Red Hat Streams for Apache Kafka 2.7

Kafka 配置调整

使用 Kafka 配置属性优化数据流

Last Updated: 2024-05-29

Red Hat Streams for Apache Kafka 2.7 Kafka 配置调整

使用 Kafka 配置属性优化数据流

法律通告

摘要

使用 Kafka 配置属性微调 Kafka 代理、生成者和消费者的操作。

目录

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 KAFKA 调整概述

微调 Kafka 部署的性能涉及根据具体要求优化各种配置属性。本节介绍了 Kafka 代理、生产者和消费者的通用配置选项。

虽然 Kafka 需要一组最小配置才能正常工作，但 Kafka 属性允许进行大量调整。通过配置属性，您可以提高延迟、吞吐量和整体效率，确保 Kafka 部署满足应用程序的需求。

为了进行有效的调整，采用方法论。从分析相关指标开始，识别潜在的瓶颈或区域改进。迭代调整配置参数，监控性能指标的影响，然后相应地优化您的设置。

有关 Apache Kafka 配置属性的更多信息，请参阅 Apache Kafka 文档。

注意

此处提供的指导提供了调整 Kafka 部署的起点。查找最佳配置取决于工作负载、基础架构和性能目标等因素。

1.1. 映射属性和值

如何指定配置属性取决于部署的类型。如果在 OCP 上部署了 Apache Kafka 的 Streams，您可以使用 **Kafka** 资源通过 **config** 属性为 Kafka 代理配置添加 Kafka 代理的配置。使用 RHEL 上的 Apache Kafka 的 Streams，您可以将配置作为环境变量添加到属性文件中。

当您向自定义资源添加 **config** 属性时，您可以使用冒号(':')来映射属性和值。

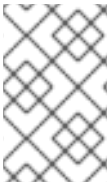
自定义资源中的配置示例

```
num.partitions:1
```

当您将属性添加为环境变量时，您可以使用等号('=')来映射属性和值。

将配置示例作为环境变量

```
num.partitions=1
```



注意

本指南中的一些示例可能会显示 OpenShift 中针对 Apache Kafka 的 Streams 的资源配置。但是，当在 RHEL 上使用 Streams for Apache Kafka 时，显示的属性同样同样适用于环境变量。

1.2. 有助于调整的工具

以下工具有助于 Kafka 调整：

- Cruise Control 生成优化建议，您可以使用它来评估和实施集群重新平衡
- Kafka 静态配额插件在代理上设置限制
- 机架配置将代理分区分散到机架中，并允许使用者从最接近的副本获取数据

其他资源

有关这些工具的更多信息，请参阅以下指南：

- 在 OpenShift 中为 Apache Kafka 配置流
- 以 KRaft 模式在 RHEL 上使用 Streams for Apache Kafka
- 在带有 ZooKeeper 的 RHEL 上使用流 for Apache Kafka

第 2 章 受管代理配置

当您在 OpenShift 上部署 Apache Kafka 的 Streams 时，您可以通过 **Kafka** 自定义资源的 **config** 属性指定代理配置。但是，某些代理配置选项由 Apache Kafka 的 Streams 直接管理。

因此，如果您在 OpenShift 中使用 Streams for Apache Kafka，则无法配置以下选项：

- **broker.id** 指定 Kafka 代理的 ID
- **log.dirs** 目录
- **zookeeper.connect** 配置以将 Kafka 与 ZooKeeper 连接
- 将 Kafka 集群公开给客户端 **的监听程序**
- 允许或拒绝用户执行操作的 **授权机制**
- 证明需要访问 Kafka 的用户的身份 **的身份验证机制**

代理 ID 从 0 开始（零），并与代理副本数对应。日志目录根据 **Kafka** 自定义资源中的 **spec.kafka.storage** 配置挂载到 **/var/lib/kafka/data/kafka-log*IDX***。*IDX* 是 Kafka 代理 pod 索引。

有关排除列表，请参阅 **KafkaClusterSpec** 模式参考。

在 RHEL 中使用流进行 Apache Kafka 时，这些排除不适用。在这种情况下，您需要在基本代理配置中添加这些属性来识别代理并提供安全访问。

RHEL 上 Apache Kafka 的流代理配置示例

```
# ...
broker.id = 1
log.dirs = /var/lib/kafka
zookeeper.connect = zoo1.my-domain.com:2181,zoo2.my-domain.com:2181,zoo3.my-
domain.com:2181
listeners = internal-1://:9092
authorizer.class.name = kafka.security.auth.SimpleAclAuthorizer
ssl.truststore.location = /path/to/truststore.jks
ssl.truststore.password = 123456
ssl.client.auth = required
# ...
```

其他资源

- 在 OCP 中配置 Kafka
- 以 KRaft 模式在 RHEL 上使用 Streams for Apache Kafka
- 在带有 ZooKeeper 的 RHEL 上使用流 for Apache Kafka

第 3 章 KAFKA 代理配置调整

使用配置属性来优化 Kafka 代理的性能。您可以使用标准 Kafka 代理配置选项，但 Apache Kafka 直接管理的属性除外。

3.1. 基本代理配置

典型的代理配置将包括与主题、线程和日志相关的属性的设置。

基本代理配置属性

```
# ...
num.partitions=1
default.replication.factor=3
offsets.topic.replication.factor=3
transaction.state.log.replication.factor=3
transaction.state.log.min.isr=2
log.retention.hours=168
log.segment.bytes=1073741824
log.retention.check.interval.ms=300000
num.network.threads=3
num.io.threads=8
num.recovery.threads.per.data.dir=1
socket.send.buffer.bytes=102400
socket.receive.buffer.bytes=102400
socket.request.max.bytes=104857600
group.initial.rebalance.delay.ms=0
zookeeper.connection.timeout.ms=6000
# ...
```

3.2. 复制主题以实现高可用性

基本主题属性为主题设置默认分区数和复制因素，这些主题将应用于没有显式设置这些属性创建的主题，包括何时自动创建主题。

```
# ...
num.partitions=1
auto.create.topics.enable=false
default.replication.factor=3
min.insync.replicas=2
replica.fetch.max.bytes=1048576
# ...
```

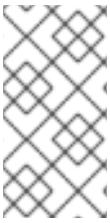
对于高可用性环境，建议将复制因素增加到至少 3 个用于主题，并将最小 in-sync 副本数量设置为复制因素小 1。

auto.create.topics.enable 属性默认为启用，以便在生产者和消费者需要时自动创建不存在的主题。如果使用自动主题创建，您可以使用 **num.partitions** 为主题设置默认分区数量。但是，此属性会被禁用，以便通过显式主题创建为主题提供更多控制。

为了实现数据持久性，需要在 *topic* 配置中设置 **min.insync.replicas**，并在 *producer* 配置中使用 **acks=all** 来发送提交确认。

使用 **replica.fetch.max.bytes** 设置复制领导分区的每个后续程序获取的最大大小（以字节为单位）。根据平均消息大小和吞吐量更改这个值。在考虑读/写缓冲区所需的总内存分配时，可用内存也必须能适应所有后续者的最大复制消息大小。

delete.topic.enable 属性默认为启用，以允许删除主题。在生产环境中，您应该禁用此属性以避免意外删除主题，从而导致数据丢失。但是，您可以临时启用它并删除主题，然后再次禁用它。



注意

在 OpenShift 上运行 Apache Kafka 的 Streams 时，主题 Operator 可以提供 operator 风格的主题管理。您可以使用 **KafkaTopic** 资源来创建主题。对于使用 **KafkaTopic** 资源创建的主题，复制因素是使用 **spec.replicas** 设置的。如果启用了 **delete.topic.enable**，您也可以使用 **KafkaTopic** 资源删除主题。

```
# ...
auto.create.topics.enable=false
delete.topic.enable=true
# ...
```

3.3. 事务和提交的内部主题设置

如果您使用事务 启用对生成者中的分区的原子写入，则事务的状态存储在内部 **__transaction_state** 主题。默认情况下，代理配置有 3 个复制因素，以及此主题的最小同步副本，这意味着 Kafka 集群中最少需要三个代理。

```
# ...
transaction.state.log.replication.factor=3
transaction.state.log.min.isr=2
# ...
```

同样，存储消费者状态的内部 **__consumer_offsets** 主题具有分区和复制因素的数量的默认设置。

```
# ...
offsets.topic.num.partitions=50
offsets.topic.replication.factor=3
# ...
```

不要在生产环境中减少这些设置。在*生产环境中*，您可以提高设置。作为例外，您可能需要减少单代理 *测试环境中的设置*。

3.4. 通过增加 I/O 线程来提高请求处理吞吐量

网络线程处理对 Kafka 集群的请求，如从客户端应用程序生成和获取请求。生成请求将置于请求队列中。响应放置在响应队列中。

每个监听器的网络线程数量应该反映了复制因素以及客户端制作者和与 Kafka 集群交互的用户的活动级别。如果您要拥有大量请求，您可以使用闲置时间线程数量来增加线程数量，以确定何时添加更多线程。

