

Apache Kafka 入门



每个企业都支持数据。我们获取信息，分析信息，对其进行操作，并创建更多作为输出。每个应用程序都会创建数据，无论是日志消息、指标、用户活动、传出消息。

我们每天都在京东、淘宝等应用程序上看到这一点，我们点击我们感兴趣的商品会变成推荐，稍后会向我们展示。如果使用微信支付购买在京东、淘宝等应用程序上看中的商品后，也会在使用微信时微信广告推荐也会推荐与你购买商品类似的商品。

本章重点介绍 Apache Kafka 及其用例的简介，此外本章还将解释 Kafka 的诞生和组件。

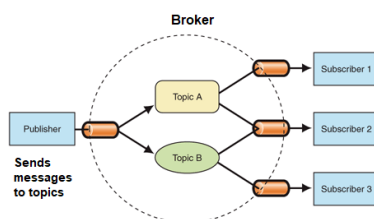
目标

本章中您将学习：

- 📖 Kafka 简介
- 📖 为什么选择 Kafka
- 📖 Kafka 应用场景
- 📖 数据生态系统
- 📖 Kafka 的诞生
- 📖 Kafka 组件

Kafka 简介

在开始了解 Apache Kafka 的具体细节之前，了解发布/订阅消息的概念以及为什么它很重要非常重要。发布/订阅消息传递是一种模式，其特征是数据（消息）的发送者（发布者）没有专门将其定向到接收方。相反，发布者以某种方式对消息进行分类，并且接收方（订阅者）订阅以接收某些类消息。Pub/sub 系统通常具有代理，这是发布消息的中心点，如图。



发布/订阅模式

Apache Kafka 是一个分布式发布-订阅邮件系统，它接收来自不同源系统的数据，并实时将数据提供给目标系统。Kafka 以 Scala 和 Java 编写，通常与大数据的实时事件流处理相关。

Apache Kafka 是解决任何软件解决方案的实时问题的解决方案，该解决方案可处理实时信息量并快速路由到多个使用者。

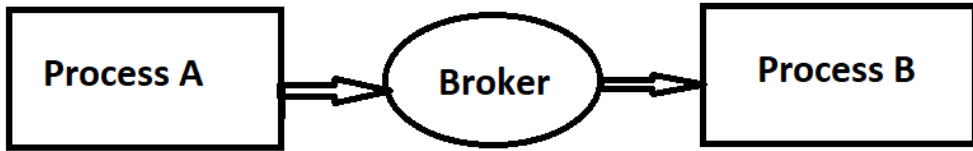
Apache Kafka 是一个开源、分布式、分区和复制的基于提交日志的发布-订阅邮件系统。Kafka 提供实时发布-订阅解决方案，可克服使用实时和批处理数据量的挑战，这些数据量可能增长数量级大于实际数据。Kafka 还支持 Hadoop 系统中的并行数据加载。

Apache Kafka 提供来自生产者和消费者的信息之间的无缝集成，而不会阻止信息的生产者，也不让生产者知道谁是最终消费者。Kafka 的体系结构由以下组件组成：

- **Topics:** Topics 就像一个类别/一个索引，它把消息一起组在一起。
- **生产者 Producer:** 将消息推送到 Kafka Topics 的流程。
- **消费者 Consumer:** 使用来自 Kafka Topics 的消息的进程。
- **分区 Partition:** 一个不可变的 Topics 消息序列，连续地附加到结构化提交日志中。
- **Kafka 代理:** 形成 Kafka 群集的一个或多个服务器。
- **消费者组 Consumer Groups:** 这些是用于加载共享的使用者组。如果使用者组使用来自一个分区的消息，则使用者组中的每个使用者都将使用不同的消息。使用者组通常用于加载共享。

为什么选择 Kafka

传统的消息代理系统（如符合 JMS 或 AMQP 标准的系统）往往具有直接连接到代理的流程，以及直接连接到流程的代理。因此，对于消息从一个进程转到另一个进程，它可以通过代理路由。这些解决方案倾向于针对灵活性和可配置的交付保证进行优化。



Apache Kafka 通常称为分布式提交日志技术，它的作用与传统的代理消息系统类似。Kafka 针对不同的用例进行了优化，然而，它们并没有集中在灵活性和交付保证上，而是倾向于关注可伸缩性和吞吐量。以下是使卡夫卡成为传统消息代理的好选择的一些其他原因。

■ 多个生产者 Multiple Producers

Kafka 能够无缝处理多个生产者，无论这些客户端是使用许多 Topics 还是同一 Topics。这使得该系统非常适合聚合来自许多前端系统的数据并使其保持一致。例如，通过许多微服务向用户提供内容的网站可以具有一个页面视图 Topics，所有服务都可以使用通用格式写入该 Topics。然后，使用者应用程序可以接收站点上所有应用程序的单个页面视图流，而无需协调来自多个 Topics 的使用，每个 Topics 一个。

