列表(ACL)规则自动控制对存储在 ZooKeeper 中的 Kafka 元数据的访问。

6.6.1. ACL 配置

ZooKeeper ACL 规则的强制由 **config/server.properties** Kafka 配置文件中的 **zookeeper.set.acl** 属性控制。

属性默认被禁用，并通过设置为 **true** 来启用：

```
zookeeper.set.acl=true
```

如果启用了 ACL 规则，当在 ZooKeeper 中创建 **znode** 时，只有创建它的 Kafka 用户才可以修改或删除它。所有其他用户都具有只读访问权限。

Kafka 仅为新创建的 ZooKeeper **znodes** 设置 ACL 规则。如果只在集群首次启动后启用 ACL，**zookeeper-security-migration.sh** 工具可以在所有现有 **znodes** 上设置 ACL。

ZooKeeper 中数据的机密性

ZooKeeper 中存储的数据包括：

- 主题名称及其配置
- 当使用 SASL SCRAM 身份验证时，salted 和 hash 用户凭证。

但是 ZooKeeper 不存储使用 Kafka 发送和接收的任何记录。ZooKeeper 中存储的数据被认为是非机密。

如果要将数据视为机密（例如，主题名称包含客户 ID），则唯一可用于保护的选项会隔离网络级别的 ZooKeeper，并只允许访问 Kafka 代理。

6.6.2. 为新的 Kafka 集群启用 ZooKeeper ACL

此流程描述了如何在新 Kafka 集群的 Kafka 配置中启用 ZooKeeper ACL。仅在 Kafka 集群首次启动前使用此流程。有关在已在运行的集群中启用 ZooKeeper ACL，请参阅 第 6.6.3 节 “在现有 Kafka 集群中启用 ZooKeeper ACL”。

先决条件

- 每个主机上安装了 Apache Kafka 的流，且配置文件可用。
- 一个 ZooKeeper 集群已被配置并运行。
- ZooKeeper 中启用了 客户端到服务器身份验证。
- 在 Kafka 代理中，ZooKeeper 身份验证被启用。
- Kafka 代理还没有启动。

流程

1. 编辑 Kafka 配置文件，在所有集群节点上将 **zookeeper.set.acl** 字段设置为 **true**。

   ```
   zookeeper.set.acl=true
   ```

2. 启动 Kafka 代理。

6.6.3. 在现有 Kafka 集群中启用 ZooKeeper ACL

此流程描述了如何在 Kafka 配置中为正在运行的 Kafka 集群启用 ZooKeeper ACL。使用 **zookeeper-security-migration.sh** 工具，在所有存在的 **znodes** 中设置 ZooKeeper ACL。**zookeeper-security-migration.sh** 作为 Apache Kafka 的 Streams 的一部分，并可在**bin** 目录中找到。

先决条件

- Kafka 集群 已配置并运行。

启用 ZooKeeper ACL

1. 编辑 Kafka 配置文件，在所有集群节点上将 **zookeeper.set.acl** 字段设置为 **true**。

   ```
   zookeeper.set.acl=true
   ```

2. 逐一重启所有 Kafka 代理。
   有关在多节点集群中重启代理的详情，请参考 第 4.3 节 "执行 Kafka 代理的安全滚动重启"。

3. 使用 **zookeeper-security-migration.sh** 工具在所有现有 ZooKeeper**znodes** 中设置 ACL。

   ```
   su - kafka
   cd /opt/kafka
   KAFKA_OPTS="-Djava.security.auth.login.config=./config/jaas.conf"; ./bin/zookeeper-
   security-migration.sh --zookeeper.acl=secure --zookeeper.connect=<ZooKeeperURL>
   exit
   ```

   例如：

   ```
   su - kafka
   cd /opt/kafka
   KAFKA_OPTS="-Djava.security.auth.login.config=./config/jaas.conf"; ./bin/zookeeper-
   ```

```
security-migration.sh --zookeeper.acl=secure --zookeeper.connect=zoo1.my-domain.com:2181
exit
```

6.7. 使用基于 OAUTH 2.0 令牌的身份验证

Apache Kafka 的流支持使用 *OAUTHBEARER* 和 *PLAIN* 机制使用 OAuth 2.0 身份验证。

OAuth 2.0 启用应用程序之间的基于令牌的标准化身份验证和授权，使用中央授权服务器签发对资源有限访问权限的令牌。

您可以配置 OAuth 2.0 身份验证，然后配置 OAuth 2.0 授权。

Kafka 代理和客户端都需要配置为使用 OAuth 2.0。OAuth 2.0 身份验证也可以与基于 simple 或 OPA 的 Kafka 授权一起使用。

使用 OAuth 2.0 身份验证时，应用程序客户端可以访问应用服务器（称为 *资源服务器*）*上的资源*，而无需公开帐户凭据。

应用程序客户端通过访问令牌作为身份验证方法传递，应用服务器也可以用来决定要授予的访问权限级别。授权服务器处理访问权限的授予和查询有关访问权限的查询。

在 Apache Kafka 的 Streams 中：

- Kafka 代理作为 OAuth 2.0 资源服务器
- Kafka 客户端充当 OAuth 2.0 应用程序客户端

Kafka 客户端在 Kafka 代理验证。代理和客户端根据需要与 OAuth 2.0 授权服务器通信，以获取或验证访问令牌。

对于 Apache Kafka 的 Streams 部署，OAuth 2.0 集成提供：

- Kafka 代理的服务器端 OAuth 2.0 支持
- 对 Kafka MirrorMaker、Kafka Connect 和 Kafka Bridge 的客户端 OAuth 2.0 支持

RHEL 上的 Apache Kafka Streams 包括两个 OAuth 2.0 库：

kafka-oauth-client

提供名为 **io.strimzi.kafka.oauth.client.JaasClientOauthLoginCallbackHandler** 的自定义登录回调处理器类。要处理 **OAUTHBEARER** 身份验证机制，请使用 Apache Kafka 提供的 **OAuthBearerLoginModule** 的登录回调处理器。

kafka-oauth-common

提供 **kafka-oauth-client** 库所需的一些功能的帮助程序库。

提供的客户端库还依赖于一些额外的第三方库，例如：**keycloak-core**、**jackson-databind** 和 **slf4j-api**。

我们建议使用 Maven 项目来打包您的客户端，以确保包含所有依赖项库。依赖项库可能会在以后的版本中有所变化。

其他资源

- OAuth 2.0 站点

6.7.1. OAuth 2.0 身份验证机制

Apache Kafka 的流支持 OAUTHBEARER 和 PLAIN 机制进行 OAuth 2.0 身份验证。这两种机制都允许 Kafka 客户端与 Kafka 代理建立经过身份验证的会话。客户端、授权服务器和 Kafka 代理之间的身份验证流因每种机制而异。

我们建议您将客户端配置为尽可能使用 OAUTHBEARER。OAUTHBEARER 提供比 PLAIN 更高的安全性，因为客户端凭证*不会*与 Kafka 代理共享。考虑仅在不支持 OAUTHBEARER 的 Kafka 客户端中使用 PLAIN。

您可以将 Kafka 代理监听程序配置为使用 OAuth 2.0 身份验证来连接客户端。如果需要，您可以在同一 **oauth** 侦听器上使用 OAUTHBEARER 和 PLAIN 机制。支持每个机制的属性必须在**oauth** 侦听器配置中明确指定。

OAUTHBEARER 概述

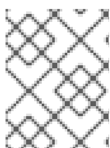
要使用 OAUTHBEARER，请将 Kafka 代理的 OAuth 身份验证监听程序配置中的 **sasl.enabled.mechanisms** 设置为 **OAUTHBEARER**。有关详细配置，请参阅 第 6.7.2 节 “OAuth 2.0 Kafka 代理配置”。

```
listener.name.client.sasl.enabled.mechanisms=OAUTHBEARER
```

许多 Kafka 客户端工具使用在协议级别为 OAUTHBEARER 提供基本支持的库。为了支持应用程序开发，Apache Kafka 的 Streams 为上游 Kafka 客户端 Java 库（但不适用于其他库）提供了一个 *OAuth 回调处理器*。因此，您不需要自行编写回调处理程序。应用客户端可以使用回调处理程序来提供访问令牌。使用 Go 等其他语言编写的客户端必须使用自定义代码连接到授权服务器并获取访问令牌。

使用 OAUTHBEARER 时，客户端发起带有 Kafka 代理进行凭证交换的会话，其中凭证采用由回调处理器提供的 bearer 令牌的形式。使用回调，您可以以三种方式之一配置令牌置备：

- 客户端 ID 和 Secret（使用 *OAuth 2.0 客户端凭证*机制）
- 一个长期的访问令牌，在配置时手动获取
- 一个长期的刷新令牌，在配置时手动获取



注意

OAUTHBEARER 身份验证只能由支持协议级别的 OAUTHBEARER 机制的 Kafka 客户端使用。

PLAIN 概述

要使用 PLAIN，请将 **PLAIN** 添加到 **sasl.enabled.mechanisms** 的值。

```
listener.name.client.sasl.enabled.mechanisms=OAUTHBEARER,PLAIN
```

PLAIN 是所有 Kafka 客户端工具使用的简单身份验证机制。要启用 PLAIN 用于 OAuth 2.0 身份验证，Apache Kafka 的流 *通过 PLAIN 服务器端回调提供 OAuth 2.0*。

客户端凭证在兼容授权服务器后集中处理，类似于使用 OAUTHBEARER 身份验证时。当与 OAuth 2.0 over PLAIN 回调一起使用时，Kafka 客户端使用以下方法之一与 Kafka 代理进行身份验证：

- 客户端 ID 和 secret（使用 *OAuth 2.0 客户端凭证*机制）
- 一个长期的访问令牌，在配置时手动获取

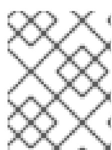
对于这两种方法，客户端必须提供 PLAIN **username** 和 **password** 属性，将凭证传递给 Kafka 代理。客户端使用这些属性传递客户端 ID 和 secret 或用户名和访问令牌。

客户端 ID 和 secret 用于获取访问令牌。

访问令牌作为 **password** 属性值传递。您可以使用或没有 **$accessToken:** 前缀来传递访问令牌。

- 如果您在监听器配置中配置令牌端点(**oauth.token.endpoint.uri**)，则需要前缀。
- 如果您没有在监听器配置中配置令牌端点(**oauth.token.endpoint.uri**)，则不需要前缀。Kafka 代理将密码解释为原始访问令牌。

如果将密码设置为访问令牌，则必须将用户名设置为 Kafka 代理从访问令牌获取的相同的主体名称。您可以使用 **oauth.username.claim**, **oauth.fallback.username.claim**, **oauth.fallback.username.prefix**, 和 **oauth.userinfo.endpoint.uri** 属性在监听器中指定用户名提取选项。用户名提取过程还取决于您的授权服务器；特别是，它将客户端 ID 映射到帐户名称。



注意

PLAIN 上的 OAuth 不支持使用（已弃用）OAuth 2.0 密码授权机制传递用户名和密码（密码授权）。

6.7.1.1. 使用属性或变量配置 OAuth 2.0

您可以使用 Java Authentication and Authorization Service(JAAS)属性或环境变量来配置 OAuth 2.0 设置。

- JAAS 属性在 **server.properties** 配置文件中配置，并传递为 **listener.name.*LISTENER-NAME*.oauthbearer.sasl.jaas.config** 属性的键值对。
- 如果使用环境变量，您仍然需要在 **server.properties** 文件中提供 **listener.name.*LISTENER-NAME*.oauthbearer.sasl.jaas.config** 属性，但您可以忽略其他 JAAS 属性。
您可以使用大写或大写环境变量命名约定。

Apache Kafka OAuth 2.0 库的 Streams 使用以以下开头的属性：

- **OAuth.** 用于配置身份验证
- **strimzi.** 到 configure OAuth 2.0 authorization

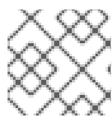
其他资源

- OAuth 2.0 Kafka 代理配置

6.7.2. OAuth 2.0 Kafka 代理配置

用于 OAuth 2.0 身份验证的 Kafka 代理配置涉及：

- 在授权服务器中创建 OAuth 2.0 客户端
- 在 Kafka 集群中配置 OAuth 2.0 身份验证



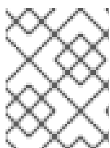
注意

与授权服务器的关系，Kafka 代理和 Kafka 客户端都被视为 OAuth 2.0 客户端。

6.7.2.1. 授权服务器上的 OAuth 2.0 客户端配置

要配置 Kafka 代理以验证会话启动期间收到的令牌，建议的做法是在授权服务器中创建一个 OAuth 2.0 *client* 定义（配置为 *confidential*），并启用了以下客户端凭证：

- 客户端 ID **kafka-broker**（例如）
- 客户端 ID 和 secret 作为身份验证机制



注意

在使用授权服务器的非公共内省端点时，您只需要使用客户端 ID 和 secret。在使用公共授权服务器端点时，通常不需要凭据，如快速本地 JWT 令牌验证一样。

6.7.2.2. Kafka 集群中的 OAuth 2.0 身份验证配置

要在 Kafka 集群中使用 OAuth 2.0 身份验证，您可以在 Kafka **server.properties** 文件中为 Kafka 集群启用 OAuth 身份验证监听程序配置。至少需要配置。您还可以配置 TLS 侦听器，其中 TLS 用于代理间通信。

您可以使用以下方法之一为授权服务器配置代理以进行令牌验证：

- 快速本地令牌验证：*JWKS* 端点与签名的 JWT 格式的访问令牌相结合
- *内省端点*

您可以配置 OAUTHBEARER 或 PLAIN 身份验证，或两者。

以下示例显示了应用 *全局* 监听器配置的最低配置，这意味着，内部代理间通信通过与应用程序客户端相同的监听程序进行。

这个示例还显示了一个特定监听程序的 OAuth 2.0 配置，在其中您可以指定 **listener.name.*LISTENER-NAME*.sasl.enabled.mechanisms** 而不是 **sasl.enabled.mechanisms**。*LISTENER-NAME* 是监听器的区分大小写的名称。在这里，我们将侦听器的 CLIENT 命名为 **listener.name.client.sasl.enabled.mechanisms**。

这个示例使用 OAUTHBEARER 身份验证。

示例：使用 JWKS 端点进行 OAuth 2.0 身份验证的最小监听程序配置

```
sasl.enabled.mechanisms=OAUTHBEARER 1
listeners=CLIENT://0.0.0.0:9092 2
listener.security.protocol.map=CLIENT:SASL_PLAINTEXT 3
listener.name.client.sasl.enabled.mechanisms=OAUTHBEARER 4
sasl.mechanism.inter.broker.protocol=OAUTHBEARER 5
inter.broker.listener.name=CLIENT 6
listener.name.client.oauthbearer.sasl.server.callback.handler.class=io.strimzi.kafka.oauth.serv
er.JaasServerOauthValidatorCallbackHandler 7
listener.name.client.oauthbearer.sasl.jaas.config=org.apache.kafka.common.security.oauthbe
arer.OAuthBearerLoginModule required \ 8
  oauth.valid.issuer.uri="https://<oauth_server_address>" \ 9
  oauth.jwks.endpoint.uri="https://<oauth_server_address>/jwks" \ 10
  oauth.username.claim="preferred_username"  \ 11
  oauth.client.id="kafka-broker" \ 12
```

```
  oauth.client.secret="kafka-secret" \ (13)
  oauth.token.endpoint.uri="https://<oauth_server_address>/token" ; (14)
listener.name.client.oauthbearer.sasl.login.callback.handler.class=io.strimzi.kafka.oauth.client
.JaasClientOauthLoginCallbackHandler (15)
listener.name.client.oauthbearer.connections.max.reauth.ms=3600000 (16)
```

1. 为通过 SASL 进行凭证交换启用 *OAUTHBEARER* 机制。
2. 配置要连接的客户端应用程序的监听程序。系统的 hostname 用作公告的主机名，客户端必须可以解析该主机名才能重新连接。本例中，侦听器名为 CLIENT。
3. 指定监听器的频道协议。SASL_SSL 用于 TLS。SASL_PLAINTEXT 用于未加密的连接（无 TLS），但存在丢失和截获 TCP 连接层的风险。
4. 为 *CLIENT* 侦听器指定 *OAUTHBEARER* 机制。客户端名称(*CLIENT*)通常在 listeners 属性中使用大写指定，对于 listener.name 属性是小写 (listener.name.client)，当为 listener.name.*client*.* 属性的一部分时是小写。
5. 指定用于代理间通信的 *OAUTHBEARER* 机制。
6. 指定用于代理间通信的监听程序。配置需要有效的规格。
7. 在客户端监听器上配置 OAuth 2.0 身份验证。
8. 配置客户端和 inter-broker 通信的身份验证设置。oauth.client.id、oauth.client.secret 和 auth.token.endpoint.uri 属性与 inter-broker 配置相关。
9. 有效的签发者 URI。只有此签发者发布的访问令牌才会被接受。例如 *：https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME*
10. JWKS 端点 URL。例如： *https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME/protocol/openid-connect/certs*。
11. 在令牌中包含实际用户名的令牌声明（或密钥）。用户名是用于识别用户的 *主体*。该值将取决于身份验证流和使用的授权服务器。如果需要，您可以使用 JsonPath 表达式，如 "['user.info'].['user.id']"，从令牌中的嵌套 JSON 属性检索用户名。
12. Kafka 代理的客户端 ID，适用于所有代理。这是在 授权服务器注册为 kafka-broker的客户端。
13. Kafka 代理的 secret，适用于所有代理。
14. 到您的授权服务器的 OAuth 2.0 令牌端点 URL。对于生产环境，请始终使用 https:// urls。例如：*https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME/protocol/openid-connect/token*。
15. 为代理间通信启用（且只需要）OAuth 2.0 身份验证。
16. （可选）当令牌过期时强制会话到期，并激活 Kafka 重新验证机制。如果指定的值小于访问令牌保留的时间，则客户端必须在实际令牌到期前重新验证。默认情况下，当访问令牌过期时，会话不会过期，客户端也不会尝试重新身份验证。

以下示例显示了 TLS 侦听器的最小配置，其中 TLS 用于代理间通信。

示例：用于 OAuth 2.0 身份验证的 TLS 侦听器配置

```
listeners=REPLICATION://kafka:9091,CLIENT://kafka:9092 (1)
listener.security.protocol.map=REPLICATION:SSL,CLIENT:SASL_PLAINTEXT (2)
listener.name.client.sasl.enabled.mechanisms=OAUTHBEARER
inter.broker.listener.name=REPLICATION
listener.name.replication.ssl.keystore.password=<keystore_password> (3)
listener.name.replication.ssl.truststore.password=<truststore_password>
listener.name.replication.ssl.keystore.type=JKS
listener.name.replication.ssl.truststore.type=JKS
listener.name.replication.ssl.secure.random.implementation=SHA1PRNG (4)
listener.name.replication.ssl.endpoint.identification.algorithm=HTTPS (5)
listener.name.replication.ssl.keystore.location=<path_to_keystore> (6)
listener.name.replication.ssl.truststore.location=<path_to_truststore> (7)
listener.name.replication.ssl.client.auth=required (8)
listener.name.client.oauthbearer.sasl.server.callback.handler.class=io.strimzi.kafka.oauth.serv
er.JaasServerOauthValidatorCallbackHandler
listener.name.client.oauthbearer.sasl.jaas.config=org.apache.kafka.common.security.oauthbe
arer.OAuthBearerLoginModule required \ (9)
  oauth.valid.issuer.uri="https://<oauth_server_address>" \
  oauth.jwks.endpoint.uri="https://<oauth_server_address>/jwks" \
  oauth.username.claim="preferred_username" ;
```

1. 相互代理通信和客户端应用程序需要单独的配置。
2. 将 *REPLICATION* 侦听器配置为使用 TLS，并将 *CLIENT* 侦听器配置为通过未加密的通道使用 SASL。客户端可能会在生产环境中使用加密的频道(SASL_SSL)。
3. ssl. 属性定义 TLS 配置。
4. 随机数生成器实施。如果没有设置，则使用 Java platform SDK 默认。
5. 主机名验证。如果设置为空字符串，则会关闭主机名验证。如果没有设置，则默认值为 HTTPS，它会强制对服务器证书进行主机名验证。
6. 侦听器的密钥存储的路径。
7. 侦听器的信任存储的路径。
8. 指定 *REPLICATION* 侦听器的客户端必须在建立 TLS 连接时与客户端证书进行身份验证（用于代理连接）。
9. 为 OAuth 2.0 配置 *CLIENT* 侦听器。与授权服务器的连接应使用安全 HTTPS 连接。

以下示例显示了使用 PLAIN 身份验证机制通过 SASL 进行凭证交换的 OAuth 2.0 身份验证的最低配置。使用快速的本地令牌验证。

示例：用于 PLAIN 验证的最小监听程序配置

```
listeners=CLIENT://0.0.0.0:9092 (1)
listener.security.protocol.map=CLIENT:SASL_PLAINTEXT (2)
listener.name.client.sasl.enabled.mechanisms=OAUTHBEARER,PLAIN (3)
sasl.mechanism.inter.broker.protocol=OAUTHBEARER (4)
inter.broker.listener.name=CLIENT (5)
listener.name.client.oauthbearer.sasl.server.callback.handler.class=io.strimzi.kafka.oauth.serv
```

```
er.JaasServerOauthValidatorCallbackHandler (6)
listener.name.client.oauthbearer.sasl.jaas.config=org.apache.kafka.common.security.oauthbe
arer.OAuthBearerLoginModule required \ (7)
  oauth.valid.issuer.uri="http://<auth_server>/auth/realms/<realm>" \ (8)
  oauth.jwks.endpoint.uri="https://<auth_server>/auth/realms/<realm>/protocol/openid-
connect/certs" \ (9)
  oauth.username.claim="preferred_username"  \ (10)
  oauth.client.id="kafka-broker" \ (11)
  oauth.client.secret="kafka-secret" \ (12)
  oauth.token.endpoint.uri="https://<oauth_server_address>/token" ; (13)
listener.name.client.oauthbearer.sasl.login.callback.handler.class=io.strimzi.kafka.oauth.client
.JaasClientOauthLoginCallbackHandler (14)
listener.name.client.plain.sasl.server.callback.handler.class=io.strimzi.kafka.oauth.server.plain
.JaasServerOauthOverPlainValidatorCallbackHandler (15)
listener.name.client.plain.sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLogi
nModule required \ (16)
  oauth.valid.issuer.uri="https://<oauth_server_address>" \ (17)
  oauth.jwks.endpoint.uri="https://<oauth_server_address>/jwks" \ (18)
  oauth.username.claim="preferred_username"  \ (19)
  oauth.token.endpoint.uri="http://<auth_server>/auth/realms/<realm>/protocol/openid-
connect/token" ; (20)
connections.max.reauth.ms=3600000 (21)
```

1. 为要连接的客户端应用程序配置监听程序（本例中为 **CLIENT** ）。系统的 **hostname** 用作公告的主机名，客户端必须可以解析该主机名才能重新连接。由于这是唯一配置的监听程序，它也用于代理间通信。

2. 将示例 **CLIENT** 侦听器配置为通过未加密的频道使用 SASL。在生产环境中，客户端应使用加密频道 (**SASL_SSL**)来保护对 TCP 连接层进行窃取和截获。

3. 为通过 SASL 和 *OAUTHBEARER* 进行凭证交换启用 *PLAIN* 身份验证机制。OAUTHBEARER 也被指定，因为代理间通信需要它。Kafka 客户端可以选择使用哪些机制进行连接。

4. 为代理间通信指定 *OAUTHBEARER* 身份验证机制。

5. 为内部代理通信指定监听器（本例中为 **CLIENT**）。配置需要有效。

6. 为 OAUTHBEARER 机制配置服务器回调处理程序。

7. 使用 *OAUTHBEARER* 机制为客户端和内部代理通信配置身份验证设置。**oauth.client.id**、**oauth.client.secret** 和 **oauth.token.endpoint.uri** 属性与 inter-broker 配置相关。

8. 有效的签发者 URI。只有来自此签发者的访问令牌才被接受。例如： *https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME*

9. JWKS 端点 URL。例如 ：*https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME/protocol/openid-connect/certs*

10. 在令牌中包含实际用户名的令牌声明（或密钥）。用户名是用于识别用户的 *主体*。该值将取决于身份验证流和使用的授权服务器。如果需要，您可以使用 JsonPath 表达式，如 **"['user.info'].['user.id']"**，从令牌中的嵌套 JSON 属性检索用户名。

11. Kafka 代理的客户端 ID，适用于所有代理。这是在 授权服务器注册为 **kafka-broker**的客户端。

12 Kafka 代理的 secret（所有代理都相同）。

13 到您的授权服务器的 OAuth 2.0 令牌端点 URL。对于生产环境，请始终使用 **https://** urls。例如 *：https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME/protocol/openid-connect/token*

14 为代理间通信启用 OAuth 2.0 身份验证。

15 为 *PLAIN* 身份验证配置服务器回调处理程序。

16 使用 *PLAIN* 身份验证为客户端通信配置身份验证设置。

oauth.token.endpoint.uri 是一个可选属性，它使用 *OAuth 2.0 客户端凭证机制*启用 OAuth 2.0 over PLAIN。

17 有效的签发者 URI。只有来自此签发者的访问令牌才被接受。例如： *https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME*

18 JWKS 端点 URL。例如 *：https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME/protocol/openid-connect/certs*

19 在令牌中包含实际用户名的令牌声明（或密钥）。用户名是用于识别用户的 *主体*。该值将取决于身份验证流和使用的授权服务器。如果需要，您可以使用 JsonPath 表达式，如 **"['user.info'].['user.id']"**，从令牌中的嵌套 JSON 属性检索用户名。

20 到您的授权服务器的 OAuth 2.0 令牌端点 URL。PLAIN 机制的其他配置。如果指定，客户端可以通过在使用 **$accessToken:** 前缀将访问令牌 作为密码 传递来通过 PLAIN 进行身份验证。

对于生产环境，请始终使用 **https://** urls。例如： *https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME/protocol/openid-connect/token*。

21 （可选）当令牌过期时强制会话到期，并激活 Kafka 重新验证机制。如果指定的值小于访问令牌保留的时间，则客户端必须在实际令牌到期前重新验证。默认情况下，当访问令牌过期时，会话不会过期，客户端也不会尝试重新身份验证。

6.7.2.3. 快速的本地 JWT 令牌验证配置

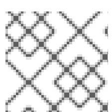
快速本地 JWT 令牌验证会在本地检查 JWT 令牌签名。

本地检查可确保令牌：

- 使用一个访问令牌的 **Bearer** 的(*typ*)声明值符合类型
- 有效（未过期）
- 具有与 **validIssuerURI** 匹配的签发者

在配置监听程序时 *指定有效的签发者 URI*，以便任何未由授权服务器发布的令牌都被拒绝。

授权服务器不需要在快速本地 JWT 令牌验证过程中联系。您可以通过指定由 OAuth 2.0 授权服务器公开的 *JWKs 端点 URI* 来激活快速本地 JWT 令牌验证。端点包含验证已签名的 JWT 令牌的公钥，这些令牌由 Kafka 客户端作为凭证发送。



注意

所有与授权服务器通信都应使用 HTTPS 执行。

对于 TLS 侦听器，您可以配置证书*信任存储*并指向信任存储文件。

快速本地 JWT 令牌验证的属性示例

```
listener.name.client.oauthbearer.sasl.jaas.config=org.apache.kafka.common.security.oauthbe
arer.OAuthBearerLoginModule required \
  oauth.valid.issuer.uri="https://<oauth_server_address>" \ (1)
  oauth.jwks.endpoint.uri="https://<oauth_server_address>/jwks" \ (2)
  oauth.jwks.refresh.seconds="300" \ (3)
  oauth.jwks.refresh.min.pause.seconds="1" \ (4)
  oauth.jwks.expiry.seconds="360" \ (5)
  oauth.username.claim="preferred_username" \ (6)
  oauth.ssl.truststore.location="<path_to_truststore_p12_file>" \ (7)
  oauth.ssl.truststore.password="<truststore_password>" \ (8)
  oauth.ssl.truststore.type="PKCS12" ; (9)
```

1. 有效的签发者 URI。只有此签发者发布的访问令牌才会被接受。例如 *: https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME*
2. JWKS 端点 URL。例如：*https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME/protocol/openid-connect/certs*。
3. 端点刷新之间的周期（默认为 300）。
4. 连续尝试刷新 JWKS 公钥之间的最小暂停（以秒为单位）。当遇到未知签名密钥时，JWKS 密钥刷新在常规定期调度后调度，且至少在最后一次刷新尝试后出现指定暂停。刷新密钥遵循 exponential backoff 规则，重试不成功刷新，且持续增加暂停，直到它到达 **oauth.jwks.refresh.seconds**。默认值为 1。
5. JWKs 证书在证书过期前被视为有效。默认为 **360 秒**。如果您指定了较长的时间，请考虑允许访问撤销的证书的风险。
6. 在令牌中包含实际用户名的令牌声明（或密钥）。用户名是用于识别用户的*主体*。该值将取决于身份验证流和使用的授权服务器。如果需要，您可以使用 JsonPath 表达式，如 **"['user.info'].['user.id']"**，从令牌中的嵌套 JSON 属性检索用户名。
7. TLS 配置中使用的信任存储的位置。
8. 访问信任存储的密码。
9. PKCS #12 格式的 truststore 类型。

6.7.2.4. OAuth 2.0 内省端点配置

使用 OAuth 2.0 内省端点进行令牌验证会将接收的访问令牌视为不透明。Kafka 代理向内省端点发送访问令牌，该端点使用验证所需的令牌信息做出响应。最重要的是，如果特定访问令牌有效，它会返回最新的信息，以及令牌何时过期的信息。

要配置基于 OAuth 2.0 内省的验证，您可以指定一个*内省端点 URI*，而不是为快速本地 JWT 令牌验证指定的 JWKs 端点 URI。根据授权服务器，通常必须指定 *client ID* 和 *client secret*，因为内省端点通常受到保护。

内省端点的属性示例

```
listener.name.client.oauthbearer.sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule required \
  oauth.introspection.endpoint.uri="https://<oauth_server_address>/introspection" \ (1)
  oauth.client.id="kafka-broker" \ (2)
  oauth.client.secret="kafka-broker-secret" \ (3)
  oauth.ssl.truststore.location="<path_to_truststore_p12_file>" \ (4)
  oauth.ssl.truststore.password="<truststore_password>" \ (5)
  oauth.ssl.truststore.type="PKCS12" \ (6)
  oauth.username.claim="preferred_username" ; (7)
```

1. OAuth 2.0 内省端点 URI。例如： *https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME/protocol/openid-connect/token/introspect*。
2. Kafka 代理的客户端 ID。
3. Kafka 代理的 secret。
4. TLS 配置中使用的信任存储的位置。
5. 访问信任存储的密码。
6. PKCS #12 格式的 truststore 类型。
7. 在令牌中包含实际用户名的令牌声明（或密钥）。用户名是用于识别用户的 *主体*。该值将取决于身份验证流和使用的授权服务器。如果需要，您可以使用 JsonPath 表达式，如 **"['user.info'].['user.id']"**，从令牌中的嵌套 JSON 属性检索用户名。

6.7.3. Kafka 代理的会话重新身份验证

您可以将 OAuth 侦听程序配置为使用 Kafka *会话在 Kafka 客户端和 Kafka 代理之间对 OAuth 2.0 会话进行重新身份验证*。这个机制在定义的时间后强制实施客户端和代理之间经过身份验证的会话的过期。当会话过期时，客户端会立即通过重复使用现有连接而不是丢弃它来启动新的会话。

会话重新身份验证默认为禁用。您可以在 **server.properties** 文件中启用它。为启用了 OAUTHBEARER 或 PLAIN 的 TLS 侦听器设置 **connections.max.reauth.ms** 属性作为 SASL 机制。

您可以指定每个监听器的会话重新身份验证。例如：

```
listener.name.client.oauthbearer.connections.max.reauth.ms=3600000
```

会话重新身份验证必须由客户端使用的 Kafka 客户端库支持。

会话重新身份验证可用于快速 *本地 JWT* 或 *内省端点* 令牌验证。

客户端重新身份验证

当代理的经过身份验证的会话过期时，客户端必须通过向代理发送一个新的有效访问令牌来重新验证到现有会话，而无需丢弃连接。

如果令牌验证成功，则使用现有连接启动新的客户端会话。如果客户端无法重新验证，代理会在进一步尝试发送或接收消息时关闭连接。如果代理上启用了重新身份验证机制，则使用 Kafka 客户端库 2.2 或更高版本的 Java 客户端会自动重新验证。

如果使用，会话重新身份验证也适用于刷新令牌。当会话过期时，客户端会使用其刷新令牌来刷新访问令牌。然后，客户端使用新的访问令牌来重新验证现有连接。

OAUTHBEARER 和 PLAIN 的会话到期

配置会话重新身份验证后，会话到期对于 OAUTHBEARER 和 PLAIN 身份验证是不同的。

对于 OAUTHBEARER 和 PLAIN，使用 *客户端 ID 和 secret* 方法：

- 代理的身份验证会话将在配置的 connections.max.reauth.ms 过期。
- 如果访问令牌在配置的时间前过期，会话将提前过期。

对于 PLAIN，使用 *长期访问令牌* 方法：

- 代理的身份验证会话将在配置的 connections.max.reauth.ms 过期。
- 如果访问令牌在配置的时间前过期，则重新身份验证将失败。虽然尝试尝试会话重新身份验证，但 PLAIN 没有刷新令牌的机制。

如果*没有*配置 connection.max.reauth.ms， OAUTHBEARER 和 PLAIN 客户端可以无限期地连接到代理，而无需重新验证。经过身份验证的会话不会因为访问令牌到期而终止。但是，这可在配置授权时考虑，例如，使用 keycloak 授权或安装自定义授权器。

其他资源

- OAuth 2.0 Kafka 代理配置
- 为 Kafka 代理配置 OAuth 2.0 支持
- kIP-368 ：允许 SASL 连接定期重新验证

6.7.4. OAuth 2.0 Kafka 客户端配置

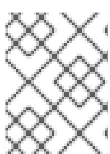
Kafka 客户端被配置为：

- 从授权服务器获取有效访问令牌（客户端 ID 和 Secret）所需的凭证
- 使用授权服务器提供的工具获取有效的长期访问令牌或刷新令牌

发送到 Kafka 代理的唯一信息是访问令牌。用于与授权服务器进行身份验证的凭证从不会发送到代理。

当客户端获取访问令牌时，不需要进一步与授权服务器通信。

最简单的机制是使用客户端 ID 和 Secret 进行身份验证。使用长期的访问令牌或长期的刷新令牌会增加复杂性，因为对授权服务器工具还有额外的依赖。



注意

如果您使用长期的访问令牌，您可能需要在授权服务器中配置客户端来提高令牌的最大生命周期。

如果 Kafka 客户端没有直接配置访问令牌，客户端会在 Kafka 会话发起授权服务器的过程中交换访问令牌的凭证。Kafka 客户端交换：

- 客户端 ID 和 Secret

- 客户端 ID、刷新令牌和（可选）secret
- 使用客户端 ID 和（可选）secret 的用户名和密码

6.7.5. OAuth 2.0 客户端身份验证流

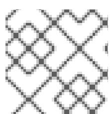
OAuth 2.0 身份验证流程取决于底层 Kafka 客户端和 Kafka 代理配置。流还必须由使用的授权服务器支持。

Kafka 代理监听程序配置决定客户端如何使用访问令牌进行身份验证。客户端可以传递客户端 ID 和机密来请求访问令牌。

如果侦听器配置为使用 PLAIN 身份验证，客户端可以通过客户端 ID 和 secret 或用户名和访问令牌进行身份验证。这些值作为 PLAIN 机制 **username** 和 **password** 属性传递。

侦听器配置支持以下令牌验证选项：

- 您可以根据 JWT 签名检查和本地令牌内省使用快速本地令牌验证，而无需联系授权服务器。授权服务器提供带有公共证书的 JWKS 端点，用于验证令牌中的签名。
- 您可以使用调用授权服务器提供的令牌内省端点。每次建立新的 Kafka 代理连接时，代理会将从客户端接收的访问令牌传递给授权服务器。Kafka 代理检查响应，以确认令牌是否有效。