要减少拥塞并规范请求流量，您可以限制请求队列中允许的请求数。当请求队列已满时，所有传入流量都会被阻断。

I/O 线程从请求队列获取请求来处理它们。添加更多线程可以提高吞吐量，但 CPU 内核和磁盘带宽的数量会实施实际的上限。至少，I/O 线程的数量应等于存储卷的数量。

```
# ...
num.network.threads=3 1
queued.max.requests=500 2
num.io.threads=8 3
num.recovery.threads.per.data.dir=4 4
# ...
```

1. Kafka 集群的网络线程数量。
2. 请求队列中允许的请求数。
3. Kafka 代理的 I/O 线程数量。
4. 在启动时用于日志载入的线程数量，并在关闭时清除。尝试设置为至少内核数。

所有代理的线程池的配置更新可能会在集群级别动态发生。这些更新仅限于当前大小的一半和两倍的当前大小。

提示

以下 Kafka 代理指标可帮助处理所需的线程数量：

- **kafka.network:type=SocketServer,name=NetworkProcessorAvgIdlePercent** 提供平均网络线程闲置的指标。
- **kafka.server:type=KafkaRequestHandlerPool,name=RequestHandlerAvgIdlePercent** 提供平均 I/O 线程闲置的指标。

如果有 0% 空闲时间，则使用所有资源，这意味着添加更多线程可能很有用。当闲置时间低于 30% 时，性能可能会开始下降。

如果线程因为磁盘数量而较慢或限制，您可以尝试增加网络请求的缓冲区的大小，以提高吞吐量：

```
# ...
replica.socket.receive.buffer.bytes=65536
# ...
```

另外，增加 Kafka 可以接收的最大字节数：

```
# ...
socket.request.max.bytes=104857600
# ...
```

3.5. 为高延迟连接增加带宽

Kafka 批处理数据，以便在从 Kafka 到客户端（如数据中心之间的连接）上实现合理的吞吐量。但是，如果高延迟问题，您可以增加缓冲区的大小来发送和接收信息。

```
# ...
socket.send.buffer.bytes=1048576
socket.receive.buffer.bytes=1048576
# ...
```

您可以使用 *bandwidth-delay 产品* 计算来估算缓冲区的最佳大小，这会乘以带往返延迟（以秒为单位）链路的最大带宽（以秒为单位）以达到最大吞吐量。

3.6. 使用 DELETE 和 COMPACT 策略管理 KAFKA 日志

Kafka 依赖于日志来存储消息数据。日志由一系列片段组成，每个片段都与基于偏移和基于时间戳的索引相关联。新消息被写入 *活跃* 片段，永远不会被修改。当服务从消费者获取请求时，将读取片段。活跃的片段 *会定期* 变为只读，并创建一个新的活跃网段来替换它。每个代理的每个主题分区只能有一个活跃网段。保留旧的片段，直到它们有资格删除。

在代理级别配置决定了日志片段的最大大小（以字节为单位），以及活跃的片段前的时间（毫秒）：

```
# ...
log.segment.bytes=1073741824
log.roll.ms=604800000
# ...
```

这些设置可以使用 **segment.bytes** 和 **segment.ms** 在主题级别上覆盖。降低或引发这些值的选择取决于删除网段的策略。较大的大小表示活跃片段包含更多信息，且会更频繁地推出。片段还有资格更频繁地删除。

在 Kafka 中，日志清理策略决定如何管理日志数据。在大多数情况下，您不需要更改集群级别的默认配置，它指定了 **删除** 清理策略，并启用 *紧凑* 清理策略使用的日志清理：

```
# ...
log.cleanup.policy=delete
log.cleaner.enable=true
# ...
```

删除清理策略

删除清理策略是所有主题的默认集群范围的策略。该策略应用于没有配置特定主题级别策略的主题。Kafka 根据基于时间的日志保留限制删除旧的片段。

紧*凑*清理策略

紧凑清理策略通常被配置为主题级策略(**cleanup.policy=compact**)。Kafka 的日志清理程序对特定主题应用压缩，只保留主题中键的最新值。您还可以将主题配置为使用这两个策略(**cleanup.policy=compact,delete**)。

为**删除策略**设置保留限制

删除清理策略与管理带有数据保留的日志相对应。当不需要永久保留数据时，该策略是合适的。您可以建立基于时间或基于大小的日志保留和清理策略，以保持绑定的日志。

当使用日志保留策略时，当达到保留限制时，非活跃日志片段会被删除。删除旧片段有助于防止超过磁盘容量。

对于基于时间的日志保留，您可以根据小时、分钟或毫秒设置保留周期：

```
# ...
log.retention.ms=1680000
# ...
```

保留周期基于消息附加到段中的时间。Kafka 使用片段中最新消息的时间戳来确定该片段是否已过期。毫秒配置的优先级超过分钟，其优先级高于小时。默认情况下，分钟和毫秒配置是 null，但三个选项提供了对您要保留数据的大量控制级别。首选项应提供给毫秒配置，因为它是唯一可以动态更新的三个属性之

一。

如果 **log.retention.ms** 设置为 -1，则不会将时间限制应用到日志保留，并且保留所有日志。但是，通常不建议此设置，因为它可能会导致出现难以重新排序的完整磁盘的问题。

对于基于大小的日志保留，您可以指定最小日志大小（以字节为单位）：

```
# ...
log.retention.bytes=1073741824
# ...
```

这意味着 Kafka 将确保至少有指定数量的日志数据可用。

例如，如果您将 **log.retention.bytes** 设置为 1000，并将 **log.segment.bytes** 设为 300，则 Kafka 将保留 4 个片段加活跃片段，确保至少有 1000 字节可用。当活跃片段已满并创建新片段时，会删除最旧的网段。此时，磁盘的大小可能会超过指定的 1000 字节，可能范围介于 1200 到 1500 字节（不包括索引文件）。

使用日志大小的潜在问题是，它不会考虑时间信息被附加到段中。您可以对清理策略使用基于时间和大小的日志保留，以获得您需要的平衡。首先达到哪个阈值会触发清理。

要在从系统中删除分段文件前添加时间延迟，您可以在所有主题的代理级别使用 **log.segment.delete.delay.ms** ：

```
# ...
log.segment.delete.delay.ms=60000
# ...
```

或在主题级别配置 **file.delete.delay.ms**。

您可以设置检查日志进行清理的频率（以毫秒为单位）：

```
# ...
log.retention.check.interval.ms=300000
# ...
```

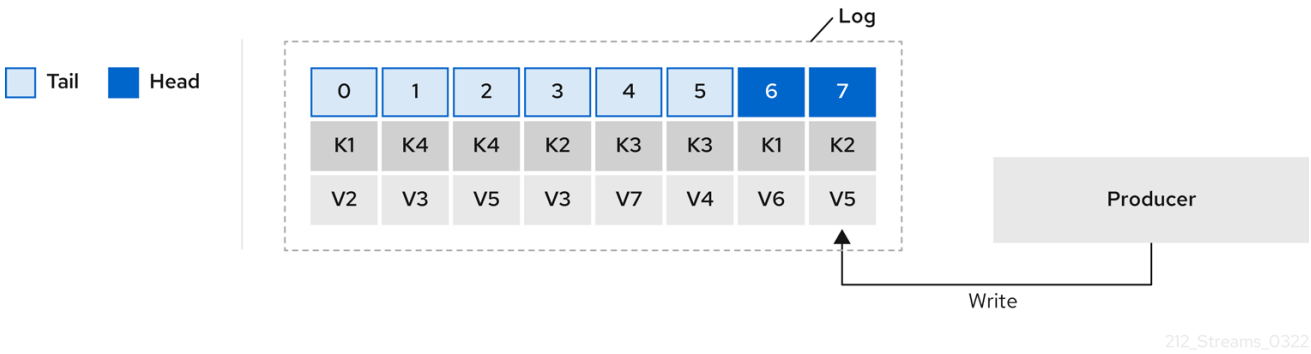
调整与日志保留设置相关的日志保留检查间隔。较小的保留大小可能需要更频繁地检查。清理的频率通常足以管理磁盘空间，但通常不会影响代理的性能。

使用紧凑策略保留最新消息

当您通过设置 **cleanup.policy=compact** 来为主题启用日志压缩时，Kafka 使用日志清理程序作为后台线程来执行压缩。紧凑策略确保保留每个消息键的最新消息，从而有效地清理旧版本的记录。当消息值可改变时，该策略是合适的，您想要保留最新的更新。

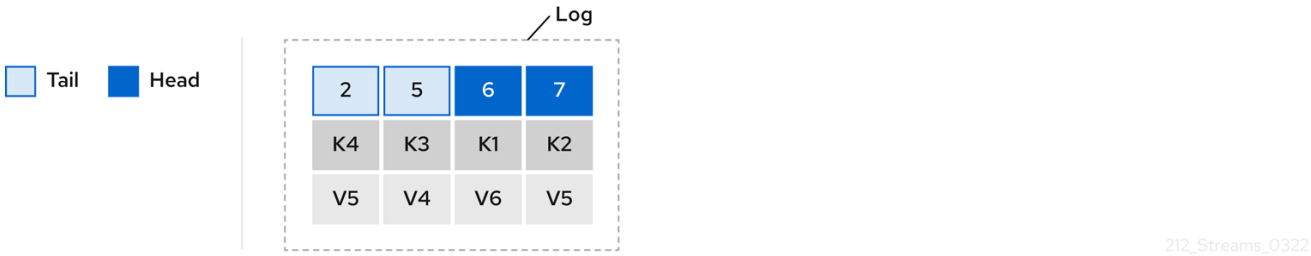
如果为日志压缩设置了清理策略，则日志 *头* 作为标准 Kafka 日志运行，并按顺序写入新消息。在紧凑的日志的 *尾部* 中，日志清理程序运行，如果稍后日志中发生具有相同键的另一个记录，则会删除记录。具有 null 值的消息也会被删除。要使用压缩，您必须有识别相关消息的密钥，因为 Kafka 保证保留每个密钥的最新信息，但它不能保证整个紧凑的日志不会包含重复。