■ 多个消费者 Multiple Consumers

Kafka 专为多个消费者设计，无需相互干扰即可读取任何单个消息流。这与许多排队系统不同，一个客户端使用消息后，它不适用于任何其他客户端。多个 Kafka 消费者可以选择作为组的一部分操作并共享流，确保整个组只处理给定消息一次。

■ 基于磁盘的保留 Disk-Based Retention

Kafka 不仅可以处理多个消费者，而且持久邮件保留意味着消费者并不总是需要实时工作。消息将提交到磁盘，并将与可配置的保留规则一起存储。这些选项可以基于每个 Topics 进行选择，允许根据消费者的需求具有不同的消息流具有不同的保留量。持久保留意味着，如果消费者由于处理速度缓慢或流量突发而落后，则不会丢失数据。这也意味着可以对使用者执行维护，使应用程序脱机一小段时间，而不关心在生产者上备份的消息或丢失消息。可以停止使用者，并且消息将保留在 Kafka 中。这允许他们重新启动和拾取处理消息，他们离开没有数据丢失。

■ 可扩展 Scalable

Kafka 灵活的可伸缩性使处理任何数量的数据变得容易。用户可以从一个代理开始作为概念证明，扩展到由三个代理组成的小型开发集群，然后随着数据规模的扩大，由数十个甚至数百个代理组成的更大集群投入生产。扩展可以在集群联机时进行，而不会影响整个系统的可用性。这也意味着由多个代理组成的集群可以处理单个代理的故障，并继续为客户机提供服务。需要容忍更多并发故障的集群可以配置更高的复制因子。

■ 高性能 High Performance

所有这些特性结合在一起，使得 Apache Kafka 成为一个在高负载下具有出色性能的发布/订阅消息传递系统。生产者、消费者和代理都可以被扩展，以轻松地处理非常大的消息流。这可以在提供从产生消息到消费者可用性的次秒级消息延迟的同时完成。



Just a minute:

_____ 处理 Kafka 消息的 Topics。

1. *producer*
2. *consumer*
3. *Kafka broker*
4. *partition*

答案:

1. *producer*

Kafka 应用场景

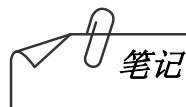
Kafka 在任何建筑中的使用方式有很多种。本节讨论 Apache Kafka 的一些流行用例

■ 信息系统 Messaging

消息代理用于将数据处理与数据生产者分离。Kafka 可以取代许多流行的消息代理，因为它提供了更好的吞吐量、内置分区、复制和容错能力。

■ 网站活动跟踪 Website Activity Tracking

Kafka 原本的应用场景要求它能重建一个用户活动追踪管线作为一个实时的发布与订阅消息源。意思就是用户在网站上的动作事件（如浏览页面、搜索、或者其它操作）被发布到每个动作对应的中心化 Topic 上。使得这些数据源能被不同场景的需求订阅到，这些场景包括实时处理、实时监控、导入 Hadoop 或用于离线处理、报表的离线数据仓库中。



笔记

活动追踪通常情况下是非常高频的，因为很多活动消息是由每个用户的页面浏览产生的。

■ 监控 Metrics

在此用例中，具有生成相同类型消息的多个应用程序的能力闪耀。应用程序定期将指标发布到 Kafka Topics，这些指标可由用于监视和警报的系统使用。它们还可用于像 Hadoop 这样的脱机系统来执行长期分析，如增长预测。

■ 日志收集 Log aggregation

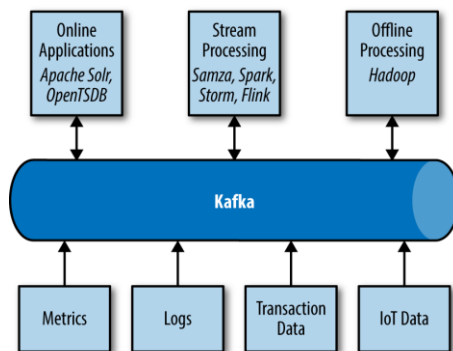
这是从服务器收集物理日志文件并将其放在中心位置（文件服务器或 HDFS）进行处理的过程。使用 Kafka 可以将日志或事件数据抽象为消息流，从而消除对文件细节的任何依赖。这还提供了较低的延迟处理和对多个数据源和分布式数据消耗的支持。

■ 流处理 Stream processing

Kafka 可用于收集的数据经过多个阶段处理的用例一个示例是从 Kafka 的 Topic 中消费的原始数据，并将其丰富或转换为新的 Kafka Topic 以供进一步使用。因此，这种处理也称为流处理。

数据生态系统 The Data Ecosystem

许多应用程序都参与到我们为数据处理而构建的环境中。我们以应用程序的形式定义了输入，这些应用程序创建数据或将数据引入系统。我们以度量、报告和其他数据产品的形式定义了输出。我们创建循环，其中一些组件从系统中读取数据，使用其他来源的数据对其进行转换，然后将其重新引入到数据基础设施中，以便在其他地方使用。这是针对许多类型的数据进行的，每种类型的数据都有其独特的内容、大小和用途。