注意

授权服务器可能只允许使用不透明访问令牌，这意味着无法进行本地令牌验证。

也可以为以下类型的身份验证配置 Kafka 客户端凭证：

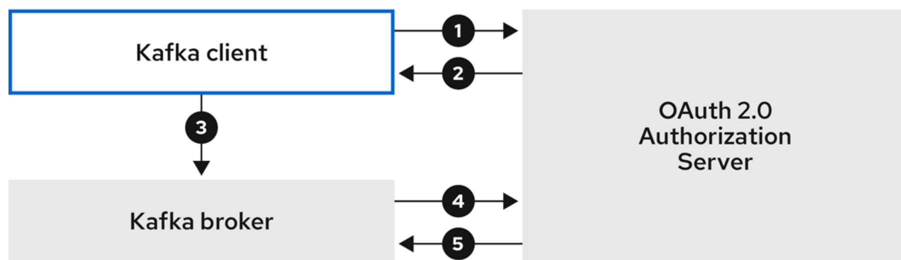
- 使用之前生成的长期访问令牌直接进行本地访问
- 与授权服务器联系，以获取要发布的新访问令牌（使用客户端 ID 和 secret，或刷新令牌，或者用户名和密码）

6.7.5.1. 使用 SASL OAUTHBEARER 机制的客户端身份验证流示例

您可以使用 SASL OAUTHBEARER 机制为 Kafka 身份验证使用以下通信流。

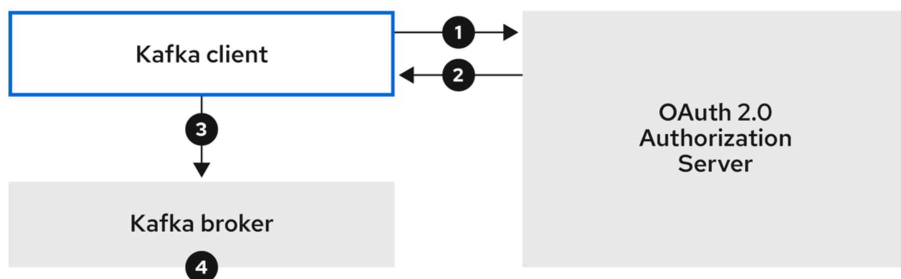
- 使用客户端 ID 和 secret 的客户端以及代理委派到授权服务器的验证
- 使用客户端 ID 和 secret 的客户端，代理执行快速本地令牌验证
- 使用长期访问令牌的客户端，带有代理委派验证到授权服务器
- 使用长期访问令牌的客户端，代理执行快速本地验证

使用客户端 ID 和 secret 的客户端以及代理委派到授权服务器的验证



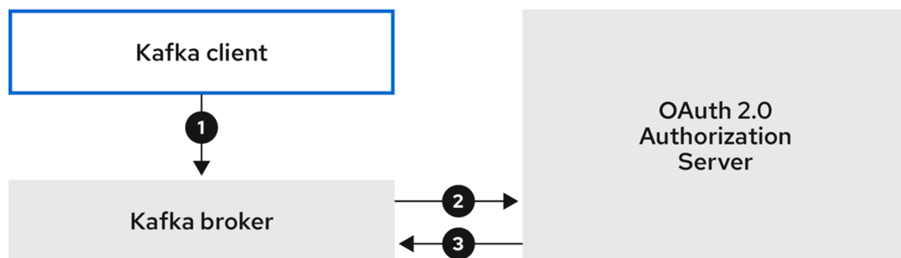
1. Kafka 客户端使用客户端 ID 和 secret 从授权服务器请求访问令牌，以及可选的刷新令牌。或者，客户端也可以使用用户名和密码进行身份验证。
2. 授权服务器生成新的访问令牌。
3. Kafka 客户端使用 SASL OAUTHBEARER 机制通过 Kafka 代理进行身份验证，以传递访问令牌。
4. Kafka 代理通过使用自己的客户端 ID 和 secret，在授权服务器上调用令牌内省端点来验证访问令牌。
5. 如果令牌有效，则会建立 Kafka 客户端会话。

使用客户端 ID 和 secret 的客户端，代理执行快速本地令牌验证



1. Kafka 客户端使用令牌端点、使用客户端 ID 和 secret 以及刷新令牌（可选）从令牌端点验证。或者，客户端也可以使用用户名和密码进行身份验证。
2. 授权服务器生成新的访问令牌。
3. Kafka 客户端使用 SASL OAUTHBEARER 机制通过 Kafka 代理进行身份验证，以传递访问令牌。
4. Kafka 代理使用 JWT 令牌签名检查和本地令牌内省验证访问令牌。

使用长期访问令牌的客户端，带有代理委派验证到授权服务器



1. Kafka 客户端使用 SASL OAUTHBEARER 机制通过 Kafka 代理进行身份验证，以传递长期访问令牌。

2. Kafka 代理通过使用自己的客户端 ID 和 secret，在授权服务器上调用令牌内省端点来验证访问令牌。

3. 如果令牌有效，则会建立 Kafka 客户端会话。

使用长期访问令牌的客户端，代理执行快速本地验证

1. Kafka 客户端使用 SASL OAUTHBEARER 机制通过 Kafka 代理进行身份验证，以传递长期访问令牌。

2. Kafka 代理使用 JWT 令牌签名检查和本地令牌内省验证访问令牌。

警告

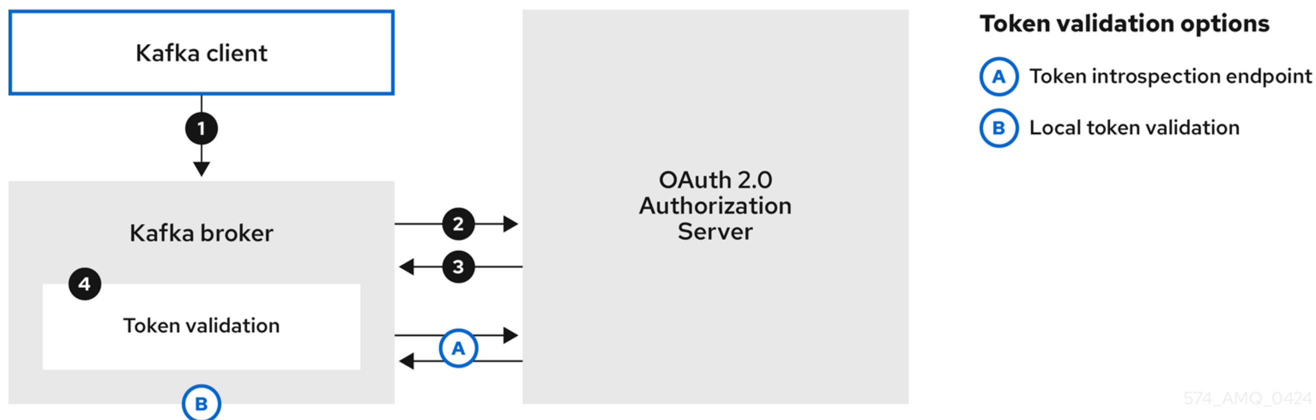
快速的本地 JWT 令牌签名验证仅适用于短期的令牌，因为如果已撤销令牌，就不会通过授权服务器检查该授权服务器。令牌到期时间写入到令牌，但可以随时进行撤销，因此不能在不联系授权服务器的情况下被考虑。任何发布的令牌都将被视为有效，直到过期为止。

6.7.5.2. 使用 SASL PLAIN 机制的客户端身份验证流示例

您可以使用 OAuth PLAIN 机制对 Kafka 身份验证使用以下通信流。

- 使用客户端 ID 和 secret 的客户端以及代理获取客户端的访问令牌
- 使用没有客户端 ID 和 secret 的长期访问令牌的客户端

使用客户端 ID 和 secret 的客户端以及代理获取客户端的访问令牌



1. Kafka 客户端或传递一个 clientId 作为用户名，以及一个 secret 作为密码。

2. Kafka 代理使用令牌端点将 clientId 和 secret 传递给授权服务器。

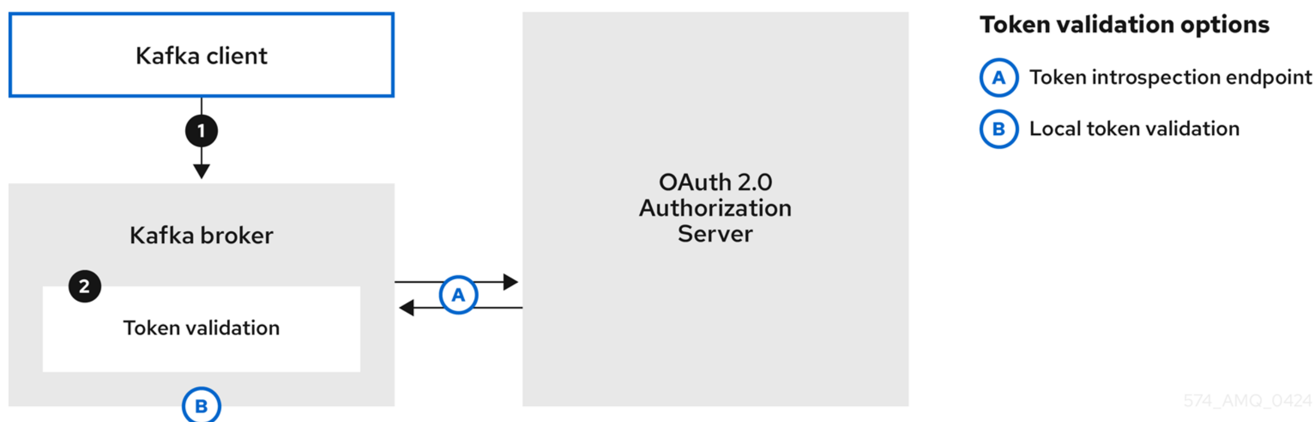
3. 如果客户端凭据无效，授权服务器会返回一个新的访问令牌或错误。

4. Kafka 代理使用以下方法之一验证令牌：

   a. 如果指定了令牌内省端点，Kafka 代理会通过调用授权服务器上的端点来验证访问令牌。如果令牌验证成功，则会建立会话。

   b. 如果使用本地令牌内省，则不会向授权服务器发出请求。Kafka 代理使用 JWT 令牌签名检查在本地验证访问令牌。

使用没有客户端 ID 和 secret 的长期访问令牌的客户端



1. Kafka 客户端会传递用户名和密码。密码提供在运行客户端前手动配置的访问令牌值。

2. 密码通过或不使用 $accessToken: 字符串前缀来传递，具体取决于 Kafka 代理侦听程序是否配置了令牌端点来进行身份验证。

   a. 如果配置了令牌端点，则密码应加上前缀 $accessToken:，以便代理知道 password 参数包含访问令牌，而不是客户端 secret。Kafka 代理将用户名解释为帐户用户名。

   b. 如果没有在 Kafka 代理监听程序上配置令牌端点（强制 no-client-credentials 模式），则密码应在没有前缀的情况下提供访问令牌。Kafka 代理将用户名解释为帐户用户名。在这个模式中，客户端不使用客户端 ID 和 secret，password 参数始终解释为原始访问令牌。

3. Kafka 代理使用以下方法之一验证令牌：

a. 如果指定了令牌内省端点，Kafka 代理会通过调用授权服务器上的端点来验证访问令牌。如果令牌验证成功，则会建立会话。

b. 如果使用本地令牌内省，则不会向授权服务器发出请求。Kafka 代理使用 JWT 令牌签名检查在本地验证访问令牌。

6.7.6. 配置 OAuth 2.0 身份验证

OAuth 2.0 用于 Kafka 客户端和流用于 Apache Kafka 组件之间的交互。

要将 OAuth 2.0 用于 Apache Kafka，您必须：

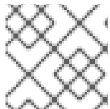
1. 为 Apache Kafka 集群和 Kafka 客户端的流配置 OAuth 2.0 授权服务器
2. 使用配置为使用 OAuth 2.0 的 Kafka 代理监听程序部署或更新 Kafka 集群
3. 更新基于 Java 的 Kafka 客户端以使用 OAuth 2.0

6.7.6.1. 将 Red Hat Single Sign-On 配置为 OAuth 2.0 授权服务器

这个步骤描述了如何将 Red Hat Single Sign-On 部署为授权服务器，并配置它以与 Apache Kafka 的 Streams 集成。

授权服务器为身份验证和授权提供了一个中央点，以及用户、客户端和权限的管理。Red Hat Single Sign-On 具有域概念，其中 *realm* 代表一组独立的用户、客户端、权限和其他配置。您可以使用默认 *master 域*，或创建新域。每个 realm 会公开自己的 OAuth 2.0 端点，这意味着应用程序客户端和应用服务器都需要使用相同的域。

要将 OAuth 2.0 与 Apache Kafka 的 Streams 搭配使用，您可以使用部署 Red Hat Single Sign-On 来创建和管理身份验证域。



注意

如果您已经部署了 Red Hat Single Sign-On，您可以跳过部署步骤并使用您的当前部署。

开始前

您将需要熟悉使用 Red Hat Single Sign-On。

有关安装和管理说明，请参阅：

- 服务器安装和配置指南
- 服务器管理指南

先决条件

- Apache Kafka 和 Kafka 的流正在运行

对于 Red Hat Single Sign-On 部署：

- 检查 Red Hat Single Sign-On 支持的配置

流程

1. 安装 Red Hat Single Sign-On。
   您可以从 ZIP 文件或使用 RPM 安装。

2. 登录到 Red Hat Single Sign-On Admin 控制台，为 Apache Kafka 创建流的 OAuth 2.0 策略。
   部署 Red Hat Single Sign-On 时会提供登录详情。

3. 创建并启用 realm。
   您可以使用现有的 master 域。

4. 如果需要，调整域的会话和令牌超时。

5. 创建名为 **kafka-broker** 的客户端。

6. 在 Settings 选项卡中设置：
   - 访问类型 以 机密
   - Standard Flow Enabled 为 **OFF** 为这个客户端禁用 Web 登录
   - Service Accounts Enabled 为 **ON**，允许此客户端在其自己的名称中进行身份验证

7. 在继续操作前，点 Save。

8. 在 Credentials 选项卡中，记录用于 Apache Kafka 集群配置的流中使用的 secret。

9. 对将连接到 Kafka 代理的任何应用程序客户端重复客户端创建步骤。
   为每个新客户端创建一个定义。

   在配置中，您将使用名称作为客户端 ID。

接下来要做什么

部署和配置授权服务器后，将 Kafka 代理配置为使用 OAuth 2.0。

6.7.6.2. 为 Kafka 代理配置 OAuth 2.0 支持

此流程描述了如何配置 Kafka 代理，以便代理监听程序可以使用授权服务器使用 OAuth 2.0 身份验证。

我们建议通过配置 TLS 侦听程序在加密接口中使用 OAuth 2.0。不建议使用 plain 监听程序。

使用支持您选择的授权服务器的属性配置 Kafka 代理，以及您要实现的授权类型。

开始前

有关 Kafka 代理监听程序的配置和验证的更多信息，请参阅：

- 监听器
- OAuth 2.0 身份验证机制

有关监听器配置中使用的属性的描述，请参阅：

- OAuth 2.0 Kafka 代理配置

先决条件

- 每个主机上安装了 Apache Kafka 的流，且配置文件可用。

- 部署 OAuth 2.0 授权服务器。

流程

1. 在 **server.properties** 文件中配置 Kafka 代理监听程序配置。
   例如，使用 OAUTHBEARER 机制：

```
sasl.enabled.mechanisms=OAUTHBEARER
listeners=CLIENT://0.0.0.0:9092
listener.security.protocol.map=CLIENT:SASL_PLAINTEXT
listener.name.client.sasl.enabled.mechanisms=OAUTHBEARER
sasl.mechanism.inter.broker.protocol=OAUTHBEARER
inter.broker.listener.name=CLIENT
listener.name.client.oauthbearer.sasl.server.callback.handler.class=io.strimzi.kafka.oa
uth.server.JaasServerOauthValidatorCallbackHandler
listener.name.client.oauthbearer.sasl.jaas.config=org.apache.kafka.common.security.
oauthbearer.OAuthBearerLoginModule required ;
listener.name.client.oauthbearer.sasl.login.callback.handler.class=io.strimzi.kafka.oau
th.client.JaasClientOauthLoginCallbackHandler
```

2. 将代理连接设置配置为 **listener.name.client.oauthbearer.sasl.jaas.config** 的一部分。
   此处的示例显示连接配置选项。

   示例 1：使用 JWKS 端点配置进行本地令牌验证

```
listener.name.client.oauthbearer.sasl.jaas.config=org.apache.kafka.common.security.
oauthbearer.OAuthBearerLoginModule required \
  oauth.valid.issuer.uri="https://<oauth_server_address>/auth/realms/<realm_name>" \

oauth.jwks.endpoint.uri="https://<oauth_server_address>/auth/realms/<realm_name>/
protocol/openid-connect/certs" \
  oauth.jwks.refresh.seconds="300" \
  oauth.jwks.refresh.min.pause.seconds="1" \
  oauth.jwks.expiry.seconds="360" \
  oauth.username.claim="preferred_username" \
  oauth.ssl.truststore.location="<path_to_truststore_p12_file>" \
  oauth.ssl.truststore.password="<truststore_password>" \
  oauth.ssl.truststore.type="PKCS12" ;
listener.name.client.oauthbearer.connections.max.reauth.ms=3600000
```

   示例 2：通过 OAuth 2.0 内省端点将令牌验证委托给授权服务器

```
listener.name.client.oauthbearer.sasl.jaas.config=org.apache.kafka.common.security.
oauthbearer.OAuthBearerLoginModule required \

oauth.introspection.endpoint.uri="https://<oauth_server_address>/auth/realms/<realm
_name>/protocol/openid-connect/introspection" \
  # ...
```

3. 如果需要，配置对授权服务器的访问。
   生产环境通常需要这一步，除非使用 *service mesh* 等技术来配置容器外的安全频道。

   a. 提供用于连接安全授权服务器的自定义信任存储。对授权服务器的访问始终需要 SSL。
      设置属性来配置信任存储。

例如：

```
listener.name.client.oauthbearer.sasl.jaas.config=org.apache.kafka.common.secur
ity.oauthbearer.OAuthBearerLoginModule required \
  # ...
  oauth.client.id="kafka-broker" \
  oauth.client.secret="kafka-broker-secret" \
  oauth.ssl.truststore.location="<path_to_truststore_p12_file>" \
  oauth.ssl.truststore.password="<truststore_password>" \
  oauth.ssl.truststore.type="PKCS12" ;
```

b. 如果证书主机名与访问 URL 主机名不匹配，您可以关闭证书主机名验证：

```
oauth.ssl.endpoint.identification.algorithm=""
```

检查可确保与授权服务器的连接是真实的。您可能想要在非生产环境中关闭验证。

4. 根据您选择的身份验证流配置附加属性：

```
listener.name.client.oauthbearer.sasl.jaas.config=org.apache.kafka.common.security.
oauthbearer.OAuthBearerLoginModule required \
  # ...

oauth.token.endpoint.uri="https://<oauth_server_address>/auth/realms/<realm_name>
/protocol/openid-connect/token" \ (1)
  oauth.custom.claim.check="@.custom == 'custom-value'" \ (2)
  oauth.scope="<scope>" \ (3)
  oauth.check.audience="true" \ (4)
  oauth.audience="<audience>" \ (5)
  oauth.valid.issuer.uri="https://https://<oauth_server_address>/auth/<realm_name>" \
(6)
  oauth.client.id="kafka-broker" \ (7)
  oauth.client.secret="kafka-broker-secret" \ (8)
  oauth.connect.timeout.seconds=60 \ (9)
  oauth.read.timeout.seconds=60 \ (10)
  oauth.http.retries=2 \ (11)
  oauth.http.retry.pause.millis=300 \ (12)
  oauth.groups.claim="$.groups" \ (13)
  oauth.groups.claim.delimiter="," \ (14)
  oauth.include.accept.header="false" ; (15)
```

(1) 到您的授权服务器的 OAuth 2.0 令牌端点 URL。对于生产环境，请始终使用 https:// urls。当使用 KeycloakAuthorizer 或启用了 OAuth 2.0 的监听程序时，需要用于代理间通信。

(2) （可选） 自定义声明检查。JsonPath 过滤器查询，用于在验证期间将其他自定义规则应用到 JWT 访问令牌。如果访问令牌不包含必要的数据，它将被拒绝。使用 *introspection* 端点方法时，自定义检查将应用到内省端点响应 JSON。

(3) （可选）传递给令牌端点 的范围 参数。获取访问令牌进行代理身份验证时使用的 *scope*。它还在使用 clientId 和 secret 的 PLAIN 客户端验证中用于 OAuth 2.0 的客户端名称。这只会影响获取令牌的能力，以及令牌的内容，具体取决于授权服务器。它不会影响监听器的令牌验证规则。

4. （可选）对象 检查。如果您的授权服务器提供 aud (audience)声明，并且希望强制进行受众检查，请将 **ouath.check.audience** 设置为 **true**。Audience 检查标识令牌的预期接收者。因此，Kafka 代理将拒绝在其 aud 声明中没有 **clientId** 的令牌。默认为 **false**。

5. （可选）传递给令牌端点的 **audience** 参数。在获取用于代理身份验证的访问令牌时，需要使用 *audience*。它还在使用 **clientId** 和 **secret** 的 PLAIN 客户端验证中用于 OAuth 2.0 的客户端名称。这只会影响获取令牌的能力，以及令牌的内容，具体取决于授权服务器。它不会影响监听器的令牌验证规则。

6. 有效的签发者 URI。只有此签发者发布的访问令牌才会被接受。（始终需要。）

7. Kafka 代理配置的客户端 ID，适用于所有代理。这是在 授权服务器注册为 **kafka-broker**的客户端。当内省端点用于令牌验证时，或使用 **KeycloakAuthorizer** 时是必需的。

8. Kafka 代理配置的 secret，适用于所有代理。当代理必须向授权服务器进行身份验证时，必须指定客户端 secret、访问令牌或刷新令牌。

9. （可选）连接到授权服务器时的连接超时（以秒为单位）。默认值为 60。

10. （可选）连接到授权服务器时读取超时（以秒为单位）。默认值为 60。

11. 将失败的 HTTP 请求重试到授权服务器的次数上限。默认值为 0，表示不会执行重试。要有效地使用这个选项，请考虑减少 **oauth.connect.timeout.seconds** 和 **oauth.read.timeout.seconds** 选项的超时时间。但请注意，重试可能会阻止当前 worker 线程可用于其他请求，如果太多请求停滞，则可能会导致 Kafka 代理无响应。

12. 尝试另一个到授权服务器的 HTTP 请求重试前等待的时间。默认情况下，这个时间被设置为零，这意味着不会应用暂停。这是因为导致失败请求的许多问题是针对每个请求的网络粘合或代理问题，可以快速解决。但是，如果您的授权服务器处于压力下或高流量，您可能希望将此选项设置为值 100 ms 或更多，以减少服务器上的负载，并增加成功重试的可能性。

13. JsonPath 查询，用于从 JWT 令牌或内省端点响应中提取组信息。默认不设置。这可供自定义授权器用于根据用户组做出授权决策。

14. 当以单一分隔的字符串返回时，用于解析组信息的分隔符。默认值为 ','(comma)。

15. （可选）将 **oauth.include.accept.header** 设置为 **false**，以从请求中删除 **Accept** 标头。如果包含标头在与授权服务器通信时导致问题，您可以使用此设置。

5. 根据您如何应用 OAuth 2.0 身份验证以及所使用的授权服务器类型，添加额外的配置设置：

```
listener.name.client.oauthbearer.sasl.jaas.config=org.apache.kafka.common.security.
oauthbearer.OAuthBearerLoginModule required \
  # ...
  oauth.check.issuer=false \ (1)
  oauth.fallback.username.claim="<client_id>" \ (2)
  oauth.fallback.username.prefix="<client_account>" \ (3)
  oauth.valid.token.type="bearer" \ (4)

oauth.userinfo.endpoint.uri="https://<oauth_server_address>/auth/realms/<realm_nam
e>/protocol/openid-connect/userinfo" ; (5)
```

1. 如果您的授权服务器不提供 iss 声明，则无法执行签发者检查。在这种情况下，将 **oauth.check.issuer** 设置为 **false**，且不指定 **oauth.valid.issuer.uri**。默认为 **true**。

2. 授权服务器可能无法提供单个属性来识别常规用户和客户端。当客户端以自己的名称进行身份验证时，服务器可能会提供 *客户端 ID*。当用户使用用户名和密码进行身份验证时，若要获

3. 当 oauth.fallback.username.claim 适用的情况下，可能还需要防止用户名声明的值与回退用户名声明之间的名称冲突。请考虑存在名为 producer 的客户端存在的情况，但也存在名为 producer 的常规用户。为了区分这两者，您可以使用此属性向客户端的用户 ID 添加前缀。

4. （仅在使用 oauth.introspection.endpoint.uri）取决于您使用的授权服务器，内省端点可能会或不返回 *令牌类型* 属性，或者可以包含不同的值。您可以指定来自内省端点的响应必须包含有效的令牌类型值。

5. （仅在使用 oauth.introspection.endpoint.uri）时，可以配置或实施授权服务器，以便在内省端点响应中提供任何可识别的信息。要获取用户 ID，您可以将 userinfo 端点的 URI 配置为回退。oauth.fallback.username.claim、oauth.fallback.username.claim 和 oauth.fallback.username.prefix 设置应用到 userinfo 端点的响应。

接下来要做什么

- 将您的 Kafka 客户端配置为使用 OAuth 2.0

6.7.6.3. 将 Kafka Java 客户端配置为使用 OAuth 2.0

配置 Kafka producer 和消费者 API，以使用 OAuth 2.0 与 Kafka 代理交互。在客户端 pom.xml 文件中添加回调插件，然后为 OAuth 2.0 配置您的客户端。

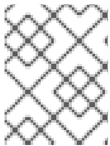
在客户端配置中指定以下内容：

- SASL（简单身份验证和安全层）安全协议：
  - SASL_SSL 用于通过 TLS 加密连接进行身份验证
  - SASL_PLAINTEXT 用于通过未加密的连接进行身份验证
    将 SASL_SSL 用于生产环境，SASL_PLAINTEXT 仅用于本地开发。使用 SASL_SSL 时，需要额外的 ssl.truststore 配置。安全连接(https://)需要 truststore 配置到 OAuth 2.0 授权服务器。要验证 OAuth 2.0 授权服务器，请将授权服务器的 CA 证书添加到客户端配置的信任存储中。您可以使用 PEM 或 PKCS #12 格式配置信任存储。
- Kafka SASL 机制：
  - OAUTHBEARER 用于使用 bearer 令牌交换的凭证
  - PLAIN 传递客户端凭证(clientId + secret)或访问令牌
- 实现 SASL 机制的 JAAS (Java 身份验证和授权服务)模块：
  - org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule 实现 OAuthbearer 机制
  - org.apache.kafka.common.security.plain.PlainLoginModule 实现普通机制

  为了可以使用 OAuthbearer 机制，还必须将自定义 io.strimzi.kafka.oauth.client.JaasClientOauthLoginCallbackHandler 类添加为回调处理程序。JaasClientOauthLoginCallbackHandler 处理 OAuth 回调到授权服务器，以便在客户端登录期间处理访问令牌。这可启用自动令牌续订，确保在用户不干预的情况下进行持续身份验证。另外，它使用 OAuth 2.0 密码授权方法为客户端处理登录凭证。

- SASL 验证方法，它支持以下验证方法：
  - OAuth 2.0 客户端凭证
  - OAuth 2.0 密码授权（已弃用）
  - 访问令牌
  - 刷新令牌

  将 SASL 身份验证属性添加为 JAAS 配置 (**sasl.jaas.config** 和 **sasl.login.callback.handler.class**)。如何配置身份验证属性取决于您用来访问 OAuth 2.0 授权服务器的身份验证方法。在此过程中，属性在属性文件中指定，然后加载到客户端配置中。



注意

您还可以将身份验证属性指定为环境变量，或指定为 Java 系统属性。对于 Java 系统属性，您可以使用 **setProperty** 设置它们，并使用 **-D** 选项在命令行中传递它们。

先决条件

- Apache Kafka 和 Kafka 的流正在运行
- 部署和配置 OAuth 2.0 授权服务器以 OAuth 访问 Kafka 代理
- 为 OAuth 2.0 配置 Kafka 代理

流程

1. 将支持 OAuth 2.0 的客户端库添加到 Kafka 客户端的 **pom.xml** 文件中：

```
<dependency>
 <groupId>io.strimzi</groupId>
 <artifactId>kafka-oauth-client</artifactId>
 <version>0.15.0.redhat-00006</version>
</dependency>
```

2. 通过在属性文件中指定以下配置来配置客户端属性：
   - 安全协议
   - SASL 机制
   - JAAS 模块和身份验证属性，具体取决于所使用的方法
     例如，我们可以将以下内容添加到 **client.properties** 文件中：

   客户端凭证机制属性

```
security.protocol=SASL_SSL (1)
sasl.mechanism=OAUTHBEARER (2)
ssl.truststore.location=/tmp/truststore.p12 (3)
ssl.truststore.password=$STOREPASS
ssl.truststore.type=PKCS12
sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule required \
```

```
  oauth.token.endpoint.uri="<token_endpoint_url>" \ (4)
  oauth.client.id="<client_id>" \ (5)
  oauth.client.secret="<client_secret>" \ (6)
  oauth.ssl.truststore.location="/tmp/oauth-truststore.p12" \ (7)
  oauth.ssl.truststore.password="$STOREPASS" \ (8)
  oauth.ssl.truststore.type="PKCS12" \ (9)
  oauth.scope="<scope>" \ (10)
  oauth.audience="<audience>" ; (11)
sasl.login.callback.handler.class=io.strimzi.kafka.oauth.client.JaasClientOauthLog
inCallbackHandler
```

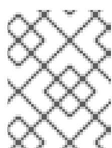
1. SASL_SSL 安全协议用于 TLS 加密连接。仅对本地开发使用SASL_PLAINTEXT。
2. 指定为 OAUTHBEARER 或 PLAIN 的 SASL 机制。
3. 用于安全访问 Kafka 集群的 truststore 配置。
4. 授权服务器令牌端点的 URI。
5. 客户端 ID，这是在授权服务器中创建 *客户端*时使用的名称。
6. 在授权服务器中创建客户端时创建的 *客户端* secret。
7. 该位置包含授权服务器的公钥证书(truststore.p12)。
8. 用于访问 truststore 的密码。
9. truststore 类型。
10. （可选）从令牌端点请求令牌的范围。授权服务器可能需要客户端来指定范围。
11. （可选）从令牌端点请求令牌的听众。授权服务器可能需要客户端来指定受众。

密码授予机制属性

```
security.protocol=SASL_SSL
sasl.mechanism=OAUTHBEARER
ssl.truststore.location=/tmp/truststore.p12
ssl.truststore.password=$STOREPASS
ssl.truststore.type=PKCS12
sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLogi
nModule required \
  oauth.token.endpoint.uri="<token_endpoint_url>" \
  oauth.client.id="<client_id>" \ (1)
  oauth.client.secret="<client_secret>" \ (2)
  oauth.password.grant.username="<username>" \ (3)
  oauth.password.grant.password="<password>" \ (4)
  oauth.ssl.truststore.location="/tmp/oauth-truststore.p12" \
  oauth.ssl.truststore.password="$STOREPASS" \
  oauth.ssl.truststore.type="PKCS12" \
  oauth.scope="<scope>" \
  oauth.audience="<audience>" ;
sasl.login.callback.handler.class=io.strimzi.kafka.oauth.client.JaasClientOauthLog
inCallbackHandler
```

-

1. 客户端 ID，这是在授权服务器中创建*客户端*时使用的名称。
2. （可选）在授权服务器中创建*客户端*时创建的客户端 secret。
3. password 授权身份验证的用户名。OAuth 密码授权配置（用户名和密码）使用 OAuth 2.0 密码授权方法。要使用密码授权，请在您的授权服务器上为客户端创建一个有限权限的用户帐户。帐户应像服务帐户一样操作。在进行身份验证需要用户帐户的环境中使用，但首先考虑使用刷新令牌。
4. 密码以授予密码身份验证。



注意

SASL PLAIN 不支持使用 OAuth 2.0 密码授权方法传递用户名和密码（密码授权）。

访问令牌属性

```
security.protocol=SASL_SSL
sasl.mechanism=OAUTHBEARER
ssl.truststore.location=/tmp/truststore.p12
ssl.truststore.password=$STOREPASS
ssl.truststore.type=PKCS12
sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule required \
  oauth.token.endpoint.uri="<token_endpoint_url>" \
  oauth.access.token="<access_token>" \ (1)
  oauth.ssl.truststore.location="/tmp/oauth-truststore.p12" \
  oauth.ssl.truststore.password="$STOREPASS" \
  oauth.ssl.truststore.type="PKCS12" ;
sasl.login.callback.handler.class=io.strimzi.kafka.oauth.client.JaasClientOauthLoginCallbackHandler
```

1. Kafka 客户端长期的访问令牌。

刷新令牌属性

```
security.protocol=SASL_SSL
sasl.mechanism=OAUTHBEARER
ssl.truststore.location=/tmp/truststore.p12
ssl.truststore.password=$STOREPASS
ssl.truststore.type=PKCS12
sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule required \
  oauth.token.endpoint.uri="<token_endpoint_url>" \
  oauth.client.id="<client_id>" \ (1)
  oauth.client.secret="<client_secret>" \ (2)
  oauth.refresh.token="<refresh_token>" \ (3)
  oauth.ssl.truststore.location="/tmp/oauth-truststore.p12" \
  oauth.ssl.truststore.password="$STOREPASS" \
```

```
  oauth.ssl.truststore.type="PKCS12" ;
sasl.login.callback.handler.class=io.strimzi.kafka.oauth.client.JaasClientOauthLog
inCallbackHandler
```

1. 客户端 ID，这是在授权服务器中创建*客户端*时使用的名称。
2. （可选）在授权服务器中创建*客户端*时创建的客户端 secret。
3. Kafka 客户端长期刷新令牌。

3. 在 Java 客户端代码中输入 OAUTH 2.0 身份验证的客户端属性。

   显示客户端属性输入示例

```
Properties props = new Properties();
try (FileReader reader = new FileReader("client.properties", StandardCharsets.UTF_8))
{
  props.load(reader);
}
```

4. 验证 Kafka 客户端是否可以访问 Kafka 代理。

6.8. 使用基于 OAUTH 2.0 令牌的授权

如果您在 Red Hat Single Sign-On 中使用 OAuth 2.0 进行基于令牌的身份验证，您还可以使用 Red Hat Single Sign-On 来配置授权规则来限制客户端对 Kafka 代理的访问。身份验证建立用户的身份。授权决定该用户的访问权限级别。

Apache Kafka 的流支持通过 Red Hat Single Sign-On Authorization Services 使用基于 OAuth 2.0 令牌的授权，它允许您集中管理安全策略和权限。

Red Hat Single Sign-On 中定义的安全策略和权限用于授予对 Kafka 代理上资源的访问权限。用户和客户端与允许对 Kafka 代理执行特定操作的策略进行匹配。

Kafka 允许所有用户默认对代理进行完全访问，并提供 AclAuthorizer 和 StandardAuthorizer 插件来配置基于访问控制列表(ACL)的授权。由这些插件管理的 ACL 规则用于根据*用户名*授予或拒绝对资源的访问，这些规则存储在 Kafka 集群本身中。但是，红帽单点登录基于 OAuth 2.0 令牌的授权在您希望实现对 Kafka 代理的访问控制方面具有更大的灵活性。另外，您可以将 Kafka 代理配置为使用 OAuth 2.0 授权和 ACL。

其他资源

- 使用基于 OAuth 2.0 令牌的身份验证
- Kafka 授权
- Red Hat Single Sign-On 文档

6.8.1. OAuth 2.0 授权机制

Apache Kafka 的 Streams 中的 OAuth 2.0 授权使用 Red Hat Single Sign-On 服务器授权服务 REST 端点通过 Red Hat Single Sign-On 扩展基于令牌的身份验证，通过在特定用户上应用定义的安全策略来扩展基于令牌的权限，并为该用户提供授予不同资源的权限列表。策略使用角色和组来匹配用户的权限。

OAuth 2.0 授权根据从 Red Hat Single Sign-On Authorization Services 用户获得的授予者列表在本地强制实施权限。

6.8.1.1. Kafka 代理自定义授权器

Red Hat Single Sign-On *authorizer* (**KeycloakAuthorizer**)提供 Apache Kafka 的 Streams。为了可以使用 Red Hat Single Sign-On 提供的授权服务的 Red Hat Single Sign-On REST 端点，您可以在 Kafka 代理上配置自定义授权器。