图 3.1. 在压缩前显示带有偏移位置的键值写入的日志



使用键识别信息，Kafka 压缩会保留日志尾部针对特定消息键中存在的最新消息（具有最高偏移），最终丢弃具有相同键的早期消息。其最新状态的消息始终可用，在日志清理程序运行时最终会删除该特定消息的任何过时的记录。您可以将消息恢复到以前的状态。即使周围记录被删除，记录也会保留其原始偏移量。因此，tail 可以具有非连续偏移。当消耗在 tail 中可用偏移时，会发现带有下一个偏移的记录。

图 3.2. 压缩后日志



如果适当，您可以在压缩过程中添加一个延迟：

```
# ...
log.cleaner.delete.retention.ms=86400000
# ...
```

删除的数据保留周期提供了在数据被不可避免删除前注意到数据的时间。

要删除与特定密钥相关的所有消息，制作者可以发送 *tombstone* 消息。tombstone 有一个 null 值，并作为标记来通知消费者删除该键的对应消息。一段时间后，只会保留 tombstone 标记。假设新消息继续进入，标记会保留在 **log.cleaner.delete.retention.ms** 指定的持续时间内，以便消费者有足够的时间识别删除。

如果没有要清理的日志，您还可以设置时间（以毫秒为单位），以将清理器放在待机中：

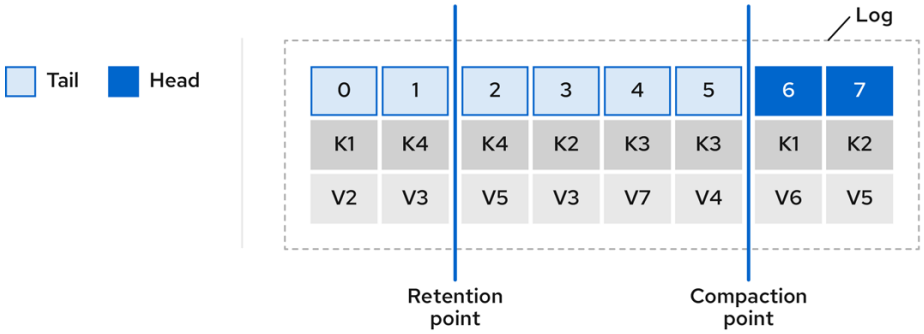
```
# ...
log.cleaner.backoff.ms=15000
# ...
```

使用组合紧凑和删除策略

如果您只选择紧凑策略，您的日志仍然可以变为任意的。在这种情况下，您可以将主题的清理策略设置为紧凑和删除日志。Kafka 应用日志压缩，删除旧版本的记录，并只保留每个密钥的最新版本。Kafka 还根据指定的基于时间的日志保留设置或基于大小的日志保留设置删除记录。

例如，在下图中，特定消息键的最新消息（具有最高偏移量）被保留到压缩点。如果保留点上有任何记录，则它们会被删除。在这种情况下，压缩过程将删除所有重复。

图 3.3. 日志保留点和压缩点



3.7. 管理高效的压缩磁盘利用率

当使用紧凑策略和日志清理器时，在 Kafka 中处理主题日志，请考虑优化内存分配。

您可以使用 deduplication 属性(**dedupe.buffer.size**)微调内存分配，它决定了为所有日志清理任务分配的内存总量。另外，您可以通过通过 **buffer.load.factor** 属性定义百分比来建立最大内存用量限制。

```
# ...
log.cleaner.dedupe.buffer.size=134217728
log.cleaner.io.buffer.load.factor=0.9
# ...
```

每个日志条目都使用 24 字节，以便您可以处理缓冲区可以在单一运行中处理多少个日志条目，并相应地调整设置。

如果要减少日志清理时间，请考虑增加日志清理线程数量：

```
# ...
log.cleaner.threads=8
# ...
```

如果您遇到 100% 磁盘带宽使用情况的问题，您可以节流日志清理 I/O，以便读/写操作的总和小于执行操作的磁盘功能的指定双值：

```
# ...
log.cleaner.io.max.bytes.per.second=1.7976931348623157E308
# ...
```

3.8. 控制消息数据的日志清除

通常，建议不要设置显式清除阈值，并让操作系统使用默认设置执行后台清除。分区复制提供比写入任何单个磁盘更高的数据持久性，因为失败的代理可以从其同步的副本中恢复。

log flush 属性控制缓存的消息数据定期写入磁盘。调度程序以毫秒为单位指定日志缓存上的检查频率：

```
# ...
log.flush.scheduler.interval.ms=2000
# ...
```

您可以根据消息保留在内存中的最长时间以及日志中的最大信息数量来控制清除的频率，然后再写入磁盘：

```
# ...
log.flush.interval.ms=50000
log.flush.interval.messages=100000
# ...
```

flushes 之间的等待时间包括进行检查的时间以及执行刷新前指定的间隔。增加清除的频率可能会影响吞吐量。

如果您使用应用程序清除管理，如果您正在使用更快速的磁盘，则设置较低冲刷阈值可能适当。

3.9. 分区重新平衡以实现可用性

分区可以在代理之间复制，以进行容错。对于给定分区，一个代理被选为 leader，并处理所有生成请求（写入日志）。在领导失败时，在其他代理中，分区会遵循其他代理为数据可靠性复制分区数据。

followers 通常不提供客户端，但 **机架** 配置允许消费者在 Kafka 集群跨越多个数据中心时消耗来自最接近的副本的消息。followers 仅操作从分区领导机复制消息，并允许在领导机失败时进行恢复。恢复需要同步的后续程序。遵循者通过向领导发送获取请求来保持同步，这将按顺序将消息返回到后续消息。如果已在领导消息中发现最近提交的消息，则后续者被视为同步。领导机通过查看后续程序请求的最后一个偏移来检查。不同步的后续者通常不符合领导机失败，除非允许未清理的领导选举机制。

您可以在考虑不同步前调整滞后时间：

```
# ...
replica.lag.time.max.ms=30000
# ...
```

Lag time 对时间设置了一个上限，以将消息复制到所有同步副本以及生成者必须等待确认的时间。如果后续程序无法获取请求，并捕获指定滞后时间的最新消息，则会从同步的副本中删除。您可以更快地减少检测失败的副本的时间，但这样做可能会增加无法同步的后续者数量。正确的滞后时间值取决于网络延迟和代理磁盘带宽。

当领导分区不再可用时，会选择其中一个同步副本作为新的领导。分区副本列表中的第一个代理称为 *首选* 领导。默认情况下，根据定期检查领导发行版，为自动分区领导重新平衡启用 Kafka。也就是说，Kafka 会检查首选领导是否为 *当前的* 领导。重新平衡可确保领导在代理和代理间均匀分布，不会超载。

您可以使用 Apache Kafka 的 Cruise Control for Streams 将副本分配找出在集群中平均平衡负载的代理。其计算需要考虑领导和后续者所经历的不同负载。失败的领导会影响 Kafka 集群的平衡，因为剩余的代理会获得领导额外分区的额外工作。

对于 Cruise Control 发现的分配，实际上，分区需要被首选领导。Kafka 可以自动确保首选领导（在可能的情况下），根据需要更改当前的领导。这样可确保集群处于 Cruise Control 找到的均衡状态。

您可以在重新平衡检查前控制重新平衡检查的频率，以及代理允许的最大 imbalance 百分比。

```
#...
auto.leader.rebalance.enable=true
```

```
leader.imbalance.check.interval.seconds=300
leader.imbalance.per.broker.percentage=10
#...
```

代理的百分比 leader imbalance 是当前代理为当前领导的分区数量和首选领导的分区数量之间的比例。您可以将百分比设置为 0，以确保首选领导一直被选择，假设它们处于同步状态。

如果检查重新平衡需要更多控制，您可以禁用自动重新平衡。然后，您可以选择何时使用 **kafka-leader-election.sh** 命令行工具触发重新平衡。

注意

提供的 Apache Kafka 的 Grafana 仪表板显示复制分区和没有活跃领导的分区的指标。

3.10. UNCLEAN 领导选举机制

对同步副本的领导选举被视为干净，因为它不会丢失数据。这是默认发生的情况。但是，如果没有同步的副本在领导方面需要什么？或许，ISR （同步副本）仅在领导磁盘结束时包含领导机。如果没有设置最少的 in-sync 副本数量，并且硬盘驱动器无法同步分区领导机时，数据将会丢失。不仅如此，*新的领导产品也无法被选举*，因为没有同步的跟随者。