Apache Kafka 为数据生态系统提供了循环系统，如图所示。它在基础设施的各个成员之间传递消息，为所有客户机提供一致的接口。当与提供消息模式的系统耦合时，生产者和消费者不再需要任何类型的紧密耦合或直接连接。组件可以随着业务案例的创建和分解而添加和删除，而生产者不需要关心谁在使用数据或使用应用程序的数量。

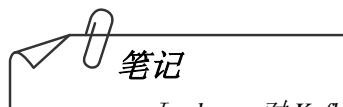
Kafka 的诞生

LinkedIn 的开发团队由 Jay Kreps 领导，他是一位首席软件工程师，他以前负责伏地魔（一种分布式密钥存储系统）的开发和开源发行。最初的团队还包括内哈·纳克赫德和后来的雷军。他们一起着手创建一个消息传递系统，既能满足监控和跟踪系统的需要，又能扩展到未来。主要目标是：

- 通过使用推拉模型来分离生产者和消费者
- 为消息传递系统中的消息数据提供持久性，以允许多个消费者
- 优化消息的高吞吐量
- 允许系统的水平扩展随着数据流的增长而增长

结果产生了一个发布/订阅消息传递系统，它具有典型的消息传递系统的接口，但存储层更像日志聚合系统。结合 apacheavro 对消息序列化的采用，Kafka 可以有效地处理度和每天数十亿条消息的用户活动跟踪。Kafka 的可扩展性帮助 LinkedIn 的使用量增长了超过 1 万亿条（截至 2015 年 8 月），每天消耗的数据量超过 1 PB

Kafka 是在 2010 年底作为一个开源项目在 GitHub 上发布的。随着它开始在开源社区引起关注，它在 2011 年 7 月被提议并被接受为 Apache 软件基金会孵化器项目。阿帕奇·卡夫卡于 2012 年 10 月从孵化器毕业。从那时起，它一直在努力，并在 LinkedIn 之外找到了一个由贡献者和提交者组成的强大社区。卡夫卡现在被用于世界上一些最大的数据管道中。2014 年秋天，Jay Kreps、Neha Narkhede 和 Jun Rao 离开 LinkedIn，成立了一家专注于为 ApacheKafka 提供开发、企业支持和培训的公司 Confluent。这两家公司，加上开源社区其他公司不断增长的贡献，继续开发和维护 Kafka，使其成为大数据管道的首选。



笔记

Jay kreps 对 Kafka 这个名字的见解：认为既然 Kafka 是一个为写作而优化的系统，那么使用一个作家的名字是有意义的。我在大学里上过很多文学课，喜欢 Franz Kafka。另外，这个名字对于开源项目来说听起来很酷。



_____从服务器收集物理日志文件并将它们放在一个中心位置（文件服务器或 HDFS）进行处理的过程。

1. 流处理
2. 日志收集
3. Metrics
4. messaging

Answer:

2. Log Aggregation

Kafka 组件

■ 消息和批处理

Kafka 内部的数据单位称为消息。如果您是从数据库背景了解 Kafka，您可以将其视为类似于行或记录。如果选中此选项，则消息中包含的数据对 Kafka 来说没有特定的格式或含义，对于 Kafka 来说，消息只是一个字节数组，因此其中包含的数据对 Kafka 来说没有特定的格式或含义。无法加载任何数据源。消息可以具有可选的元数据位（称为键）。消息可以具有可选的元数据位（称为键）。密钥也是一个字节数组，与消息一样，对 Kafka 没有特定的含义。当消息以更可控的方式写入分区时，使用键；当消息以更可控的方式写入分区时，使用键。最简单的这种方案是生成密钥的一致哈希，然后通过获取哈希 modulo 的结果，即 Topics 中分区的总数，来选择该消息的分区号。这可以确保具有相同密钥的消息始终写入同一分区。

为了提高效率，消息被成批写入 Kafka。批处理只是消息的集合，所有这些消息都是针对同一 Topics 和分区生成的。每个消息在网络上的单独往返会导致过多的开销，而将消息收集到一个批中可以减少这种开销。当然，这是延迟和吞吐量之间的折衷：批处理越大，单位时间内可以处理的消息越多，但单个消息传播所需的时间就越长。批处理通常也被压缩，以提供更高效的数据传输和存储，但需要一些处理能力。

■ 摘要 Schemas

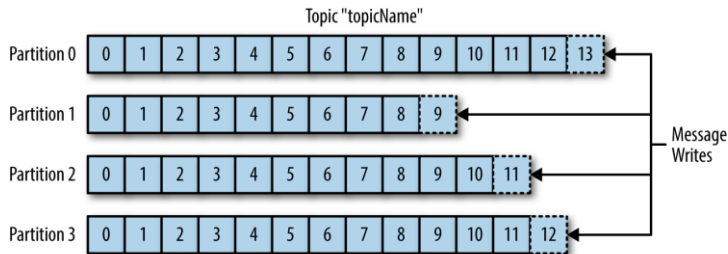
另外，也可以把消息本身理解成一个不透明的数组，这样就可以很容易地理解消息本身的不透明性。根据应用程序的个别需要，消息模式有许多可用的选项。简单化的系统，