授权程序根据需要从授权服务器获取授予权限的列表，并在 Kafka Broker 上本地强制实施授权，为每个客户端请求做出快速授权决策。

6.8.2. 配置 OAuth 2.0 授权支持

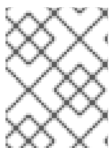
这个步骤描述了如何使用 Red Hat Single Sign-On Authorization Services 将 Kafka 代理配置为使用 OAuth 2.0 授权服务。

开始前

考虑某些用户所需的访问权限或希望限制某些用户。您可以使用 Red Hat Single Sign-On *组*、*角色*、*客户端*和 *用户在* Red Hat Single Sign-On 中配置访问权限的组合。

通常，组用于根据机构部门或地理位置匹配用户。和 角色用于根据其功能匹配用户。

使用红帽单点登录，您可以在 LDAP 中存储用户和组，而客户端和角色不能以这种方式存储。存储和对用户数据的访问可能是您选择配置授权策略的一个因素。



注意

无论在 Kafka 代理上实现的授权是什么，超级用户始终对 Kafka 代理具有不受限制的访问。

先决条件

- Apache Kafka 的流必须配置为在 Red Hat Single Sign-On 中使用 OAuth 2.0 进行基于令牌的身份验证。设置授权时，您可以使用相同的 Red Hat Single Sign-On 服务器端点。
- 您需要了解如何管理 Red Hat Single Sign-On Authorization Services 的策略和权限，如 Red Hat Single Sign-On 文档所述。

流程

1. 访问 Red Hat Single Sign-On Admin 控制台，或使用 Red Hat Single Sign-On Admin CLI 为设置 OAuth 2.0 身份验证时创建的 Kafka 代理客户端启用授权服务。
2. 使用 Authorization Services 为客户端定义资源、授权范围、策略和权限。
3. 通过分配角色和组，将权限绑定到用户和客户端。
4. 将 Kafka 代理配置为使用 Red Hat Single Sign-On 授权。
将以下内容添加到 Kafka **server.properties** 配置文件中，以在 Kafka 中安装授权器：

```
authorizer.class.name=io.strimzi.kafka.oauth.server.authorizer.KeycloakAuthorizer
principal.builder.class=io.strimzi.kafka.oauth.server.OAuthKafkaPrincipalBuilder
```

5. 为 Kafka 代理添加配置以访问授权服务器和授权服务。
   在此我们显示作为额外属性添加到 **server.properties** 的示例配置，但您也可以使用大写或大写的命名规则将它们定义为环境变量。

```
strimzi.authorization.token.endpoint.uri="https://<auth_server_address>/auth/realms/REALM-NAME/protocol/openid-connect/token" (1)
strimzi.authorization.client.id="kafka" (2)
```

   (1) 到 Red Hat Single Sign-On 的 OAuth 2.0 令牌端点 URL。对于生产环境，请始终使用 **https://** urls。

   (2) 在启用了授权服务的 Red Hat Single Sign-On 中 OAuth 2.0 客户端定义的客户端 ID。通常，**kafka** 被用作 ID。

6. （可选）为特定 Kafka 集群添加配置。
   例如：

```
strimzi.authorization.kafka.cluster.name="kafka-cluster" (1)
```

   (1) 特定 Kafka 集群的名称。名称用于目标权限，因此可以在同一 Red Hat Single Sign-On 域中管理多个集群。默认值为 **kafka-cluster**。

7. （可选）与简单授权决定：

```
strimzi.authorization.delegate.to.kafka.acl="true" (1)
```

   (1) 如果 Red Hat Single Sign-On Authorization Services 策略无法访问，将授权委派给 Kafka **AclAuthorizer**。默认值为 **false**。

8. （可选）将 TLS 连接的配置添加到授权服务器。
   例如：

```
strimzi.authorization.ssl.truststore.location=<path_to_truststore> (1)
strimzi.authorization.ssl.truststore.password=<my_truststore_password> (2)
strimzi.authorization.ssl.truststore.type=JKS (3)
strimzi.authorization.ssl.secure.random.implementation=SHA1PRNG (4)
strimzi.authorization.ssl.endpoint.identification.algorithm=HTTPS (5)
```

   (1) 包含证书的信任存储的路径。

   (2) truststore 的密码。

   (3) truststore 类型。如果没有设置，则使用默认的 Java 密钥存储类型。

   (4) 随机数生成器实施。如果没有设置，则使用 Java platform SDK 默认。

   (5) 主机名验证。如果设置为空字符串，则会关闭主机名验证。如果没有设置，则默认值为 **HTTPS**，它会强制对服务器证书进行主机名验证。

9. （可选）配置从授权服务器刷新授权。授予刷新作业的工作原理，方法是枚举活跃的令牌并请求每个令牌的最新授权。
例如：

```
strimzi.authorization.grants.refresh.period.seconds="120" (1)
strimzi.authorization.grants.refresh.pool.size="10" (2)
strimzi.authorization.grants.max.idle.time.seconds="300" (3)
strimzi.authorization.grants.gc.period.seconds="300" (4)
strimzi.authorization.reuse.grants="false" (5)
```

1. 指定授权服务器的授予的频率（默认为每分钟一次）。要关闭刷新以进行调试，请将 设置为 "0"。
2. 指定授予刷新作业使用的线程池大小（并行级别）。默认值为 "5"。
3. 缓存中闲置授权的时间（以秒为单位）。默认值为 300。
4. 连续运行作业之间的时间（以秒为单位）。默认值为 300。
5. 控制是否为新会话获取最新的授权。禁用后，从 Red Hat Single Sign-On 检索并缓存该用户的权限。默认值为 true。

10. （可选）在与授权服务器通信时配置网络超时。
例如：

```
strimzi.authorization.connect.timeout.seconds="60" (1)
strimzi.authorization.read.timeout.seconds="60" (2)
strimzi.authorization.http.retries="2" (3)
```

1. 连接到 Red Hat Single Sign-On 令牌端点时的连接超时（以秒为单位）。默认值为 60。
2. 连接到 Red Hat Single Sign-On 令牌端点时读取超时（以秒为单位）。默认值为 60。
3. 重新尝试（不暂停）授权服务器的 HTTP 请求失败的次数上限。默认值为 0，表示不会执行重试。要有效地使用这个选项，请考虑减少 **strimzi.authorization.connect.timeout.seconds** 和 **strimzi.authorization.read.timeout.seconds** 选项的超时时间。但请注意，重试可能会阻止当前 worker 线程可用于其他请求，如果太多请求停滞，则可能会导致 Kafka 代理无响应。

11. （可选）为令牌验证和授权启用 OAuth 2.0 指标：

```
oauth.enable.metrics="true" (1)
```

1. 控制是否启用或禁用 OAuth 指标。默认值为 false。

12. （可选）从请求中删除 Accept 标头：

```
oauth.include.accept.header="false" (1)
```

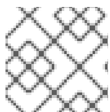
1. 如果包含标头在与授权服务器通信时导致问题，则设置为 false。默认值为 true。

13. 通过以客户端或具有特定角色的用户访问 Kafka 代理来验证配置的权限，确保它们具有必要的访问权限，或者没有应该具有访问权限。

6.9. 使用基于 OPA 策略的授权

开源策略代理(OPA)是一个开源策略引擎。您可以将 OPA 与 Apache Kafka 的 Streams 集成，以作为基于策略的授权机制，允许在 Kafka 代理上进行客户端操作。

从客户端发出请求时，OPA 将根据为 Kafka 访问定义的策略评估请求，然后允许或拒绝请求。



注意

红帽不支持 OPA 服务器。

其他资源

- 打开 Policy Agent 网站

6.9.1. 定义 OPA 策略

在将 OPA 与 Apache Kafka 的 Streams 集成之前，请考虑如何定义策略以提供精细的访问控制。

您可以为 Kafka 集群、消费者组和主题定义访问控制。例如，您可以定义一个授权策略，允许从制作者客户端写入到特定代理主题的访问。

为此，策略可能会指定：

- 与制作者客户端关联的用户主体和主机地址
- 客户端允许的操作
- 策略适用的 资源类型 (topic)和资源名称

允许或拒绝决策将写入到策略中，并根据提供的请求和客户端识别数据提供响应。

在我们的示例中，生成者客户端必须满足策略才能写入该主题。

6.9.2. 连接到 OPA

要启用 Kafka 访问 OPA 策略引擎以查询访问控制策略，您可以在您的 Kafka **server.properties** 文件中配置自定义 OPA authorizer 插件 (**kafka-authorizer-opa-*VERSION*.jar**）。

客户端发出请求时，OPA 策略引擎将通过指定的 URL 地址和 REST 端点来查询 OPA 策略引擎，该端点必须是定义的策略的名称。

该插件提供了客户端请求的详细信息 - 用户主体、操作和资源 - 以 JSON 格式针对策略进行检查。详细信息将包括客户端的唯一身份；例如，使用 TLS 身份验证时将区分名称与客户端证书进行区分。

OPA 使用数据向插件提供响应 - *true* 或 *false* - 允许或拒绝请求。

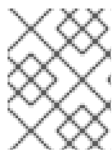
6.9.3. 配置 OPA 授权支持

这个步骤描述了如何将 Kafka 代理配置为使用 OPA 授权。

开始前

考虑某些用户所需的访问权限或希望限制某些用户。您可以使用 *用户和* Kafka *资源* 的组合来定义 OPA 策略。

可以设置 OPA 从 LDAP 数据源加载用户信息。



注意

无论在 Kafka 代理上实现的授权是什么，超级用户始终对 Kafka 代理具有不受限制的访问。

先决条件

- 每个主机上安装了 Apache Kafka 的流，且配置文件可用。
- OPA 服务器必须可用于连接。
- Kafka 的 OPA 授权器插件。

流程

1. 编写授权客户端请求对 Kafka 代理执行操作所需的 OPA 策略。
   请参阅 定义 OPA 策略。

   现在，将 Kafka 代理配置为使用 OPA。

2. 为 Kafka 安装 OPA 授权器插件。
   请参阅 连接到 OPA。

   确保插件文件包含在 Kafka 类路径中。

3. 在 Kafka **server.properties** 配置文件中添加以下内容以启用 OPA 插件：

```
authorizer.class.name: com.bisnode.kafka.authorization.OpaAuthorizer
```

4. 在 Kafka 代理的 **server.properties** 中添加其他配置来访问 OPA 策略引擎和策略。
   例如：

```
opa.authorizer.url=https://OPA-ADDRESS/allow 1
opa.authorizer.allow.on.error=false 2
opa.authorizer.cache.initial.capacity=50000 3
opa.authorizer.cache.maximum.size=50000 4
opa.authorizer.cache.expire.after.seconds=600000 5
super.users=User:alice;User:bob 6
```

1. （必需）授权器插件将查询的策略的 OAuth 2.0 令牌端点 URL。在本例中，策略称为 allow。
2. 标志指定在授权器插件无法与 OPA 策略引擎连接时，默认是否允许或拒绝客户端访问。
3. 本地缓存的初始容量（以字节为单位）。使用缓存，以便插件不必查询每个请求的 OPA 策略引擎。
4. 本地缓存的最大容量（以字节为单位）。

5. 本地缓存的时间（以毫秒为单位），方法是从 OPA 策略引擎重新加载。

6. 被视为超级用户的用户主体列表，以便在不查询 Open Policy Agent 策略的情况下始终允许它们。

有关身份验证和授权选项的信息，请参阅 Open Policy Agent 网站。

5. 通过使用具有正确授权的客户端访问 Kafka 代理来验证配置的权限。

第 7 章 创建和管理主题

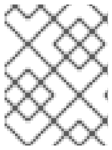
Kafka 中的消息始终从主题发送到或接收。本章论述了如何创建和管理 Kafka 主题。

7.1. 分区和副本

主题始终被分成一个或多个分区。分区充当分片。这意味着，由制作者发送的每个消息始终仅写入单个分区。

每个分区都可以有一个或多个副本，它们存储在集群中的不同代理中。在创建主题时，您可以使用 *复制因素* 配置副本数。*复制因子* 定义在集群中保留的副本数。给定分区的一个副本将被选为领导。领导副本由生产者用于发送新消息并供消费者使用消息。其他副本将遵循副本。后续者复制领导。

如果领导失败，则其中一个同步后续者将自动成为新的领导。每个服务器充当其部分分区的领导者，对其他分区的遵循，因此负载在群集内取得良好平衡。



注意

复制因素决定了副本数量，包括领导机和后续程序。例如，如果您将复制因子设置为 3，则有一个领导和两个后续副本。

7.2. 消息保留

消息保留策略定义信息存储在 Kafka 代理中的时长。它可以根据时间、分区大小或两者定义。

例如，您可以定义消息应该保留：

- 7 天
- 直到分区有 1GB 的消息。达到限制后，将删除最旧的消息。
- 7 天或直至达到 1GB 限制。首先将使用任何限制。

警告

Kafka 代理将信息存储在日志片段中。只有在创建新日志片段时，才会删除其保留策略的消息。当以前的日志片段超过配置的日志片段大小时，会创建新的日志片段。此外，用户可以请求定期创建新的片段。

Kafka 代理支持紧凑策略。

对于带有紧凑策略的主题，代理总是只保留每个键的最后一条消息。具有相同密钥的旧消息将从分区中删除。因为紧凑是定期执行的操作，所以当具有相同键的新消息发送到分区时不会立即发生。相反，可能需要稍等片刻，直到旧的消息被删除为止。

有关消息保留配置选项的更多信息，请参阅 第 7.5 节 “主题配置”。

7.3. TOPIC 自动创建

默认情况下，如果制作者或消费者尝试从不存在的主题发送或接收信息，则 Kafka 会自动创建一个主题。此行为由 auto.create.topics.enable 配置属性管理，该配置属性默认设置为 true。

在生产环境中，建议禁用自动主题创建。要做到这一点，在 Kafka 配置属性文件中将 auto.create.topics.enable 设置为 false ：

禁用自动主题创建

```
auto.create.topics.enable=false
```

7.4. 主题删除

Kafka 提供了用于防止删除主题的选项，由 delete.topic.enable 属性控制。默认情况下，此属性设置为 true，允许删除主题。

但是，在 Kafka 配置属性文件中将其设置为 false 将禁用删除主题。在这种情况下，尝试删除主题将返回成功状态，但主题本身不会被删除。

禁用主题删除

```
delete.topic.enable=false
```

7.5. 主题配置

自动创建的主题将使用默认主题配置，可在代理属性文件中指定。但是，在手动创建主题时，可以在创建时指定其配置。也可以在创建主题后更改主题配置。手动创建主题的主要主题配置选项有：

cleanup.policy
: 将保留策略配置为 delete 或 compact。删除策略 将删除旧记录。紧凑 策略将启用日志压缩。默认值为 delete。有关日志压缩的更多信息，请参阅 Kafka 网站。

compression.type
: 指定用于存储消息的压缩。有效值为 gzip、snappy、lz4、 未压缩 （无压缩）和 生成者 （包含生产者使用的压缩代码）。默认值为 producer。

max.message.bytes
: Kafka 代理允许的最大消息大小，以字节为单位。默认值为 1000012。

min.insync.replicas
: 要被视为成功，必须同步的最小副本数才被视为成功。默认值为：1。

retention.ms
: 保留日志片段的最大毫秒数。早于这个值的日志片段将被删除。默认值为 604800000 (7 天)。

retention.bytes
: 分区将保留的最大字节数。当分区大小超过这个限制后，将删除最旧的日志片段。-1 表示没有限制。默认值为 -1。

segment.bytes
: 单个提交日志段文件的最大文件大小（以字节为单位）。当片段达到其大小时，将启动新的网段。默认值为 1073741824 字节 (1 gibibyte)。

自动创建主题的默认值可使用类似选项在 Kafka 代理配置中指定：

log.cleanup.policy

请参阅上面的 cleanup.policy。

compression.type

请参阅上述 compression.type。

message.max.bytes

请参阅上面的 max.message.bytes。

min.insync.replicas

请参阅上面的 min.insync.replicas。

log.retention.ms

请参阅上面的 retention.ms。

log.retention.bytes

请参阅上面的 retention.bytes。

log.segment.bytes

请参阅上面的 segment.bytes。

default.replication.factor

自动创建的主题的默认复制因素。默认值为 1。

num.partitions

自动创建主题的默认分区数。默认值为 1。

7.6. 内部主题

内部主题由 Kafka 代理和客户端在内部创建和使用。Kafka 有几个内部主题，其中两个用于存储消费者偏移(__consumer_offsets)和事务状态(__transaction_state)。

__consumer_offsets 和 __transaction_state 主题可以使用以前缀 offsets.topic. 和 transaction.state.log. 开头的专用 Kafka 代理配置选项来配置。

最重要的配置选项有：

offsets.topic.replication.factor

__consumer_offsets 主题的副本数。默认值为 3。

offsets.topic.num.partitions

__consumer_offsets 主题的分区数。默认值为 50。

transaction.state.log.replication.factor

__transaction_state 主题的副本数。默认值为 3。

transaction.state.log.num.partitions

__transaction_state 主题的分区数。默认值为 50。

transaction.state.log.min.isr

必须确认对 __transaction_state 主题的写入的最小副本数才被视为成功。如果无法满足此最小值，则生成者将失败，但会异常。默认值为 2。

7.7. 创建主题

使用 kafka-topics.sh 工具管理主题。kafka-topics.sh 是 Apache Kafka 发行版本的 Streams 的一部分，可在 bin 目录中找到。

先决条件

- 每个主机上安装了 Apache Kafka 的流，且配置文件可用。

创建主题

1. 使用 **kafka-topics.sh** 实用程序创建主题并指定以下内容：
   - 在 **--bootstrap-server** 选项中的 Kafka 代理的主机和端口。
   - 要在 **--create** 选项中创建的新主题。
   - **--topic** 选项中的主题名称。
   - **--partitions** 选项中的分区数量。
   - **--replication-factor** 选项中的主题复制因素。
   您还可以使用 **--config** 选项覆盖一些默认主题配置选项。这个选项可多次使用来覆盖不同的选项。

   ```
   /opt/kafka/bin/kafka-topics.sh --bootstrap-server <broker_address> --create --topic
   <TopicName> --partitions <NumberOfPartitions> --replication-factor
   <ReplicationFactor> --config <Option1>=<Value1> --config <Option2>=<Value2>
   ```

   创建名为 mytopic的主题的命令示例

   ```
   /opt/kafka/bin/kafka-topics.sh --bootstrap-server localhost:9092 --create --topic
   mytopic --partitions 50 --replication-factor 3 --config cleanup.policy=compact --
   config min.insync.replicas=2
   ```

2. 验证主题是否使用 **kafka-topics.sh** 是否存在。

   ```
   /opt/kafka/bin/kafka-topics.sh --bootstrap-server <broker_address> --describe --topic
   <TopicName>
   ```

   描述名为 mytopic的主题的命令示例

   ```
   /opt/kafka/bin/kafka-topics.sh --bootstrap-server localhost:9092 --describe --topic
   mytopic
   ```

7.8. 列出和描述主题

kafka-topics.sh 工具可用于列出和描述主题。**kafka-topics.sh** 是 Apache Kafka 发行版本的 Streams 的一部分，可在 **bin** 目录中找到。

先决条件

- 每个主机上安装了 Apache Kafka 的流，且配置文件可用。

描述主题

1. 使用 **kafka-topics.sh** 工具描述主题并指定以下内容：
   - 在 **--bootstrap-server** 选项中的 Kafka 代理的主机和端口。

- 使用 --describe 选项指定您要描述主题。
- 主题名称必须在 --topic 选项中指定。
- 当省略 --topic 选项时，它描述了所有可用的主题。

```
/opt/kafka/bin/kafka-topics.sh --bootstrap-server <broker_host>:<port> --describe --topic <topic_name>
```

描述名为 mytopic的主题的命令示例

```
/opt/kafka/bin/kafka-topics.sh --bootstrap-server localhost:9092 --describe --topic mytopic
```

命令列出属于此主题的所有分区和副本。它还列出所有主题配置选项。

7.9. 修改主题配置

kafka-configs.sh 工具可用于修改主题配置。kafka-configs.sh 是 Apache Kafka 发行版本的 Streams 的一部分，可在 bin 目录中找到。

先决条件

- 每个主机上安装了 Apache Kafka 的流，且配置文件可用。

修改主题配置

1. 使用 kafka-configs.sh 工具获取当前的配置。
   - 在 --bootstrap-server 选项中指定 Kafka 代理的主机和端口。
   - 将 --entity-type 设置为 topic，将 --entity-name 设置为主题的名称。
   - 使用 --describe 选项获取当前的配置。

   ```
   /opt/kafka/bin/kafka-configs.sh --bootstrap-server <broker_host>:<port> --entity-type topics --entity-name <topic_name> --describe
   ```

   获取名为 mytopic的主题配置的命令示例

   ```
   /opt/kafka/bin/kafka-configs.sh --bootstrap-server localhost:9092 --entity-type topics --entity-name mytopic --describe
   ```

2. 使用 kafka-configs.sh 工具更改配置。
   - 在 --bootstrap-server 选项中指定 Kafka 代理的主机和端口。
   - 将 --entity-type 设置为 topic，将 --entity-name 设置为主题的名称。
   - 使用 --alter 选项修改当前配置。
   - 在选项 --add-config 中指定您要添加或更改的选项。

```
/opt/kafka/bin/kafka-configs.sh --bootstrap-server <broker_host>:<port> --entity-type topics --entity-name <topic_name> --alter --add-config <option>=<value>
```

更改名为 mytopic的主题配置的命令示例

```
/opt/kafka/bin/kafka-configs.sh --bootstrap-server localhost:9092 --entity-type topics --entity-name mytopic --alter --add-config min.insync.replicas=1
```

3. 使用 kafka-configs.sh 工具删除现有配置选项。

   - 在 --bootstrap-server 选项中指定 Kafka 代理的主机和端口。
   - 将 --entity-type 设置为 topic，将 --entity-name 设置为主题的名称。
   - 使用 --delete-config 选项删除现有配置选项。
   - 在选项 --remove-config 中指定您要删除的选项。

```
/opt/kafka/bin/kafka-configs.sh --bootstrap-server <broker_host>:<port> --entity-type topics --entity-name <topic_name> --alter --delete-config <option>
```

更改名为 mytopic的主题配置的命令示例

```
/opt/kafka/bin/kafka-configs.sh --bootstrap-server localhost:9092 --entity-type topics --entity-name mytopic --alter --delete-config min.insync.replicas
```

7.10. 删除主题

kafka-topics.sh 工具可用于管理主题。kafka-topics.sh 是 Apache Kafka 发行版本的 Streams 的一部分，可在 bin 目录中找到。

先决条件

- 每个主机上安装了 Apache Kafka 的流，且配置文件可用。

删除主题

1. 使用 kafka-topics.sh 工具删除主题。

   - 在 --bootstrap-server 选项中的 Kafka 代理的主机和端口。
   - 使用 --delete 选项指定应删除现有主题。
   - 主题名称必须在 --topic 选项中指定。

```
/opt/kafka/bin/kafka-topics.sh --bootstrap-server <broker_host>:<port> --delete --topic <topic_name>
```

创建名为 mytopic的主题的命令示例

```
/opt/kafka/bin/kafka-topics.sh --bootstrap-server localhost:9092 --delete --topic mytopic
```

2. **使用 kafka-topics.sh 验证主题是否已删除。**

```
/opt/kafka/bin/kafka-topics.sh --bootstrap-server <broker_host>:<port> --list
```

列出所有主题的命令示例

```
/opt/kafka/bin/kafka-topics.sh --bootstrap-server localhost:9092 --list
```

第 8 章 在 KAFKA CONNECT 中使用 APACHE KAFKA 的 STREAMS

使用 Kafka Connect 在 Kafka 和外部系统之间流数据。Kafka Connect 提供了一个框架来移动大量数据，同时保持可扩展性和可靠性。Kafka Connect 通常用于将 Kafka 与 Kafka 集群外部的数据库、存储和消息传递系统集成。

Kafka Connect 以独立或分布式模式运行。

独立模式

在独立模式中，Kafka Connect 在单一节点中运行。独立模式用于开发和测试。

分布式模式

在分布式模式中，Kafka Connect 在一个或多个 worker 节点上运行，工作负载分布在它们中。分布式模式主要用于生产环境。

Kafka Connect 使用连接器插件，为不同类型的外部系统实现连接。有两种连接器插件：接收器和源。sink 连接器将数据从 Kafka 流传输到外部系统。源连接器将来自外部系统的数据流传输到 Kafka。

您还可以使用 Kafka Connect REST API 创建、管理和监控连接器实例。

连接器配置指定要从或写入的源或接收器连接器和 Kafka 主题等详情。如何管理配置取决于您是否在独立或分布式模式下运行 Kafka Connect。

- 在独立模式中，您可以通过 Kafka Connect REST API 将连接器配置作为 JSON 提供，也可以使用属性文件来定义配置。
- 在分布式模式中，您只能通过 Kafka Connect REST API 以 JSON 提供连接器配置。

处理大量信息

您可以调整配置以处理大量信息。如需更多信息，请参阅 处理大量信息。

8.1. 在独立模式中使用 KAFKA 连接

在 Kafka Connect 独立模式中，连接器与 Kafka Connect worker 进程在同一个节点上运行，该进程作为单个 JVM 中的单个进程运行。这意味着 worker 进程和连接器共享相同的资源，如 CPU、内存和磁盘。

8.1.1. 在独立模式中配置 Kafka 连接

要在独立模式下配置 Kafka Connect，请编辑 **config/connect-standalone.properties** 配置文件。以下选项是最重要的。

bootstrap.servers

用作 Kafka 的 bootstrap 连接的 Kafka 代理地址列表。例如： **kafka0.my-domain.com:9092,kafka1.my-domain.com:9092,kafka2.my-domain.com:9092**。

key.converter

用于将消息密钥转换为 Kafka 格式的类。例如，**org.apache.kafka.connect.json.JsonConverter**.

value.converter

用于将消息有效负载转换为 Kafka 格式的类。例如，**org.apache.kafka.connect.json.JsonConverter**.

offset.storage.file.filename

指定存储偏移数据的文件。

连接器插件使用 bootstrap 地址打开到 Kafka 代理的客户端连接。要配置这些连接，请使用以 **producer.** 或 **consumer.** 为前缀的标准 Kafka producer 和使用者配置选项。

8.1.2. 在独立模式中运行 Kafka 连接

在独立模式中配置并运行 Kafka Connect。

先决条件

- 每个主机上安装了 Apache Kafka 的流，且配置文件可用。
- 您已在属性文件中指定连接器配置。
  您还可以使用 Kafka Connect REST API 管理连接器。

流程

1. 编辑 **/opt/kafka/config/connect-standalone.properties** Kafka 连接配置文件，并将 **bootstrap.server** 设置为指向您的 Kafka 代理。例如：

   ```
   bootstrap.servers=kafka0.my-domain.com:9092,kafka1.my-
   domain.com:9092,kafka2.my-domain.com:9092
   ```

2. 使用配置文件启动 Kafka 连接并指定一个或多个连接器配置。

   ```
   su - kafka
   /opt/kafka/bin/connect-standalone.sh /opt/kafka/config/connect-standalone.properties
   connector1.properties
   [connector2.properties ...]
   ```

3. 验证 Kafka Connect 是否正在运行。

   ```
   jcmd | grep ConnectStandalone
   ```

8.2. 在分布式模式中使用 KAFKA CONNECT

在分布式模式中，Kafka Connect 作为 worker 进程集群运行，每个 worker 在单独的节点上运行。连接器可以在集群中的任何 worker 上运行，从而提高可扩展性和容错能力。连接器由 worker 管理，它们相互协调，以分发工作并确保每个连接器在任何给定时间都在单一节点上运行。

8.2.1. 在分布式模式下配置 Kafka 连接

要在分布式模式下配置 Kafka Connect，请编辑 **config/connect-distributed.properties** 配置文件。以下选项是最重要的。

bootstrap.servers

用作 Kafka 的 bootstrap 连接的 Kafka 代理地址列表。例如：**kafka0.my-domain.com:9092,kafka1.my-domain.com:9092,kafka2.my-domain.com:9092**。

key.converter

用于将消息密钥转换为 Kafka 格式的类。例如，**org.apache.kafka.connect.json.JsonConverter**.

value.converter

用于将消息有效负载转换为 Kafka 格式的类。例如，**org.apache.kafka.connect.json.JsonConverter**.

group.id

分布式 Kafka Connect 集群的名称。这必须是唯一的，且不得与另一个消费者组 ID 冲突。默认值为 **connect-cluster**。

config.storage.topic

用于存储连接器配置的 Kafka 主题。默认值为 **connect-configs**。

offset.storage.topic

用于存储偏移的 Kafka 主题。默认值为 **connect-offset**。

status.storage.topic

用于 worker 节点状态的 Kafka 主题。默认值为 **connect-status**。

Apache Kafka 的 Streams 在分布式模式下包括 Kafka Connect 的示例配置文件 - 请参阅 Streams for Apache Kafka 安装目录中的 **config/connect-distributed.properties**。

连接器插件使用 bootstrap 地址打开到 Kafka 代理的客户端连接。要配置这些连接，请使用以 **producer.** 或 **consumer.** 为前缀的标准 Kafka producer 和使用者配置选项。

8.2.2. 在分布式模式下运行 Kafka Connect

在分布式模式下运行 Kafka Connect。

先决条件

- 每个主机上安装了 Apache Kafka 的流，且配置文件可用。

运行集群

1. 编辑所有 Kafka Connect worker 节点上的 **/opt/kafka/config/connect-distributed.properties** Kafka Connect 配置文件。
   - 设置 **bootstrap.server** 选项以指向 Kafka 代理。
   - 设置 **group.id** 选项。
   - 设置 **config.storage.topic** 选项。
   - 设置 **offset.storage.topic** 选项。
   - 设置 **status.storage.topic** 选项。
     例如：

```
bootstrap.servers=kafka0.my-domain.com:9092,kafka1.my-domain.com:9092,kafka2.my-domain.com:9092
group.id=my-group-id
config.storage.topic=my-group-id-configs
offset.storage.topic=my-group-id-offsets
status.storage.topic=my-group-id-status
```

2. 使用所有 Kafka Connect 节点上的 **/opt/kafka/config/connect-distributed.properties** 配置文件启动 Kafka Connect worker。

```
su - kafka
/opt/kafka/bin/connect-distributed.sh /opt/kafka/config/connect-distributed.properties
```

3. 验证 Kafka Connect 是否正在运行。

```
jcmd | grep ConnectDistributed
```

4. 使用 Kafka Connect REST API 管理连接器。

8.3. 管理连接器

Kafka Connect REST API 提供用于直接创建、更新和删除连接器的端点。您还可以使用 API 检查连接器的状态或更改日志记录级别。当您通过 API 创建连接器时，您可以提供连接器的配置详情作为 API 调用的一部分。

您还可以添加和管理连接器作为插件。插件被打包为 JAR 文件，其中包含通过 Kafka Connect API 实现连接器的类。您只需要在 classpath 中指定插件，或将其添加到 Kafka Connect 的插件路径中，以便在启动时运行连接器插件。

除了使用 Kafka Connect REST API 或插件来管理连接器外，您还可以在独立模式中运行 Kafka Connect 时使用属性文件添加连接器配置。要做到这一点，您可以在启动 Kafka Connect worker 进程时指定属性文件的位置。属性文件应包含连接器的配置详情，包括连接器类、源和目标主题，以及任何所需的身份验证或序列化设置。

8.3.1. 限制对 Kafka Connect API 的访问

Kafka Connect REST API 可以被经过身份验证的用户访问，并知道端点 URL，其中包括主机名/IP 地址和端口号。仅将对 Kafka Connect API 的访问限制为可信用户，以防止未经授权的操作和潜在的安全问题。

为提高安全性，我们建议为 Kafka Connect API 配置以下属性：

- (Kafka 3.4 或更高版本) **org.apache.kafka.disallowed.login.modules**，来专门排除不安全的登录模块
- **connector.client.config.override.policy** 设置为 **NONE**，以防止连接器配置覆盖 Kafka Connect 配置及其使用的使用者和制作者

8.3.2. 配置连接器

使用 Kafka Connect REST API 或属性文件来创建、管理和监控连接器实例。在独立或分布式模式中使用 Kafka Connect 时，您可以使用 REST API。在独立模式中使用 Kafka Connect 时，您可以使用属性文件。

8.3.2.1. 使用 Kafka Connect REST API 管理连接器

使用 Kafka Connect REST API 时，您可以通过向 Kafka Connect REST API 发送 **PUT** 或 **POST** HTTP 请求来动态创建连接器，在请求正文中指定连接器配置详情。

提示

使用 **PUT** 命令时，它具有相同的命令来启动和更新连接器。

REST 接口默认侦听端口 8083，并支持以下端点：

GET /connectors
: 返回现有连接器列表。

POST /connectors
: 创建连接器。请求正文必须是带有连接器配置的 JSON 对象。

GET /connectors/<connector_name>
: 获取有关特定连接器的信息。

GET /connectors/<connector_name>/config
: 获取特定连接器的配置。

PUT /connectors/<connector_name>/config
: 更新特定连接器的配置。

GET /connectors/<connector_name>/status
: 获取特定连接器的状态。

GET /connectors/<connector_name>/tasks
: 获取特定连接器的任务列表

GET /connectors/<connector_name>/tasks/*<task_id>*/status
: 获取特定连接器的任务状态

PUT /connectors/<connector_name>/pause
: 暂停连接器及其所有任务。连接器将停止处理任何信息。

PUT /connectors/<connector_name>/stop
: 停止连接器及其所有任务。连接器将停止处理任何信息。从运行停止连接器可能更适合长时间运行，而不是只暂停。

PUT /connectors/<connector_name>/resume
: 恢复暂停的连接器。

POST /connectors/<connector_name>/restart
: 如果连接器失败，请重启它。

POST /connectors/<connector_name>/tasks/*<task_id>*/restart
: 重启特定的任务。

DELETE /connectors/<connector_name>
: 删除连接器。

GET /connectors/<connector_name>/topics
: 获取特定连接器的主题。

PUT /connectors/<connector_name>/topics/reset
: 为特定连接器清空一组活跃的主题。

GET /connectors/<connector_name>/offsets
: 获取连接器的当前偏移量。

DELETE /connectors/<connector_name>/offsets
: 为连接器重置偏移，该连接器必须处于已停止状态。

PATCH /connectors/<connector_name>/offsets
: 为连接器调整 偏移 属性（使用请求的偏移属性），该连接器必须处于已停止状态。

GET /connector-plugins

获取所有支持的连接器插件的列表。

GET /connector-plugins/<connector_plugin_type>/config

获取连接器插件的配置。

PUT /connector-plugins/<connector_type>/config/validate

验证连接器配置。

8.3.2.2. 指定连接器配置属性

要配置 Kafka Connect 连接器，您需要指定源或接收器连接器的配置详情。有两种方法可以做到这一点：通过 Kafka Connect REST API，使用 JSON 提供配置，或使用属性文件来定义配置属性。每种连接器类型可用的特定配置选项可能有所不同，但这两种方法都提供指定必要设置的灵活方法。

以下选项适用于所有连接器：

name

连接器的名称，它在当前 Kafka Connect 实例中必须是唯一的。

connector.class

连接器插件的类。例如，**org.apache.kafka.connect.file.FileStreamSinkConnector**。

tasks.max

指定连接器可以使用的最大任务数量。任务可让连接器并行执行工作。连接器可能会创建比指定更少的任务。

key.converter

用于将消息密钥转换为 Kafka 格式的类。这会覆盖 Kafka Connect 配置设置的默认值。例如，**org.apache.kafka.connect.json.JsonConverter**.

value.converter

用于将消息有效负载转换为 Kafka 格式的类。这会覆盖 Kafka Connect 配置设置的默认值。例如，**org.apache.kafka.connect.json.JsonConverter**.