您可以配置 Kafka 如何处理领导失败：

```
# ...
unclean.leader.election.enable=false
# ...
```

默认情况下，不干净的领导选举机制被禁用，这意味着不同步的副本无法成为领导。使用干净的领导选举机制时，如果在旧领导丢失时没有其他代理，则 Kafka 会在消息写入或读取前等待该领导机在线。不干净的领导选举机制意味着不同步的副本可能会成为领导机，但您会面临丢失消息的风险。您做出的选择取决于您的要求是否优先使用可用性或持久性。

您可以在主题级别覆盖特定主题的默认配置。如果您无法承担数据丢失的风险，请保留默认配置。

3.11. 避免不必要的消费者组重新平衡

对于加入新消费者组的用户，您可以添加一个延迟，以避免对代理进行不必要的重新平衡：

```
# ...
group.initial.rebalance.delay.ms=3000
# ...
```

延迟是协调器等待成员加入的时间长度。延迟时间越长，所有成员都将一次加入并避免重新平衡的可能性。但是，延迟也会阻止组消耗到周期结束为止。

第 4 章 KAFKA 消费者配置调整

使用配置属性来优化 Kafka 用户的性能。在调整您的消费者时，您的主要关注将确保他们能够高效地处理数据量。与生成者调优一样，准备好进行增量更改，直到消费者按预期工作。

在调整消费者时，仔细考虑以下方面，因为它们会对其性能和行为有严重影响：

扩展

消费者组通过将负载分布到多个消费者，从而提高可扩展性和吞吐量，从而并行处理消息。主题分区数量决定了您可以实现的最大并行级别，因为一个分区只能分配给消费者组中的一个消费者。

消息排序

如果主题中的绝对排序非常重要，请使用单分区主题。消费者按照提交到代理的顺序在单个分区中观察信息，这意味着 Kafka 只为单一分区中的消息提供排序保证。也可以维护特定于各个实体（如用户）的消息排序。如果创建新实体，您可以创建一个新的主题，专用于该实体。您可以使用唯一 ID，如用户 ID，作为消息密钥，并将具有相同键的所有消息路由到主题内的单个分区。

偏移重置策略

设置适当的偏移策略可确保消费者消耗来自所需起点的消息，并相应地处理消息处理。默认 Kafka reset 值 **是最新的**，它从分区的结尾开始，因此根据消费者的行为和分区状态，可能会丢失一些信息。将 **auto.offset.reset** 设置为 **earliest** 可确保连接新 **group.id** 时，所有消息都会从日志开始检索。

保护访问

通过设置用户帐户来管理对 Kafka 的安全访问，为身份验证、加密和授权实施安全措施。

4.1. 基本消费者配置

每个消费者都需要 connection 和 deserializer 属性。通常，为跟踪添加客户端 ID 是不错的做法。

在消费者配置中，无论后续配置是什么：

- 消费者从给定的偏移中获取，并按顺序消耗消息，除非将偏移更改为跳过或重新读取消息。
- 代理不知道消费者是否处理响应，即使将偏移提交到 Kafka，因为偏移可能会发送到集群中的不同代理。

基本消费者配置属性

```
# ...
bootstrap.servers=localhost:9092 1
key.deserializer=org.apache.kafka.common.serialization.StringDeserializer 2
value.deserializer=org.apache.kafka.common.serialization.StringDeserializer 3
client.id=my-client 4
group.id=my-group-id 5
# ...
```

1. （必需）告诉消费者使用 Kafka 代理的 *host:port* bootstrap 服务器地址连接到 Kafka 集群。消费者使用地址来发现并连接到集群中的所有代理。如果服务器停机，请使用逗号分隔的列表来指定两个或三个地址，但不需要提供集群中所有代理的列表。如果您使用 loadbalancer 服务公开 Kafka 集群，则只需要该服务的地址，因为可用性由 loadbalancer 处理。
2. （必需） Deserializer 将从 Kafka 代理获取的字节数转换为消息密钥。
3. （必需） Deserializer 将从 Kafka 代理获取的字节数转换为消息值。

4 （可选）客户端的逻辑名称，用于日志和指标来识别请求源。id 也可以根据处理时间配额来节流消费者。

5 （条件）消费者 *需要* 组 id 才能加入消费者组。

4.2. 使用消费者组扩展数据消耗

用户组共享一个通常由来自给定主题的一个或多个制作者生成的大型数据流。消费者使用 **group.id** 属性分组，允许消息分散到成员中。组中的一个消费者被选为领导机，决定如何将分区分配给组中的消费者。每个分区只能分配给一个消费者。

如果您还没有作为分区的用户数量，您可以通过添加具有相同 **group.id** 的更多消费者实例来扩展数据消耗。在组中添加比分区更多的消费者将有助于吞吐量，但这意味着在待机上有消费者能够停止工作。如果您可以使用较少的使用者达到吞吐量目标，您可以节省资源。

同一消费者组中的消费者发送偏移提交和心跳到同一代理。消费者向 Kafka 代理发送心跳，以指示其在消费者组中的活动。因此组中消费者的数量越大，代理上的请求负载越高。

```
# ...
group.id=my-group-id 1
# ...
```

1 使用组 ID 将消费者添加到消费者组中。

4.3. 选择正确的分区分配策略

选择适当的分区分配策略，它决定了 Kafka 主题分区如何在组中的消费者实例之间分布。

以下类支持分区策略：

- **org.apache.kafka.clients.consumer.RangeAssignor**
- **org.apache.kafka.clients.consumer.RoundRobinAssignor**
- **org.apache.kafka.clients.consumer.StickyAssignor**
- **org.apache.kafka.clients.consumer.CooperativeStickyAssignor**

使用 **partition.assignment.strategy** 使用者配置属性指定类。**范围** 分配策略为每个消费者分配一组分区，在您要将相关数据一起处理时非常有用。

或者，为消费者间相等的分区分布选择 **循环** 分配策略，这是需要并行处理高吞吐量的情况。

如需更稳定的分区分配，请考虑 **粘性** 和 **合作粘性** 策略。粘性策略旨在在重新平衡过程中维护分配的分区。如果消费者之前分配了某些分区，则粘性策略会在重新平衡后保留具有相同消费者的相同分区，同时只撤销和重新分配实际移至另一个消费者的分区。将分区分配保留原位可减少分区移动的开销。合作粘性策略还支持合作重新平衡，实现来自未重新分配的分区的不间断消耗。

如果没有可用的策略适合您的数据，您可以创建一个根据您的特定要求量身定制的自定义策略。

4.4. 消息排序保证

Kafka 代理从请求代理从主题、分区和偏移位置列表中发送消息的用户接收请求。

消费者按照提交到代理的顺序在单个分区中观察信息，这意味着 Kafka **只** 为单一分区中的消息提供排序保证。相反，如果消费者消耗来自多个分区的消息，则不同分区中消息的顺序与消费者观察到的消息顺序不一定反映了发送它们的顺序。

如果您希望严格排序来自一个主题的消息，请为每个消费者使用一个分区。

4.5. 优化消费者吞吐量和延迟

控制客户端应用程序调用 **KafkaConsumer.poll （）** 时返回的消息数量。

使用 **fetch.max.wait.ms** 和 **fetch.min.bytes** 属性来增加由 Kafka 代理使用者获取的最小数据量。基于时间的批处理使用 **fetch.max.wait.ms** 进行配置，并且基于大小的批处理则使用 **fetch.min.bytes** 来配置。

如果消费者或代理中的 CPU 使用率很高，这可能是因为消费者有太多请求。您可以调整 **fetch.max.wait.ms** 和 **fetch.min.bytes** 属性，以便在较大的批处理中交付较少的请求和信息。通过调整更高，吞吐量会降低延迟。如果生成的数据量较低，您也可以调整更高的数据。

例如，如果您将 **fetch.max.wait.ms** 设置为 500ms，并且 **fetch.min.bytes** 设为 16384 字节，当 Kafka 从消费者收到获取请求时，它将在达到第一个阈值时响应。

相反，您可以调整 **fetch.max.wait.ms** 和 **fetch.min.bytes** 属性，以提高端到端延迟。

```
# ...
fetch.max.wait.ms=500 (1)
fetch.min.bytes=16384 (2)
# ...
```

1. 代理在完成获取请求前等待的时间（毫秒）。默认值为 **500** 毫秒。
2. 如果使用最小批处理大小，则会在达到最小时发送请求，或者已排队消息的时间超过 **fetch.max.wait.ms** （以更早的时间为准）。添加延迟可让批处理将消息递增到批处理大小。

通过增加获取请求大小来降低延迟

使用 **fetch.max.bytes** 和 **max.partition.fetch.bytes** 属性增加消费者从 Kafka 代理获取的最大数据量。

fetch.max.bytes 属性设置一次从代理获取的数据量的最大限制（以字节为单位）。

max.partition.fetch.bytes 会以字节为单位设置每个分区返回的最大限制，每个分区必须始终大于代理或主题配置中设置的 **max.message.bytes** 字节数。

客户端可消耗的最大内存量大约计算为：

```
NUMBER-OF-BROKERS * fetch.max.bytes and NUMBER-OF-PARTITIONS *
max.partition.fetch.bytes
```