如 JavaScript 对象表示法 (JSON) 和可扩展标记语言 (XML)，易于使用和人类可读。但是，它们缺乏诸如健壮的类型处理和模式版本之间的兼容性等特性。许多 Kafka 开发人员喜欢使用 Apache Avro，这是一个最初为 Hadoop 开发的序列化框架。Avro 提供了一种紧凑的序列化格式；模式与消息有效负载分离，并且在更改时不需要生成代码；以及强大的数据类型和模式演化，具有向后和向前兼容性。

在 Kafka 中，一致的数据格式是很重要的，因为它允许消息的读写分离。当这些任务紧密耦合时，订阅消息的应用程序必须更新以处理新的数据格式，同时处理旧格式。只有这样，才能更新发布消息的应用程序以使用新格式。通过使用定义良好的模式并将它们存储在一个公共存储库中，Kafka 中的消息可以在不需要协调的情况下被理解。

■ Topics 和分区 Topics and Partitions

Kafka 中的消息被分类为 Topics。Topics 最接近的类比是数据库表或文件系统中的文件夹。Topics 还被分解为多个 Partitions。回到“提交日志”描述，分区是单个日志。消息以仅追加的方式写入它，并按顺序从头到尾读取。请注意，由于 Topics 通常具有多个分区，因此不能保证整个 Topics（只需在单个分区内）的消息时间排序。图中显示了一个包含四个分区的 Topics，每个分区的末尾都追加写入。分区也是 Kafka 提供冗余性和可扩展性的方式。每个分区可以托管在不同的服务器上，这意味着单 Topics 可以跨多个服务器水平缩放，以提供远远超出单个服务器能力的性能。



具有多个分区的 Topic

在 Kafka 这样的系统中讨论数据时，经常使用术语流。通常，流被认为是一个单独的数据主题，而不考虑分区的数量。这表示从生产者到消费者的单一数据流。在讨论流处理时，这种引用消息的方式最为常见，即当框架（其中包括 Kafka Streams、Apache Samza 和 Storm）实时操作消息时。这种操作方法可以与离线框架（即 Hadoop）设计用于稍后处理批量数据的方式相比较。

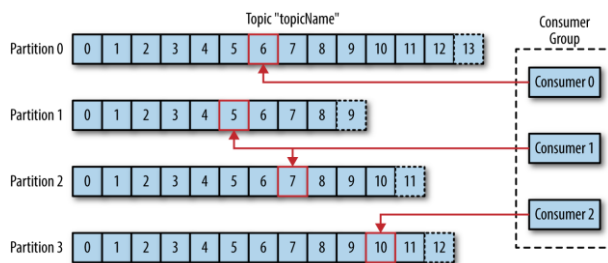
■ 生产者和消费者 Producers and Consumers

Kafka 客户端是系统的用户，有两种基本类型：生产者和消费者。还有用于数据集成的高级客户端 API Kafka Connect API 和用于流处理的 Kafka 流。高级客户机使用生产者和消费者作为构建块，并在顶部提供更高级别的功能。

Producer 创建新消息。在其他发布/订阅系统中，这些可能被称为发布者或写入者。一般来说，将针对特定主题生成消息。默认情况下，生产者不关心特定消息写入到哪个分区，它会在主题的所有分区上均匀地平衡消息。在某些情况下，生产者将消息定向到特定的分区。这通常是通过使用消息密钥和分区器来完成的，该分区器将生成密钥的哈希并将其映射到特定分区。这可以确保使用给定密钥生成的所有消息都将写入同一分区。生产者还可以使用遵循其他业务规则的自定义分区器将消息映射到分区。

消费者阅读信息。在其他发布/订阅系统中，这些客户端可以称为订阅服务器或读卡器。使用者订阅一个或多个主题，并按照消息产生的顺序读取消息。消费者通过跟踪消息的偏移量来跟踪它已经消费了哪些消息。偏移量是另一个元数据位，它是一个整数值，Kafka 在每一条消息产生时都会不断增加。给定分区中的每条消息都有一个唯一的偏移量。通过在 Zookeeper 或 Kafka 中存储每个分区上一次使用的消息的偏移量，使用者可以停止并重新启动，而不会丢失位置。

消费者作为消费者群体的一部分工作，消费者群体是一个或多个消费者共同消费一个主题。该组确保每个分区仅由一个成员使用。在图中，一个组中有三个消费者在消费一个主题。其中两个使用者分别从一个分区工作，而第三个使用者在两个分区工作。使用者到分区的映射通常称为使用者对分区的所有权。