必须至少为接收器连接器设置以下选项之一：

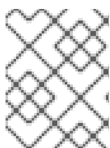
topics

以逗号分隔的主题列表，用作输入。

topics.regex

用作输入的 Java 正则表达式。

有关所有其他选项，请参阅 Apache Kafka 文档中的 连接器属性。



注意

Apache Kafka 的 Streams 在 Apache Kafka 安装目录的 Streams 中包含示例连接器配置文件 **config/connect-file-sink.properties** 和 **config/connect-file-source.properties**。

其他资源

- Kafka Connect REST API OpenAPI 文档

8.3.3. 使用 Kafka Connect API 创建连接器

使用 Kafka Connect REST API 创建用于 Kafka Connect 的连接器。

先决条件

- Kafka Connect 安装。

流程

1. 使用连接器配置准备 JSON 有效负载。例如：

```
{
  "name": "my-connector",
  "config": {
  "connector.class": "org.apache.kafka.connect.file.FileStreamSinkConnector",
    "tasks.max": "1",
    "topics": "my-topic-1,my-topic-2",
    "file": "/tmp/output-file.txt"
  }
}
```

2. 发送 POST 请求到 *<KafkaConnectAddress>*:8083/connectors，以创建连接器。以下示例使用 curl ：

```
curl -X POST -H "Content-Type: application/json" --data @sink-connector.json
http://connect0.my-domain.com:8083/connectors
```

3. 通过向 < *KafkaConnectAddress* > :8083/connectors 发送 GET 请求来验证连接器是否已部署。以下示例使用 curl ：

```
curl http://connect0.my-domain.com:8083/connectors
```

8.3.4. 使用 Kafka Connect API 删除连接器

使用 Kafka Connect REST API 从 Kafka Connect 中删除连接器。

先决条件

- Kafka Connect 安装。

删除连接器

1. 通过向 *<KafkaConnectAddress>*:8083/connectors/*<ConnectorName>* 发送 GET 请求来验证连接器是否存在。以下示例使用 curl ：

```
curl http://connect0.my-domain.com:8083/connectors
```

2. 要删除连接器，请发送 DELETE 请求到 <*KafkaConnectAddress& gt;* :8083/connectors。以下示例使用 curl ：

```
curl -X DELETE http://connect0.my-domain.com:8083/connectors/my-connector
```

3. 通过向 < *KafkaConnectAddress* > :8083/connectors 发送 GET 请求来验证连接器是否已删除。以下示例使用 curl ：

```
curl http://connect0.my-domain.com:8083/connectors
```

8.3.5. 添加连接器插件

Kafka 提供示例连接器，用作开发连接器的起点。以下连接器示例包括在 Apache Kafka 的 Streams 中：

FileStreamSink
: 从 Kafka 主题读取数据，并将数据写入一个文件中。

FileStreamSource
: 从文件中读取数据，并将数据发送到 Kafka 主题。

这两个连接器都包含在 **libs/connect-file-<kafka_version>.redhat-<build>.jar** 插件中。

要使用 Kafka Connect 中的连接器插件，您可以将其添加到 classpath 中，或者在 Kafka Connect 属性文件中指定插件路径，并将插件复制到位置。

在 classpath 中指定示例连接器

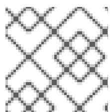
```
CLASSPATH=/opt/kafka/libs/connect-file-<kafka_version>.redhat-<build>.jar
opt/kafka/bin/connect-distributed.sh
```

设置插件路径

```
plugin.path=/opt/kafka/connector-plugins,/opt/connectors
```

plugin.path 配置选项可以包含以逗号分隔的路径列表。

如果需要，您可以添加更多连接器插件。Kafka Connect 在启动时搜索并运行连接器插件。



注意

当在分布式模式下运行 Kafka Connect 时，所有 worker 节点上都必须提供插件。

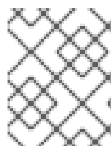
第 9 章 使用带有 MIRRORMAKER 2 的 APACHE KAFKA 的 STREAMS

使用 MirrorMaker 2 在两个或多个活跃 Kafka 集群之间复制数据，并在数据中心之间复制数据。

要配置 MirrorMaker 2，请编辑 **config/connect-mirror-maker.properties** 配置文件。如果需要，您可以为 MirrorMaker 2 启用分布式追踪。

处理大量信息

您可以调整配置以处理大量信息。如需更多信息，请参阅 处理大量信息。



注意

MirrorMaker 2 具有之前版本的 MirrorMaker 不支持的功能。但是，您可以将 MirrorMaker 2 配置为用于旧模式。

9.1. 配置主动/主动或主动/被动模式

您可以在*主动/被动*或*主动/主动*集群配置中使用 MirrorMaker 2。

主动/主动集群配置

主动/主动配置有两个主动集群双向复制数据。应用程序可以使用任一集群。每个集群都可以提供相同的数据。这样，您可以在不同的地理位置提供相同的数据。因为消费者组在两个集群中都活跃，复制主题的使用者偏移不会重新同步到源集群。

主动/被动集群配置

主动/被动配置具有主动集群将数据复制到被动集群。被动集群保持在待机状态。在出现系统失败时，您可以使用被动集群进行数据恢复。

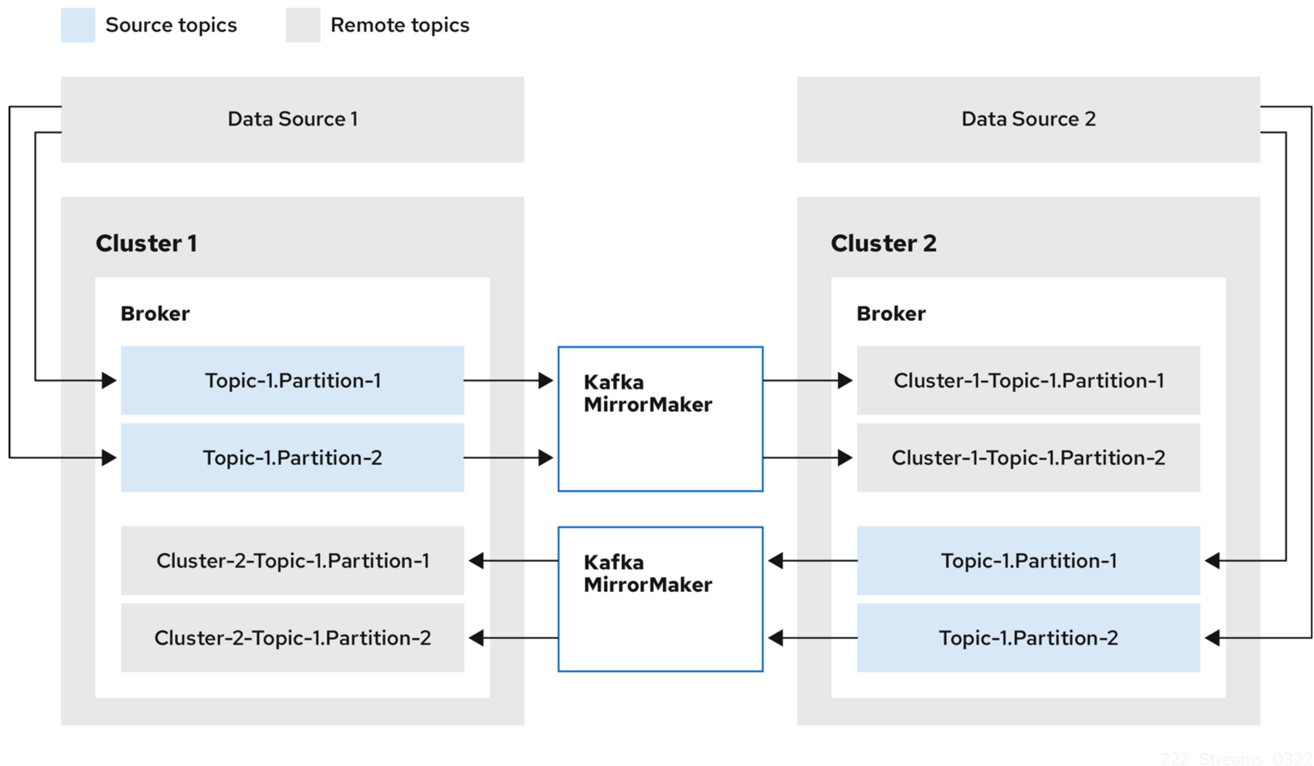
预期的结构是，生成者和消费者仅连接到活跃集群。每个目标目的地都需要一个 MirrorMaker 2 集群。

9.1.1. 双向复制（主动/主动）

MirrorMaker 2 架构支持 *主动/主动集群配置中* 的双向复制。

每个集群使用 *source* 和 *remote* 主题的概念复制其他集群的数据。由于同一主题存储在每个集群中，因此远程主题由 MirrorMaker 2 自动重命名，以代表源集群。原始集群的名称前面是主题名称的前面。

图 9.1. 主题重命名



通过标记原始集群，主题不会复制到该集群。

在配置需要数据聚合的架构时，通过 *远程主题* 复制的概念非常有用。消费者可以订阅同一集群中的源和目标主题，而无需单独的聚合集群。

9.1.2. 单向复制（主动/被动）

MirrorMaker 2 架构支持 *主动/被动集群* 配置中的单向复制。

您可以使用 *主动/被动集群* 配置来备份或将数据迁移到另一个集群。在这种情况下，您可能不希望自动重命名远程主题。

您可以通过将 **IdentityReplicationPolicy** 添加到源连接器配置来覆盖自动重命名。应用此配置后，主题会保留其原始名称。

9.2. 配置 MIRRORMAKER 2 连接器

将 MirrorMaker 2 连接器配置用于编配 Kafka 集群之间的数据同步的内部连接器。

MirrorMaker 2 由以下连接器组成：

MirrorSourceConnector

源连接器将主题从源集群复制到目标集群。它还复制 ACL，且是 **MirrorCheckpointConnector** 才能运行所必需的。

MirrorCheckpointConnector

checkpoint 连接器会定期跟踪偏移。如果启用，它还在源和目标集群之间同步消费者组偏移。

MirrorHeartbeatConnector

heartbeat 连接器会定期检查源和目标集群之间的连接。

下表描述了连接器属性以及您配置为使用它们的连接器。

表 9.1. MirrorMaker 2 连接器配置属性

属性	sourceConnector	checkpointConnector	heartbeatConnector
admin.timeout.ms 管理任务的超时，如检测新主题。默认值为 **60000** (1 分钟)。	✓	✓	✓
replication.policy.class 定义远程主题命名约定的策略。默认为 **org.apache.kafka.connect.mirror.DefaultReplicationPolicy**。	✓	✓	✓
replication.policy.separator 在目标集群中用于主题命名的分隔符。默认情况下，分隔符设置为点(.)。分隔符配置仅适用于 **DefaultReplicationPolicy** 复制策略类，用于定义远程主题名称。**IdentityReplicationPolicy** 类不使用属性，因为主题会保留其原始名称。	✓	✓	✓
consumer.poll.timeout.ms 轮询源集群时超时。默认值为 **1000** (1 秒)。	✓	✓	
offset-syncs.topic.location **offset-syncs** 主题的位置，可以是 **源**（默认）或 **目标集群**。	✓	✓	
topic.filter.class 选择要复制的主题的主题过滤器。默认为 **org.apache.kafka.connect.mirror.DefaultTopicFilter**。	✓	✓	
config.property.filter.class 用于选择要复制的主题配置属性的主题过滤器。默认为 **org.apache.kafka.connect.mirror.DefaultConfigPropertyFilter**。	✓		

属性	sourceConnector	checkpointConnector	heartbeatConnector
config.properties.exclude 不应复制的主题配置属性。支持以逗号分隔的属性名称和正则表达式。	✓		
offset.lag.max 在同步远程分区前，最大允许（不同步）偏移滞后。默认值为 **100**。	✓		
offset-syncs.topic.replication.factor 内部 **offset-syncs** 主题的复制因素。默认值为 **3**。	✓		
refresh.topics.enabled 启用检查新主题和分区。默认为 **true**。	✓		
refresh.topics.interval.seconds 主题刷新的频率。默认为 **600** (10 分钟)。默认情况下，检查源集群中的新主题每 10 分钟进行一次。您可以通过在源连接器配置中添加 **refresh.topics.interval.seconds** 来更改频率。	✓		
replication.factor 新主题的复制因素。默认值为 **2**。	✓		
sync.topic.acls.enabled 启用从源集群同步 ACL。默认为 **true**。如需更多信息，请参阅 第 9.5 节 “ACL 规则同步”。	✓		
sync.topic.acls.interval.seconds ACL 同步的频率。默认为 **600** (10 分钟)。	✓		
sync.topic.configs.enabled 启用从源集群同步主题配置。默认为 **true**。	✓		

属性	sourceConnector	checkpointConnector	heartbeatConnector
sync.topic.configs.interval.seconds 主题配置同步的频率。默认 **600** (10 分钟)。	✓		
checkpoints.topic.replication.factor 内部 **检查点** 主题的复制因素。默认值为 **3**。		✓	
emit.checkpoints.enabled 启用将消费者偏移同步到目标集群。默认为 **true**。		✓	
emit.checkpoints.interval.seconds 消费者偏移同步的频率。默认值为 **60** (1 分钟)。		✓	
group.filter.class 组过滤器，以选择要复制的消费者组。默认为 **org.apache.kafka.connect.mirror.DefaultGroupFilter**。		✓	
refresh.groups.enabled 启用检查新的消费者组。默认为 **true**。		✓	
refresh.groups.interval.seconds 消费者组刷新的频率。默认为 **600** (10 分钟)。		✓	
sync.group.offsets.enabled 启用将消费者组偏移同步到目标集群 **__consumer_offsets** 主题。默认为 **false**。		✓	
sync.group.offsets.interval.seconds 消费者组偏移同步的频率。默认值为 **60** (1 分钟)。		✓	

属性	sourceConnector	checkpointConnector	heartbeatConnector
emit.heartbeats.enabled 在目标集群中启用连接检查。默认为 **true**。			✓
emit.heartbeats.interval.seconds 连接检查的频率。默认为 **1** (1 秒)。			✓
heartbeats.topic.replication.factor 内部 **心跳** 主题的复制因素。默认值为 **3**。			✓

9.2.1. 更改消费者组偏移主题的位置

MirrorMaker 2 使用内部主题跟踪消费者组的偏移。

offset-syncs 主题
: **offset-syncs** 主题映射复制主题元数据的源和目标偏移。

checkpoints 主题
: **checkpoints** 主题映射源和目标集群中每个消费者组中复制的主题分区的最后提交偏移量。

因为它们被 MirrorMaker 2 内部使用，所以您不会直接与这些主题交互。

MirrorCheckpointConnector 为偏移跟踪发出 *检查点*。**checkpoints** 主题的偏移通过配置以预先确定的间隔进行跟踪。这两个主题都允许从故障转移上的正确偏移位置完全恢复复制。

offset-syncs 主题的位置是 源集群。您可以使用 **offset-syncs.topic.location** 连接器配置将其更改为 目标集群。您需要对包含该主题的集群进行读/写访问。使用目标集群作为 **offset-syncs** 主题的位置，您也可以使用 MirrorMaker 2，即使您只有对源集群的读访问权限。

9.2.2. 同步消费者组偏移

__consumer_offsets 主题存储各个消费者组的提交偏移信息。偏移同步会定期将源集群的消费者组的使用者偏移转移到目标集群的使用者偏移量中。

偏移同步在*主动/被动*配置中特别有用。如果主动集群停机，消费者应用程序可以切换到被动(standby)集群，并从最后一个传输的偏移位置获取。

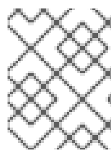
要使用主题偏移同步，请通过将 **sync.group.offsets.enabled** 添加到检查点连接器配置来启用同步，并将属性设置为 **true**。默认情况下禁用同步。

在源连接器中使用 **IdentityReplicationPolicy** 时，还必须在检查点连接器配置中进行配置。这样可确保为正确的主题应用镜像的消费者偏移。

消费者偏移仅针对目标集群中未激活的消费者组同步。如果消费者组位于目标集群中，则无法执行同步，并返回 **UNKNOWN_MEMBER_ID** 错误。

如果启用，则会定期从源集群同步偏移。您可以通过在检查点连接器配置中添加 **sync.group.offsets.interval.seconds** 和 **emit.checkpoints.interval.seconds** 来更改频率。属性指定同步消费者组偏移的频率，以及为偏移跟踪发送检查点的频率。这两个属性的默认值为 60 秒。您还可以使用 **refresh.groups.interval.seconds** 属性更改检查新消费者组的频率，该属性默认为每 10 分钟执行。

由于同步基于时间，因此消费者到被动集群的任何切换都可能会导致一些消息重复。



注意

如果您有使用 Java 编写的应用程序，您可以使用 **RemoteClusterUtils.java** 工具通过应用同步偏移。实用程序从 **checkpoints** 主题获取消费者组的远程偏移。

9.2.3. 决定使用 heartbeat 连接器的时间

heartbeat 连接器发出心跳来检查源和目标 Kafka 集群之间的连接。内部 心跳 主题从源集群复制，这意味着 heartbeat 连接器必须连接到源集群。**heartbeat** 主题位于目标集群上，它允许它执行以下操作：

- 识别它要从中镜像数据的所有源集群
- 验证镜像进程的存活度和延迟

这有助于确保进程不会因为任何原因而卡住或已停止。虽然 heartbeat 连接器是监控 Kafka 集群之间的镜像进程的有价值的工具，但并非总是需要使用它。例如，如果您的部署具有低网络延迟或少量主题，您可能需要使用日志消息或其他监控工具来监控镜像过程。如果您决定不使用 heartbeat 连接器，只需从 MirrorMaker 2 配置中省略它。

9.2.4. 对齐 MirrorMaker 2 连接器的配置

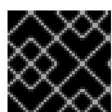
为确保 MirrorMaker 2 连接器正常工作，请确保在连接器之间保持一致某些配置设置。具体来说，请确保以下属性在所有适用的连接器中具有相同的值：

- **replication.policy.class**
- **replication.policy.separator**
- **offset-syncs.topic.location**
- **topic.filter.class**

例如，source、检查点和 heartbeat 连接器的 **replication.policy.class** 的值必须相同。不匹配或缺失的设置会导致数据复制或偏移同步出现问题，因此必须使用同一设置保持所有相关连接器配置。

9.3. 连接器生成者和消费者配置

MirrorMaker 2 连接器使用内部生产者和消费者。如果需要，您可以配置这些制作者和消费者来覆盖默认设置。



重要

生产者和消费者配置选项取决于 MirrorMaker 2 的实施，并可能随时更改。

生产者和消费者配置适用于所有连接器。您在 **config/connect-mirror-maker.properties** 文件中指定配置。

使用属性文件以以下格式覆盖制作者和消费者的任何默认配置：

- *<source_cluster_name>*.consumer.*<property>*
- *<source_cluster_name>*.producer.*<property>*
- *<target_cluster_name>*.consumer.*<property>*
- *<target_cluster_name>*.producer.*<property>*

以下示例演示了如何配置制作者和消费者。尽管为所有连接器设置了属性，但有些配置属性仅与某些连接器相关。

连接器制作者和消费者的配置示例

```
clusters=cluster-1,cluster-2

# ...
cluster-1.consumer.fetch.max.bytes=52428800
cluster-2.producer.batch.size=327680
cluster-2.producer.linger.ms=100
cluster-2.producer.request.timeout.ms=30000
```

9.4. 指定最大任务数

连接器创建负责在 Kafka 中移动数据的任务。每个连接器由一个或多个任务组成，它们分布到运行任务的一组 worker pod 中。在复制大量分区或同步大量消费者组的偏移时，增加任务数量可以帮助解决性能问题。

任务并行运行。为 worker 分配一个或多个任务。单个任务由一个 worker pod 处理，因此您不需要多个 worker pod 超过任务。如果有多个任务，worker 会处理多个任务。

您可以使用 **tasks.max** 属性指定 MirrorMaker 配置中的最大连接器任务数量。在不指定最大任务数量的情况下，默认设置是单个任务。

heartbeat 连接器始终使用单个任务。

为源和检查点连接器启动的任务数量是最大可能任务数和 **tasks.max** 的值之间的较低值。对于源连接器，可能的最大任务数是从源集群复制的每个分区。对于检查点连接器，可能的最大任务数都是从源集群复制的每个消费者组。在设置最多任务数量时，请考虑分区数量和支持进程的硬件资源。

如果基础架构支持处理开销，增加任务数量可以提高吞吐量和延迟。例如，添加更多任务可减少在有大量分区或消费者组时轮询源集群所需的时间。

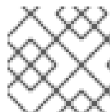
MirrorMaker 连接器的 tasks.max 配置

```
clusters=cluster-1,cluster-2
# ...
tasks.max = 10
```

默认情况下，MirrorMaker 2 每 10 分钟检查新消费者组。您可以调整 **refresh.groups.interval.seconds** 配置以更改频率。在调整降低时请小心。更频繁的检查可能会对性能造成负面影响。

9.5. ACL 规则同步

如果使用 AclAuthorizer，则管理对代理访问权限的 ACL 规则也适用于远程主题。读取源主题的用户可以读取其远程的等效内容。

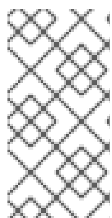


注意

OAuth 2.0 授权不支持以这种方式访问远程主题。

9.6. 在专用模式下运行 MIRRORMAKER 2

使用 MirrorMaker 2 通过配置同步 Kafka 集群间的数据。此流程演示了如何配置和运行专用的单节点 MirrorMaker 2 集群。专用集群使用 Kafka Connect worker 节点在 Kafka 集群间镜像数据。



注意

也可以在分布式模式下运行 MirrorMaker 2。MirrorMaker 2 在专用和分布式模式中作为连接器运行。在运行专用 MirrorMaker 集群时，连接器会在 Kafka Connect 集群中配置。因此，这允许直接访问 Kafka Connect 集群、运行额外连接器并使用 REST API。如需更多信息，请参阅 Apache Kafka 文档。

通过在 旧模式下运行 MirrorMaker 2，仍支持之前的 MirrorMaker版本。

配置必须指定：

- 每个 Kafka 集群
- 每个集群的连接信息，包括 TLS 身份验证
- 复制流和方向
    - 集群到集群
    - topic 的主题
- 复制规则
- 提交偏移跟踪间隔

此流程描述了如何通过在属性文件中创建配置来实现 MirrorMaker 2，然后在使用 MirrorMaker 脚本文件设置连接时传递属性。

您可以指定您要从源集群复制的主题和消费者组。您可以指定源和目标集群的名称，然后指定要复制的主题和消费者组。

在以下示例中，指定了主题和消费者组，用于从集群 1 复制到 2。

复制特定主题和消费者组的配置示例

```
clusters=cluster-1,cluster-2
cluster-1->cluster-2.topics = topic-1, topic-2
cluster-1->cluster-2.groups = group-1, group-2
```

您可以提供名称列表或使用正则表达式。默认情况下，如果您未设置这些属性，则会复制所有主题和消费者组。您还可以使用 RCU 作为正则表达式复制所有主题和消费者组。 但是，尝试只指定您需要指定主题和消费者组，以避免在集群中造成不必要的额外负载。

开始前

./config/connect-mirror-maker.properties 中提供了示例配置属性文件。

先决条件

- 您需要在一个您要复制的每个 Kafka 集群节点的主机上安装 Apache Kafka 的 Streams。

流程

1. 在文本编辑器中打开示例属性文件，或创建一个新文件，并编辑该文件使其包含连接信息以及每个 Kafka 集群的复制流程。
以下示例显示了连接两个集群的配置，即 *cluster-1* 和 *cluster-2* 双向。集群名称可通过集群属性配置。

MirrorMaker 2 配置示例

```
clusters=cluster-1,cluster-2 (1)

cluster-1.bootstrap.servers=<cluster_name>-kafka-bootstrap-<project_name_one>:443
(2)
cluster-1.security.protocol=SSL (3)
cluster-1.ssl.truststore.password=<truststore_name>
cluster-1.ssl.truststore.location=<path_to_truststore>/truststore.cluster-1.jks_
cluster-1.ssl.keystore.password=<keystore_name>
cluster-1.ssl.keystore.location=<path_to_keystore>/user.cluster-1.p12

cluster-2.bootstrap.servers=<cluster_name>-kafka-bootstrap-<project_name_two>:443
(4)
cluster-2.security.protocol=SSL (5)
cluster-2.ssl.truststore.password=<truststore_name>
cluster-2.ssl.truststore.location=<path_to_truststore>/truststore.cluster-2.jks_
cluster-2.ssl.keystore.password=<keystore_name>
cluster-2.ssl.keystore.location=<path_to_keystore>/user.cluster-2.p12

cluster-1->cluster-2.enabled=true (6)
cluster-2->cluster-1.enabled=true (7)
cluster-1->cluster-2.topics=.* (8)
cluster-2->cluster-1.topics=topic-1, topic-2 (9)
cluster-1->cluster-2.groups=.* (10)
cluster-2->cluster-1.groups=group-1, group-2 (11)

replication.policy.separator=- (12)
sync.topic.acls.enabled=false (13)
refresh.topics.interval.seconds=60 (14)
refresh.groups.interval.seconds=60 (15)
```

(1) 每个 Kafka 集群都使用其别名来标识。

(2) 使用 *bootstrap address* 和端口 *443* 用于 *cluster-1* 的连接信息。两个集群都使用端口 *443* 连接到使用 OpenShift *Routes* 的 Kafka。

(3) ssl. 属性定义 *cluster-1* 的 TLS 配置。

4. *cluster-2* 的连接信息。
5. ssl. 属性定义 *cluster-2* 的 TLS 配置。
6. 从 *cluster-1* 启用复制流到 *cluster-2*。
7. 从 *cluster-2* 启用复制流到 *cluster-1*。
8. 将所有主题从 *cluster-1* 复制到 *cluster-2*。源连接器复制指定的主题。checkpoint 连接器跟踪指定主题的偏移。
9. 将特定主题从 *cluster-2* 复制到 *cluster-1*。
10. 将所有消费者组从 *cluster-1* 复制到 *cluster-2*。checkpoint 连接器复制指定的消费者组。
11. 将特定消费者组从 *cluster-2* 复制到 *cluster-1*。
12. 定义用于重命名远程主题的分隔符。
13. 启用后，ACL 将应用到同步主题。默认值为 false。
14. 检查之间的期间，检查要同步的新主题。
15. 检查要同步的新消费者组之间的周期。

2. OPTION：如果需要，请添加一个策略来覆盖远程主题的自动重命名。该主题不会用源集群的名称来附加名称，而是保留其原始名称。
此可选设置用于主动/被动备份和数据迁移。

```
replication.policy.class=org.apache.kafka.connect.mirror.IdentityReplicationPolicy
```

3. OPTION：如果要同步消费者组偏移，请添加配置来启用和管理同步：

```
refresh.groups.interval.seconds=60
sync.group.offsets.enabled=true (1)
sync.group.offsets.interval.seconds=60 (2)
emit.checkpoints.interval.seconds=60 (3)
```

1. 用于同步消费者组偏移的可选设置，这对于在主动/被动配置中恢复非常有用。默认不启用同步。
2. 如果启用了使用者组偏移的同步，您可以调整同步的频率。
3. 调整检查偏移跟踪的频率。如果您更改了偏移同步的频率，您可能需要调整这些检查的频率。

4. 在目标集群中启动 ZooKeeper 和 Kafka：

```
su - kafka
/opt/kafka/bin/zookeeper-server-start.sh -daemon \
/opt/kafka/config/zookeeper.properties
```

```
/opt/kafka/bin/kafka-server-start.sh -daemon \
/opt/kafka/config/server.properties
```

5. 使用您在属性文件中定义的集群连接配置和复制策略启动 MirrorMaker：

```
/opt/kafka/bin/connect-mirror-maker.sh \
/opt/kafka/config/connect-mirror-maker.properties
```

MirrorMaker 在集群之间设置连接。

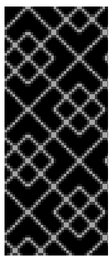
6. 对于每个目标集群，验证主题是否被复制：

```
/opt/kafka/bin/kafka-topics.sh --bootstrap-server <broker_address> --list
```

9.7. (DEPRECTAED)在旧模式中使用 MIRRORMAKER 2

这个步骤描述了如何将 MirrorMaker 2 配置为在旧模式中使用它。旧模式支持之前版本的 MirrorMaker。

MirrorMaker 脚本 **/opt/kafka/bin/kafka-mirror-maker.sh** 可以在旧模式下运行 MirrorMaker 2。



重要

Kafka MirrorMaker 1（称为文档中的 *MirrorMaker*）已在 Apache Kafka 3.0.0 中弃用，并将在 Apache Kafka 4.0.0 中删除。因此，Kafka MirrorMaker 1 也已在 Apache Kafka 的 Streams 中弃用。当使用 Apache Kafka 4.0.0 时，Kafka MirrorMaker 1 将从 Apache Kafka 的 Streams 中删除。作为替换，将 MirrorMaker 2 与 **IdentityReplicationPolicy** 搭配使用。

先决条件

您需要当前与 MirrorMaker 旧版本搭配使用的属性文件。

- **/opt/kafka/config/consumer.properties**
- **/opt/kafka/config/producer.properties**

流程

1. 编辑 MirrorMaker **consumer.properties** 和 **producer.properties** 文件，以关闭 MirrorMaker 2 功能。
例如：

```
replication.policy.class=org.apache.kafka.mirror.LegacyReplicationPolicy (1)

refresh.topics.enabled=false (2)
refresh.groups.enabled=false
emit.checkpoints.enabled=false
emit.heartbeats.enabled=false
sync.topic.configs.enabled=false
sync.topic.acls.enabled=false
```

(1) 模拟之前的 MirrorMaker 版本。

2. MirrorMaker 2 禁用了功能，包括内部 *检查点* 和 *心跳* 主题

2. 保存更改，并使用您在之前版本的 MirrorMaker 中使用的属性文件重启 MirrorMaker：

```
su - kafka /opt/kafka/bin/kafka-mirror-maker.sh \
--consumer.config /opt/kafka/config/consumer.properties \
--producer.config /opt/kafka/config/producer.properties \
--num.streams=2
```

consumer 属性提供源集群和 producer 属性的配置，提供目标集群配置。

MirrorMaker 在集群之间设置连接。

3. 在目标集群中启动 ZooKeeper 和 Kafka：

```
su - kafka
/opt/kafka/bin/zookeeper-server-start.sh -daemon
/opt/kafka/config/zookeeper.properties
```

```
su - kafka
/opt/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/config/server.properties
```

4. 对于目标集群，验证主题是否被复制：

```
/opt/kafka/bin/kafka-topics.sh --bootstrap-server <broker_address> --list
```

第 10 章 为 KAFKA 组件配置日志记录

在配置属性中直接配置 Kafka 组件的日志记录级别。您还可以动态更改 Kafka 代理、Kafka Connect 和 MirrorMaker 2 的代理级别。

增加日志级别详情，如从 INFO 升级到 DEBUG，有助于对 Kafka 集群进行故障排除。但是，更详细的日志也可能对性能造成负面影响，并更难以诊断问题。

10.1. 配置 KAFKA 日志记录属性

Kafka 组件使用 Log4j 框架进行错误日志记录。默认情况下，日志记录配置使用以下属性文件从 classpath 或 **config** 目录中读取：

- Kafka 和 ZooKeeper 的 **log4j.properties**
- Kafka Connect 和 MirrorMaker 2 的 **connect-log4j.properties**

如果没有明确设置，日志记录器会继承每个文件中的 **log4j.rootLogger** 日志记录级别配置。您可以更改这些文件中的日志级别。您还可以为其他日志记录器添加和设置日志记录级别。

您可以使用 **KAFKA_LOG4J_OPTS** 环境变量来更改日志记录属性文件的位置和名称，该变量供组件的启动脚本使用。

传递 Kafka 代理使用的日志属性文件的名称和位置

```
su - kafka
export KAFKA_LOG4J_OPTS="-Dlog4j.configuration=file:/my/path/to/log4j.properties"; \
/opt/kafka/bin/kafka-server-start.sh \
/opt/kafka/config/server.properties
```

传递 ZooKeeper 使用的日志属性文件的名称和位置

```
su - kafka
export KAFKA_LOG4J_OPTS="-Dlog4j.configuration=file:/my/path/to/log4j.properties"; \
/opt/kafka/bin/zookeeper-server-start.sh -daemon \
/opt/kafka/config/zookeeper.properties
```

传递 Kafka Connect 使用的日志属性文件的名称和位置

```
su - kafka
export KAFKA_LOG4J_OPTS="-Dlog4j.configuration=file:/my/path/to/connect-
log4j.properties"; \
/opt/kafka/bin/connect-distributed.sh \
/opt/kafka/config/connect-distributed.properties
```

传递 MirrorMaker 2 使用的日志属性文件的名称和位置

```
su - kafka
export KAFKA_LOG4J_OPTS="-Dlog4j.configuration=file:/my/path/to/connect-
log4j.properties"; \
/opt/kafka/bin/connect-mirror-maker.sh \
/opt/kafka/config/connect-mirror-maker.properties
```

10.2. 为 KAFKA 代理日志记录器动态更改日志记录级别

Kafka 代理日志记录由每个代理中的代理日志记录器提供。在运行时动态更改代理日志记录器的日志记录级别，而无需重启代理。

您还可以动态地将代理日志记录器重置为其默认日志记录级别。

先决条件

- 每个主机上安装了 Apache Kafka 的流，且配置文件可用。
- Kafka 正在运行。

流程

1. 切换到 kafka 用户：

```
su - kafka
```

2. 使用 kafka-configs.sh 工具列出代理的所有代理日志记录器：

```
/opt/kafka/bin/kafka-configs.sh --bootstrap-server <broker_address> --describe --entity-type broker-loggers --entity-name BROKER-ID
```

例如，对于代理 0 ：

```
/opt/kafka/bin/kafka-configs.sh --bootstrap-server localhost:9092 --describe --entity-type broker-loggers --entity-name 0
```

这会返回每个日志记录器的日志记录级别：TRACE, DEBUG, INFO, WARN, ERROR, 或 FATAL。

例如：

```
#...
kafka.controller.ControllerChannelManager=INFO sensitive=false synonyms={}
kafka.log.TimeIndex=INFO sensitive=false synonyms={}
```

3. 更改一个或多个代理日志记录器的日志级别。使用 --alter 和 --add-config 选项，并使用双引号将每个日志记录器及其级别指定为逗号分隔的列表。

```
/opt/kafka/bin/kafka-configs.sh --bootstrap-server <broker_address> --alter --add-config "LOGGER-ONE=NEW-LEVEL,LOGGER-TWO=NEW-LEVEL" --entity-type broker-loggers --entity-name BROKER-ID
```

例如，对于代理 0 ：

```
/opt/kafka/bin/kafka-configs.sh --bootstrap-server localhost:9092 --alter --add-config "kafka.controller.ControllerChannelManager=WARN,kafka.log.TimeIndex=WARN" --entity-type broker-loggers --entity-name 0
```

如果成功，则返回：

```
Completed updating config for broker: 0.
```

重置代理日志记录器
您可以使用 kafka-configs.sh 工具将一个或多个代理日志记录器重置为其默认日志记录级别。使用 --alter 和 --delete-config 选项，并使用双引号将每个代理日志记录器指定为用逗号分开的列表：

```
/opt/kafka/bin/kafka-configs.sh --bootstrap-server localhost:9092 --alter --delete-config
"LOGGER-ONE,LOGGER-TWO" --entity-type broker-loggers --entity-name BROKER-ID
```

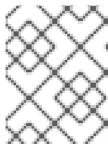
其他资源

- Apache Kafka 文档中的 更新 代理配置

10.3. 动态更改 KAFKA CONNECT 和 MIRRORMAKER 2 的日志记录级别

在运行时动态更改 Kafka Connect worker 或 MirrorMaker 2 连接器的日志记录级别，而无需重启。

使用 Kafka Connect API 为 worker 或连接器日志记录器临时更改日志级别。Kafka Connect API 提供了一个 admin/loggers 端点来获取或修改日志记录级别。当您使用 API 更改日志级别时，connect-log4j.properties 配置文件中的 logger 配置不会改变。如果需要，您可以永久更改配置文件中的日志记录级别。



注意

您只能在分布式或独立模式中时更改 MirrorMaker 2 的日志记录级别。专用 MirrorMaker 2 集群没有 Kafka Connect REST API，因此无法更改日志级别。

Kafka Connect API 的默认监听程序在端口 8083 上，用于此流程。您可以更改或添加更多监听程序，并使用 admin.listeners 配置启用 TLS 身份验证。

admin 端点的监听程序配置示例

```
admin.listeners=https://localhost:8083
admin.listeners.https.ssl.truststore.location=/path/to/truststore.jks
admin.listeners.https.ssl.truststore.password=123456
admin.listeners.https.ssl.keystore.location=/path/to/keystore.jks
admin.listeners.https.ssl.keystore.password=123456
```

如果您不希望 admin 端点可用，您可以通过指定空字符串来禁用它。

禁用 admin 端点的监听程序配置示例

```
admin.listeners=
```

先决条件

- 每个主机上安装了 Apache Kafka 的流，且配置文件可用。
- Kafka 正在运行。
- Kafka Connect 或 MirrorMaker 2 正在运行。

流程

1. 切换到 **kafka** 用户：

```
su - kafka
```

2. 检查 **connect-log4j.properties** 文件中配置的日志记录器的当前日志记录级别：

```
$ cat /opt/kafka/config/connect-log4j.properties

# ...
log4j.rootLogger=INFO, stdout, connectAppender
# ...
log4j.logger.org.reflections=ERROR
```

使用 curl 命令检查 Kafka Connect API 的 **admin/loggers** 端点中的日志记录级别：

```
curl -s http://localhost:8083/admin/loggers/ | jq

{
  "org.reflections": {
    "level": "ERROR"
  },
  "root": {
    "level": "INFO"
  }
}
```

jq 以 JSON 格式打印输出。列表显示标准的**org** 和 **root** 级别的日志程序，以及特定的带有改变的日志级别的日志程序。

如果您在 Kafka Connect 中为 **admin.listeners** 配置配置 TLS (Transport Layer Security)身份验证，则 loggers 端点的地址是为 **admin.listeners** 指定的值，如 https，如 **https://localhost:8083**。

您还可以获取特定日志记录器的日志级别：

```
curl -s
http://localhost:8083/admin/loggers/org.apache.kafka.connect.mirror.MirrorCheckpoint
Connector | jq

{
  "level": "INFO"
}
```

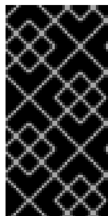
3. 使用 PUT 方法更改日志记录器的日志级别：

```
curl -Ss -X PUT -H 'Content-Type: application/json' -d '{"level": "TRACE"}'
http://localhost:8083/admin/loggers/root

{
  # ...

  "org.reflections": {
    "level": "TRACE"
```

```
  },
  "org.reflections.Reflections": {
    "level": "TRACE"
  },
  "root": {
    "level": "TRACE"
  }
}
```

如果您更改了 root 日志程序，则默认使用了 root 日志级别的日志程序的日志级别也会被改变。

第 11 章 使用 KAFKA 静态配额插件对代理设置限制



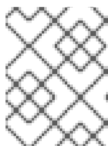
重要

Kafka 静态配额插件只是一个技术预览。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实施任何技术预览功能。此技术预览功能为您提供对即将推出的产品创新的早期访问，允许您在开发过程中测试并提供反馈。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。

使用 *Kafka 静态配额*插件在 Kafka 集群中的代理上设置吞吐量和存储限制。您可以通过在 Kafka 配置文件中添加属性来启用插件和设置限制。您可以设置字节阈值和存储配额，以在与代理交互的客户端上放置限制。

您可以为生成者和消费者带宽设置字节阈值。总限制分布在访问代理的所有客户端中。例如，您可以为制作者设置 40 MBps 的字节阈值。如果两个制作者正在运行，则它们各自限制为 20 MBps 的吞吐量。

存储配额在软限制和硬限制之间节流 Kafka 磁盘存储限制。限制适用于所有可用磁盘空间。生产者在软限制和硬限制之间逐渐减慢。限制可防止磁盘填满速度超过其容量。完整磁盘可能会导致出现问题。硬限制是最大存储限制。



注意

对于 JBOD 存储，限制适用于所有磁盘。如果代理使用两个 1 TB 磁盘，且配额为 1.1 TB，则一个磁盘可能会填满，另一个磁盘将大约为空。

先决条件

- 每个主机上安装了 Apache Kafka 的流，且配置文件可用。

流程

1. 编辑 Kafka 配置属性文件。
   本例中显示了插件属性。

   Kafka 静态配额插件配置示例

   ```
   # ...
   client.quota.callback.class=io.strimzi.kafka.quotas.StaticQuotaCallback (1)
   client.quota.callback.static.produce=1000000 (2)
   client.quota.callback.static.fetch=1000000 (3)
   client.quota.callback.static.storage.soft=400000000000 (4)
   client.quota.callback.static.storage.hard=500000000000 (5)
   client.quota.callback.static.storage.check-interval=5 (6)
   # ...
   ```

   (1) 加载 Kafka Static Quota 插件。

   (2) 设置制作者字节阈值。本示例中为 1 Mbps.