如果内存用量可以容纳它，您可以增加这两个属性的值。通过允许每个请求中的更多数据，随着获取请求减少，延迟会得到提高。

```
# ...
fetch.max.bytes=52428800 (1)
max.partition.fetch.bytes=1048576 (2)
# ...
```

1. 获取请求返回的最大数据量（以字节为单位）。
2. 每个分区返回的最大数据量（以字节为单位）。

4.6. 在提交偏移时避免数据丢失或重复

Kafka *auto-commit 机制允许* 使用者自动提交消息的偏移。如果启用，消费者将从轮询代理接收的偏移以 5000ms 间隔提交。

auto-commit 机制比较方便，但它引入了数据丢失和重复的风险。如果消费者获取并转换了很多消息，但系统会在执行 auto-commit 时与消费者缓冲区中处理的消息崩溃，则该数据会丢失。如果系统在处理消息后崩溃，但在执行 auto-commit 前，数据会在重新平衡后在另一个消费者实例上重复。

auto-committing 可以避免仅在下一次轮询到代理或消费者关闭前处理所有消息时数据丢失。

要最小化数据丢失或重复的可能性，您可以将 **enable.auto.commit** 设置为 **false**，并开发您的客户端应用程序来更好地控制提交偏移。或者，您可以使用 **auto.commit.interval.ms** 来减少提交之间的间隔。

```
# ...
enable.auto.commit=false (1)
# ...
```

1. 自动提交设置为 false，以提供更多对提交偏移的控制。

通过将 **enable.auto.commit** 设置为 **false**，您可以在 **执行所有** 处理后提交偏移，并且消息已被使用。例如，您可以设置应用程序来调用 Kafka **commitSync** 和 **commitAsync** 提交 API。

commitSync API 在从轮询返回的消息批处理中提交偏移量。完成批处理中的所有消息后，会调用 API。如果使用 **commitSync** API，则应用程序不会轮询新消息，直到批处理中的最后一个偏移提交为止。如果这对吞吐量造成负面影响，您可以更频繁地提交，也可以使用 **commitAsync** API。**commitAsync** API 不会等待代理响应提交请求，而是在重新平衡时创建更多重复的风险。常见方法是将应用程序中的提交 API 与在关闭消费者或重新平衡前使用的 **commitSync** API 相结合，以确保最终提交成功。

4.6.1. 控制事务消息

考虑在生成者端使用事务 ID 和启用 idempotence (**enable.idempotence=true**)，以保证精确发送一次。然后，您可以使用 **isolation.level** 属性来控制消费者读取事务消息的方式。

isolation.level 属性有两个有效的值：

- **read_committed**
- **read_uncommitted** (default)

使用 **read_committed** 来确保仅由消费者读取提交的事务消息。但是，这会导致端到端延迟增加，因为消费者将无法返回消息，直到代理编写记录事务结果的事务标记(*提交* 或 *中止*)。

```
# ...
enable.auto.commit=false
isolation.level=read_committed (1)
# ...
```

1. 设置 **read_committed**，以便只有提交的消息才会被消费者读取。

4.7. 恢复失败以避免数据丢失

如果消费者组中的故障，Kafka 提供了一个重新平衡协议，用于有效检测和恢复。为最大程度降低这些故障的潜在影响，一个关键策略是调整 **max.poll.records** 属性，以平衡高效处理与系统稳定性。此属性决定了消费者可在单个轮询中获取的最大记录数。微调 **max.poll.records** 有助于维护受控的消耗率，防止消费者不认为自己或 Kafka 代理。

另外，Kafka 还提供高级配置属性，如 **session.timeout.ms** 和 **heartbeat.interval.ms**。这些设置通常为更具体的用例保留，在标准场景中可能不需要调整。

session.timeout.ms 属性指定消费者可以在不向 Kafka 代理发送心跳的情况下可以进入的最大时间，以指示它在消费者组中处于活跃状态。如果消费者无法在会话超时内发送心跳，则被视为不活跃。标记为 inactive 的消费者会触发主题对分区的重新平衡。设置 **session.timeout.ms** 属性值太低，可能会导致误报结果，而设置得太高可能会导致从故障中恢复。

heartbeat.interval.ms 属性决定了消费者向 Kafka 代理发送心跳的频率。连续心跳之间有一个较短的间隔，可以更快地检测消费者故障。heartbeat 间隔必须较低，通常由第三个，而不是会话超时。减少心跳间隔可减少意外重新平衡的几率，但更频繁的心跳会增加代理资源的开销。

4.8. 管理偏移策略

使用 **auto.offset.reset** 属性来控制消费者没有提交偏移时的行为方式，或者提交的偏移不再有效或删除。

假设您首次部署使用者应用，并且从现有的主题读取消息。由于第一次使用 **group.id**，因此 **__consumer_offsets** 主题不包含此应用的任何偏移信息。新应用可以开始处理日志开始的所有现有消息，或者仅处理新消息。默认重置值为 **latest**（从分区末尾开始），因此缺少一些消息。为避免数据丢失，但要增加处理量，将 **auto.offset.reset** 设置为 **earliest** 以在分区的头部开始。

另外，请考虑使用 **最早** 的选项来避免在为代理配置偏移保留周期(**offsets.retention.minutes**)时丢失消息。如果消费者组或独立消费者不活跃，并在保留期间提交没有偏移，则之前提交的偏移将从 **__consumer_offsets** 中删除。

```
# ...
heartbeat.interval.ms=3000 1
session.timeout.ms=45000 2
auto.offset.reset=earliest 3
# ...
```

1. 根据预期的重新平衡调整较低的心跳间隔。
2. 如果在超时时间到期前 Kafka 代理没有接收心跳，则消费者将从消费者组中删除，并启动重新平衡。如果代理配置具有 **group.min.session.timeout.ms** 和 **group.max.session.timeout.ms**，则会话超时值必须在该范围内。
3. 设置为 **earliest** 以返回到分区的起始位置，并在未提交偏移时避免数据丢失。

如果在单个获取请求中返回的数据量较大，则在消费者处理数据前可能会出现超时。在这种情况下，您可以降低 **max.partition.fetch.bytes** 或增加 **session.timeout.ms**。

4.9. 最小化重新平衡的影响

在组中活跃消费者间重新平衡分区是进行以下所需的时间：

- 消费者提交其偏移

- 要形成的新消费者组
- 为组成员分配分区的组领导者
- 组中的消费者，接收其分配并开始获取

重新平衡过程可能会增加服务的停机时间，特别是在消费者组集群滚动重启后重复发生时。

在这种情况下，您可以通过为组中的每个消费者实例分配唯一标识符(**group.instance.id**)来引入 *静态成员资格*。静态成员资格使用持久性，以便在会话超时后重启过程中识别消费者实例。因此，消费者维护其主题分区的分配，从而减少在故障或重启后重新加入组时不必要的重新平衡。

另外，调整 **max.poll.interval.ms** 配置可能会阻止由延长的处理任务导致的重新平衡，允许您指定对新消息的轮询之间的最大间隔。使用 **max.poll.records** 属性在每个轮询期间从消费者缓冲区返回的记录数上限。减少记录数量可让消费者更有效地处理较少的消息。如果冗长的消息处理不可避免，请考虑将此类任务卸载到 worker 线程池。这种并行处理方法可防止因为大量记录的消费者造成延迟和潜在的重新平衡。

```
# ...
group.instance.id=UNIQUE-ID (1)
max.poll.interval.ms=300000 (2)
max.poll.records=500 (3)
# ...
```

1. 唯一的实例 id 确保新的消费者实例接收相同的主题分区分配。
2. 设置间隔以检查消费者正在继续处理消息。
3. 设置从消费者返回的已处理记录的数量。

第 5 章 KAFKA PRODUCER 配置调整

使用配置属性来优化 Kafka producer 的性能。您可以使用标准 Kafka producer 配置选项。调整配置以最大化吞吐量可能会增加延迟，反之亦然。您需要对生成者配置进行试验和调优，以获得所需的平衡。

在配置制作者时，仔细考虑以下方面，因为它们会严重影响其性能和行为：

压缩

通过在网络发送前压缩消息，您可以节省网络带宽并降低磁盘存储要求，但由于压缩和解压缩进程而增加 CPU 使用率增加。

批处理

当生成者发送消息时，调整批处理大小和时间间隔可能会影响吞吐量和延迟。

分区

Kafka 集群中的分区策略可以通过并行和负载平衡支持制作者，其中生产者可以同时写入多个分区，每个分区接收相等的信息共享。其他策略可能包括容错的主题复制。

保护访问

通过设置用户帐户来管理对 Kafka 的安全访问，为身份验证、加密和授权实施安全措施。

5.1. 基本制作者配置

每个制作者都需要 connection 和 serializer 属性。通常，最好为跟踪添加客户端 ID，并使用生成者压缩来减小请求的批处理大小。

在基本制作者配置中：

- 无法保证分区中的消息顺序。
- 到达代理的消息不能保证持久性。

基本制作者配置属性

```
# ...
bootstrap.servers=localhost:9092 1
key.serializer=org.apache.kafka.common.serialization.StringSerializer 2
value.serializer=org.apache.kafka.common.serialization.StringSerializer 3
client.id=my-client 4
compression.type=gzip 5
# ...
```