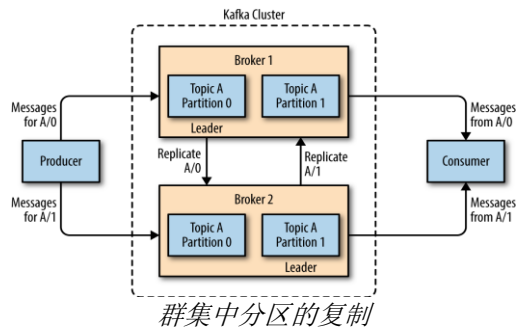


来自 topic 正在读取的消费者组

通过这种方式，消费者可以横向扩展以使用包含大量消息的主题。此外，如果单个使用者失败，组的其余成员将重新平衡正在使用的分区，以接管丢失的成员。

代理和集群 Brokers and Clusters

单个 Kafka 服务器称为代理。代理接收来自生产者的消息，为其分配偏移量，并将消息提交到磁盘上的存储。它还为用户提供服务，响应分区的提取请求，并响应已提交到磁盘的消息。根据特定的硬件及其性能特性，单个代理可以轻松地处理数千个分区和数百万条消息/秒。Kafka 代理被设计成集群的一部分。在代理集群中，一个代理还将充当集群控制器（从集群的活动成员中自动选择）。控制器负责管理操作，包括为代理分配分区和监视代理故障。一个分区由集群中的一个代理拥有，这个代理称为分区的领导者。一个分区可以分配给多个代理，这将导致分区被复制（如下图所示）。这在分区中提供了冗余的消息，这样当代理失败时，另一个代理可以接管领导权。但是，在该分区上操作的所有消费者和生产者都必须连接到领导者。



Apache Kafka 的一个关键特性是保留功能，即在一段时间内持久地存储消息。Kafka 代理配置了主题的默认保留设置，保留消息一段时间（例如，7 天）或直到主题达到某个字节大小（例如 1GB）。一旦达到这些限制，邮件就会过期并被删除，因此保留配置是任何时候可用的最小数据量。还可以使用各自的保留设置来配置各个主题，以便仅在邮件有用时存储这些邮件。例如，跟踪主题可能保留几天，而应用程序度量可能只保留几个小时。主题也可以配置为日志压缩，这意味着 Kafka 将只保留使用特定密钥生成的最后一条消息。这对于变更类型的数据非常有用，因为只有最后一次更新才是最新的。



Just a minute:

_____代理设计为作为群集的一部分运行。

- 1.Kafka
- 2.Schema
3. HBase
- 4.producer

Answer:

1. Kafka



活动 1.1: *Apache Kafka* 案例研究

练习题

1. 对 producer（生产者）描述正确的是
 - a. 使用来自 Kafka topics 的消息的进程
 - b. 将消息推送到 Kafka topics 的进程
 - c. 作为群集的一部分运行。
 - d. 是消息的持久存储一段时间
2. 单个 Kafka 服务器称为 _____
 - a. Producer
 - b. consumer
 - c. broker
 - d. Partitioner
3. 识别有关生产者和消费者的错误陈述
 - a. 生产者发布新消息
 - b. 消费者订阅消息
 - c. 消费者不是消费者组的一部分
 - d. 生产者发布消息到 Kafka topics
4. Kafka 灵活的可扩展性使其易于处理
 - a. 少量数据
 - b. 大量数据
 - c. 不处理数据
 - d. 有时处理数据
5. 以下哪一项描述正确？
 1. Apache Kafka 的一个关键功能是保留功能，它是一段时间内消息的持久存储
 2. 消费者作为消费者组的一部分，该组是一个或多个消费者，他们使用共同的 topic.
 - a. 1 正确，2 错误
 - b. 1 错误，2 正确
 - c. 两个都对
 - d. 两个都错

总结

本章中，您了解到：

- **Apache Kafka** 是一个分布式发布-订阅消息系统，它接收来自不同源系统的数据，并实时将数据提供给目标系统。
- **Kafka 优点**
 1. 多个生产者
 2. 多个消费者
 3. 基于磁盘保留
 4. 可扩展
 5. 高性能
- **Kafka 应用场景**
 1. 消息系统
 2. 网络活动跟踪
 3. Metrics
 4. 日志收集
 5. 流处理
- **Apache Kafka 为数据生态系统提供循环系统**
- 它在所有客户端的各个成员之间传递消息，为所有客户端提供一致的接口。
- **Kafka** 于 2010 年早些时候作为开源项目发布在 **GitHub** 上。随着它开始引起开源社区的关注，它于 2011 年 7 月被提议并被接受为 **Apache** 软件基金会的孵化器项目。
- **Kafka 构成**
 1. 消息系统和批处理
 2. Schemas
 3. Topics 和分区
 4. 生产者和消费者
 5. 代理和集群

安装 Kafka



本章描述如何开始使用 Apache-Kafka 代理，包括如何设置 Apache Zookeeper，Kafka 使用它来存储代理的元数据。本章还将介绍 Kafka 部署的基本配置选项，以及选择运行代理的正确硬件的标准。