   (3) 设置消费者字节阈值。本示例中为 1 Mbps.

   (4) 为存储设置较低软限制。在本示例中为 400 GB。

5. 为存储设置更高的硬限制。本例中为 500 GB。
6. 设置存储检查间隔（以秒为单位）。本示例中为 5 秒。您可以将其设置为 0 来禁用检查。

2. 使用默认配置文件启动 Kafka 代理。

```
su - kafka
/opt/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/config/server.properties
```

3. 验证 Kafka 代理是否正在运行。

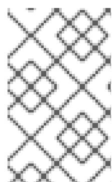
```
jcmd | grep Kafka
```

第 12 章 添加和删除 KAFKA 代理和 ZOOKEEPER 节点

在 Kafka 集群中，管理添加和删除代理和 ZooKeeper 节点对于维护稳定且可扩展的系统至关重要。当您添加到可用代理数量时，您可以为代理中的主题配置默认复制因素和最小同步副本。您可以使用动态重新配置来添加和删除 ZooKeeper 节点，而无需中断。

12.1. 通过添加或删除代理来扩展集群

通过添加代理来扩展 Kafka 集群，可以提高集群的性能和可靠性。添加更多代理会增加可用资源，允许集群处理更大的工作负载并处理更多信息。它还可以通过提供更多副本和备份来提高容错。相反，删除使用率不足的代理可减少资源消耗并提高效率。必须仔细执行扩展，以避免中断或数据丢失。通过在集群中的所有代理间重新分发分区，每个代理的资源使用率会减少，这可以提高集群的整体吞吐量。



注意

要增加 Kafka 主题的吞吐量，您可以增加该主题的分区数量。这允许在集群中的不同代理之间共享主题的负载。但是，如果每个代理受特定资源（如 I/O）的限制，则添加更多分区不会增加吞吐量。在这种情况下，您需要在集群中添加更多代理。

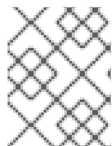
在运行多节点 Kafka 集群时添加代理会影响作为副本的集群中的代理数量。主题的实际复制因素由 **default.replication.factor** 和 **min.insync.replicas** 的设置决定，以及可用代理的数量。例如，一个复制因素为 3 表示主题的每个分区在三个代理之间复制，确保在代理失败时容错。

副本配置示例

```
default.replication.factor = 3
min.insync.replicas = 2
```

当您添加或删除代理时，Kafka 不会自动重新分配分区。执行此操作的最佳方法是使用 Cruise Control。在扩展集群或缩减时，您可以使用 Cruise Control 的 **add-brokers** 和 **remove-brokers** 模式。

- 在扩展 Kafka 集群后，使用 **add-brokers** 模式，将现有代理中的分区副本移到新添加的代理中。
- 在缩减 Kafka 集群前，使用 **remove-brokers** 模式，将分区副本移出要删除的代理。



注意

在缩减代理时，您无法指定要从集群中删除的特定 pod。相反，代理删除过程从最高数字的 pod 开始。

12.2. 在 ZOOKEEPER 集群中添加节点

在不停止整个集群的情况下，使用动态重新配置 从 ZooKeeper 集群添加节点。动态重新配置允许 ZooKeeper 更改一组组成 ZooKeeper 集群的节点成员资格，而不中断。

先决条件

- ZooKeeper 配置文件中启用了动态重新配置(**reconfigEnabled=true**)。
- 启用了 ZooKeeper 身份验证，您可以使用身份验证机制访问新的服务器。

流程

为您要添加的每个 ZooKeeper 服务器执行以下步骤，一次一个：

1. 在 ZooKeeper 集群中添加服务器，如 第 4.1 节 “运行多节点 ZooKeeper 集群”所述，然后启动 ZooKeeper。
2. 请注意新服务器的 IP 地址和配置的访问端口。
3. 为服务器启动 **zookeeper-shell** 会话。从可以访问集群的机器中运行以下命令（如果可以访问集群，这可能是 ZooKeeper 节点或本地机器之一）。

```
su - kafka
/opt/kafka/bin/zookeeper-shell.sh <ip-address>:<zk-port>
```

4. 在 shell 会话中，运行 ZooKeeper 节点，输入以下行将新服务器作为投票成员添加到仲裁中：

```
reconfig -add server.<positive-id> = <address1>:<port1>:<port2>[:role];[<client-port-address>:]<client-port>
```

例如：

```
reconfig -add server.4=172.17.0.4:2888:3888:participant;172.17.0.4:2181
```

其中 **<positive-id>** 是新服务器 ID4。

对于两个端口，**<port1>** *2888* 用于 ZooKeeper 服务器之间的通信，**<port2>** *3888* 用于领导选举机制。

新配置传播到 ZooKeeper 集群中的其他服务器；新服务器现在是仲裁的完整成员。

12.3. 从 ZOOKEEPER 集群中删除节点

在不停止整个集群的情况下，使用动态重新配置 从 ZooKeeper 集群中删除节点。动态重新配置允许 ZooKeeper 更改一组组成 ZooKeeper 集群的节点成员资格，而不中断。

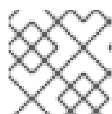
先决条件

- ZooKeeper 配置文件中启用了动态重新配置(**reconfigEnabled=true**)。
- 启用了 ZooKeeper 身份验证，您可以使用身份验证机制访问新的服务器。

流程

为您删除的每个 ZooKeeper 服务器同时执行以下步骤：

1. 登录到在缩减后会保留的其中一个服务器上的 **zookeeper-shell**（例如，服务器 1）。



注意

使用为 ZooKeeper 集群配置的验证机制访问服务器。

2. 删除服务器，如 server 5。

```
reconfig -remove 5
```

3. **取消激活您删除的服务器。**

第 13 章 使用 CRUISE CONTROL 进行集群重新平衡

Cruise Control 是一个用于自动化 Kafka 操作的开源系统，如监控集群工作负载、根据预定义的限制重新平衡集群，并检测和修复异常情况。它包含四个主要组件 - Load Monitor、 Anomaly Detector 和 Executor- 以及用于客户端交互的 REST API。

您可以使用 Cruise Control *来重新平衡* Kafka 集群。在 Red Hat Enterprise Linux 中， Cruise Control for Apache Kafka 作为单独的 ziped 发行版本提供。

Apache Kafka 的流使用 REST API 来支持以下 Cruise Control 功能：

- 从优化目标生成优化方案。
- 根据优化提议，重新平衡 Kafka 集群。

  优化目标

  优化目标描述了从重新平衡实现的特定目标。例如，目标可能是在代理之间更均匀地分布主题副本。您可通过配置更改要包含的目标。目标定义为一个硬目标或软目标。您可以通过 Cruise Control 部署配置添加硬目标。您还具有适合这些类别的主要、默认和用户提供的目标。

  - 硬 目标是预先设置的，必须满足优化建议才能成功进行。
  - 为了成功优化提议，无需满足软目标。如果表示满足所有硬目标，则可以设置它们。
  - 主要目标从 Cruise Control 中继承。其中一些预先设置为硬目标。默认在优化提议中使用主要目标。
  - 默认情况下，默认目标与主目标相同。您可以指定您自己的一组默认目标。
  - 用户提供的目标是配置用于生成特定优化器的默认目标子集。

  优化提议

  优化建议由您要从重新平衡实现的目标组成。您可以生成优化建议，以创建提议的更改概述，以及重新平衡的结果。目标以特定优先级顺序进行评估。然后您可以选择批准或拒绝提议。您可以使用调整的目标集合拒绝再次运行它。
  您可以通过向以下 API 端点之一发出请求并批准优化建议。

  - /rebalance 端点用于运行一个完整的重新平衡操作。
  - 在扩展 Kafka 集群时添加代理后，/add_broker 端点要重新平衡。
  - 缩减 Kafka 集群时，在删除代理前要重新平衡的 /remove_broker 端点。

您可以通过配置属性文件配置优化目标。Apache Kafka 的流提供了 Cruise Control 的示例属性文件。

13.1. CRUISE CONTROL 组件和功能

Cruise Control 包括四个主要组件 - Load Monitor、 Anomaly Detector 和 Executor- 以及用于客户端交互的 REST API。Apache Kafka 的流使用 REST API 来支持以下 Cruise Control 功能：

- 从优化目标生成优化方案。
- 根据优化提议，重新平衡 Kafka 集群。

优化目标

优化目标描述了从重新平衡实现的特定目标。例如，目标可能是在代理之间更均匀地分布主题副本。您可通过配置更改要包含的目标。目标定义为一个硬目标或软目标。您可以通过 Cruise Control 部署配置添加硬目标。您还具有适合这些类别的主要、默认和用户提供的目标。

- 硬 目标是预先设置的，必须满足优化建议才能成功进行。
- 为了成功优化提议，无需满足软目标。如果表示满足所有硬目标，则可以设置它们。
- 主要目标从 Cruise Control 中继承。其中一些预先设置为硬目标。默认在优化提议中使用主要目标。
- 默认情况下，默认目标与主目标相同。您可以指定您自己的一组默认目标。
- 用户提供的目标是 配置用于生成特定优化提议的默认目标子集。

优化提议

优化建议由您要从重新平衡实现的目标组成。您可以生成优化建议，以创建提议的更改概述，以及重新平衡的结果。目标以特定优先级顺序进行评估。然后您可以选择批准或拒绝提议。您可以使用调整的目标集合拒绝再次运行它。
您可以在三种模式之一中生成优化方案。

- **full** 是默认模式，运行完整重新平衡。
- **add-brokers** 是扩展 Kafka 集群时添加代理后使用的模式。
- **remove-brokers** 是缩减 Kafka 集群时删除代理之前使用的模式。

目前还不支持其他 Cruise Control 功能，包括自我修复、通知、编写目标以及更改主题复制因素。

其他资源

- Cruise Control 文档

13.2. 下载 CRUISE CONTROL

可以从红帽网站下载 Cruise Control 的 ZIP 文件发布。您可以从 Apache Kafka 软件下载页面的 Streams for Apache Kafka 下载最新版本的 Red Hat Streams for Apache Kafka。

流程

1. 从红帽客户门户网站下载 Red Hat Streams for Apache Kafka Cruise Control归档的最新版本。
2. 创建 **/opt/cruise-control** 目录：

```
sudo mkdir /opt/cruise-control
```

3. 将 Cruise Control ZIP 文件的内容提取到新目录中：

```
unzip amq-streams-<version>-cruise-control-bin.zip -d /opt/cruise-control
```

4. 将 **/opt/cruise-control** 目录的所有权更改为 **kafka** 用户：

```
sudo chown -R kafka:kafka /opt/cruise-control
```

13.3. 部署 CRUISE CONTROL METRICS REPORTER

在开始 Cruise Control 前，您必须将 Kafka 代理配置为使用提供的 Cruise Control Metrics Reporter。Metrics Reporter 的文件由 Apache Kafka 安装工件的 Streams 提供。

在运行时加载时，Metrics Reporter 会将指标发送到 **__CruiseControlMetrics** 主题，其中有三个自动创建的主题之一。Cruise Control 使用这些指标来创建和更新工作负载模型，并计算优化建议。

先决条件

- 每个主机上安装了 Apache Kafka 的流，且配置文件可用。
- 以 **kafka** 用户身份登录 Red Hat Enterprise Linux。

流程

对于 Kafka 集群中的每个代理，一次一个：

1. 停止 Kafka 代理：

   ```
   /opt/kafka/bin/kafka-server-stop.sh
   ```

2. 编辑 Kafka 配置属性文件，以配置 Cruise Control Metrics Reporter。

   a. 将 **CruiseControlMetricsReporter** 类添加到 **metric.reporters** 配置选项。不要删除任何现有的指标报告器。

   ```
   metric.reporters=com.linkedin.kafka.cruisecontrol.metricsreporter.CruiseControlMetricsReporter
   ```

   b. 添加以下配置选项和值：

   ```
   cruise.control.metrics.topic.auto.create=true
   cruise.control.metrics.topic.num.partitions=1
   cruise.control.metrics.topic.replication.factor=1
   ```

   这些选项允许 Cruise Control Metrics Reporter 创建 **__CruiseControlMetrics** 主题，其日志清理策略为 **DELETE**。如需更多信息，请参阅 Auto-created topics 和 Log cleanup policy for Cruise Control Metrics topic。

3. 如果需要，配置 SSL。

   a. 在 Kafka 配置属性文件中，通过设置相关的客户端配置属性，在 Cruise Control Metrics Reporter 和 Kafka 代理之间配置 SSL。
   Metrics Reporter 接受带有 **cruise.control.metrics.reporter** 前缀的所有特定于标准制作者的配置属性。例如： **cruise.control.metrics.reporter.ssl.truststore.password**。

   b. 在 Cruise Control 属性文件中(**/opt/cruise-control/config/cruisecontrol.properties**)通过设置相关的客户端配置属性在 Kafka 代理和 Cruise Control 服务器之间配置 SSL。
   Cruise Control 从 Kafka 中继承 SSL 客户端属性选项，并将这些属性用于所有 Cruise Control 服务器客户端。

4. 重启 Kafka 代理：

   ```
   /opt/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/config/server.properties
   ```

有关在多节点集群中重启代理的详情，请参考 第 4.3 节 “执行 Kafka 代理的安全滚动重启”。

5. 对剩余的代理重复步骤 1-5。

13.4. 配置并启动 CRUISE CONTROL

配置 Cruise Control 使用的属性，然后使用 **kafka-cruise-control-start.sh** 脚本启动 Cruise Control 服务器。服务器托管到整个 Kafka 集群的单个机器上。

Cruise Control 启动时会自动创建三个主题。如需更多信息，请参阅 自动创建的主题。

先决条件

- 以 **kafka** 用户身份登录 Red Hat Enterprise Linux。
- 您已下载了 Cruise Control。
- 您已部署了 Cruise Control Metrics Reporter。

流程

1. 编辑 Cruise Control 属性文件 (**/opt/cruise-control/config/cruisecontrol.properties**)。
2. 配置以下示例配置中显示的属性：

```
# The Kafka cluster to control.
bootstrap.servers=localhost:9092 (1)

# The replication factor of Kafka metric sample store topic
sample.store.topic.replication.factor=2 (2)

# The configuration for the BrokerCapacityConfigFileResolver (supports JBOD, non-JBOD, and heterogeneous CPU core capacities)
#capacity.config.file=config/capacity.json
#capacity.config.file=config/capacityCores.json
capacity.config.file=config/capacityJBOD.json (3)

# The list of goals to optimize the Kafka cluster for with pre-computed proposals
default.goals={List of default optimization goals} (4)

# The list of supported goals
goals={list of main optimization goals} (5)

# The list of supported hard goals
hard.goals={List of hard goals} (6)

# How often should the cached proposal be expired and recalculated if necessary
proposal.expiration.ms=60000 (7)

# The zookeeper connect of the Kafka cluster
zookeeper.connect=localhost:2181 (8)
```

(1) Kafka 代理的主机和端口号（始终为端口 9092）。

2. Kafka 指标示例存储主题的复制因素。如果要在单节点 Kafka 和 ZooKeeper 集群中评估 Cruise Control，请将此属性设置为 1。对于生产环境，请将此属性设置为 2 或以上。

3. 为代理资源设置最大容量限制的配置文件。使用适用于 Kafka 部署配置的文件。如需更多信息，请参阅 容量配置。

4. 使用完全限定域名(FQDN)的以逗号分隔的默认优化目标列表。许多主要优化目标（请参见 5）已设为默认优化目标；如果需要，您可以添加或删除目标。更多信息请参阅 第 13.5 节 “优化目标概述”。

5. 使用 FQDN 以逗号分隔的主要优化目标列表。要完全排除用于生成优化提议的目标，请将它们从列表中删除。更多信息请参阅 第 13.5 节 “优化目标概述”。

6. 使用 FQDN 以逗号分隔的硬目标列表。七大主要优化目标已设置为硬目标；如果需要，您可以添加或删除目标。更多信息请参阅 第 13.5 节 “优化目标概述”。

7. 刷新从默认优化目标生成的缓存的优化建议的时间间隔（以毫秒为单位）。更多信息请参阅 第 13.6 节 “优化提议概述”。

8. ZooKeeper 连接的主机和端口号（始终端口 2181）。

3. 启动 Cruise Control 服务器。默认情况下，服务器在端口 9092 上启动；（可选）指定不同的端口。

```
cd /opt/cruise-control/
./kafka-cruise-control-start.sh config/cruisecontrol.properties <port_number>
```

4. 要验证 Cruise Control 是否正在运行，请将 GET 请求发送到 Cruise Control 服务器的 /state 端点：

```
curl -X GET 'http://<cc_host>:<cc_port>/kafkacruisecontrol/state'
```

自动创建的主题
下表显示了在 Cruise Control 启动时自动创建的三个主题。Cruise Control 需要这些主题才能正常工作，且不得删除或更改。

表 13.1. 自动创建的主题

自动创建的主题	创建人	功能
__CruiseControlMetrics	Cruise Control Metrics Reporter	将 Metrics Reporter 中的原始指标存储在每个 Kafka 代理中。
__KafkaCruiseControlPartitionMetricSamples	Sything Control	存储每个分区的派生指标。它们由指标示例聚合器创建。
__KafkaCruiseControlModelTrainingSamples	Sything Control	存储用于创建 Cluster Workload Model 的指标样本。

要确保在自动创建的主题中*禁用*日志压缩，请确保配置 Cruise Control Metrics Reporter，如第 13.3 节“部署 Cruise Control Metrics Reporter”所述。日志压缩可以删除 Cruise Control 所需的记录，并防止它正常工作。

其他资源

- Cruise Control Metrics 主题的日志清理策略

13.5. 优化目标概述

优化目标是在 Kafka 集群中重新发布工作负载和资源利用率的限制。要重新平衡 Kafka 集群，Cruise Control 使用优化目标来 生成优化建议。

13.5.1. 优先级的目标顺序

Red Hat Enterprise Linux 上的 Apache Kafka 流支持 Cruise Control 项目中开发的所有优化目标。支持的目标（以优先级默认降序排列）如下：

1. 机架感知性
2. 每个代理对一组主题的最小领导副本数
3. 副本容量
4. 容量：磁盘容量、网络入站容量、网络出站容量
5. CPU 容量
6. 副本分发
7. 潜在的网络输出
8. 资源分布：磁盘使用分发、网络入站使用分布、网络出站使用分布
9. 领导字节速率分布
10. 主题副本分发
11. CPU 用量分布
12. 领导副本分发
13. 首选领导选举机制
14. Kafka Assigner 磁盘用量发行版本
15. intra-broker 磁盘容量
16. intra-broker 磁盘用量

有关每个优化目标的更多信息，请参阅 Cruise Control Wiki 中的 目标。

13.5.2. Cruise Control 属性文件中的目标配置

您可以在 **cruise-control/config/** 目录中的 **cruisecontrol.properties** 文件中设置优化目标。Cruise Control 具有硬优化目标的配置，必须满足以及主要、默认和用户提供的优化目标。

您可以在以下配置中指定以下类型的优化目标：

- Main goals – **cruisecontrol.properties** 文件
- Hard goals – **cruisecontrol.properties** 文件
- Default goals – **cruisecontrol.properties** 文件
- User-provided goals – runtime 参数

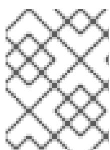
（可选）在运行时设置 用户提供的 优化目标，作为请求到 **/rebalance** 端点的参数。

优化目标取决于代理资源的任何 容量限制。

13.5.3. 硬和软优化目标

硬目标是在优化提议时 *必须满足* 的目标。没有作为硬目标配置的目标被成为*软目标*。您可以将软目标视为 *best effort* 目标：它们不需要在优化建议方面满足，但包含在优化计算中。

Cruise Control 将计算满足所有硬目标以及尽可能多的软目标（按优先级顺序）的优化建议。满足所有硬目标的优化方法由 Analyzer 拒绝，*不会*向用户发送。



注意

例如，您可能有一个软目标来在集群间平均分配主题的副本（主题分布目标）。如果这样做可让所有配置的硬目标满足，则 Cruise Control 将忽略这个目标。

在 Cruise Control 中，以下 主要优化目标 被预先设置为硬目标：

> RackAwareGoal; MinTopicLeadersPerBrokerGoal; ReplicaCapacityGoal; DiskCapacityGoal; NetworkInboundCapacityGoal; NetworkOutboundCapacityGoal; CpuCapacityGoal

要更改硬目标，请编辑 **cruisecontrol.properties** 文件的 **hard.goals** 属性，并使用其完全限定的域名指定目标。

增加硬目标数量可减少 Cruise Control 计算并生成有效优化方案的可能性。

13.5.4. 主要优化目标

所有用户都提供了主要的优化目标。没有在主优化目标中列出的目标在 Cruise Control 操作中不可用。

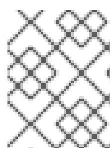
以下主要优化目标在 **cruisecontrol.properties** 文件的 **goals** 属性中预先设置（以降序排列）：

> RackAwareGoal; MinTopicLeadersPerBrokerGoal; ReplicaCapacityGoal; DiskCapacityGoal; NetworkInboundCapacityGoal; NetworkOutboundCapacityGoal; ReplicaDistributionGoal; PotentialNwOutGoal; DiskUsageDistributionGoal; NetworkInboundUsageDistributionGoal; NetworkOutboundUsageDistributionGoal; CpuUsageDistributionGoal; TopicReplicaDistributionGoal; LeaderReplicaDistributionGoal; LeaderBytesInDistributionGoal; PreferredLeaderElectionGoal

为降低复杂度，建议不要更改预先设定的主要优化目标，除非需要完全排除一个或多个目标来生成优化建议。如果需要，可以在配置中修改主要优化目标的优先级顺序，以实现默认优化目标。

要修改预设主要优化目标，以降序在 **goals** 属性中指定目标列表。使用 **cruisecontrol.properties** 文件中所示的完全限定域名。

您必须至少指定一个主要目标，否则 Cruise Control 将会崩溃。



注意

如果更改了预先设置的主要优化目标，您必须确保配置的 hard.goals 是您配置的主要优化目标的子集。否则，在生成优化建议时会出现错误。

13.5.5. 默认优化目标

Cruise Control *使用默认的优化目标*列表来生成*缓存的优化建议*。更多信息请参阅 第 13.6 节 “优化提议概述”。

您可以通过设置 用户提供的优化目标，在运行时覆盖默认优化目标。

以下默认优化目标在 **cruisecontrol.properties** 文件的 **default.goals** 属性中预先设置，以降序排列：

> RackAwareGoal; MinTopicLeadersPerBrokerGoal; ReplicaCapacityGoal; DiskCapacityGoal; NetworkInboundCapacityGoal; NetworkOutboundCapacityGoal; CpuCapacityGoal; ReplicaDistributionGoal; PotentialNwOutGoal; DiskUsageDistributionGoal; NetworkInboundUsageDistributionGoal; NetworkOutboundUsageDistributionGoal; CpuUsageDistributionGoal; TopicReplicaDistributionGoal; LeaderReplicaDistributionGoal; LeaderBytesInDistributionGoal

您必须至少指定一个默认目标，否则 Cruise Control 将会崩溃。

要修改默认优化目标，以降序在 **default.goals** 属性中指定目标列表。默认目标必须是主要优化目标的子集；使用完全限定域名。

13.5.6. 用户提供的优化目标

*用户提供的优化目标*会缩小为特定优化提议配置的默认目标。您可以根据需要将它们设置为 HTTP 请求到 **/rebalance** 端点的参数。更多信息请参阅 第 13.9 节 “生成优化建议”。

用户提供的优化目标可以为不同的场景生成优化建议。例如，您可能想要在不考虑磁盘容量或磁盘利用率的情况下在 Kafka 集群中优化领导副本分布。因此，您将请求发送到 **/rebalance** 端点，其中包含一个用于领导副本分发的目标。

用户提供的优化目标必须：

- 包括所有配置的硬目标，或发生错误
- 是主要优化目标的子集

要在优化提议中忽略配置的硬目标，请将 **skip_hard_goals_check=true** 参数添加到请求中。

其他资源

- Cruise Control 配置
- Cruise Control Wiki 中的配置

13.6. 优化提议概述

optimization proposal 是要生成一个更加均衡的 Kafka 集群、在代理中平均分配分区工作负载的建议概述。

每个优化建议均基于一组用于生成它的 优化目标，受代理资源配置的任何容量限制。

所有优化的提议都是对提议重新平衡的影响的*估算*。您可以批准或拒绝提议。在不生成优化建议的情况下，您无法批准集群重新平衡。

您可以使用以下端点之一运行优化建议：

- /rebalance
- /add_broker
- /remove_broker

13.6.1. 重新平衡端点

当您发送 POST 请求来生成优化建议时，您可以指定重新平衡端点。

/rebalance

/rebalance 端点通过在集群中的所有代理间移动副本来运行完全重新平衡。

/add_broker

add_broker 端点在扩展 Kafka 集群后通过添加一个或多个代理来使用。通常，在扩展 Kafka 集群后，新的代理仅用于托管新创建的主题的分区。如果没有创建新主题，则不会使用新添加的代理，现有代理仍保留在同一负载中。通过在向集群添加代理后立即使用 add_broker 端点，重新平衡操作会将副本从现有代理移到新添加的代理中。您可以在 POST 请求中将新代理指定为 brokerid 列表。

/remove_broker

在缩减 Kafka 集群前，使用 /remove_broker 端点删除一个或多个代理。如果您缩减 Kafka 集群，代理也会关闭，即使它们托管副本。这可能会导致复制的分区，并可能导致某些分区位于其最小 ISR 下（同步副本）。为了避免这种潜在问题，/remove_broker 端点会将副本从要删除的代理移出。当这些代理不再托管副本时，可以安全地运行缩减操作。您可以在 POST 请求中指定您要删除的代理作为 brokerid 列表。

通常，使用 /rebalance 端点通过在代理间分散负载来重新平衡 Kafka 集群。只有在您要扩展集群或缩减并相应地重新平衡副本时，才使用 /add-broker 端点和 /remove_broker 端点。

运行重新平衡的过程实际上在三个不同的端点之间相同。唯一的区别是列出添加或将添加到请求的代理。

13.6.2. 批准或拒绝优化建议

优化提议摘要显示了所提议的更改范围。通过 Cruise Control API 对 HTTP 请求返回概述。

当您向 /rebalance 端点发出 POST 请求时，响应中会返回一个优化提议概述。

返回优化提议概述

```
curl -v -X POST 'cruise-control-server:9090/kafkacruisecontrol/rebalance'
```

使用摘要决定是否批准或拒绝优化提议。

批准优化提议

您可以通过向 /rebalance 端点发出 POST 请求并将dryrun 参数设置为 false 来批准优化提议（默认为 true）。Cruise Control 将提议应用到 Kafka 集群，并启动集群重新平衡操作。

拒绝优化方案

如果您选择不批准优化方案，您可以更改优化目标或更新任何重新平衡性能调优选项，然后生成另一个提议。您可以在没有 dryrun 参数的情况下重新发送请求，以生成新的优化建议。

使用优化建议来评估重新平衡所需的移动。例如，概述描述了 inter-broker 和 intra-broker 移动。inter-broker 重新平衡在独立代理间移动数据。在使用 JBOD 存储配置时，intra-broker 重新平衡可在同一代理上的磁盘之间移动数据。即使您没有提前并批准提议，此类信息也很有用。

因为在重新平衡时在 Kafka 集群中出现额外的负载，您可能会拒绝优化过程或延迟其批准。

在以下示例中，提议建议在不同的代理间重新平衡数据。重新平衡涉及在代理间移动 55 分区副本，包括 12MB 数据。虽然分区副本间的移动对性能有高影响，但数据总数不大。如果总数据量较大，您可以拒绝提议，或者在批准重新平衡以限制 Kafka 集群性能的影响时的时间。

重新平衡性能调优选项有助于降低数据移动的影响。如果可扩展重新平衡周期，您可以将重新平衡分成较小的批处理。一次减少数据移动会减少集群的负载。

优化提议概述示例

```
Optimization has 55 inter-broker replica (12 MB) moves, 0 intra-broker
replica (0 MB) moves and 24 leadership moves with a cluster model of 5
recent windows and 100.000% of the partitions covered.
Excluded Topics: [].
Excluded Brokers For Leadership: [].
Excluded Brokers For Replica Move: [].
Counts: 3 brokers 343 replicas 7 topics.
On-demand Balancedness Score Before (78.012) After (82.912).
Provision Status: RIGHT_SIZED.
```

这个提议还会将 24 个分区领导机移到不同的代理中，这对性能的影响较低。

balancedness 分数是优化提议前后 Kafka 集群的整体平衡量。平衡分数基于优化目标。如果满足所有目标，则分数为 100。当一个目标不满足时，分数会降低。比较均衡分数，以查看 Kafka 集群是否低于重新平衡。

provision 状态指示当前集群配置是否支持优化目标。检查 provision 状态，以查看是否应添加或删除代理。

表 13.2. 优化提议置备状态

状态	描述
RIGHT_SIZED	集群有适当的代理数来满足优化目标。
UNDER_PROVISIONED	集群已置备，需要更多代理来满足优化目标。
OVER_PROVISIONED	集群过度置备，需要较少的代理来满足优化目标。
UNDECIDED	状态不相关，或者尚未决定。

13.6.3. 优化提议概述属性

下表描述了优化提议中包含的属性。

表 13.3. 优化提议概述中包含的属性

属性	描述
n inter-broker replica (y MB) moves	**n** ：将在独立代理之间移动的分区副本数量。 **在重新平衡操作期间影响性能** ：高。 **y MB** ：将移动到独立代理的每个分区副本的大小总和。 **在重新平衡操作期间影响性能** ：变量.集群重新平衡所需的时间越大，完成集群重新平衡所需的时间。
n intra-broker replica (y MB) moves	**n** ：集群代理磁盘之间传输的分区副本数量。 **重新平衡操作期间的性能影响** ： 高，但少于 **inter-broker 副本移动**。 **y MB** ：将在同一代理的磁盘之间移动的每个分区副本的大小总和。 **在重新平衡操作期间影响性能** ：变量.集群重新平衡所需的时间越大，完成集群重新平衡所需的时间。在同一代理的磁盘间移动大量数据比独立代理之间的影响较低（请参阅 **inter-broker replica moves**）。
n 排除主题	优化建议中分区副本/领导移动计算以外的主题数量。 您可以使用以下方法之一排除主题： 在 **cruisecontrol.properties** 文件中指定 **topics.excluded.from.partition.movement** 属性中的正则表达式。 在对 **/rebalance** 端点的 POST 请求中，在 **excluded_topics** 参数中指定正则表达式。 与正则表达式匹配的主题列在响应中，并将在集群重新平衡中排除。
N 领导移动	**n** ：其领导要切换到不同副本的分区数量。 **在重新平衡操作期间影响性能** ：低.
n 最近的窗口	**n** ：优化提议的指标窗口数量。
N% 的分区覆盖	**n%**: 优化建议涵盖的 Kafka 集群中分区的百分比。
On-demand Balancedness Score Before (nn.yyy) After (nn.yyy)	Kafka 集群的整体平衡的测量。 Cruise Control 根据多个因素（目标在 **default.goal** 或用户提供的目标列表中，为每一个优化目标分配了一个 **Balancedness Score**）。**按需保证分数**的计算方式为：从 100 中减去每个违反的软目标 **Balancedness Score** 的总和。 **Before** score 基于 Kafka 集群的当前配置。**After** 分数基于生成的优化提议。

13.6.4. 缓存的优化方案

Cruise Control 根据配置的默认优化目标维护*缓存的优化建议*。从工作负载模型生成，缓存的优化方案每 15 分钟更新一次，以反映 Kafka 集群的当前状态。

当使用以下目标配置时，返回最新缓存的优化方案：

- 默认优化目标
- 用户提供的优化目标，可在当前缓存的提议中满足

要更改缓存的优化提议刷新间隔，请编辑 **cruisecontrol.properties** 文件中的 **proposal.expiration.ms** 设置。考虑快速更改集群的间隔较短，尽管这会在 Cruise Control 服务器上增加负载。