1. （必需）告诉制作者使用 Kafka 代理的 *host:port* bootstrap 服务器地址连接到 Kafka 集群。生产者使用地址来发现并连接到集群中的所有代理。如果服务器停机，请使用逗号分隔的列表来指定两个或三个地址，但不需要提供集群中的所有代理的列表。
2. （必需）Serializer 将每个消息的密钥转换为字节，然后再发送到代理。
3. （必需）Serializer 将每个消息的值转换为字节，然后再发送到代理。
4. （可选）客户端的逻辑名称，用于日志和指标来识别请求源。
5. （可选）压缩消息的代码c，这些消息会被发送，并可能以压缩格式存储，然后在到达消费者时解压缩。压缩可用于提高吞吐量并减少存储负载，但可能不适用于压缩或解压缩成本的低延迟应用程序。

5.2. 数据持久性

消息发送确认可最大程度降低消息丢失的可能性。默认情况下，将 acks 属性设置为 **acks=all** 时启用确认。要控制制作者从代理等待确认的时间，并处理发送消息的潜在延迟，您可以使用 **delivery.timeout.ms** 属性。

确认消息发送

```
# ...
acks=all (1)
delivery.timeout.ms=120000 (2)
# ...
```

1. **acks=all** 强制领导副本将消息复制到特定数量的后续人员，然后再确认消息请求被成功收到。
2. 等待完整发送请求的最长时间（毫秒）。您可以将值设为 **MAX_LONG**，以委派给 Kafka 的重试次数。默认值为 **120000** 或 2 分钟。

acks=all 设置提供了最强的交付保证，但它会增加生产者发送消息和接收确认之间的延迟。如果您不要求这种强保证，则 **acks=0** 或 **acks=1** 的设置不提供交付保证，或者仅确认领导副本已将记录写入其日志中。

使用 **acks=all** 时，领导机会等待所有同步副本确认消息发送。主题的 **min.insync.replicas** 配置设定了同步副本确认所需的最小所需数量。确认数量包括领导和跟着者的数量。

典型的起点是使用以下配置：

- 制作者配置：
  - **acks=all** (default)
- 主题复制的代理配置：
  - **default.replication.factor=3** (default = **1**)
  - **min.insync.replicas=2** (default = **1**)

在创建主题时，您可以覆盖默认的复制因素。您还可以在主题配置的主题级别覆盖 **min.insync.replicas**。

Apache Kafka 的 Streams 在示例配置文件中使用此配置来进行 Kafka 的多节点部署。

下表描述了此配置的工作方式，具体取决于复制领导副本的后续者可用性。

表 5.1. 遵循可用性

可用和同步的后续者数量	致谢	生产者可以发送消息？
2	领导等待 2 个后续确认	是
1	领导等待 1 个跟随者的确认	是
0	领导机引发异常	否

3 的主题复制因素会创建一个领导副本和两个后续者。在此配置中，如果单个后续者不可用，则生成者可以继续。有些延迟可能会发生 whilst 从同步副本中删除失败的代理或创建新领导。如果第二个后续人也不可用，消息发送将无法成功。领导者不会发送成功消息，而不是向制作者发送错误(*而非充足的副本*)。生产者引发等同的异常。通过 **重试** 配置，生成者可以重新发送失败的消息请求。

注意

如果系统失败，缓冲区中数据的风险会丢失。

5.3. 订购的交付

幂等的生成者会避免在消息发送一次时重复。ID 和序列号分配给消息，以确保发送顺序，即使出现失败情况。如果您使用 **acks=all** 用于数据一致性，使用 idempotency 对排序的交付有意义。默认情况下，为制作者启用 idempotency。启用 idempotency 后，您可以将并发 in-flight 请求的数量设置为最多 5 个，以便保留消息排序。

使用 idempotency 排序交付

```
# ...
enable.idempotence=true 1
max.in.flight.requests.per.connection=5 2
acks=all 3
retries=2147483647 4
# ...
```

1. 设置为 **true** 以启用幂等制作者。
2. 随着幂等的交付，动态请求数可能大于 1，同时仍然提供消息顺序保证。默认为 5 个 in-flight 请求。
3. 将 a **ck** 设置为 **all**。
4. 设置重新发送失败消息请求的尝试次数。

如果您选择不使用 **acks=all** 并因为性能成本而禁用 idempotency，请将 in-flight （未确认）请求数量设置为 1 来保留顺序。否则，在 *Message-A* 已写入代理后，*Message-* A 才会成功。

在没有 idempotency 的情况下排序交付

```
# ...
enable.idempotence=false 1
max.in.flight.requests.per.connection=1 2
retries=2147483647
# ...
```

1. 设置为 **false** 以禁用幂等制作者。
2. 将动态请求数设置为正好 **1**。

5.4. 可靠性保证

在对单个分区进行一次写入一次，idempotence 非常有用。事务与 idempotence 一起使用时，允许跨多个分区的一次写入一次。

使用相同事务 ID 发送的事务消息只生成一次，因此只会*所有*都成功写入到相应的日志，或*所有都没有*写入。

```
# ...
enable.idempotence=true
max.in.flight.requests.per.connection=5
acks=all
retries=2147483647
transactional.id=UNIQUE-ID (1)
transaction.timeout.ms=900000 (2)
# ...
```

1. 指定唯一事务 ID。
2. 在返回超时错误前设置事务的最大允许时间（以毫秒为单位）。默认值为 **900000** 或 15 分钟。

选择 **transactional.id** 非常重要，以便保持事务保证。每个事务 ID 都应该用于一组唯一的主题分区。例如，可以使用主题分区名称的外部映射到事务 ID，或使用避免冲突的功能计算主题分区名称中的事务 ID 来实现。

5.5. 为吞吐量和延迟优化制作者

通常，系统的要求是满足给定延迟内消息的特定吞吐量目标。例如，以每秒 500,000 条消息为目标，在 2 秒内确认消息的 95%。

您的制作者的消息语义（消息排序和持久性）可能由您的应用程序的要求定义。例如，您可能没有选择使用 **acks=0** 或 **acks=1**，而不破坏一些重要属性或应用程序提供的保证。

代理重启会对高百分比统计产生重大影响。例如，在较长的时间内，代理重启过程中会造成 99 百分比的延迟。在设计基准测试中或比较基准测试与生产环境中看到的性能号进行比较时，这值得考虑。

根据您的目标，Kafka 提供了很多配置参数和技术，用于调优生成者性能以获得吞吐量和延迟。

消息批处理(linger.ms 和 batch.size)

消息批处理延迟会按希望向同一代理发送更多消息，从而使它们被批量批量化到单个生成请求中。批处理在返回更高吞吐量时返回延迟之间很折现。基于时间的批处理使用 **linger.ms** 进行配置，并且基于大小的批处理则使用 **batch.size** 进行配置。

压缩(**compression.type**)

消息压缩会增加生成者（压缩消息的 CPU 时间）的延迟，但可以使请求（以及可能的磁盘写入）更小，从而提高吞吐量。无论是值得压缩还是值得使用的最佳压缩，都将取决于所发送的消息。压缩发生在调用 **KafkaProducer.send （）** 的线程上，因此如果此方法的延迟关系，您应该考虑使用更多线程。

pipelining (**max.in.flight.requests.per.connection**)

pipelining 表示在收到上一个请求的响应前发送更多请求。一般来说，管道流意味着更好的吞吐量，最多可达到其他影响的阈值，如更糟糕的批处理，开始处理对吞吐量的影响。

降低延迟

当应用程序调用 **KafkaProducer.send （）** 方法时，在发送前会处理一系列操作：

- 拦截器 ：消息由任何配置的拦截器处理。
- 序列化 ：消息被序列化为适当的格式。
- 分区分配 ：每个消息都分配给一个特定的分区。
- 压缩：消息被压缩以节省网络带宽。
- 批处理：压缩的消息添加到特定于分区的队列中的批处理中。

在这些操作过程中，**send （）** 方法会完全阻止。如果 **buffer.memory** 已满或者元数据不可用，它也会被阻断。

批处理将保留在队列中，直到出现以下情况之一：

- 批处理已满（根据 **batch.size**）。
- **linger.ms** 引入的延迟已通过。
- 发件人准备好将其他分区的批处理分配到同一代理，并可包含此批处理。
- 生成者正在清除或关闭。

为最大程度降低 **send （）** 阻塞对延迟的影响，请优化批处理和缓冲配置。使用 **linger.ms** 和 **batch.size** 属性将更多消息批处理到单个生成请求中，以获得更高的吞吐量。

```
# ...
linger.ms=100 1
batch.size=16384 2
buffer.memory=33554432 3
# ...
```

1. **linger.ms** 属性会以毫秒为单位添加一个延迟，以便在请求中累积和发送大量信息。默认值为 **0**。
2. 如果使用了最大 **batch.size**，则在达到最大值时，将发送请求，或者已排队消息的时间超过 **linger.ms** （以更早的时间为准）。添加延迟可让批处理将消息递增到批处理大小。
3. 缓冲区大小必须至少与批处理大小相同，并能够容纳缓冲、压缩和动态请求。