本章重点介绍 Kafka 的入门。

目标

在本章中，您将学到：

- 安装 Java
- 安装 Zookeeper
- 安装 Kafka 代理
- 代理配置
- Kafka 控制台工具
- Kafka 代理属性列表

设置 kafka 的环境

Apache Kafka 是一个 Java 应用程序，可以在许多操作系统上运行。这包括 Windows、MacOS、Linux 等。本章中的安装步骤将侧重于在 Linux 环境中设置和使用 Kafka，因为这是安装 Kafka 的最常见操作系统。这也是部署 Kafka 以供一般使用的推荐操作系统。

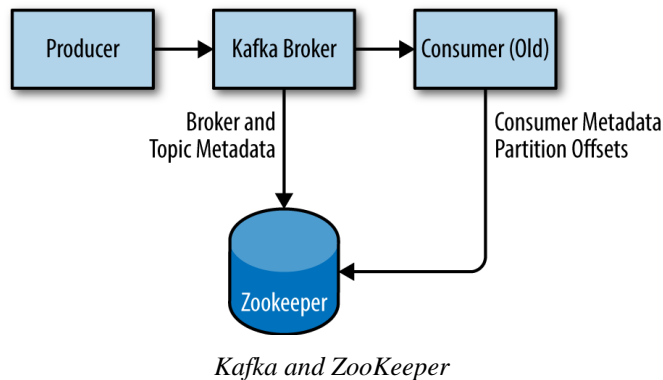
安装 Java

■ 安装 Java

在安装 Zookeeper 或 Kafka 之前，您需要一个 Java 环境的设置和运行。这应该是 Java8 版本，可以是您的操作系统提供的版本，也可以是直接从 java.com 网站。当开发一个完整的 Java 应用程序时，kafkkeeper 和 Java 应用程序开发包可能会更方便。安装步骤将假定您已经在/usr/java/jdk1.8.0_51 中安装了 jdk8 更新 51。

安装 Zookeeper

Apache Kafka 使用 Zookeeper 存储有关 Kafka 群集的元数据以及使用者客户端详细信息，如图 2-1 所示。虽然使用 Kafka 分发中包含的脚本运行 Zookeeper 服务器是微不足道的，但从分发中安装 Zookeeper 的完整版本是微不足道的。



Kafka 已经通过 Zookeeper 稳定的 3.4.6 版本进行了广泛的测试，可以从 apache.org 网站在 <http://bit.ly/2sDWSgJ>

独立服务器

下面的示例在/usr/local/Zookeeper 中安装具有基本配置的 Zookeeper，并将其数据存储在/var/lib/Zookeeper 中：

```
# tar -zxvf zookeeper-3.4.6.tar.gz
```

```
# mv zookeeper-3.4.6 /usr/local/zookeeper
# mkdir -p /var/lib/zookeeper
# cat > /usr/local/zookeeper/conf/zoo.cfg << EOF
> tickTime=2000
> dataDir=/var/lib/zookeeper
> clientPort=2181
> EOF
# export JAVA_HOME=/usr/java/jdk1.8.0_51
# /usr/local/zookeeper/bin/zkServer.sh start
JMX enabled by default
Using config: /usr/local/zookeeper/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
#
```

现在，您可以通过连接到客户端端口并发送四个字母的命令 `srvr`:

```
# telnet localhost 2181
Trying ::1...
Connected to localhost.
Escape character is '^]'.
srvr
Zookeeper version: 3.4.6-1569965, built on 02/20/2014 09:09 GMT
Latency min/avg/max: 0/0/0
Received: 1
Sent: 0
Connections: 1
Outstanding: 0
Zxid: 0x0
Mode: standalone
Node count: 4
Connection closed by foreign host.
#
```

安装 Kafka 代理

Kafka 代理接收来自生产者的消息，并将它们存储在由唯一偏移键的磁盘上。Kafka 代理允许使用者按 `topic`、分区和偏移提取消息。Kafka 代理可以通过使用 Zookeeper 直接或间接地相互共享信息来创建 Kafka 群集。Kafka 群集只有一个代理充当控制器。

■ 安装 Kafka 代理

配置好 Java 和 Zookeeper 之后，就可以安装 Apache Kafka 了。Kafka 的最新版本可以在 http://kafka.apache.org/download_loads.html。在发布时，该版本是在 Scala 2.11.0 下运行的 0.9.0.1 版本。

以下示例在 `/usr/local/Kafka` 中安装 Kafka，配置为使用先前启动的 Zookeeper 服务，并存储 `/tmp/Kafka logs` 中存储的消息日志段：

```
# tar -zxvf kafka_2.11-0.9.0.1.tgz
# mv kafka_2.11-0.9.0.1 /usr/local/kafka
```

```
# mkdir /tmp/kafka-logs
# export JAVA_HOME=/usr/java/jdk1.8.0_51
# /usr/local/kafka/bin/kafka-server-start.sh -daemon
/usr/local/kafka/config/server.properties
#
```