其他资源

- 优化目标概述
- 生成优化建议
- 启动集群重新平衡

13.7. 重新平衡性能调优概述

您可以调整集群重新平衡的几个性能调优选项。这些选项控制如何执行重新平衡中的分区副本和领导移动，以及分配给重新平衡操作的带宽。

分区重新分配命令
优化建议 由单独的分区重新分配命令组成。当您启动提议时，Cruise Control 服务器会将这些命令应用到 Kafka 集群。

分区重新分配命令由以下一种操作组成：

- 分区移动 ：使分区副本及其数据传输到新位置。分区移动可以采用两种形式之一：
    - inter-broker movement ：分区副本被移到不同代理上的日志目录中。
    - intra-broker movement ：分区副本被移到同一代理的不同日志目录中。
- 领导移动 ：尝试切换分区副本的领导。

Cruise Control 在批处理中向 Kafka 集群发出分区重新分配命令。重新平衡期间集群的性能会受到每个批处理中包含的每种移动数量的影响。

要配置分区重新分配命令，请参阅重新平衡调整选项。

副本移动策略
集群重新平衡性能也会受到 *副本移动策略的影响*，该策略应用于分区重新分配命令的批处理。默认情况下，Cruise Control 使用 **BaseReplicaMovementStrategy**，它按照生成顺序应用命令。但是，如果提议的早期有一些非常大的分区重新分配，则此策略可能会减慢其他重新分配的应用程序。

Cruise Control 提供了三种替代副本移动策略，可用于优化提议：

- **PrioritizeSmallReplicaMovementStrategy**: Order reassignments 以升序大小为单位。
- **PrioritizeLargeReplicaMovementStrategy**: Order reassignments 以降序排列。

- **PostponeUrpReplicaMovementStrategy**: 优先考虑为没有处于不同步状态的分区的服务的重新分配。

这些策略可以配置为序列。第一个策略尝试使用其内部逻辑比较两个分区重新分配。如果重新分配等同于，那么它将按顺序传递给下一个策略，以确定顺序等。

要配置副本移动策略，请参阅重新平衡调整选项。

重新平衡调整选项
Cruise Control 提供多个配置选项用于调优重新平衡参数。这些选项通过以下方式设置：

- 作为属性，在 **cruisecontrol.properties** 文件中默认的 Cruise Control 配置中
- 作为 POST 请求中的参数到 **/rebalance** 端点

下表总结了这两种方法的相关配置。

表 13.4. 重新平衡性能调优配置

Cruise Control 属性	KafkaRebalance 参数	默认	描述
num.concurrent.partition.movements.per.broker	**concurrent_partition_movements_per_broker**	5	每个分区重新分配批处理中的最大 inter-broker 分区移动数
num.concurrent.intra.broker.partition.movements	**concurrent_intra_broker_partition_movements**	2	每个分区重新分配批处理中的最大 intra-broker 分区移动数
num.concurrent.leader.movements	**concurrent_leader_movements**	1000	每个分区重新分配批处理中的最大分区领导更改数
default.replication.throttle	**replication_throttle**	null（无限制）	分配给分区重新分配的带宽（每秒字节数）

Cruise Control 属性	KafkaRebalance 参数	默认	描述
default.replica.movement.strategies	**replica_movement_strategies**	**BaseReplicaMovementStrategy**	用于决定为生成的提议执行分区重新分配命令的顺序（优先级顺序）的列表。有三个策略：**PrioritizeSmallReplicaMovementStrategy**, **PrioritizeLargeReplicaMovementStrategy** 和 **PostponeUrpReplicaMovementStrategy**。对于 server 设置，请使用带有策略类的完全限定名称的列表（将 **com.linkedin.kafka.cruisecontrol.executor.strategy.** to the each class name 的开始）。对于重新平衡参数，请使用副本移动策略的类名称的逗号分隔列表。

更改默认设置会影响重新平衡完成所需的时间，以及重新平衡期间在 Kafka 集群上放置的负载。使用较低值可减少负载，但会增加所花费的时间，反之亦然。

其他资源

- Cruise Control Wiki 中的配置
- Cruise Control Wiki 中的 REST API

13.8. CRUISE CONTROL 配置

config/cruisecontrol.properties 文件包含 Cruise Control 的配置。该文件由以下类型之一属性组成：

- 字符串
- Number
- 布尔值

您可以指定并配置 Cruise Control Wiki 的 Configurations 部分中列出的所有属性。

容量配置

Cruise Control 使用 *容量限制* 来确定某些基于资源的优化目标是否被破坏。如果将一个或多个基于资源的目标设置为硬目标，则尝试优化会失败，然后中断。这可防止优化用于生成优化建议。

您可以在 cruise-control/config 中的以下三个 .json 文件中为 Kafka 代理资源指定容量限制：

- capacityJBOD.json ：用于 JBOD Kafka 部署（默认文件）。
- capacity.json: 用于非JBOD Kafka 部署，每个代理都有相同的 CPU 内核数。
- capacityCores.json: 用于在每个代理都有不同 CPU 内核数的非JBOD Kafka 部署中使用。

在 cruisecontrol.properties 中的 capacity.config.file 属性中设置文件。所选文件将用于代理容量解析。例如：

```
capacity.config.file=config/capacityJBOD.json
```

可以在上述单元中为以下代理资源设置容量限制：

- DISK ：以 MB 为单位的磁盘存储
- CPU: CPU 使用率为百分比(0-100)或内核数
- NW_IN ：入站网络吞吐量（KB 每秒）
- NW_OUT ：出站网络吞吐量（KB 每秒）

要将相同的容量限制应用到由 Cruise Control 监控的每个代理，请为代理 ID -1 设置容量限制。要为单个代理设置不同的容量限制，请指定每个代理 ID 及其容量配置。

容量限制配置示例

```
{
  "brokerCapacities":[
    {
      "brokerId": "-1",
      "capacity": {
        "DISK": "100000",
        "CPU": "100",
        "NW_IN": "10000",
        "NW_OUT": "10000"
      },
      "doc": "This is the default capacity. Capacity unit used for disk is in MB, cpu is in
percentage, network throughput is in KB."
    },
    {
      "brokerId": "0",
      "capacity": {
        "DISK": "500000",
        "CPU": "100",
        "NW_IN": "50000",
        "NW_OUT": "50000"
      },
      "doc": "This overrides the capacity for broker 0."
```

```
      }
    ]
  }
```

如需更多信息，请参阅 Cruise Control Wiki 中的填充 容量配置文件。

Cruise Control Metrics 主题的日志清理策略
非常重要的一点是，自动创建的 __CruiseControlMetrics 主题（请参阅额 auto-created topics）有一个日志清理策略 DELETE 而不是 COMPACT。否则，Cruise Control 所需的记录可能会被删除。

如 第 13.3 节 "部署 Cruise Control Metrics Reporter"所述，在 Kafka 配置文件中设置以下选项可确保 COMPACT 日志清理策略被正确设置：

- cruise.control.metrics.topic.auto.create=true
- cruise.control.metrics.topic.num.partitions=1
- cruise.control.metrics.topic.replication.factor=1

如果在 Cruise Control Metrics Reporter 中的主题自动创建为 *disabled*（cruise.control.metrics.topic.auto.create=false），但在 Kafka 集群中是 *enabled*，__CruiseControlMetrics 主题仍然由代理自动创建。在这种情况下，您必须使用 kafka-configs.sh 工具将 __CruiseControlMetrics 主题的日志清理策略改为 DELETE。

1. 获取 __CruiseControlMetrics 主题的当前配置：

```
opt/kafka/bin/kafka-configs.sh --bootstrap-server <broker_address> --entity-type
topics --entity-name __CruiseControlMetrics --describe
```

2. 在主题配置中更改日志清理策略：

```
/opt/kafka/bin/kafka-configs.sh --bootstrap-server <broker_address> --entity-type
topics --entity-name __CruiseControlMetrics --alter --add-config cleanup.policy=delete
```

如果在 Cruise Control Metrics Reporter *和* Kafka 集群中都 *禁用了* 主题自动创建，则必须手动创建 __CruiseControlMetrics 主题，然后使用 kafka-configs.sh 工具将它配置为使用 DELETE 日志清理策略。

更多信息请参阅 第 7.9 节 "修改主题配置"。

日志记录配置
Cruise Control 将 log4j1 用于所有服务器日志记录。要更改默认配置，请编辑 /opt/cruise-control/config/log4j.properties 文件中的 log4j.properties 文件。

在更改生效前，您必须重启 Cruise Control 服务器。

13.9. 生成优化建议

当您向 /rebalance 端点发出 POST 请求时，Cruise Control 会根据提供的优化目标生成一个优化建议来重新平衡 Kafka 集群。您可以使用优化提议的结果来重新平衡 Kafka 集群。

您可以使用以下端点之一运行优化建议：

- /rebalance

- /add_broker
- /remove_broker

您使用的端点取决于您是否正在在 Kafka 集群中已在运行的所有代理进行重新平衡；或者要在扩展后或缩减 Kafka 集群前重新平衡。如需更多信息，请参阅使用代理扩展 重新平衡端点。

优化建议作为*空运行*方式生成，除非提供了 **dryrun** 参数，并将其设置为 **false**。在"dry run mode"中，Cruise Control 会生成优化建议和估算结果，但不通过重新平衡集群来启动提议。

您可以分析优化提议中返回的信息，并决定是否批准它。

使用以下参数向端点发出请求：

dryRun

type: boolean, default: true

告知 Cruise 控制是否只生成优化提议（**true**），或生成优化建议并执行集群重新平衡 (**false**)。

当 **dryrun=true** （默认值）时，您还可以传递 **verbose** 参数，以返回有关 Kafka 集群状态的更多详细信息。这包括应用优化提议前后每个 Kafka 代理上负载的指标，以及 before 和 after 值之间的区别。

excluded_topics

类型： regex

与优化提议的计算中排除的主题匹配的正则表达式。

目标

type: 字符串列表，默认为配置的 **default.goals** 列表

用户提供的优化目标列表，用于准备优化方法。如果没有提供目标，则会使用 **cruisecontrol.properties** 文件中配置的 **default.goals** 列表。

skip_hard_goals_check

type: boolean, default: **false**

默认情况下，Cruise Control 会检查用户提供的优化目标（在 **goals** 参数中）包含所有配置的硬目标（在 **hard.goals**中）。如果您提供不是配置的 **hard.goals** 子集的目标，请求会失败。

如果要生成不使用所有配置的 **hard.goals** 优化目标，则将 **skip_hard_goals_check** 设置为 **true**。

json

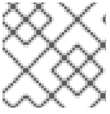
type: boolean, default: **false**

控制 Cruise Control 服务器返回的响应类型。如果没有提供，或设置为 **false**，Cruise Control 会返回格式的文本，以便在命令行上显示。如果要以编程方式提取返回的信息元素，请设置 **json=true**。这将返回 JSON 格式的文本，这些文本可以传送到 **jq** 等工具，或在脚本和程序中解析它们。

详细

type: boolean, default: **false**

控制 Cruise Control 服务器返回的响应中详情级别。可与 **dryrun=true** 一起使用。



注意

其他参数可用。如需更多信息，请参阅 Cruise Control Wiki 中的 REST API。

先决条件

- Kafka 正在运行。
- 您已配置了 Cruise Control。
- （可选）您已在 主机上安装了新代理，以包含在重新平衡中。

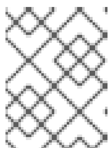
流程

1. 使用对 /rebalance、/add_broker 或 /remove_broker 端点的 POST 请求来生成优化建议。

   使用默认目标到 /rebalance 的请求示例

   ```
   curl -v -X POST 'cruise-control-server:9090/kafkacruisecontrol/rebalance'
   ```

   缓存的优化提议会立即返回。

   注意

   如果返回 NotEnoughValidWindows， Cruise Control 尚未记录足够的指标数据，以生成优化器。等待几分钟，然后重新发送请求。

   使用指定目标向 /rebalance 的请求示例

   ```
   curl -v -X POST 'cruise-control-server:9090/kafkacruisecontrol/rebalance?
   goals=RackAwareGoal,ReplicaCapacityGoal'
   ```

   如果请求满足提供的目标，则立即返回缓存的优化提议。否则，会使用提供的目标生成新的优化方案；这需要更长的时间来计算。您可以通过在请求中添加 ignore_proposal_cache=true 参数来强制实施此行为。

   使用指定目标向 /rebalance 的请求示例，而无需硬目标

   ```
   curl -v -X POST 'cruise-control-server:9090/kafkacruisecontrol/rebalance?
   goals=RackAwareGoal,ReplicaCapacityGoal,ReplicaDistributionGoal&skip_hard_goal
   _check=true'
   ```

   包含指定代理的 /add_broker 请求示例

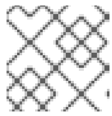
   ```
   curl -v -X POST 'cruise-control-server:9090/kafkacruisecontrol/add_broker?
   brokerid=3,4'
   ```

   请求仅包含新代理的 ID。例如，此请求添加 ID 为 3 和 4 的代理。在重新平衡时，副本会从现有代理移到新代理中。

   排除指定代理的 /remove_broker 请求示例

```
curl -v -X POST 'cruise-control-server:9090/kafkacruisecontrol/remove_broker?brokerid=3,4'
```

请求仅包含被排除的代理 ID。例如，此请求排除 ID 为 3 和 4 的代理。在重新平衡时，副本会从要删除的代理移到其他现有代理中。



注意

如果要删除的代理包含排除主题，则副本仍会移动。

2. 查看响应中包含的优化提议。属性描述待处理的集群重新平衡操作。
   提议包含所提出优化的高级概述，以及各个默认优化目标的总结，以及执行过程后的预期集群状态。

   请特别注意以下信息：

   - 在 重新平衡概述后集群负载。如果满足您的要求，您应该使用高级别概述来评估所提议更改的影响。
   - **n inter-broker replica (y MB) moves** 表示在代理间移动的数据量。值越大，重新平衡过程中对 Kafka 集群的潜在性能影响越大。
   - **n intra-broker replica (y MB) moves** 表示代理本身（生成磁盘）中多少数据。数值越大，对各个代理的潜在性能影响越大（尽管小于 **n inter-broker replica (y MB) moves**）。
   - 领导机移动的数量。这在重新平衡过程中对集群的性能有可忽略的影响。

异步响应
默认情况下，Cruise Control REST API 端点会在 10 秒后超时，虽然提议生成将继续在服务器上。如果最新缓存的优化器尚未就绪，或者用户提供的优化目标是通过 **ignore_proposal_cache=true** 指定，则可能会出现超时。

要允许您稍后检索优化过程，请记录请求的唯一标识符，该标识符在 **/rebalance** 端点的响应标头中提供。

要使用 **curl** 获取响应，请指定 verbose (**-v**)选项：

```
curl -v -X POST 'cruise-control-server:9090/kafkacruisecontrol/rebalance'
```

以下是一个标头示例：

```
* Connected to cruise-control-server (::1) port 9090 (#0)
> POST /kafkacruisecontrol/rebalance HTTP/1.1
> Host: cc-host:9090
> User-Agent: curl/7.70.0
> Accept: /
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Date: Mon, 01 Jun 2023 15:19:26 GMT
< Set-Cookie: JSESSIONID=node01wk6vjzjj12go13m81o7no5p7h9.node0; Path=/
< Expires: Thu, 01 Jan 1970 00:00:00 GMT
< User-Task-ID: 274b8095-d739-4840-85b9-f4cfaaf5c201
< Content-Type: text/plain;charset=utf-8
< Cruise-Control-Version: 2.0.103.redhat-00002
```

```
< Cruise-Control-Commit_Id: 58975c9d5d0a78dd33cd67d4bcb497c9fd42ae7c
< Content-Length: 12368
< Server: Jetty(9.4.26.v20200117-redhat-00001)
```

如果优化提议在超时时间内没有就绪，您可以重新提交 POST 请求，这一次包括标头中原始请求的 User-Task-ID ：

```
curl -v -X POST -H 'User-Task-ID: 274b8095-d739-4840-85b9-f4cfaaf5c201' 'cruise-control-server:9090/kafkacruisecontrol/rebalance'
```

接下来要做什么

第 13.10 节 “批准优化提议”

13.10. 批准优化提议

如果您对最近生成的优化器满意，您可以指示 Cruise Control 启动集群重新平衡并开始重新分配分区。

在生成优化提议并启动集群重新平衡之间尽可能少的时间。如果因为您生成了原始优化提议，集群状态可能会改变。因此，启动的集群重新平衡可能与您检查的不同。如果有疑问，首先要生成新的优化方案。

一个集群会重新平衡，状态为"Active"，一次才能进行。

先决条件

- 您已从 Cruise 控制中生成了优化的提议。

流程

1. 使用 dryrun=false 参数向 /rebalance、 /add_broker 或 /remove_broker 端点发送 POST 请求：如果您使用 /add_broker 或 /remove_broker 端点来生成包含或排除代理的提议，请使用同一端点来执行使用或不使用指定代理进行重新平衡。

   对 /rebalance的请求示例

   ```
   curl -X POST 'cruise-control-server:9090/kafkacruisecontrol/rebalance?dryrun=false'
   ```

   对 /add_broker的请求示例

   ```
   curl -v -X POST 'cruise-control-server:9090/kafkacruisecontrol/add_broker?dryrun=false&brokerid=3,4'
   ```

   对 /remove_broker的请求示例

   ```
   curl -v -X POST 'cruise-control-server:9090/kafkacruisecontrol/remove_broker?dryrun=false&brokerid=3,4'
   ```

   Cruise Control 启动集群重新平衡并返回优化建议。

2. 检查优化提议中总结的更改。如果更改不是您期望的， 您可以停止重新平衡。

3. 使用 /user_tasks 端点检查集群重新平衡的进度。集群重新平衡状态为 "Active"。
   查看 Cruise Control 服务器上执行的所有集群重新平衡任务：

```
curl 'cruise-control-server:9090/kafkacruisecontrol/user_tasks'

USER TASK ID      CLIENT ADDRESS  START TIME     STATUS  REQUEST URL
c459316f-9eb5-482f-9d2d-97b5a4cd294d  0:0:0:0:0:0:0:1       2020-06-01_16:10:29 UTC
Active      POST /kafkacruisecontrol/rebalance?dryrun=false
445e2fc3-6531-4243-b0a6-36ef7c5059b4  0:0:0:0:0:0:0:1       2020-06-01_14:21:26 UTC
Completed   GET /kafkacruisecontrol/state?json=true
05c37737-16d1-4e33-8e2b-800dee9f1b01  0:0:0:0:0:0:0:1       2020-06-01_14:36:11 UTC
Completed   GET /kafkacruisecontrol/state?json=true
aebae987-985d-4871-8cfb-6134ecd504ab  0:0:0:0:0:0:0:1       2020-06-01_16:10:04 UTC
```

4. 要查看特定集群重新平衡任务的状态，请提供 **user-task-ids** 参数和任务 ID：

```
curl 'cruise-control-server:9090/kafkacruisecontrol/user_tasks?user_task_ids=c459316f-9eb5-482f-9d2d-97b5a4cd294d'
```

（可选）在缩减时删除代理

成功重新平衡后，您可以停止排除的任何代理来缩减 Kafka 集群。

1. 检查正在删除的每个代理是否没有任何实时分区(**log.dirs**)。

```
ls -l <LogDir> | grep -E '^d' | grep -vE '[a-zA-Z0-9.-]+\.[a-z0-9]+-delete$'
```

如果日志目录与正则表达式 **\.[a-z0-9]-delete$** 不匹配，则活动分区仍然存在。如果您有活跃的分区，请检查重新平衡已完成，或配置优化建议。您可以再次运行提议。在继续下一步之前，请确保没有活动的分区。

2. 停止代理。

```
su - kafka
/opt/kafka/bin/kafka-server-stop.sh
```

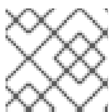
3. 确认代理已停止。

```
jcmd | grep kafka
```

13.11. 停止活跃的集群重新平衡

您可以停止当前正在进行的集群重新平衡。

这指示 Cruise Control 完成当前分区重新分配的批处理，然后停止重新平衡。当重新平衡停止后，已完成的分区重新分配已被应用；因此，与重新平衡操作开始前，Kafka 集群的状态会有所不同。如果需要进一步重新平衡，您应该生成新的优化方案。



注意

在 intermediate（停止）状态中的 Kafka 集群的性能可能比初始状态更糟。

先决条件

- 集群重新平衡正在进行（由状态"Active"）指定。

流程

流程

- 发送 POST 请求到 **/stop_proposal_execution** 端点：

```
curl -X POST 'cruise-control-server:9090/kafkacruisecontrol/stop_proposal_execution'
```

其他资源

- 生成优化建议

第 14 章 使用 CRUISE CONTROL 修改主题复制因素

向 Cruise Control REST API 的 /topic_configuration 端点发出请求，以修改主题配置，包括复制因素。

先决条件

- 以 kafka 用户身份登录 Red Hat Enterprise Linux。
- 您已配置了 Cruise Control。
- 您已部署了 Cruise Control Metrics Reporter。

流程

1. 启动 Cruise Control 服务器。默认情况下，服务器在端口 9092 上启动；（可选）指定不同的端口。

```
cd /opt/cruise-control/
./kafka-cruise-control-start.sh config/cruisecontrol.properties <port_number>
```

2. 要验证 Cruise Control 是否正在运行，请将 GET 请求发送到 Cruise Control 服务器的 /state 端点：

```
curl -X GET 'http://<cc_host>:<cc_port>/kafkacruisecontrol/state'
```

3. 使用 --describe 选项运行 bin/kafka-topics.sh 命令，并检查目标主题的当前复制因素：

```
/opt/kafka/bin/kafka-topics.sh \
  --bootstrap-server localhost:9092 \
  --topic <topic_name> \
  --describe
```

4. 更新主题的复制因素：

```
curl -X POST 'http://<cc_host>:<cc_port>/kafkacruisecontrol/topic_configuration?
topic=<topic_name>&replication_factor=<new_replication_factor>&dryrun=false'
```

例如，curl -X POST 'localhost:9090/kafkacruisecontrol/topic_configuration?topic=topic1&replication_factor=3&dryrun=false'。

5. 使用 --describe 选项运行 bin/kafka-topics.sh 命令，以查看对主题的更改结果。

第 15 章 使用分区重新分配工具

在扩展 Kafka 集群时，您可能需要添加或删除代理并更新分区的发布或主题的复制因素。要更新分区和主题，您可以使用 **kafka-reassign-partitions.sh** 工具。

工具可用于重新分配分区，并在代理间平衡分区分布以提高性能。您还可以更改主题的复制因素。但是，建议您使用 Cruise Control 进行自动分区重新分配和集群重新平衡，并更改主题复制因素。Cruise Control 可以在不停机的情况下将主题从一个代理移到另一个代理，这是重新分配分区的最高效方法。

15.1. 分区重新分配工具概述

分区重新分配工具提供以下功能来管理 Kafka 分区和代理：

重新分发分区副本
: 通过添加或删除代理来扩展集群，并将 Kafka 分区从大量载入的代理移到使用不足的代理中。要做到这一点，您必须创建一个分区重新分配计划，标识要移动哪些主题和分区，以及移动它们的位置。建议对这类操作进行 Cruise Control，因为它 可自动执行集群重新平衡过程。

扩展和缩减主题复制因素
: 增加或减少 Kafka 主题的复制因素。要做到这一点，您必须创建一个分区重新分配计划，标识分区之间的现有复制分配，并使用复制因素更改更新分配。

更改首选领导
: 更改 Kafka 分区的首选领导。如果当前首选领导不可用，或者要在集群中的代理间重新分发负载，这很有用。要做到这一点，您必须创建一个分区重新分配计划，通过更改副本顺序为每个分区指定新的首选领导。

更改日志目录以使用特定的 JBOD 卷
: 更改 Kafka 代理的日志目录以使用特定的 JBOD 卷。如果要将 Kafka 数据移到不同的磁盘或存储设备中，这非常有用。要做到这一点，您必须创建一个分区重新分配计划，该计划为每个主题指定新日志目录。

15.1.1. 生成分区重新分配计划

分区重新分配工具(**kafka-reassign-partitions.sh**)通过生成分区分配计划来工作，指定应将哪些分区从当前代理移到新代理中。

如果您对计划满意，您可以执行它。然后，该工具执行以下操作：

- 将分区数据迁移到新代理
- 更新 Kafka 代理上的元数据以反映新分区分配
- 触发 Kafka 代理的滚动重启，以确保新分配生效

分区重新分配工具有三个不同的模式：

--generate
: 取一组主题和代理，并生成 *重新分配 JSON 文件*，这会导致将这些主题的分区分配给这些代理。由于这对整个主题进行操作，因此当您只需要重新分配某些主题的一些分区时，无法使用它。

--execute
: 取一个 *重新分配 JSON 文件*，并将其应用到集群中的分区和代理。因此，获得分区的代理遵循分区领导机。对于给定分区，当新代理发现并加入 ISR（同步副本）后，旧代理将停止为后续，并将删除其副本。

--verify

使用与 **--execute** 步骤相同的 *重新分配 JSON 文件*， **-- verify** 会检查文件中所有分区是否已移至其预期的代理中。如果重新分配已完成， **-- verify** 也会删除任何生效的流量节流(**--throttle**)。除非被删除， throttles 将继续影响集群，即使重新分配完成后也是如此。

在任何给定时间，集群中只能有一个重新分配运行，且无法取消正在运行的重新分配。如果需要取消重新分配，请等待它完成，然后执行另一个重新分配来恢复第一个重新分配的影响。**kafka-reassign-partitions.sh** 将打印这个 reversion 的重新分配 JSON 作为其输出的一部分。非常大的重新分配应该被分解为多个较小的重新分配，以防需要停止 in-progress 重新分配。

15.1.2. 在分区重新分配 JSON 文件中指定主题

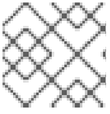
kafka-reassign-partitions.sh 工具使用一个重新分配 JSON 文件，该文件指定要重新分配的主题。如果要移动特定分区，您可以生成重新分配 JSON 文件或手动创建文件。

一个基本的重新分配 JSON 文件在以下示例中显示结构，它描述了属于两个 Kafka 主题的三个分区。每个分区都会被重新分配给一组新的副本，这些副本由代理 ID 识别。版本、**topic**、分区、和 **replicas** 属性都是必需的。

分区重新分配 JSON 文件结构示例

```
{
  "version": 1, (1)
  "partitions": [ (2)
    {
      "topic": "example-topic-1", (3)
      "partition": 0, (4)
      "replicas": [1, 2, 3] (5)
    },
    {
      "topic": "example-topic-1",
      "partition": 1,
      "replicas": [2, 3, 4]
    },
    {
      "topic": "example-topic-2",
      "partition": 0,
      "replicas": [3, 4, 5]
    }
  ]
}
```

1. 重新分配 JSON 文件格式的版本。目前只支持版本 1，因此应始终为 1。
2. 指定重新分配的分区的数组。
3. 分区所属的 Kafka 主题的名称。
4. 被重新分配的分区的 ID。
5. 应该为此分区分配为副本的代理 ID 的排序数组。列表中的第一个代理是领导副本。



注意

不包含在 JSON 中的分区不会改变。

如果您只使用主题数组指定 主题，分区重新分配工具会重新分配属于指定主题的所有分区。

为主题重新分配所有分区的重新分配 JSON 文件结构示例

```
{
  "version": 1,
  "topics": [
    { "topic": "my-topic"}
  ]
}
```

15.1.3. 在 JBOD 卷间重新分配分区

当在 Kafka 集群中使用 JBOD 存储时，您可以在特定卷及其日志目录间重新分配分区（每个卷只有一个日志目录）。

要将分区重新分配给特定卷，请在重新分配 JSON 文件中为每个分区添加 **log_dirs** 值。每个 **log_dirs** 数组包含与 **replicas** 数组相同的条目数，因为每个副本都应分配给特定的日志目录。**log_dirs** 数组包含一个到日志目录的绝对路径或特殊值 **any**。**any** 值表示 Kafka 可以为该副本选择任何可用的日志目录，这在 JBOD 卷间重新分配分区时很有用。

使用日志目录重新分配 JSON 文件结构示例

```
{
  "version": 1,
  "partitions": [
    {
      "topic": "example-topic-1",
      "partition": 0,
      "replicas": [1, 2, 3]
      "log_dirs": ["/var/lib/kafka/data-0/kafka-log1", "any", "/var/lib/kafka/data-1/kafka-log2"]
    },
    {
      "topic": "example-topic-1",
      "partition": 1,
      "replicas": [2, 3, 4]
      "log_dirs": ["any",  "/var/lib/kafka/data-2/kafka-log3", "/var/lib/kafka/data-3/kafka-log4"]
    },
    {
      "topic": "example-topic-2",
      "partition": 0,
      "replicas": [3, 4, 5]
      "log_dirs": ["/var/lib/kafka/data-4/kafka-log5", "any",  "/var/lib/kafka/data-5/kafka-log6"]
    }
  ]
}
```

15.1.4. 分区重新分配节流

分区重新分配可能会是一个较慢的过程，因为它涉及在代理间传输大量数据。为了避免对客户端产生不利

影响，您可以限制重新分配过程。使用带有 **kafka-reassign-partitions.sh** 工具的 **--throttle** 参数，以节流重新分配。您可以为代理间移动分区指定每秒的最大阈值（以字节为单位）。例如，**-- throttle 5000000** 为移动分区设置 50 MBps 的最大阈值。

节流可能会导致重新分配完成所需的时间。

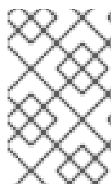
- 如果节流过低，则新分配的代理将无法与要发布的记录保持同步，且重新分配永远不会完成。
- 如果节流过高，客户端将会受到影响。

例如，对于生成者，这可能比等待确认的正常延迟高。对于消费者，这可能因为轮询之间延迟更高的吞吐量导致的吞吐量下降。

15.2. 添加代理后重新分配分区

在增加 Kafka 集群中的代理数量后，使用 **kafka-reassign-partitions.sh** 工具生成的重新分配文件来重新分配分区。重新分配文件应该描述了如何将分区重新分配给放大 Kafka 集群中的代理。您可以将文件中指定的重新分配应用到代理，然后验证新分区分配。

此流程描述了使用 TLS 的安全扩展过程。您需要一个使用 TLS 加密和 mTLS 身份验证的 Kafka 集群。



注意

虽然您可以使用 **kafka-reassign-partitions.sh** 工具，但建议使用 Cruise Control进行自动分区重新分配和集群重新平衡。Cruise Control 可以在不停机的情况下将主题从一个代理移到另一个代理，这是重新分配分区的最高效方法。

先决条件

- 现有的 Kafka 集群。
- 安装了 额外 AMQ 代理的新机器。
- 您已创建了 JSON 文件，以指定如何将分区重新分配给放大集群中的代理。
  在此过程中，我们为名为 **my-topic** 的主题重新分配所有分区。名为 **topics.json** 的 JSON 文件指定主题，用于生成 **reassignment.json** 文件。

示例 JSON 文件指定 my-topic

```
{
 "version": 1,
 "topics": [
  { "topic": "my-topic"}
 ]
}
```

流程

1. 使用与集群中其他代理相同的设置为新代理创建一个配置文件，但 **broker.id** 除外，它应该是没有被任何其他代理使用的数字。
2. 启动新的 Kafka 代理，将您在上一步中创建的配置文件作为 **kafka-server-start.sh** 脚本的参数：

```
su - kafka
/opt/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/config/server.properties
```

3. 验证 Kafka 代理是否正在运行。

```
jcmd | grep Kafka
```

4. 为每个新代理重复上述步骤。

5. 如果没有这样做，请使用 **kafka-reassign-partitions.sh** 工具生成名为 **reassignment.json** 的重新分配 JSON 文件。

生成重新分配 JSON 文件的命令示例

```
/opt/kafka/bin/kafka-reassign-partitions.sh \
  --bootstrap-server localhost:9092 \
  --topics-to-move-json-file topics.json \ 1
  --broker-list 0,1,2,3,4 \ 2
  --generate
```

1 指定主题的 JSON 文件。

2 用于包含在操作中的 kafka 集群中的代理 ID。这假设代理 4 已被添加。

显示当前和提议的副本分配的 JSON 文件示例

```
Current partition replica assignment
{"version":1,"partitions":[{"topic":"my-topic","partition":0,"replicas":[0,1,2],"log_dirs":
["any","any","any"]},{"topic":"my-topic","partition":1,"replicas":[1,2,3],"log_dirs":
["any","any","any"]},{"topic":"my-topic","partition":2,"replicas":[2,3,0],"log_dirs":
["any","any","any"]}]}

Proposed partition reassignment configuration
{"version":1,"partitions":[{"topic":"my-topic","partition":0,"replicas":
[0,1,2,3],"log_dirs":["any","any","any","any"]},{"topic":"my-
topic","partition":1,"replicas":[1,2,3,4],"log_dirs":["any","any","any","any"]},
{"topic":"my-topic","partition":2,"replicas":[2,3,4,0],"log_dirs":
["any","any","any","any"]}]}
```

如果您需要稍后恢复更改，请将此文件的副本保存到本地。

6. 使用 **--execute** 选项运行分区重新分配。

```
/opt/kafka/bin/kafka-reassign-partitions.sh \
  --bootstrap-server localhost:9092 \
  --reassignment-json-file reassignment.json \
  --execute
```

如果要节流复制，您还可以通过 **--throttle** 选项，并传递一个 inter-broker 节流率（以字节/秒为单位）。例如：

```
/opt/kafka/bin/kafka-reassign-partitions.sh \
```

```
  --bootstrap-server localhost:9092 \
  --reassignment-json-file reassignment.json \
  --throttle 5000000 \
  --execute
```

7. 使用 **--verify** 选项验证重新分配是否已完成。

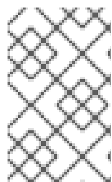
```
/opt/kafka/bin/kafka-reassign-partitions.sh \
  --bootstrap-server localhost:9092 \
  --reassignment-json-file reassignment.json \
  --verify
```

当 **--verify** 命令报告正在移动的每个分区都成功完成时，重新分配已完成。最后 **--verify** 也具有删除任何重新分配节流的影响。

15.3. 在删除代理前重新分配分区

在减少 Kafka 集群中的代理数量前，使用 **kafka-reassign-partitions.sh** 工具生成的重新分配文件来重新分配分区。重新分配文件必须描述如何将分区重新分配给 Kafka 集群中的剩余代理。您可以将文件中指定的重新分配应用到代理，然后验证新分区分配。首先删除编号最高的 pod 中的代理。

此流程描述了使用 TLS 的安全扩展过程。您需要一个使用 TLS 加密和 mTLS 身份验证的 Kafka 集群。



注意

虽然您可以使用 **kafka-reassign-partitions.sh** 工具，但建议使用 Cruise Control进行自动分区重新分配和集群重新平衡。Cruise Control 可以在不停机的情况下将主题从一个代理移到另一个代理，这是重新分配分区的最高效方法。

先决条件

- 现有的 Kafka 集群。
- 您已创建了 JSON 文件，以指定如何将分区重新分配给减少集群中的代理。
  在此过程中，我们为名为 **my-topic** 的主题重新分配所有分区。名为 **topics.json** 的 JSON 文件指定主题，用于生成 **reassignment.json** 文件。

示例 JSON 文件指定 **my-topic**

```
{
  "version": 1,
  "topics": [
    { "topic": "my-topic"}
  ]
}
```

流程

1. 如果没有这样做，请使用 **kafka-reassign-partitions.sh** 工具生成名为 **reassignment.json** 的重新分配 JSON 文件。

   生成重新分配 JSON 文件的命令示例

```
/opt/kafka/bin/kafka-reassign-partitions.sh \
  --bootstrap-server localhost:9092 \
  --topics-to-move-json-file topics.json \ (1)
  --broker-list 0,1,2,3 \ (2)
  --generate
```

1. 指定主题的 JSON 文件。
2. 用于包含在操作中的 kafka 集群中的代理 ID。这假设代理 4 已被删除。

显示当前和提议的副本分配的 JSON 文件示例

```
Current partition replica assignment
{"version":1,"partitions":[{"topic":"my-topic","partition":0,"replicas":
[3,4,2,0],"log_dirs":["any","any","any","any"]},{"topic":"my-
topic","partition":1,"replicas":[0,2,3,1],"log_dirs":["any","any","any","any"]},
{"topic":"my-topic","partition":2,"replicas":[1,3,0,4],"log_dirs":
["any","any","any","any"]}]}

Proposed partition reassignment configuration
{"version":1,"partitions":[{"topic":"my-topic","partition":0,"replicas":[0,1,2],"log_dirs":
["any","any","any"]},{"topic":"my-topic","partition":1,"replicas":[1,2,3],"log_dirs":
["any","any","any"]},{"topic":"my-topic","partition":2,"replicas":[2,3,0],"log_dirs":
["any","any","any"]}]}
```