增加吞吐量

您可以使用自定义分区程序将消息定向到指定分区来替换默认值，来提高消息请求的吞吐量。

```
# ...
partitioner.class=my-custom-partitioner 1

# ...
```

1. 指定自定义分区器的类名称。

第 6 章 处理大量信息

如果您的 Apache Kafka 部署的流需要处理大量信息，您可以使用配置选项优化吞吐量和延迟。

生产者和消费者配置可帮助控制对 Kafka 代理的请求大小和频率。有关配置选项的更多信息，请参阅以下内容：

- 生成者的 Apache Kafka 配置文档
- 消费者的 Apache Kafka 配置文档

您还可以将相同的配置选项与 Kafka Connect 运行时源连接器（包括 MirrorMaker 2）和接收器连接器使用的制作者和消费者一起使用。

源连接器

- 来自 Kafka Connect 运行时的制作者向 Kafka 集群发送信息。
- 对于 MirrorMaker 2，因为源系统是 Kafka，因此消费者从源 Kafka 集群检索信息。

sink 连接器

- Kafka Connect 运行时中的消费者从 Kafka 集群检索信息。

对于消费者，您可以增加在单个获取请求中获取的数据量，以减少延迟。您可以使用 **fetch.max.bytes** 和 **max.partition.fetch.bytes** 属性增加 fetch 请求大小。您还可以使用 **max.poll.records** 属性设置从消费者缓冲区返回的消息数量的最大限制。

对于 MirrorMaker 2，在源连接器级别（**consumer.***）配置 **fetch.max.bytes**, **max.partition.fetch.bytes**, 和 **max.poll.records** 值，因为它们与从源获取消息的特定消费者相关。

对于制作者，您可以增加在单个生成请求中发送的消息批处理的大小。您可以使用 **batch.size** 属性增加批处理大小。更大的批处理大小可减少准备好发送的未完成消息的数量，以及消息队列中积压的大小。发送到同一分区的消息将一起批处理。当达到批处理大小时，生成请求将发送到目标集群。通过增加批处理大小，生成请求会延迟，更多的消息添加到批处理中，并同时发送到代理。当您只有几个处理大量消息的主题分区时，这可以提高吞吐量。

考虑制作者为合适的生产批处理大小处理的记录的数量和大小。

使用 **linger.ms** 添加等待时间（以毫秒为单位），以延迟生产负载减少时生成请求。delay 表示如果记录位于最大批处理大小下，则可以向批处理添加更多记录。

在连接器级别 (**producer.override.***) 配置 **batch.size** 和 **linger.ms** 值，因为它们与向目标 Kafka 集群发送信息的特定制作者相关。

对于 Kafka Connect 源连接器，到目标 Kafka 集群的数据流管道如下：

Kafka Connect 源连接器的数据流管道

External data source →(Kafka Connect tasks) source message queue → producer buffer → target Kafka topic

对于 Kafka Connect sink 连接器，数据流管道到目标外部数据源，如下所示：

Kafka Connect sink 连接器的数据流管道

source Kafka topic → (Kafka Connect tasks) sink message queue → consumer buffer → external data source

对于 MirrorMaker 2，数据将管道镜像到目标 Kafka 集群，如下所示：

MirrorMaker 2 的数据镜像管道

source Kafka topic → (Kafka Connect tasks) source message queue → producer buffer → target Kafka topic

producer 会将缓冲区中的信息发送到目标 Kafka 集群中的主题。在发生这种情况时，Kafka Connect 任务继续轮询数据源，以将消息添加到源消息队列中。

源连接器的制作者缓冲区的大小使用 **producer.override.buffer.memory** 属性设置。任务在清除缓冲区前等待指定的超时时间(**offset.flush.timeout.ms**)。这应该有足够的时间供代理和提交的偏移数据确认。源任务不会等待生成者在提交偏移前清空消息队列，但关机期间除外。

如果生产者无法满足源消息队列中消息的吞吐量，则缓冲区将被阻止，直到缓冲区中的空间在 **max.block.ms** 绑定的时间段内有空间可用。在此期间，缓冲区中仍然发送任何未确认的信息。在确认和刷新这些消息前，不会向缓冲区添加新消息。

您可以尝试以下配置更改，将未处理消息的底层源消息队列保持为可管理的大小：

- 将默认值（以毫秒为单位）增加 **offset.flush.timeout.ms**
- 确保有足够的 CPU 和内存资源
- 通过执行以下操作增加并行运行的任务数量：
  - 使用 **tasksMax** 属性增加并行运行的任务数量
  - 使用 **replicas** 属性增加运行任务的 worker 节点数量

根据可用的 CPU 和内存资源和 worker 节点数量，请考虑可以并行运行的任务数量。您可能需要调整配置值，直到它们有所需的效果。

6.1. 为高卷信息配置 KAFKA CONNECT

Kafka Connect 从源外部数据系统获取数据，并将其传递给 Kafka Connect 运行时制作者，使其复制到目标集群。

以下示例显示了使用 KafkaConnect 自定义资源的 **Kafka Connect** 配置。

用于处理大量消息的 Kafka Connect 配置示例

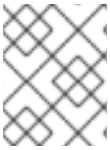
```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true"
spec:
  replicas: 3
  config:
    offset.flush.timeout.ms: 10000
    # ...
  resources:
```

```
    requests:
      cpu: "1"
      memory: 2Gi
    limits:
      cpu: "2"
      memory: 2Gi
  # ...
```

为源连接器添加生成者配置，该连接器通过 **KafkaConnector** 自定义资源进行管理。

用于处理大量信息的源连接器配置示例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: my-source-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: org.apache.kafka.connect.file.FileStreamSourceConnector
  tasksMax: 2
  config:
    producer.override.batch.size: 327680
    producer.override.linger.ms: 100
    # ...
```



注意

FileStreamSourceConnector 和 **FileStreamSinkConnector** 作为示例连接器提供。有关将其部署为 **KafkaConnector** 资源的详情，请参考 部署 **KafkaConnector** 资源。

为接收器连接器添加消费者配置。

用于处理大量消息的接收器连接器配置示例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: my-sink-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: org.apache.kafka.connect.file.FileStreamSinkConnector
  tasksMax: 2
  config:
    consumer.fetch.max.bytes: 52428800
    consumer.max.partition.fetch.bytes: 1048576
    consumer.max.poll.records: 500
    # ...
```

如果您使用 Kafka Connect API 而不是 **KafkaConnector** 自定义资源来管理连接器，您可以将连接器配置添加为 JSON 对象。

为处理大量信息添加源连接器配置的 curl 请求示例

```
curl -X POST \
  http://my-connect-cluster-connect-api:8083/connectors \
  -H 'Content-Type: application/json' \
  -d '{ "name": "my-source-connector",
    "config":
    {
      "connector.class":"org.apache.kafka.connect.file.FileStreamSourceConnector",
      "file": "/opt/kafka/LICENSE",
      "topic":"my-topic",
      "tasksMax": "4",
      "type": "source"
      "producer.override.batch.size": 327680
      "producer.override.linger.ms": 100
    }
}'
```

6.2. 为高卷消息配置 MIRRORMAKER 2

MirrorMaker 2 从源集群获取数据，并将其传递给 Kafka Connect 运行时制作者，以便将其复制到目标集群。

以下示例显示了使用 **KafkaMirrorMaker2** 自定义资源的 MirrorMaker 2 的配置。

处理大量消息的 MirrorMaker 2 配置示例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker2
spec:
  version: 3.7.0
  replicas: 1
  connectCluster: "my-cluster-target"
  clusters:
  - alias: "my-cluster-source"
    bootstrapServers: my-cluster-source-kafka-bootstrap:9092
  - alias: "my-cluster-target"
    config:
      offset.flush.timeout.ms: 10000
    bootstrapServers: my-cluster-target-kafka-bootstrap:9092
  mirrors:
  - sourceCluster: "my-cluster-source"
    targetCluster: "my-cluster-target"
    sourceConnector:
      tasksMax: 2
      config:
        producer.override.batch.size: 327680
        producer.override.linger.ms: 100
        consumer.fetch.max.bytes: 52428800
        consumer.max.partition.fetch.bytes: 1048576
        consumer.max.poll.records: 500
    # ...
  resources:
    requests:
      cpu: "1"
```