Kafka 代理启动，我们就可以通过对集群执行一些简单的操作来验证它是否在工作，创建一个测试主题，生成一些消息，并使用相同的消息。

■ 创建并验证 topic:

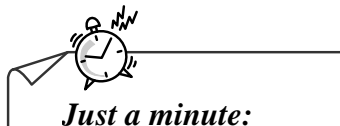
```
# /usr/local/kafka/bin/kafka-topics.sh --create --zookeeper
localhost:2181
--replication-factor 1 --partitions 1 --topic test
Created topic "test".
# /usr/local/kafka/bin/kafka-topics.sh --zookeeper
localhost:2181
--describe --topic test
Topic:test PartitionCount:1 ReplicationFactor:1 Configs:
Topic: test Partition: 0 Leader: 0 Replicas: 0 Isr: 0
#
```

■ 测试主题生成消息

```
# /usr/local/kafka/bin/kafka-console-producer.sh --broker-list
localhost:9092 --topic test
Test Message 1
Test Message 2
^D
#
```

■ 测试使用来自主题的消息

```
# /usr/local/kafka/bin/kafka-console-consumer.sh --zookeeper
localhost:2181 --topic test --from-beginning
Test Message 1
Test Message 2
^C
Consumed 2 messages
#
```



一个 *Kafka* 代理接收来自_____的消息，并将它们存储在以唯一偏移量键控的磁盘上

1. 生产者
2. 消费者
3. *Kafka* 代理
4. 分区

答案:

1. 生产者



活动 2.1: 安装和启动 Kafka。

代理配置

与 Kafka 发行版一起提供的 Broker（代理）配置足以运行独立服务器作为概念证明，但对于大多数安装来说，这是不够的。Kafka 有许多配置选项，可控制设置和调优的所有方面。许多选项可以留给默认设置，因为它们处理 Kafka 代理的调优方面，在您有特定的用例要处理和需要调整这些设置的特定用例之前，这些方面将不适用。

■ General Broker

在为除单个服务器上的独立代理之外的任何环境部署 Kafka 时，应该检查多个代理配置。这些参数处理代理的基本配置，并且必须更改大多数参数才能与其他代理一起在集群中正常运行。

■ broker.id

每个 Kafka 代理都必须有一个整数标识符，该标识符是使用代理.id 配置。默认情况下，此整数设置为 0，但可以是任何值。最重要的是整数在一个 Kafka 簇中必须是唯一的。此号码的选择是任意的，如果需要，可以在代理之间移动，以执行维护任务。一个好的指导原则是将这个值设置为主机固有的值，这样在执行维护时，将代理 ID 号映射到主机并不麻烦。例如，如果主机名包含唯一的编号（例如 host1）。example.com 网站，主机 2。example.com 网站等），这是一个很好的选择经纪人.id 价值观。

■ port

示例配置文件使用 TCP 端口 9092 上的侦听器启动 Kafka。这可以通过更改端口配置参数设置为任何可用端口。请记住，如果选择低于 1024 的端口，则必须作为根启动 Kafka。将 Kafka 作为根运行不建议配置。

■ zookeeper.connect

使用 Zookeeper.connect 配置参数设置用于存储代理元数据的 Zookeeper 的位置。示例配置使用在本地主机上的端口 2181 上运行的 Zookeeper，该端口指定为本地主机：2181

- 主机名、主机名或 Zookeeper 服务器的 IP 地址
- 端口，服务器的客户端端口号
- /路径，可选的 Zookeeper 路径，用作卡夫卡的 chroot 环境

集群。如果省略，则使用根路径。如果指定了 chroot 路径但不存在，则代理将在启动时创建它。

■ log.dirs

Kafka 将所有消息保存到磁盘，这些日志段存储在日志目录配置。这是本地系统上以逗号分隔的路径列表。如果指定了多个路径，代理将以“最少使用”的方式存储分区，其中一个分区的日志段存储在同一路径中。

■ Topic 默认值

Kafka 服务器配置为创建的主题指定了许多默认配置。其中几个参数，包括分区计数和消息保留，可以使用管理工具为每个主题设置。服务器配置中的默认值应设置为适合集群中大多数主题的基线值。一些参数显示在表中并附有说明。

参数	说明
<code>num.partitions</code>	<code>num.partitions</code> 参数确定创建新主题时使用的分区数，主要是在启用自动主题创建时（这是默认设置）。
<code>log.retention.ms</code>	默认值是在配置文件中使用的 <code>log.retention.ms</code> 参数指定的，它被设置为 168 小时或一周。
<code>log.retention.bytes</code>	消息保留值的总字节数是使用 <code>log.retention.bytes</code> 参数设置的，它将应用于每个分区。
<code>log.segment.bytes</code>	日志段已达到 <code>log.segment.bytes</code> 参数指定的大小（默认值为 1 GB），日志段将关闭并打开一个新的日志段。日志段一旦关闭，就可以考虑过期。