如果您需要稍后恢复更改，请将此文件的副本保存到本地。

2. 使用 **--execute** 选项运行分区重新分配。

```
/opt/kafka/bin/kafka-reassign-partitions.sh \
  --bootstrap-server localhost:9092 \
  --reassignment-json-file reassignment.json \
  --execute
```

如果要节流复制，您还可以通过 **--throttle** 选项，并传递一个 inter-broker 节流率（以字节/秒为单位）。例如：

```
/opt/kafka/bin/kafka-reassign-partitions.sh \
  --bootstrap-server localhost:9092 \
  --reassignment-json-file reassignment.json \
  --throttle 5000000 \
  --execute
```

3. 使用 **--verify** 选项验证重新分配是否已完成。

```
/opt/kafka/bin/kafka-reassign-partitions.sh \
  --bootstrap-server localhost:9092 \
  --reassignment-json-file reassignment.json \
  --verify
```

当 **--verify** 命令报告正在移动的每个分区都成功完成时，重新分配已完成。最后 **--verify** 也具有删除任何重新分配节流的影响。

4. 检查正在删除的每个代理是否没有任何实时分区(log.dirs)。

```
ls -l <LogDir> | grep -E '^d' | grep -vE '[a-zA-Z0-9.-]+\.[a-z0-9]+-delete$'
```

如果日志目录与正则表达式 \.[a-z0-9]-delete$ 不匹配，则活动分区仍然存在。如果您有活跃的分区，检查重新分配已完成，或者在重新分配 JSON 文件中的配置。您可以再次运行重新分配。在继续下一步之前，请确保没有活动的分区。

5. 停止代理。

```
su - kafka
/opt/kafka/bin/kafka-server-stop.sh
```

6. 确认 Kafka 代理已停止。

```
jcmd | grep kafka
```

15.4. 更改主题的复制因素

使用 **kafka-reassign-partitions.sh** 工具更改 Kafka 集群中主题的复制因素。这可以通过重新分配文件来完成，以描述如何更改主题副本。

先决条件

- 现有的 Kafka 集群。
- 您已创建了 JSON 文件，以指定要在操作中包含的主题。
  在此过程中，名为 **my-topic** 的主题有 4 个副本，而我们希望将其减小到 3。名为**topics.json** 的 JSON 文件指定主题，用于生成 **reassignment.json** 文件。

示例 JSON 文件指定 my-topic

```
{
  "version": 1,
  "topics": [
    { "topic": "my-topic"}
  ]
}
```

流程

1. 如果没有这样做，请使用 **kafka-reassign-partitions.sh** 工具生成名为 **reassignment.json** 的重新分配 JSON 文件。

   生成重新分配 JSON 文件的命令示例

```
/opt/kafka/bin/kafka-reassign-partitions.sh \
  --bootstrap-server localhost:9092 \
  --topics-to-move-json-file topics.json \ (1)
  --broker-list 0,1,2,3,4 \ (2)
  --generate
```

1. 指定主题的 JSON 文件。
2. 用于包含在操作中的 kafka 集群中的代理 ID。

显示当前和提议的副本分配的 JSON 文件示例

```
Current partition replica assignment
{"version":1,"partitions":[{"topic":"my-topic","partition":0,"replicas":
[3,4,2,0],"log_dirs":["any","any","any","any"]},{"topic":"my-
topic","partition":1,"replicas":[0,2,3,1],"log_dirs":["any","any","any","any"]},
{"topic":"my-topic","partition":2,"replicas":[1,3,0,4],"log_dirs":
["any","any","any","any"]}]}

Proposed partition reassignment configuration
{"version":1,"partitions":[{"topic":"my-topic","partition":0,"replicas":
[0,1,2,3],"log_dirs":["any","any","any","any"]},{"topic":"my-
topic","partition":1,"replicas":[1,2,3,4],"log_dirs":["any","any","any","any"]},
{"topic":"my-topic","partition":2,"replicas":[2,3,4,0],"log_dirs":
["any","any","any","any"]}]}
```

如果您需要稍后恢复更改，请将此文件的副本保存到本地。

2. 编辑 **reassignment.json**，以从每个分区中删除副本。
例如，使用 jq 删除主题的每个分区的列表中最后一个副本：

删除每个分区的最后一个主题副本

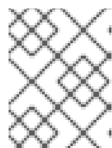
```
jq '.partitions[].replicas |= del(.[-1])' reassignment.json > reassignment.json
```

显示更新的副本的重新分配文件示例

```
{"version":1,"partitions":[{"topic":"my-topic","partition":0,"replicas":[0,1,2],"log_dirs":
["any","any","any","any"]},{"topic":"my-topic","partition":1,"replicas":
[1,2,3],"log_dirs":["any","any","any","any"]},{"topic":"my-
topic","partition":2,"replicas":[2,3,4],"log_dirs":["any","any","any","any"]}]}
```

3. 使用 **--execute** 选项使主题副本更改。

```
/opt/kafka/bin/kafka-reassign-partitions.sh \
  --bootstrap-server localhost:9092 \
  --reassignment-json-file reassignment.json \
  --execute
```



注意

从代理中删除副本不需要任何代理数据移动，因此不需要节流复制。如果要添加副本，则可能需要更改节流率。

4. 使用 **--verify** 选项，验证对主题副本的更改是否已完成。

```
/opt/kafka/bin/kafka-reassign-partitions.sh \
  --bootstrap-server localhost:9092 \
```

```
  --reassignment-json-file reassignment.json \
  --verify
```

当 --verify 命令报告正在移动的每个分区都成功完成时，重新分配已完成。最后 --verify 也具有删除任何重新分配节流的影响。

5. 使用 --describe 选项运行 bin/kafka-topics.sh 命令，以查看对主题的更改结果。

```
/opt/kafka/bin/kafka-topics.sh \
  --bootstrap-server localhost:9092 \
  --describe
```

减少主题副本数的结果

```
my-topic  Partition: 0  Leader: 0  Replicas: 0,1,2 Isr: 0,1,2
my-topic  Partition: 1  Leader: 2  Replicas: 1,2,3 Isr: 1,2,3
my-topic  Partition: 2  Leader: 3  Replicas: 2,3,4 Isr: 2,3,4
```

第 16 章 设置分布式追踪

分布式追踪允许您跟踪分布式系统中的应用程序间事务的进度。在微服务架构中，追踪跟踪服务间事务的进度。跟踪数据对于监控应用程序性能和目标系统和最终用户应用程序的问题非常有用。

在 Apache Kafka 的流中，追踪有助于对消息的端到端跟踪：从源系统到 Kafka，然后从 Kafka 到目标系统和应用程序。它补充了 JMX 指标 中可以查看的指标，以及组件日志记录器。

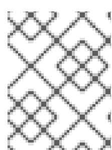
以下 Kafka 组件内置了对追踪的支持：

- Kafka Connect
- MirrorMaker
- MirrorMaker 2
- Apache Kafka Bridge 的流

Kafka 代理不支持追踪。

您可以将追踪配置添加到组件的属性文件中。

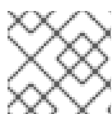
要启用追踪，您可以设置环境变量，并将追踪系统库添加到 Kafka 类路径。对于 Jaeger tracing，您可以使用 Jaeger Exporter 为 OpenTelemetry 添加追踪工件。



注意

Apache Kafka 的流不再支持 OpenTracing。如果您之前将 OpenTracing 与 Jaeger 搭配使用，我们建议您改为使用 OpenTelemetry。

要在 Kafka producer、消费者和 Kafka Streams API 应用程序中启用追踪，您可以 *检测* 应用程序代码。在检测程序时，客户端会生成 trace 数据；例如，当生成消息或向日志写入偏移时。



注意

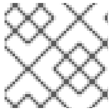
对于除 Apache Kafka 的流外的应用程序和系统设置追踪不在此内容范围内。

16.1. 流程概述

要为 Apache Kafka 设置流追踪，请按照以下步骤操作：

- 为 Kafka Connect、MirrorMaker 2 和 MirrorMaker 设置追踪：
  - 为 Kafka Connect 启用追踪
  - 为 MirrorMaker 2 启用追踪
  - 为 MirrorMaker 启用追踪
- 为客户端设置追踪：
  - 为 Kafka 客户端初始化 Jaeger tracer
- 使用 tracers 检测客户端：
  - 用于追踪的工具生成者和消费者

- 用于追踪的工具 Kafka Streams 应用程序



注意

有关为 Kafka Bridge 启用追踪的详情，请参考使用 Apache Kafka Bridge 的 Streams。

16.2. 追踪选项

在 Jaeger tracing 系统中使用 OpenTelemetry。

OpenTelemetry 提供独立于追踪或监控系统的 API 规格。

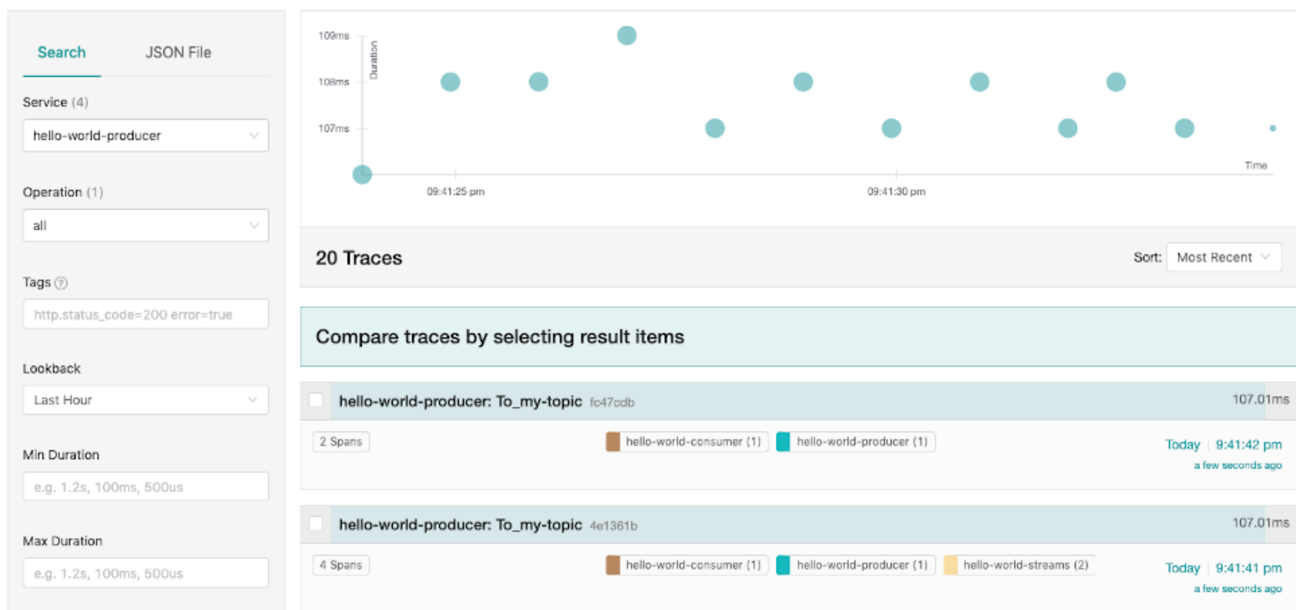
您可以使用 API 检测应用程序代码以进行追踪。

- 检测的应用程序会为分布式系统中的单个请求生成 *trace*。
- 跟踪由 *范围* 组成，用于定义一段时间内的特定工作单元。

Jaeger 是基于微服务的分布式系统的追踪系统。

- Jaeger 用户界面允许您查询、过滤和分析 trace 数据。

Jaeger 用户界面显示一个简单的查询



其他资源

- Jaeger 文档
- OpenTelemetry 文档

16.3. 用于追踪的环境变量

在为 Kafka 组件启用追踪或为 Kafka 客户端初始化 tracer 时，请使用环境变量。

追踪环境变量可能会改变。有关最新信息，请参阅 OpenTelemetry 文档。

下表描述了用于设置 tracer 的关键环境变量。

表 16.1. OpenTelemetry 环境变量

属性	必需	Description
OTEL_SERVICE_NAME	是	OpenTelemetry 的 Jaeger tracing 服务的名称。
OTEL_EXPORTER_JAEGER_ENDPOINT	是	用于追踪的导出器。
OTEL_TRACES_EXPORTER	是	用于追踪的导出器。默认设置为 **otlp**。如果使用 Jaeger tracing，则需要将此环境变量设置为 **jaeger**。如果您使用另一个追踪实现，请指定使用的导出器。

16.4. 为 KAFKA CONNECT 启用追踪

使用配置属性为 Kafka Connect 启用分布式追踪。只有 Kafka Connect 本身生成和使用的消息才会被 traced。要跟踪 Kafka Connect 和外部系统之间发送的消息，您必须在连接器中为这些系统配置追踪。

您可以启用使用 OpenTelemetry 的追踪。

流程

1. 将追踪工件添加到 **opt/kafka/libs** 目录中。
2. 在相关的 Kafka Connect 配置文件中配置生成者和消费者追踪。
   - 如果您以独立模式运行 Kafka 连接，请编辑 **/opt/kafka/config/connect-standalone.properties** 文件。
   - 如果您以分布式模式运行 Kafka Connect，请编辑 **/opt/kafka/config/connect-distributed.properties** 文件。

   在配置文件中添加以下追踪拦截器属性：

   OpenTelemetry 的属性

   ```
   producer.interceptor.classes=io.opentelemetry.instrumentation.kafkaclients.TracingProducerInterceptor
   consumer.interceptor.classes=io.opentelemetry.instrumentation.kafkaclients.TracingConsumerInterceptor
   ```

   启用追踪后，您可以在运行 Kafka Connect 脚本时初始化追踪。
3. 保存配置文件。
4. 设置用于追踪 的环境变量。
5. 使用配置文件作为参数（以及任何连接器属性）在独立或分布式模式下启动 Kafka 连接：

   在独立模式中运行 Kafka 连接

```
su - kafka
/opt/kafka/bin/connect-standalone.sh \
/opt/kafka/config/connect-standalone.properties \
connector1.properties \
[connector2.properties ...]
```

在分布式模式下运行 Kafka Connect

```
su - kafka
/opt/kafka/bin/connect-distributed.sh /opt/kafka/config/connect-distributed.properties
```

现在，启用了 Kafka Connect 的内部使用者和制作者。

16.5. 为 MIRRORMAKER 2 启用追踪

通过在 MirrorMaker 2 属性文件中定义 Interceptor 属性，为 MirrorMaker 2 启用分布式追踪。信息在 Kafka 集群之间追踪。跟踪数据记录消息进入并离开 MirrorMaker 2 组件。

您可以启用使用 OpenTelemetry 的追踪。

流程

1. 将追踪工件添加到 opt/kafka/libs 目录中。

2. 在 opt/kafka/config/connect-mirror-maker.properties 文件中配置制作者和消费者追踪。
在配置文件中添加以下追踪拦截器属性：

   OpenTelemetry 的属性

   ```
   header.converter=org.apache.kafka.connect.converters.ByteArrayConverter
   producer.interceptor.classes=io.opentelemetry.instrumentation.kafkaclients.TracingProducerInterceptor
   consumer.interceptor.classes=io.opentelemetry.instrumentation.kafkaclients.TracingConsumerInterceptor
   ```

   ByteArrayConverter 会阻止 Kafka Connect 将消息标头（包含追踪 ID）转换为 base64 编码。这样可确保在源和目标集群中消息都相同。

   启用追踪后，您可以在运行 Kafka MirrorMaker 2 脚本时初始化追踪。

3. 保存配置文件。

4. 设置用于追踪 的环境变量。

5. 使用生成者和消费者配置文件作为参数启动 MirrorMaker 2：

   ```
   su - kafka
   /opt/kafka/bin/connect-mirror-maker.sh \
   /opt/kafka/config/connect-mirror-maker.properties
   ```

   MirrorMaker 2 的内部使用者和制作者现在被启用用于追踪。

16.6. 为 MIRRORMAKER 启用追踪

通过将 Interceptor 属性作为消费者和制作者配置参数传递，为 MirrorMaker 启用分布式追踪。消息从源集群追踪到目标集群。跟踪数据记录消息进入和离开 MirrorMaker 组件。

您可以启用使用 OpenTelemetry 的追踪。

流程

1. 将追踪工件添加到 opt/kafka/libs 目录中。
2. 在 /opt/kafka/config/producer.properties 文件中配置生产者追踪。
   添加以下追踪拦截器属性：

   OpenTelemetry 的 producer 属性

   ```
   producer.interceptor.classes=io.opentelemetry.instrumentation.kafkaclients.TracingPr
   oducerInterceptor
   ```

3. 保存配置文件。
4. 在 /opt/kafka/config/consumer.properties 文件中配置消费者追踪。
   添加以下追踪拦截器属性：

   OpenTelemetry 的消费者属性

   ```
   consumer.interceptor.classes=io.opentelemetry.instrumentation.kafkaclients.TracingC
   onsumerInterceptor
   ```

   启用追踪后，您可以在运行 Kafka MirrorMaker 脚本时初始化追踪。

5. 保存配置文件。
6. 设置用于追踪 的环境变量。
7. 使用生成者和消费者配置文件作为参数启动 MirrorMaker：

   ```
   su - kafka
   /opt/kafka/bin/kafka-mirror-maker.sh \
   --producer.config /opt/kafka/config/producer.properties \
   --consumer.config /opt/kafka/config/consumer.properties \
   --num.streams=2
   ```

   MirrorMaker 的内部使用者和制作者现已启用用于追踪。

16.7. 初始化 KAFKA 客户端的追踪

为 OpenTelemetry 初始化 tracer，然后检测您的客户端应用程序以进行分布式追踪。您可以检测 Kafka producer 和消费者客户端，以及 Kafka Streams API 应用程序。

使用一组 追踪环境变量 配置和初始化 tracer。

流程

在每个客户端应用程序中添加 tracer 的依赖项：

1. 将 Maven 依赖项添加到客户端应用程序的 pom.xml 文件中：

OpenTelemetry 的依赖项

```
<dependency>
   <groupId>io.opentelemetry.semconv</groupId>
   <artifactId>opentelemetry-semconv</artifactId>
   <version>1.21.0-alpha</version>
</dependency>
<dependency>
   <groupId>io.opentelemetry</groupId>
   <artifactId>opentelemetry-exporter-otlp</artifactId>
   <version>1.34.1</version>
   <exclusions>
      <exclusion>
         <groupId>io.opentelemetry</groupId>
         <artifactId>opentelemetry-exporter-sender-okhttp</artifactId>
      </exclusion>
   </exclusions>
</dependency>
<dependency>
   <groupId>io.opentelemetry</groupId>
   <artifactId>opentelemetry-exporter-sender-grpc-managed-channel</artifactId>
   <version>1.34.1</version>
   <scope>runtime</scope>
</dependency>
<dependency>
   <groupId>io.opentelemetry</groupId>
   <artifactId>opentelemetry-sdk-extension-autoconfigure</artifactId>
   <version>1.34.1</version>
</dependency>
<dependency>
   <groupId>io.opentelemetry.instrumentation</groupId>
   <artifactId>opentelemetry-kafka-clients-2.6</artifactId>
   <version>1.32.0-alpha</version>
</dependency>
<dependency>
   <groupId>io.opentelemetry</groupId>
   <artifactId>opentelemetry-sdk</artifactId>
   <version>1.34.1</version>
</dependency>
<dependency>
   <groupId>io.opentelemetry</groupId>
   <artifactId>opentelemetry-exporter-sender-jdk</artifactId>
   <version>1.34.1-alpha</version>
   <scope>runtime</scope>
</dependency>
<dependency>
   <groupId>io.grpc</groupId>
   <artifactId>grpc-netty-shaded</artifactId>
   <version>1.61.0</version>
</dependency>
```

2. 使用追踪环境变量定义 tracer 的配置。

3. 创建一个 tracer，它使用环境变量初始化：

为 OpenTelemetry 创建 tracer

为 OpenTelemetry 创建 tracer

```
OpenTelemetry ot = GlobalOpenTelemetry.get();
```

4. 将 tracer 注册为全局 tracer：

```
GlobalTracer.register(tracer);
```

5. 检测您的客户端：
   - 第 16.8 节 “用于追踪的制作者和消费者的工具”
   - 第 16.9 节 “为追踪检测 Kafka Streams 应用程序”

16.8. 用于追踪的制作者和消费者的工具

检测应用程序代码，以便在 Kafka 生成者和消费者中启用追踪。使用 decorator 模式或拦截器来检测您的 Java 生成者和消费者应用程序代码以进行追踪。然后，您可以在从主题生成或检索消息时记录 trace。

OpenTelemetry 检测项目提供类来支持生成者和消费者的工具。

decorator 检测
: 对于 decorator 检测，请为追踪创建修改后的制作者或消费者实例。

拦截器检测
: 对于拦截器检测，请将追踪功能添加到消费者或生成者配置中。

先决条件

- 您已为 客户端 初始化了追踪。
  您可以通过在项目中添加追踪 JAR 作为依赖项来启用生成者和消费者应用程序的检测。

流程

在每个制作者和消费者应用的应用程序代码中执行这些步骤。使用 decorator 模式或拦截器（拦截器）检测您的客户端应用程序代码。

- 要使用 decorator 模式，请创建一个修改后的制作者或消费者实例来发送或接收消息。
  您传递了原始 KafkaProducer 或 KafkaConsumer 类。

  OpenTelemetry 的 decorator 检测示例

```
// Producer instance
Producer < String, String > op = new KafkaProducer < > (
    configs,
    new StringSerializer(),
    new StringSerializer()
    );
    Producer < String, String > producer = tracing.wrap(op);
KafkaTracing tracing = KafkaTracing.create(GlobalOpenTelemetry.get());
producer.send(...);

//consumer instance
Consumer<String, String> oc = new KafkaConsumer<>(
    configs,
```

```
    new StringDeserializer(),
    new StringDeserializer()
    );
    Consumer<String, String> consumer = tracing.wrap(oc);
consumer.subscribe(Collections.singleton("mytopic"));
ConsumerRecords<Integer, String> records = consumer.poll(1000);
ConsumerRecord<Integer, String> record = ...
SpanContext spanContext = TracingKafkaUtils.extractSpanContext(record.headers(),
tracer);
```

- 要使用拦截器，请在生成者或消费者配置中设置拦截器类。
  您以通常的方式使用 KafkaProducer 和 KafkaConsumer 类。TracingProducerInterceptor 和 TracingConsumerInterceptor interceptor 类负责追踪功能。

  使用拦截器的制作者配置示例

```
senderProps.put(ProducerConfig.INTERCEPTOR_CLASSES_CONFIG,
    TracingProducerInterceptor.class.getName());

KafkaProducer<Integer, String> producer = new KafkaProducer<>(senderProps);
producer.send(...);
```

  使用拦截器的消费者配置示例

```
consumerProps.put(ConsumerConfig.INTERCEPTOR_CLASSES_CONFIG,
    TracingConsumerInterceptor.class.getName());

KafkaConsumer<Integer, String> consumer = new KafkaConsumer<>
(consumerProps);
consumer.subscribe(Collections.singletonList("messages"));
ConsumerRecords<Integer, String> records = consumer.poll(1000);
ConsumerRecord<Integer, String> record = ...
SpanContext spanContext = TracingKafkaUtils.extractSpanContext(record.headers(),
tracer);
```

16.9. 为追踪检测 KAFKA STREAMS 应用程序

检测应用程序代码，以便在 Kafka Streams API 应用程序中启用追踪。使用 decorator 模式或拦截器来检测您的 Kafka Streams API 应用程序以进行追踪。然后，您可以在从主题生成或检索消息时记录 trace。

decorator 检测

对于 decorator 检测，请为追踪创建一个修改后的 Kafka Streams 实例。对于 OpenTelemetry，您需要创建一个自定义 TracingKafkaClientSupplier 类，以提供 Kafka Streams 的追踪工具。

拦截器检测

对于拦截器检测，在 Kafka Streams producer 和消费者配置中添加追踪功能。

先决条件

- 您已为客户端 初始化了追踪。
  您可以通过在项目中添加追踪 JAR 作为依赖项来启用 Kafka Streams 应用程序中的检测。
- 要使用 OpenTelemetry 检测 Kafka Streams，您需要编写自定义 TracingKafkaClientSupplier。

- 自定义 TracingKafkaClientSupplier 可以扩展 Kafka 的DefaultKafkaClientSupplier，覆盖生成者和消费者创建方法，将实例嵌套与遥测相关的代码。

  自定义 TracingKafkaClientSupplier 示例

```
private class TracingKafkaClientSupplier extends DefaultKafkaClientSupplier {
  @Override
  public Producer<byte[], byte[]> getProducer(Map<String, Object> config) {
    KafkaTelemetry telemetry = KafkaTelemetry.create(GlobalOpenTelemetry.get());
    return telemetry.wrap(super.getProducer(config));
  }

  @Override
  public Consumer<byte[], byte[]> getConsumer(Map<String, Object> config) {
    KafkaTelemetry telemetry = KafkaTelemetry.create(GlobalOpenTelemetry.get());
    return telemetry.wrap(super.getConsumer(config));
  }

  @Override
  public Consumer<byte[], byte[]> getRestoreConsumer(Map<String, Object> config)
{
    return this.getConsumer(config);
  }

  @Override
  public Consumer<byte[], byte[]> getGlobalConsumer(Map<String, Object> config) {
    return this.getConsumer(config);
  }
}
```

流程

为每个 Kafka Streams API 应用程序执行这些步骤。

- 要使用 decorator 模式，创建一个 TracingKafkaClientSupplier 供应商接口的实例，然后为 KafkaStreams 提供供应商接口。

  decorator 检测示例

```
KafkaClientSupplier supplier = new TracingKafkaClientSupplier(tracer);
KafkaStreams streams = new KafkaStreams(builder.build(), new
StreamsConfig(config), supplier);
streams.start();
```

- 要使用拦截器，请在 Kafka Streams producer 和消费者配置中设置拦截器类。TracingProducerInterceptor 和 TracingConsumerInterceptor interceptor 类负责追踪功能。

  使用拦截器的制作者和消费者配置示例

```
props.put(StreamsConfig.PRODUCER_PREFIX +
ProducerConfig.INTERCEPTOR_CLASSES_CONFIG,
TracingProducerInterceptor.class.getName());
props.put(StreamsConfig.CONSUMER_PREFIX +
ConsumerConfig.INTERCEPTOR_CLASSES_CONFIG,
TracingConsumerInterceptor.class.getName());
```

16.10. 使用 OPENTELEMETRY 指定追踪系统

您可以指定 OpenTelemetry 支持的其他追踪系统，而不是默认的 Jaeger 系统。

如果要在 OpenTelemetry 中使用另一个追踪系统，请执行以下操作：

1. 将追踪系统库添加到 Kafka 类路径。
2. 将追踪系统的名称添加为额外的 exporter 环境变量。

   不使用 Jaeger 时的额外环境变量

```
OTEL_SERVICE_NAME=my-tracing-service
OTEL_TRACES_EXPORTER=zipkin (1)
OTEL_EXPORTER_ZIPKIN_ENDPOINT=http://localhost:9411/api/v2/spans (2)
```

(1) 追踪系统的名称。在本例中，指定了 Zipkin。

(2) 侦听 span 的特定所选导出器的端点。在本例中，指定了 Zipkin 端点。

其他资源

- OpenTelemetry 导出器值

16.11. 为 OPENTELEMETRY 指定自定义 SPAN 名称

追踪 *span* 是 Jaeger 中的逻辑工作单元，包括操作名称、开始时间和持续时间。span 具有内置名称，但您可以在使用的 Kafka 客户端检测中指定自定义范围名称。

指定自定义范围名称是可选的，只有在生成者和消费者客户端检测或 Kafka Streams 检测中使用 decorator 模式时才适用。

无法通过 OpenTelemetry 直接指定自定义 span 名称。相反，您可以通过向客户端应用程序添加代码来提取额外的标签和属性来检索范围名称。

提取属性的代码示例

```
//Defines attribute extraction for a producer
private static class ProducerAttribExtractor implements AttributesExtractor < ProducerRecord
< ? , ? > , Void > {
   @Override
   public void onStart(AttributesBuilder attributes, ProducerRecord < ? , ? > producerRecord) {
      set(attributes, AttributeKey.stringKey("prod_start"), "prod1");
   }
   @Override
   public void onEnd(AttributesBuilder attributes, ProducerRecord < ? , ? > producerRecord,
@Nullable Void unused, @Nullable Throwable error) {
      set(attributes, AttributeKey.stringKey("prod_end"), "prod2");
   }
}
//Defines attribute extraction for a consumer
private static class ConsumerAttribExtractor implements AttributesExtractor <
ConsumerRecord < ? , ? > , Void > {
   @Override
```

```
    public void onStart(AttributesBuilder attributes, ConsumerRecord < ? , ? > producerRecord)
{
        set(attributes, AttributeKey.stringKey("con_start"), "con1");
    }
    @Override
    public void onEnd(AttributesBuilder attributes, ConsumerRecord < ? , ? > producerRecord,
@Nullable Void unused, @Nullable Throwable error) {
        set(attributes, AttributeKey.stringKey("con_end"), "con2");
    }
}
//Extracts the attributes
public static void main(String[] args) throws Exception {
        Map < String, Object > configs = new HashMap < >
(Collections.singletonMap(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
"localhost:9092"));
        System.setProperty("otel.traces.exporter", "jaeger");
        System.setProperty("otel.service.name", "myapp1");
        KafkaTracing tracing = KafkaTracing.newBuilder(GlobalOpenTelemetry.get())
            .addProducerAttributesExtractors(new ProducerAttribExtractor())
            .addConsumerAttributesExtractors(new ConsumerAttribExtractor())
            .build();
```

第 17 章 使用 KAFKA EXPORTER

Kafka Exporter 是一个开源项目，用于增强对 Apache Kafka 代理和客户端的监控。

Kafka Exporter 由 Streams for Apache Kafka 提供，用于使用 Kafka 集群部署，从与偏移、消费者组、消费者滞后和主题相关的 Kafka 代理中提取额外的指标数据。

例如，使用指标数据来帮助识别速度较慢的用户。

lag 数据作为 Prometheus 指标公开，然后可在 Grafana 中显示这些指标数据进行分析。

如果您已经使用 Prometheus 和 Grafana 监控内置 Kafka 指标，您可以将 Prometheus 配置为同时提取 Kafka Exporter Prometheus 端点。

Kafka 通过 JMX 公开指标，然后可作为 Prometheus 指标导出。如需更多信息，请参阅使用 JMX 监控集群。

17.1. 消费者滞后

消费者滞后表示消息的速度与消息的消耗的差别。具体来说，给定消费者组的消费者滞后指示分区中最后一个消息与当前由该消费者获取的消息之间的延迟。lag 反映了与分区日志末尾相关的消费者偏移位置。

这种差异有时被称为生成者偏移和消费者偏移之间的 *delta*，在 Kafka 代理主题分区中的读写位置。

假设主题将 100 个消息流过一秒。生成者偏移（主题分区头）和消费者读取的最后偏移之间的 1000 个消息表示有 10 秒的延迟。

监控消费者滞后的重要性

对于依赖于实时数据的处理的应用程序，监控消费者来判断其是否不会变得太大。整个过程越好，流程从实时处理目标中得到的增长。

例如，消费者滞后可能是消耗太多的旧数据（这些数据尚未被清除）或出现计划外的关闭。

减少消费者滞后

减少滞后的典型操作包括：

- 通过添加新消费者来扩展消费者组
- 增加消息的保留时间，以保留在主题中
- 添加更多磁盘容量以增加消息缓冲

减少消费者滞后的操作取决于底层基础架构，支持 Apache Kafka 的用例流。例如，分发的消费者不太可能从其磁盘缓存中获取请求的服务从代理服务。在某些情况下，可能可以接受自动丢弃信息，直到消费者发现为止。

17.2. KAFKA EXPORTER 警报规则示例

特定于 Kafka Exporter 的警报通知规则示例如下：

UnderReplicatedPartition

警告警告主题正在复制时的警报，并且代理没有复制充足的分区。如果主题有一个或多个重复分区，默认配置是警报。该警报可能会表示 Kafka 实例已停机，或者 Kafka 集群过载。可能需要计划重启 Kafka 代理才能重启复制过程。

TooLargeConsumerGroupLag

警告警告消费者组的滞后对于特定主题分区而言太大。默认配置是 1000 记录。大型滞后可能表示消费者太慢，并落于生产者。

NoMessageForTooLong

警告在一段时间内没有收到消息的警报。时间段的默认配置为 10 分钟。延迟可能是防止制作者将消息发布到主题的配置问题。

您可以根据您的特定需求调整警报规则。

其他资源

有关设置警报规则的更多信息，请参阅 Prometheus 文档中的 配置。

17.3. KAFKA EXPORTER 指标

Lag 信息由 Kafka Exporter 公开，作为 Grafana 中表示的 Prometheus 指标。

Kafka Exporter 会公开代理、主题和消费者组的指标数据。

表 17.1. 代理指标输出

Name	信息
kafka_brokers	Kafka 集群中的代理数量

表 17.2. 主题指标输出

Name	信息
kafka_topic_partitions	主题的分区数
kafka_topic_partition_current_offset	代理的当前主题分区偏移
kafka_topic_partition_oldest_offset	代理的最旧的主题分区偏移
kafka_topic_partition_in_sync_replica	主题分区的 in-sync 副本数
kafka_topic_partition_leader	主题分区的领导代理 ID
kafka_topic_partition_leader_is_preferred	如果主题分区正在使用首选代理，显示 **1**
kafka_topic_partition_replicas	本主题分区的副本数
kafka_topic_partition_under_replicated_partition	如果主题分区复制不足，显示 **1**。

表 17.3. 消费者组指标输出

Name	信息
kafka_consumergroup_current_offset	消费者组的当前主题分区偏移
kafka_consumergroup_lag	主题分区中消费者组的当前大约滞后

17.4. 运行 KAFKA EXPORTER

运行 Kafka Exporter，在 Grafana 仪表板中公开 Prometheus 指标。

下载并安装 Kafka Exporter 软件包，以使用带有 Apache Kafka 的 Streams 的 Kafka Exporter。您需要一个 Streams for Apache Kafka 订阅才能下载并安装软件包。

先决条件

- 每个主机上安装了 Apache Kafka 的流，且配置文件可用。
- 您有一个 Apache Kafka 订阅。

此流程假设您已经访问 Grafana 用户界面，并添加了 Prometheus 作为数据源。

流程

1. 安装 Kafka Exporter 软件包：

```
dnf install kafka_exporter
```

2. 验证软件包是否已安装：

```
dnf info kafka_exporter
```

3. 使用适当的配置参数值运行 Kafka Exporter：

```
kafka_exporter --kafka.server=<kafka_bootstrap_address>:9092 --kafka.version=3.7.0
--<my_other_parameters>
```

参数需要双假设惯例，如 **--kafka.server**。

表 17.4. Kafka Exporter 配置参数

选项	描述	默认
kafka.server	Kafka 服务器的 host/post 地址。	**kafka:9092**
kafka.version	Kafka 代理版本。	**1.0.0**
group.filter	指定指标中包含的消费者组的正则表达式。	**adtrust** (all)

选项	描述	默认
topic.filter	指定指标中包含的主题的正则表达式。	**adtrust** (all)
sasl.<*parameter*>	使用 SASL/PLAIN 身份验证启用和连接到 Kafka 集群的参数，使用用户名和密码。	**false**
tls.<*parameter*>	启用使用 TLS 身份验证连接到 Kafka 集群的参数，以及可选的证书和密钥。	**false**
web.listen-address	公开指标的端口地址。	**:9308**
web.telemetry-path	公开指标的路径。	**/metrics**
log.level	日志记录配置，以记录给定严重性(debug、info、warn、error、fatal)或更高严重性的消息。	**info**
log.enable-sarama	启用 Sarama 日志记录的布尔值，这是 Kafka Exporter 使用的 Go 客户端库。	**false**
legacy.partitions	布尔值，以启用从非活动主题分区以及活跃的分区中获取指标。如果您希望 Kafka Exporter 返回不活跃分区的指标，设置为 **true**。	**false**