```
    memory: Gi
  limits:
    cpu: "2"
    memory: 4Gi
```

6.3. 检查 MIRRORMAKER 2 消息流

如果使用 Prometheus 和 Grafana 监控部署，您可以检查 MirrorMaker 2 消息流。

带有 Apache Kafka 的 Streams 提供的 MirrorMaker 2 Grafana 仪表板示例显示了以下与 flush 管道相关的指标。

- Kafka Connect 的未处理消息队列中的消息数量
- producer 缓冲区的可用字节数
- 偏移提交超时（以毫秒为单位）

您可以使用这些指标来量化是否需要根据消息卷调整配置。

其他资源

- 指标简介
- 添加 Kafka Connect 连接器

第 7 章 处理大量消息大小

Kafka 的默认批处理大小为 1MB，这是大多数用例中最大吞吐量的最佳选择。Kafka 可以在减少吞吐量下容纳更大的批处理，假设有足够的磁盘容量。

大型消息大小可以通过以下四种方式进行处理：

1. 代理、生产者和消费者配置为容纳更大的消息大小。
2. 生产者消息压缩 将压缩消息写入日志。
3. 基于参考的消息传递仅发送对消息有效负载中某些其他系统中存储的数据的引用。
4. 内联消息传递将信息分成使用相同键的块，然后使用流处理器（如 Kafka Streams）在输出中合并。

除非您要处理非常大的消息，否则建议使用配置方法。基于参考的消息和消息压缩选项涵盖了大多数其他情况。对于这些选项，必须小心，以避免引入性能问题。

7.1. 配置 KAFKA 组件以处理更大的信息

大型消息可能会影响系统性能，并在消息处理中引入复杂性。如果无法避免它们，则有可用的配置选项。要高效地处理较大的消息并防止消息流中的块，请考虑调整以下配置：

- 调整最大记录批处理大小：
  - 在代理级别设置 **message.max.bytes**，以支持所有主题的较大的记录批处理大小。
  - 在主题级别上设置 **max.message.bytes**，以支持单个主题的较大的记录批处理大小。
- 增加每个分区跟踪器获取的最大消息大小(**replica.fetch.max.bytes**)。
- 增加制作者的批处理大小(**batch.size**)，以增加单个生成请求中发送的消息批处理大小。
- 为制作者(**max.request.size**)和消费者配置最大请求大小(**fetch.max.bytes**)，以容纳更大的记录批处理。
- 设置一个更高的最大限制(**max.partition.fetch.bytes**)，以为每个分区返回多少数据。

确保批处理请求的最大大小至少为 **message.max.bytes**，以适应最大的记录批处理大小。

代理配置示例

```
message.max.bytes: 10000000
replica.fetch.max.bytes: 10485760
```

生成者配置示例

```
batch.size: 327680
max.request.size: 10000000
```

消费者配置示例

```
fetch.max.bytes: 10000000
max.partition.fetch.bytes: 10485760
```

-

也可以配置由 Kafka Bridge、Kafka Connect 和 MirrorMaker 2 等其他 Kafka 组件使用的制作者和消费者，以更有效地处理更大的信息。

Kafka Bridge

使用特定的制作者和消费者配置属性配置 Kafka Bridge：

- **producer.config** for producers
- **consumer.config** 用于消费者

Kafka Bridge 配置示例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  # ...
  producer:
    config:
      batch.size: 327680
      max.request.size: 10000000
  consumer:
    config:
      fetch.max.bytes: 10000000
      max.partition.fetch.bytes: 10485760
      # ...
```

Kafka Connect

对于 Kafka Connect，配置负责使用制作者和消费者配置属性的前缀来发送和接收消息的源和接收器连接器：

- 源连接器用来向 Kafka 集群发送消息的制作者的 **producer.override**
- sink 连接器用于从 Kafka 集群检索消息 **的消费者**

Kafka Connect 源连接器配置示例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: my-source-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  # ...
  config:
    producer.override.batch.size: 327680
    producer.override.max.request.size: 10000000
    # ...
```

Kafka Connect sink 连接器配置示例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: my-sink-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  # ...
  config:
    consumer.fetch.max.bytes: 10000000
    consumer.max.partition.fetch.bytes: 10485760
    # ...
```

MirrorMaker 2

对于 MirrorMaker 2，使用制作者和消费者配置属性的前缀配置源 Kafka 集群检索信息的源连接器：

- 用于将数据复制到目标 Kafka 集群的运行时 Kafka Connect producer 的 **producer.override**
- sink 连接器用于从源 Kafka 集群检索消息 **的消费者**

MirrorMaker 2 源连接器配置示例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker2
spec:
  # ...
  mirrors:
  - sourceCluster: "my-cluster-source"
    targetCluster: "my-cluster-target"
    sourceConnector:
      tasksMax: 2
      config:
        producer.override.batch.size: 327680
        producer.override.max.request.size: 10000000
        consumer.fetch.max.bytes: 10000000
        consumer.max.partition.fetch.bytes: 10485760
        # ...
```

7.2. 制作者压缩

对于生成者配置，您可以指定一个 **compression.type**，如 Gzip，它被应用到制作者生成的数据的批处理。使用代理配置 **compression.type=producer** （默认），代理会保留使用制作者的任何压缩。每当制作者和主题压缩不匹配时，代理必须在将批处理附加到日志前再次压缩批处理，这会影响代理性能。

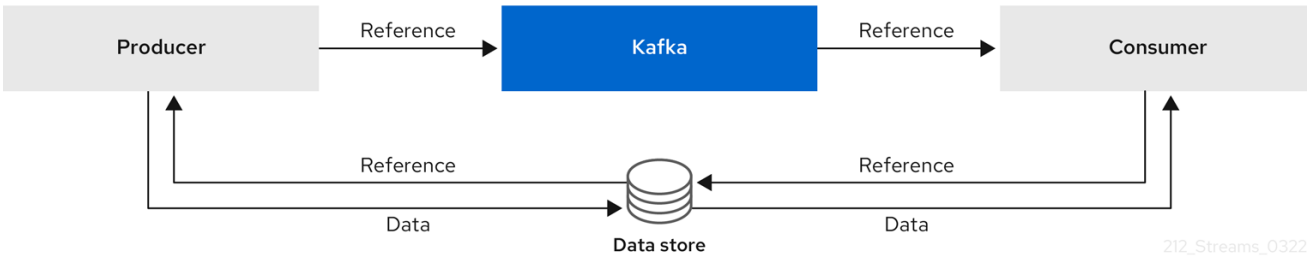
压缩还会在生产者和解压缩开销上增加额外的处理开销，但在批处理中包含更多的数据，因此当消息数据压缩良好时，吞吐量通常很有用。

将生成者压缩与批处理大小的微调相结合，以促进最佳吞吐量。使用指标有助于量化所需的平均批处理大小。

7.3. 基于参考的消息传递

当您不知道消息有多大时，基于参考的消息传递对于数据复制非常有用。外部数据存储必须快速、持久且高度可用，才能使此配置正常工作。数据被写入数据存储，并返回对数据的引用。producer 发送一条消息，其中包含对 Kafka 的引用。消费者从消息中获取引用，并使用它来从数据存储中获取数据。

7.4. 基于参考的消息传递流



当消息传递需要更多行程时，端到端延迟会增加。这个方法的另一个显著缺陷是，在清理 Kafka 消息时，外部系统中没有自动清理数据。混合方法是仅将大型消息发送到数据存储并直接处理标准化消息。

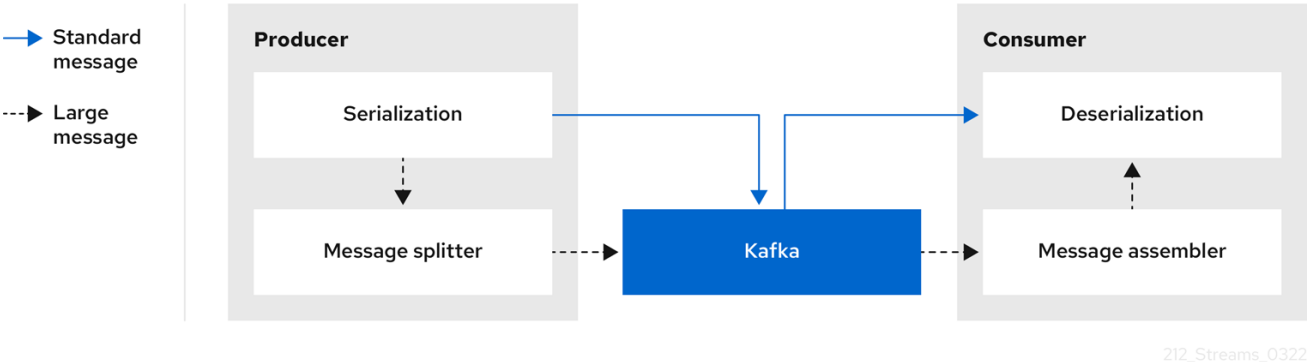
内联消息传递

内联消息传递非常复杂，但它对基于参考的消息等外部系统没有开销。

如果消息太大，则生成客户端应用必须序列化，然后对数据进行阻塞。然后，生成者使用 Kafka **ByteArraySerializer**，或者在发送前再次序列化每个块。消费者跟踪消息和缓冲区块，直到它有完整的消息。消耗客户端应用程序接收块，这些块在进行序列化前被编译。根据每个块消息集合的第一个或最后一个块的偏移量，将完整消息发送到消耗应用程序的剩余部分。

消费者应仅在接收和处理所有消息块后提交其偏移，以确保准确跟踪消息交付并防止重新平衡期间发生重复。块可能会在不同的网段之间分布。消费者处理应该覆盖在随后删除片段时块不可用的可能性。

图 7.1. 内联消息传递流



内联消息传递在消费者端具有性能开销，因为需要缓冲，特别是在并行处理一系列大型消息时。大型消息的块可能会变为交集，因此如果缓冲区中另一个大消息的块不完整，则无法提交消息的所有区块。因此，通常通过持久保留消息块或实施提交逻辑来支持缓冲。

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