您可以使用 **kafka_exporter --help** 来获取有关属性的信息。

4. 配置 Prometheus 以监控 Kafka 导出器指标。
有关配置 Prometheus 的更多信息，请参阅 Prometheus 文档。

5. 启用 Grafana 以显示 Prometheus 公开的 Kafka 导出器指标数据。
如需更多信息，请参阅在 Grafana 中呈现 Kafka Exporter 指标。

更新 Kafka Exporter

使用带有 Apache Kafka 安装的 Streams 的 Kafka Exporter 的最新版本。

要检查更新，请使用：

```
dnf check-update
```

要更新 Kafka Exporter，请使用：

```
dnf update kafka_exporter
```

17.5. 在 GRAFANA 中显示 KAFKA EXPORTER 指标

使用 Kafka Exporter Prometheus 指标作为数据源，您可以创建一个 Grafana chart 的仪表板。

例如，在指标数据中，您可以创建以下 Grafana chart：

- 每秒的消息（来自主题）
- 每分钟的消息（来自主题）
- 消费者组滞后
- 每分钟消耗的消息（按消费者组）

当收集指标数据一段时间时，Kafka Exporter chart 会被填充。

使用 Grafana chart 分析滞后，检查操作是否对受影响的消费者组产生影响。例如，如果对 Kafka 代理进行了调整以减少滞后，仪表板将显示 *Lag by consumer group* 图表下降，*Messages consumed per minute* 图表增加。

其他资源

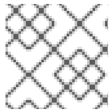
- Kafka Exporter 的仪表板示例
- Grafana 文档

第 18 章 升级 APACHE KAFKA 和 KAFKA 的流

在不停机的情况下升级 Kafka 集群。Apache Kafka 2.7 的流支持并使用 Apache Kafka 版本 3.7.0。Kafka 3.6.0 仅支持升级到 Apache Kafka 2.7 的 Streams 的目的。当您安装 Apache Kafka 的最新版本流时，您可以升级到最新的 Kafka 版本。

18.1. 升级先决条件

在开始升级过程前，请确定您熟悉 Red Hat Enterprise Linux 发行注记中 Apache Kafka 2.7 的 Streams for Apache Kafka 2.7 中描述的任何升级更改。



注意

有关如何升级到该版本的信息，请参阅支持 Apache Kafka 特定版本的文档。

18.2. 更新 KAFKA 版本

当使用 ZooKeeper 进行集群管理时，在 **Kafka** 资源的配置中需要更新 Kafka 版本 (**Kafka.spec.kafka.version**)及其 inter-broker 协议版本(**inter.broker.protocol.version**)。Kafka 的每个版本都有了一个内部代理协议的兼容版本。inter-broker 协议用于代理间通信。协议的次要版本通常会增加以匹配 Kafka 的次要版本，如上表中所示。inter-broker 协议版本在 **Kafka** 资源中设置 cluster wide。要更改它，您可以编辑 **Kafka.spec.kafka.config** 中的 **inter.broker.protocol.version** 属性。

下表显示了 Kafka 版本之间的区别：

表 18.1. Kafka 版本的不同

Apache Kafka 版本流	Kafka 版本	inter-broker 协议版本	日志消息格式版本	ZooKeeper 版本
2.7	3.7.0	3.7	3.7	3.8.3
2.6	3.6.0	3.6	3.6	3.8.3

- Kafka 3.7.0 支持在生产环境中使用。
- Kafka 3.6.0 仅支持升级到 Apache Kafka 2.7 的 Streams 的目的。

日志消息格式版本

当制作者发送消息到 Kafka 代理时，消息会使用特定的格式进行编码。格式可能会在 Kafka 发行版本之间改变，因此消息指定它们编码的消息格式版本。

用于设置特定消息格式版本的属性如下：

- 主题的 **message.format.version** 属性
- Kafka 代理的 **log.message.format.version** 属性

从 Kafka 3.0.0，消息格式版本值被假定为与 **inter.broker.protocol.version** 匹配，且不需要设置。该值反映了使用的 Kafka 版本。

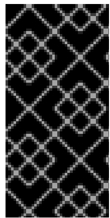
当升级到 Kafka 3.0.0 或更高版本时，您可以在更新 **inter.broker.protocol.version** 时删除这些设置。否则，您可以根据您要升级到的 Kafka 版本设置消息格式版本。

由在 Kafka 代理中设置的 **log.message.format.version** 定义的 **message.format.version** 的默认值。您可以通过修改主题配置来手动设置主题的 **message.format.version**。

Kafka 版本的滚动更新更改

当 Kafka 版本被更新时，Cluster Operator 会启动对 Kafka 代理的滚动更新。进一步的滚动更新依赖于 **inter.broker.protocol.version** 和 **log.message.format.version** 的配置。

如果 **Kafka.spec.kafka.config** 包含…	Cluster Operator 启动…
inter.broker.protocol.version 和 **log.message.format.version**。	单个滚动更新。更新后，必须手动更新 **inter.broker.protocol.version**，后跟 **log.message.format.version**。更改每个将触发进一步的滚动更新。
inter.broker.protocol.version 或 **log.message.format.version**。	两个滚动更新。
没有 **inter.broker.protocol.version** 或 **log.message.format.version** 的配置。	两个滚动更新。



重要

从 Kafka 3.0.0，当 **inter.broker.protocol.version** 设置为 **3.0** 或更高版本时，**log.message.format.version** 选项会被忽略，且不需要设置。代理的 **log.message.format.version** 属性和主题的 **message.format.version** 属性已弃用，并将在以后的 Kafka 发行版本中删除。

作为 Kafka 升级的一部分，Cluster Operator 为 ZooKeeper 启动滚动更新。

- 即使 ZooKeeper 版本没有改变，也会进行单个滚动更新。
- 如果 Kafka 的新版本需要新的 ZooKeeper 版本，则会进行额外的滚动更新。

18.3. 升级客户端的策略

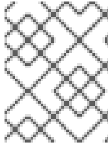
升级 Kafka 客户端可确保它们从新版本的 Kafka 中引入的功能、修复和改进中受益。升级的客户端保持与其他升级的 Kafka 组件的兼容性。客户端的性能和稳定性也可能有所改进。

考虑升级 Kafka 客户端和代理的最佳方法，以确保平稳过渡。所选升级策略取决于您是否首先升级代理或客户端。从 Kafka 3.0 开始，您可以以任何顺序独立和升级代理和客户端。升级客户端或代理的决定首先取决于几个因素，如需要升级的应用程序数量以及可以容忍的停机时间。

如果您在代理前升级客户端，一些新功能可能无法正常工作，因为它们还没有被代理支持。但是，代理可以处理使用不同版本运行的生产者和消费者，并支持不同的日志消息版本。

18.4. 升级 KAFKA 代理和 ZOOKEEPER

在主机机器上升级 Kafka 代理和 ZooKeeper，以使用最新版本的 Apache Kafka。您可以更新安装文件，然后配置并重启所有 Kafka 代理以使用新的 inter-broker 协议版本。执行这些步骤后，数据会使用新的 inter-broker 协议版本在 Kafka 代理之间传输。



注意

从 Kafka 3.0.0 开始，消息格式版本值被假定为与 **inter.broker.protocol.version** 匹配，不需要设置。该值反映了使用的 Kafka 版本。

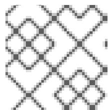
先决条件

- 以 **kafka** 用户身份登录 Red Hat Enterprise Linux。
- 您已在单独的主机上安装 Kafka 和其他 Kafka 组件。
更多信息请参阅 第 3.1 节 “安装环境”。
- 您已下载了 安装文件。

流程

对于 Apache Kafka 集群的流中的每个 Kafka 代理，一次一个：

1. 从 Apache Kafka 软件下载页面 的 Streams for Apache Kafka 归档下载。



注意

如有提示，登录到您的红帽帐户。

2. 在命令行中，创建一个临时目录并提取 **amq-streams-<version>-bin.zip** 文件的内容。

```
mkdir /tmp/kafka
unzip amq-streams-<version>-bin.zip -d /tmp/kafka
```

3. 如果运行，停止 ZooKeeper 和主机上运行的 Kafka 代理。

```
/opt/kafka/bin/zookeeper-server-stop.sh
/opt/kafka/bin/kafka-server-stop.sh
jcmd | grep zookeeper
jcmd | grep kafka
```

如果您在多节点集群中运行 Kafka，请参阅 第 4.3 节 “执行 Kafka 代理的安全滚动重启”。

4. 从现有安装中删除 **libs** 和 **bin** 目录：

```
rm -rf /opt/kafka/libs /opt/kafka/bin
```

5. 从临时目录中复制 **libs** 和 **bin** 目录：

```
cp -r /tmp/kafka/kafka_<version>/libs /opt/kafka/
cp -r /tmp/kafka/kafka_<version>/bin /opt/kafka/
```

6. 如果需要，更新 **config** 目录中的配置文件以反映新版本中的任何更改。

7. 删除临时目录。

```
rm -r /tmp/kafka
```

8. 编辑 /opt/kafka/config/server.properties 属性文件。
   将 inter.broker.protocol.version 和 log.message.format.version 属性设置为*当前*版本。

   例如，如果从 Kafka 版本 3.6.0 升级到 3.7.0，则当前版本是 3.6。

   ```
   inter.broker.protocol.version=3.6
   log.message.format.version=3.6
   ```

   为您要从中升级的 Kafka 版本(3.5、3.6 等)使用正确的版本。将 inter.broker.protocol.version 保持不变在当前设置中，可确保代理可以在升级过程中继续相互通信。

   如果没有配置属性，请使用当前版本添加它们。

   如果您要从 Kafka 3.0.0 或更高版本升级，则只需要设置 inter.broker.protocol.version。

9. 重启更新的 ZooKeeper 和 Kafka 代理：

   ```
   /opt/kafka/bin/zookeeper-server-start.sh -daemon
   /opt/kafka/config/zookeeper.properties
   /opt/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/config/server.properties
   ```

   Kafka 代理和 ZooKeeper 开始为最新的 Kafka 版本使用二进制文件。

   有关在多节点集群中重启代理的详情，请参考 第 4.3 节 "执行 Kafka 代理的安全滚动重启"。

10. 验证重启的 Kafka 代理是否有分区副本。
    使用 kafka-topics.sh 工具来确保代理中包含的所有副本都重新同步。具体步骤请参阅 列出和描述主题。

    在后续步骤中，更新您的 Kafka 代理以使用新的 inter-broker 协议版本。

    一次更新每个代理。

警告

完成以下步骤后，无法降级 Apache Kafka 的流。

11. 根据您选择的 策略来升级客户端，升级所有客户端应用程序以使用客户端二进制文件的新版本。

12. 在 /opt/kafka/config/server.properties 文件中，将 inter.broker.protocol.version 属性设置为 3.7 ：

    ```
    inter.broker.protocol.version=3.7
    ```

13. 在命令行中停止您修改的 Kafka 代理：

    ```
    /opt/kafka/bin/kafka-server-stop.sh
    ```

14. 检查 Kafka 没有运行：

```
jcmd | grep kafka
```

15. 重启您修改的 Kafka 代理：

```
/opt/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/config/server.properties
```

16. 检查 Kafka 是否正在运行：

```
jcmd | grep kafka
```

17. 如果您要从 Kafka 3.0.0 之前的版本升级，请在 **/opt/kafka/config/server.properties** 文件中将 **log.message.format.version** 属性设置为 **3.7** ：

```
log.message.format.version=3.7
```

18. 在命令行中停止您修改的 Kafka 代理：

```
/opt/kafka/bin/kafka-server-stop.sh
```

19. 检查 Kafka 没有运行：

```
jcmd | grep kafka
```

20. 重启您修改的 Kafka 代理：

```
/opt/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/config/server.properties
```

21. 检查 Kafka 是否正在运行：

```
jcmd | grep kafka
```

22. 验证重启的 Kafka 代理是否有分区副本。
使用 **kafka-topics.sh** 工具来确保代理中包含的所有副本都重新同步。具体步骤请参阅 列出和描述主题。

23. 如果升级使用了它，请从 **server.properties** 文件中删除旧的 **log.message.format.version** 配置。

18.5. 升级 KAFKA 组件

在主机机器上升级 Kafka 组件以使用 Apache Kafka 的最新版本的 Streams。您可以使用 Apache Kafka 安装文件的 Streams 来升级以下组件：

- Kafka Connect
- MirrorMaker
- Kafka Bridge （区分 ZIP 文件）

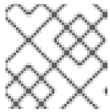
先决条件

- 以 kafka 用户身份登录 Red Hat Enterprise Linux。
- 您已下载了 安装文件。
- 您已 升级 Kafka。
  如果 Kafka 组件在与 Kafka 相同的主机上运行，则在升级时还需要停止并启动 Kafka。

流程

对于运行 Kafka 组件实例的每个主机：

1. 从 Streams for Apache Kafka 软件下载页面 下载 Apache Kafka 或 Kafka Bridge 安装文件。

   注意

   如有提示，登录到您的红帽帐户。

2. 在命令行中，创建一个临时目录并提取 amq-streams-<version>-bin.zip 文件的内容。

   ```
   mkdir /tmp/kafka
   unzip amq-streams-<version>-bin.zip -d /tmp/kafka
   ```

   对于 Kafka Bridge，提取 amq-streams-<version>-bridge-bin.zip 文件。

3. 如果运行，请停止主机上运行的 Kafka 组件。
4. 从现有安装中删除 libs 和 bin 目录：

   ```
   rm -rf /opt/kafka/libs /opt/kafka/bin
   ```

5. 从临时目录中复制 libs 和 bin 目录：

   ```
   cp -r /tmp/kafka/kafka_<version>/libs /opt/kafka/
   cp -r /tmp/kafka/kafka_<version>/bin /opt/kafka/
   ```

6. 如果需要，更新 config 目录中的配置文件以反映新版本中的任何更改。
7. 删除临时目录。

   ```
   rm -r /tmp/kafka
   ```

8. 使用适当的脚本和属性文件启动 Kafka 组件。

   以独立模式启动 Kafka 连接

   ```
   /opt/kafka/bin/connect-standalone.sh \
   /opt/kafka/config/connect-standalone.properties <connector1>.properties
   [<connector2>.properties ...]
   ```

   以分布式模式启动 Kafka 连接

   ```
   /opt/kafka/bin/connect-distributed.sh \
   /opt/kafka/config/connect-distributed.properties
   ```

以专用模式启动 MirrorMaker 2

```
/opt/kafka/bin/connect-mirror-maker.sh \
/opt/kafka/config/connect-mirror-maker.properties
```

启动 Kafka Bridge

```
su - kafka
./bin/kafka_bridge_run.sh \
--config-file=<path>/application.properties
```

9. 验证 Kafka 组件是否正在运行，并按预期生成或消耗数据。

在独立模式中验证 Kafka 连接正在运行

```
jcmd | grep ConnectStandalone
```

在分布式模式中验证 Kafka 连接正在运行

```
jcmd | grep ConnectDistributed
```

在专用模式中验证 MirrorMaker 2 正在运行

```
jcmd | grep mirrorMaker
```

通过检查日志来验证 Kafka Bridge 正在运行

```
HTTP-Kafka Bridge started and listening on port 8080
HTTP-Kafka Bridge bootstrap servers localhost:9092
```

第 19 章 使用 JMX 监控集群

收集指标对于了解 Kafka 部署的健康状态和性能至关重要。通过监控指标，您可以在问题变得至关重要前主动识别问题，并根据资源分配和容量规划做出明智的决策。如果没有指标，您可能会对 Kafka 部署的行为有有限的可见性，这有助于进行故障排除。设置指标可节省长时间运行的时间和资源，并帮助确保 Kafka 部署的可靠性。

Kafka 组件使用 Java 管理扩展(JMX)通过指标共享管理信息。这些指标对于监控 Kafka 集群的性能和总体健康状况至关重要。与许多其他 Java 应用程序一样，Kafka 使用 Managed Beans (MBeans)提供指标数据以监控工具和仪表板。JMX 在 JVM 级别上运行，允许外部工具从 Kafka 组件连接和检索管理信息。要连接到 JVM，这些工具通常需要在同一台计算机上运行，并且默认具有相同的用户特权。

19.1. 启用 JMX 代理

使用 JVM 系统属性启用对 Kafka 组件的 JMX 监控。使用 **KAFKA_JMX_OPTS** 环境变量设置启用 JMX 监控所需的 JMX 系统属性。运行 Kafka 组件的脚本使用这些属性。

流程

1. 使用 JMX 属性设置 **KAFKA_JMX_OPTS** 环境变量以启用 JMX 监控。

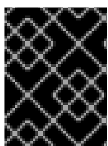
   ```
   export KAFKA_JMX_OPTS=-Dcom.sun.management.jmxremote=true
     -Dcom.sun.management.jmxremote.port=<port>
     -Dcom.sun.management.jmxremote.authenticate=false
     -Dcom.sun.management.jmxremote.ssl=false
   ```

   将 <port> 替换为您要 Kafka 组件侦听 JMX 连接的端口名称。

2. 将 **org.apache.kafka.common.metrics.JmxReporter** 添加到 **server.properties** 文件中的 **metric.reporters**。

   ```
   metric.reporters=org.apache.kafka.common.metrics.JmxReporter
   ```

3. 使用适当的脚本启动 Kafka 组件，如 **bin/kafka-server-start.sh** 用于代理，或 **bin/connect-distributed.sh** 用于 Kafka Connect。



重要

建议您配置身份验证和 SSL 来保护远程 JMX 连接。有关执行此操作所需的系统属性的更多信息，请参阅 Oracle 文档。

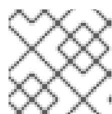
19.2. 禁用 JMX 代理

通过更新 **KAFKA_JMX_OPTS** 环境变量来禁用 Kafka 组件的 JMX 监控。

流程

1. 设置 **KAFKA_JMX_OPTS** 环境变量以禁用 JMX 监控。

   ```
   export KAFKA_JMX_OPTS=-Dcom.sun.management.jmxremote=false
   ```

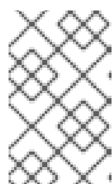


注意

禁用 JMX 监控时不需要指定其他 JMX 属性，如端口、身份验证和 SSL 属性。

2. 在 Kafka **server.properties** 文件中，将 **auto.include.jmx.reporter** 设置为 **false**。

```
auto.include.jmx.reporter=false
```



注意

auto.include.jmx.reporter 属性已弃用。从 Kafka 4 中，只有 **org.apache.kafka.common.metrics.JmxReporter** 添加到属性文件中的 **metric.reporters** 配置中时，才会启用 JMXReporter。

3. 使用适当的脚本启动 Kafka 组件，如 **bin/kafka-server-start.sh** 用于代理，或 **bin/connect-distributed.sh** 用于 Kafka Connect。

19.3. 指标命名约定

在使用 Kafka JMX 指标时，务必要了解用于识别和检索特定指标的命名约定。Kafka JMX 指标使用以下格式：

指标格式

```
<metric_group>:type=<type_name>,name=<metric_name><other_attribute>=<value>
```

- <metric_group> 是指标组的名称
- <type_name> 是指标类型的名称
- <metric_name> 是特定指标的名称
- <other_attribute> 代表零个或多个附加属性

例如，**BytesInPerSec** 指标是 **kafka.server** 组中的 **BrokerTopicMetrics** 类型：

```
kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec
```

在某些情况下，指标可能包含实体的 ID。例如，当监控特定客户端时，指标格式包括客户端 ID：

特定客户端的指标

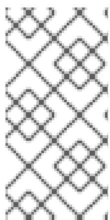
```
kafka.consumer:type=consumer-fetch-manager-metrics,client-id=<client_id>
```

同样，指标也可以进一步缩小到特定客户端和主题：

特定客户端和主题的指标

```
kafka.consumer:type=consumer-fetch-manager-metrics,client-id=<client_id>,topic=<topic_id>
```

了解这些命名约定将允许您准确指定您要监控和分析的指标。



注意

要查看 Strimzi 安装的可用 JMX 指标的完整列表，您可以使用 JConsole 等图形工具。JConsole 是一个 Java 监控和管理控制台，允许您监控和管理 Java 应用程序，包括 Kafka。通过使用其进程 ID 连接到运行 Kafka 组件的 JVM，工具的用户界面允许您查看指标列表。

19.4. 分析 KAFKA JMX 指标以进行故障排除

JMX 提供了收集有关 Kafka 代理的指标，以监控和管理其性能和资源使用情况。通过分析这些指标，可以诊断和解决高 CPU 用量、内存泄漏、线程争用以及响应时间较慢的代理问题。某些指标可以找出这些问题的根本原因。

JMX 指标还深入了解 Kafka 集群的整体健康状况和性能。它们有助于监控系统吞吐量、延迟和可用性、诊断问题并优化性能。本节探索使用 JMX 指标来帮助识别常见问题，并深入了解 Kafka 集群的性能。

使用 Prometheus 和 Grafana 等工具收集和绘制这些指标，您可以视觉化返回的信息。这在检测问题或优化性能方面特别有用。随着时间的推移图表指标还可帮助识别趋势和预测资源消耗。

19.4.1. 检查已复制分区

对于最佳性能，平衡 Kafka 集群非常重要。在均衡集群中，分区和领导层在所有代理中平均分布，I/O 指标则反映了这一点。另外，您还可以使用 **kafka-topics.sh** 工具获得复制分区列表并识别有问题的代理。如果复制分区的数量波动或多个代理显示高请求延迟，这通常表示集群中需要调查的性能问题。另一方面，集群中许多代理报告的非复制分区数量通常表示集群中的其中一个代理处于离线状态。

使用 **kafka-topics.sh** 工具中的 **describe --under-replicated-partitions** 选项来显示集群中当前复制的分区的信息。这些是副本比配置的复制因素少的分区。

如果输出为空，则 Kafka 集群没有复制的分区。否则，输出显示没有同步或可用的副本。

在以下示例中，每个分区只有 3 个中的 2 个副本同步，ISR（同步副本）中缺少一个副本。

从命令行返回重复分区的信息

```
bin/kafka-topics.sh --bootstrap-server :9092 --describe --under-replicated-partitions

Topic: topic-1 Partition: 0 Leader: 4 Replicas: 4,2,3 Isr: 4,3
Topic: topic-1 Partition: 1 Leader: 3 Replicas: 2,3,4 Isr: 3,4
Topic: topic-1 Partition: 2 Leader: 3 Replicas: 3,4,2 Isr: 3,4
```

以下是检查 I/O 和重复分区的一些指标：

检查重复分区的指标

```
kafka.server:type=ReplicaManager,name=PartitionCount (1)
kafka.server:type=ReplicaManager,name=LeaderCount (2)
kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec (3)
kafka.server:type=BrokerTopicMetrics,name=BytesOutPerSec (4)
kafka.server:type=ReplicaManager,name=UnderReplicatedPartitions (5)
kafka.server:type=ReplicaManager,name=UnderMinIsrPartitionCount (6)
```

(1) 集群中所有主题的分区总数。

2. 集群中所有主题的领导总数。
3. 每个代理每秒传入的字节数。
4. 每个代理每秒传出字节数。
5. 集群中所有主题的复制分区数量。
6. 最小 ISR 下的分区数量。

如果为高可用性设置了主题配置，则主题至少为 3 的复制因素，最小同步副本的数量小于复制因素 1 个，没有复制的分区仍然可以使用。相反，低于最低 ISR 的分区会减少可用性。您可以使用 **kafka-topics.sh** 工具中的 **kafka.server:type=ReplicaManager,name=UnderMinIsrPartitionCount** 指标和 **under-min-isr-partitions** 选项来监控它们。

提示

使用 Cruise Control 自动监控和重新平衡 Kafka 集群的任务，以确保分区负载均匀分布。如需更多信息，请参阅 第 13 章 *使用 Cruise Control 进行集群重新平衡*。

19.4.2. 识别 Kafka 集群中的性能问题

集群指标中的激增可能代表一个代理问题，这通常与存储设备缓慢或故障的其他进程中的计算 restraint 相关。如果操作系统或硬件级别没有问题，则 Kafka 集群的负载是不平衡，一些分区因为与同一 Kafka 主题中的其他流量相比，有些分区接收分散流量。

要预测 Kafka 集群中的性能问题，监控 **RequestHandlerAvgIdlePercent** 指标非常有用。**RequestHandlerAvgIdlePercent** 提供有关集群行为的良好总体指示器。此指标的值介于 0 到 1 之间。下面的值 0.7 表示线程在时间忙碌的 30%，性能开始降级。如果值低于 50%，则可能会出现问题，特别是在集群需要扩展或重新平衡时。在 30% 时，集群很少可用。

另一个有用的指标是 **kafka.network:type=Processor,name=IdlePercent**，您可以使用它来监控 Kafka 集群中网络处理器闲置的扩展（作为百分比）。指标有助于识别处理器是否已超过或未被充分利用。

为确保最佳性能，请将 **num.io.threads** 属性等于系统中处理器数量，包括超线程处理器。如果集群是均衡的，但单个客户端更改了其请求模式，并导致问题，减少集群的负载或增加代理数量。

务必要注意，在单个代理上的单个磁盘故障可能会严重影响整个集群的性能。由于生成者客户端连接到所有导致主题的分区的代理，并且这些分区在整个集群中平均分布，因此执行代理会减慢生成请求的速度，并导致生成者中的压力降低到所有代理。RAID（冗余磁盘阵列）存储配置将多个物理磁盘驱动器合并到一个逻辑单元中有助于防止这个问题。

以下是检查 Kafka 集群性能的一些指标：

检查 Kafka 集群性能的指标

```
kafka.server:type=KafkaRequestHandlerPool,name=RequestHandlerAvgIdlePercent (1)
# attributes: OneMinuteRate, FifteenMinuteRate
kafka.server:type=socket-server-metrics,listener=([-.\w]+),networkProcessor=([\d]+) (2)
# attributes: connection-creation-rate
kafka.network:type=RequestChannel,name=RequestQueueSize (3)
kafka.network:type=RequestChannel,name=ResponseQueueSize (4)
```

```
kafka.network:type=Processor,name=IdlePercent,networkProcessor=([-.\w]+) (5)
kafka.server:type=KafkaServer,name=TotalDiskReadBytes (6)
kafka.server:type=KafkaServer,name=TotalDiskWriteBytes (7)
```

1. Kafka 代理线程池中的请求处理器线程的平均空闲百分比。OneMinuteRate 和 FifteenMinuteRate 属性分别显示最后一分钟和十五分钟的请求率。
2. 在 Kafka 代理的特定网络处理器中创建新连接的速率。listener 属性引用侦听器的名称，networkProcessor 属性则引用网络处理器的 ID。connection-creation-rate 属性显示每秒连接创建的速度。
3. 请求队列的当前大小。
4. 响应队列的当前大小。
5. 指定网络处理器闲置的时间百分比。networkProcessor 指定要监控的网络处理器的 ID。
6. Kafka 服务器从磁盘读取的字节数。
7. Kafka 服务器写入磁盘的字节数。

19.4.3. 识别 Kafka 控制器中的性能问题

Kafka 控制器负责管理集群的整体状态，如代理注册、分区重新分配和主题管理。Kafka 集群中控制器的问题很难诊断，通常属于 Kafka 本身中的错误类别。当代理看起来正常或主题创建没有正确发生时，控制器问题可能会作为代理元数据没有同步、离线副本。

没有多种方法来监控控制器，但您可以监控活跃的控制器计数和控制器队列大小。如果出现问题，监控这些指标会给出一个高级别的指示器。虽然队列大小中的激增是预期的，但如果这个值持续增加，或者保持稳定值且没有丢弃，但它表示控制器可能会卡住。如果您遇到这个问题，您可以将控制器移到不同的代理中，这需要关闭当前控制器的代理。

以下是检查 Kafka 控制器性能的一些指标：

检查 Kafka 控制器性能的指标

```
kafka.controller:type=KafkaController,name=ActiveControllerCount (1)
kafka.controller:type=KafkaController,name=OfflinePartitionsCount (2)
kafka.controller:type=ControllerEventManager,name=EventQueueSize (3)
```

1. Kafka 集群中的活跃控制器数量。值 1 表示只有一个活跃的控制器，即所需状态。
2. 当前离线的分区数。如果这个值持续增加或保持高值，则控制器可能会出现问题。
3. 控制器中事件队列的大小。事件是控制器必须执行的操作，如创建新主题或将分区移到新代理。如果值持续增加或保持高值，则控制器可能会卡住且无法执行所需的操作。

19.4.4. 识别请求的问题

您可以使用 RequestHandlerAvgIdlePercent 指标来确定请求是否慢。另外，请求指标可以识别哪些特定请求遇到延迟和其他问题。

要有效地监控 Kafka 请求，收集两个关键指标至关重要： count 和 99th percentile 延迟（也称为 tail 延迟）。

count 指标表示在特定时间间隔内处理的请求数。它可让您了解 Kafka 集群处理的请求卷，并帮助识别流量中的 spikes 或丢弃。

99th percentile 延迟测量请求延迟，这是处理请求的时间。它代表了处理 99% 请求的持续时间。但是，它不会提供有关剩余 1% 请求的确切持续时间的信息。换句话说，99th percentile 的延迟指标告知您在一定时间内处理 99% 的请求，剩余的 1% 可能需要更长时间，但这种剩余 1% 的精确持续时间未知。99%ile 的选择主要侧重于大多数请求，并排除可以偏移结果的排除者。

此指标在识别与大多数请求相关的性能问题和瓶颈时特别有用，但不给出少量请求所遇到的最大延迟的完整列表。

通过收集和分析 count 和 99th percentile 延迟指标，您可以了解 Kafka 集群的整体性能和健康状况，以及正在处理请求的延迟。

以下是检查 Kafka 请求性能的一些指标：

检查请求性能的指标

```
# requests: EndTxn, Fetch, FetchConsumer, FetchFollower, FindCoordinator, Heartbeat, InitProducerId,
# JoinGroup, LeaderAndIsr, LeaveGroup, Metadata, Produce, SyncGroup, UpdateMetadata (1)
kafka.network:type=RequestMetrics,name=RequestsPerSec,request=([\w]+) (2)
kafka.network:type=RequestMetrics,name=RequestQueueTimeMs,request=([\w]+) (3)
kafka.network:type=RequestMetrics,name=TotalTimeMs,request=([\w]+) (4)
kafka.network:type=RequestMetrics,name=LocalTimeMs,request=([\w]+) (5)
kafka.network:type=RequestMetrics,name=RemoteTimeMs,request=([\w]+) (6)
kafka.network:type=RequestMetrics,name=ThrottleTimeMs,request=([\w]+) (7)
kafka.network:type=RequestMetrics,name=ResponseQueueTimeMs,request=([\w]+) (8)
kafka.network:type=RequestMetrics,name=ResponseSendTimeMs,request=([\w]+) (9)
# attributes: Count, 99thPercentile (10)
```

1. 请求类型以中断请求指标。
2. Kafka 代理每秒处理的速率。
3. 请求在处理代理请求队列前等待的时间（以毫秒为单位）。
4. 请求完成的总时间（以毫秒为单位），从代理接收到响应回客户端的时间。
5. 请求在本地机器上处理由代理处理的时间（以毫秒为单位）。
6. 请求被集群中的其他代理处理的时间（以毫秒为单位）。
7. 请求被代理节流的时间（毫秒）。当代理决定客户端快速发送太多请求时，会进行节流，需要减慢。
8. 在向客户端发送前，响应在代理的响应队列中等待的时间（以毫秒为单位）。
9. 在代理生成响应后，响应需要发送到客户端的时间（以毫秒为单位）。
10. 对于所有请求指标，Count 和 99thPercentile 属性显示已处理的请求总数，以及最慢的 1% 请求完成的时间。

19.4.5. 使用指标检查客户端的性能

通过分析客户端指标，您可以监控连接到代理的 Kafka 客户端(producers 和 consumers)的性能。这有助于识别代理日志中突出显示的问题，如用户经常启动其消费者组、高请求故障率或频繁断开连接。

以下是检查 Kafka 客户端性能的一些指标：

检查客户端请求性能的指标

```
kafka.consumer:type=consumer-metrics,client-id=([-.\w]+) (1)
# attributes: time-between-poll-avg, time-between-poll-max
kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+) (2)
# attributes: heartbeat-response-time-max, heartbeat-rate, join-time-max, join-rate, rebalance-rate-per-hour
kafka.producer:type=producer-metrics,client-id=([-.\w]+) (3)
# attributes: buffer-available-bytes, bufferpool-wait-time, request-latency-max, requests-in-flight
# attributes: txn-init-time-ns-total, txn-begin-time-ns-total, txn-send-offsets-time-ns-total, txn-commit-time-ns-total, txn-abort-time-ns-total
# attributes: record-error-total, record-queue-time-avg, record-queue-time-max, record-retry-rate, record-retry-total, record-send-rate, record-send-total
```

1. (consumer)轮询请求之间的平均时间和最长时间，这有助于确定消费者是否经常轮询消息以保持消息流。time-between-poll-avg 和 time-between-poll-max 属性分别显示消费者成功轮询之间的平均和最长时间（以毫秒为单位）。
2. (consumer)指标来监控 Kafka 用户与代理协调器之间的协调过程。属性与心跳、加入和重新平衡过程相关。
3. (producer)指标来监控 Kafka producer 的性能。属性与缓冲区使用量、请求延迟、动态请求、事务处理和记录处理相关。

19.4.6. 使用指标检查主题和分区的性能

主题和分区的指标有助于诊断 Kafka 集群中的问题。当无法收集客户端指标时，您还可以使用它们调试特定客户端的问题。

以下是检查特定主题和分区性能的一些指标：

检查主题和分区性能的指标

```
#Topic metrics
kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec,topic=([-.\w]+) (1)
kafka.server:type=BrokerTopicMetrics,name=BytesOutPerSec,topic=([-.\w]+) (2)
kafka.server:type=BrokerTopicMetrics,name=FailedFetchRequestsPerSec,topic=([-.\w]+) (3)
kafka.server:type=BrokerTopicMetrics,name=FailedProduceRequestsPerSec,topic=([-.\w]+) (4)
kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec,topic=([-.\w]+) (5)
kafka.server:type=BrokerTopicMetrics,name=TotalFetchRequestsPerSec,topic=([-.\w]+) (6)
kafka.server:type=BrokerTopicMetrics,name=TotalProduceRequestsPerSec,topic=([-.\w]+) (7)
#Partition metrics
kafka.log:type=Log,name=Size,topic=([-.\w]+),partition=([\d]+)) (8)
```

```
kafka.log:type=Log,name=NumLogSegments,topic=([-.\w]+),partition=([\d]+)) 9
kafka.log:type=Log,name=LogEndOffset,topic=([-.\w]+),partition=([\d]+)) 10
kafka.log:type=Log,name=LogStartOffset,topic=([-.\w]+),partition=([\d]+)) 11
```

1. 特定主题每秒传入的字节数。
2. 特定主题每秒传出字节数。
3. 特定主题每秒失败的请求率。
4. 生成特定主题每秒失败的请求数。
5. 特定主题每秒的传入消息率。
6. 特定主题的每秒获取请求（成功和失败）的总率。
7. 特定主题的每秒获取请求（成功和失败）的总率。
8. 特定分区的日志的大小（以字节为单位）。
9. 特定分区中的日志片段数。
10. 特定分区日志中最后一个消息的偏移量。
11. 特定分区日志中第一个消息的偏移

其他资源

- Apache Kafka 文档了解 可用指标的完整列表
- Prometheus 文档
- Grafana 文档

附录 A. 使用您的订阅

Apache Kafka 的流通过软件订阅提供。要管理您的订阅，请访问红帽客户门户中的帐户。

访问您的帐户

1. 转至 access.redhat.com。
2. 如果您还没有帐户，请创建一个帐户。
3. 登录到您的帐户。

激活订阅

1. 转至 access.redhat.com。
2. 导航到 My Subscriptions。
3. 导航到 激活订阅 并输入您的 16 位激活号。

下载 Zip 和 Tar 文件

要访问 zip 或 tar 文件，请使用客户门户网站查找下载的相关文件。如果您使用 RPM 软件包，则不需要这一步。

1. 打开浏览器并登录红帽客户门户网站 产品下载页面，网址为 access.redhat.com/downloads。
2. 在 INTEGRATION AND AUTOMATION 目录中找到 Apache Kafka for Apache Kafka的流。
3. 选择 Apache Kafka 产品所需的流。此时会打开 Software Downloads 页面。
4. 单击组件的 Download 链接。

使用 DNF 安装软件包

要安装软件包以及所有软件包的依赖软件包，请使用：

```
dnf install <package_name>
```

要从本地目录中安装之前下载的软件包，请使用：

```
dnf install <path_to_download_package>
```

更新于 2024-04-30