默认 Topics 的参数和说明



每个 *Kafka* 代理都必须有一个_____标识符

1. *String*
2. *Integer*
3. *Float*
4. 分区

答案:

2. *Integer*



Kafka 控制台工具

Kafka 提供了命令行工具来管理主题、用户组、消费和发布消息等等。Kafka 控制台脚本对于基于 Unix 和 Windows 平台是不同的。

您可能需要根据您的平台添加扩展。

Linux: 位于 bin/with.sh 扩展名的脚本。

Windows: 脚本位于 bin\windows\ 中，扩展名为.bat。

创建 Kafka 主题

在本节中，我们将看到创建 Kafka topic 以及列表 topic、描述 topic 和更改 topic。

1. 创建 Kafka 主题

我们需要使用以下语法来创建 Kafka topic

```
kafka-topics --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic test
```

上面的语法将创建一个只有一个分区而没有复制的 topic。

2. Describe a topic:

以下语法用于 describe topic

```
kafka-topics --zookeeper localhost:2181 --describe --topic test
```

3. Topics 列表:

您可以使用以下语法获取 topics 列表。

```
kafka-topics --zookeeper localhost:2181 -list
```

4. 更改 Topic:

下面给出的语法用于更改 Kafka 中的 topic

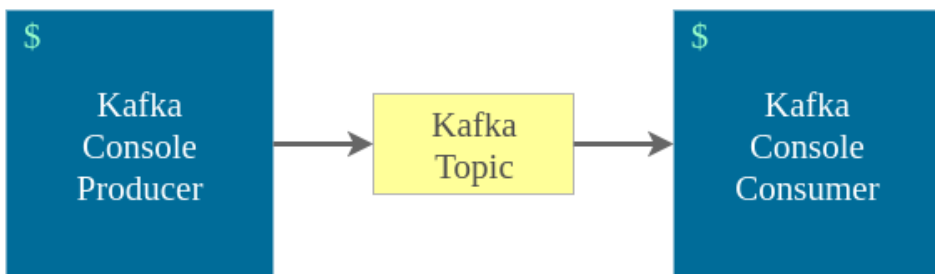
```
# change configuration
kafka-topics --zookeeper localhost:2181 --alter --topic test --
configmax.message.bytes=128000
# add a partition
kafka-topics --zookeeper localhost:2181 --alter --topic test --
partitions 2
```



活动 2.2：创建、列出、描述和删除主题

Kafka CLI 制作人

此工具用于将消息写入 topic。它通常不像控制台使用者那样有用，但当消息采用基于文本的格式时，它很有用。



因此，此工具允许您从命令行生成消息。

- 向 topic 发送简单的字符串消息：

```
kafka-console-producer --broker-list localhost:9092 --topic test
here is a message
here is another message
^D
```

注意：每一新行都是一条新消息，键入 ctrl+D 或 ctrl+C 停止

- 用 key 发送信息：

```
kafka-console-producer --broker-list localhost:9092 --topic test-
topic \
--property parse.key=true \
--property key.separator=,
key 1, message 1
key 2, message 2
null, message 3
^D
```

- 从文件发送消息

```
kafka-console-producer --broker-list localhost:9092 --topic
test_topic < file.log
```

Kafka CLI 使用者

`kafka-console-consumer` 是一个使用者命令行：从 Kafka 主题读取数据并写入标准输出（控制台）。此工具允许您使用主题中的消息来使用旧的使用者实现，将 `---bootstrap-server` 换为 `--zookeeper`。

- **显示简单信息：**

```
kafka-console-consumer --bootstrap-server localhost:9092 --topic test
```

- **使用旧邮件：**

要查看旧邮件，可以使用 `--from-beginning`

- **显示 key-value 信息：**

```
kafka-console-consumer --bootstrap-server localhost:9092 --topic test-  
topic \  
--property print.key=true \  
--property key.separator=,
```

- **kafka-simple-consumer-shell**

此工具允许您使用来自特定分区的消息。偏移量和副本。

有用参数：

- `partition`：要从中使用的特定分区（默认为全部）
- `offset`：起始偏移量。使用 `-2` 从开始使用消息，`-1` 从结尾使用。
- `max-messages`：要打印的邮件数
- `replica`：复制副本，默认为代理头（`-1`）

例子：

```
kafka-simple-consumer-shell \  
--broker-list localhost:9092 \  
--partition 1 \  
--offset 4 \  
--max-messages 3 \  
--topic test-topic
```

从 `topic 测试 topic` 的偏移量 4 开始显示来自分区 1 的 3 条消息

Kafka CLI 用户组

Kafka 消费者属于特定的消费群体。消费者组基本上表示应用程序的名称。为了在消费者组中使用消息，使用“-group”命令。此工具允许您列出、描述或删除消费者组。有关消费者组的信息，请参阅本文

如果仍然使用旧的使用者实现，请将--bootstrap-server 替换为--zookeeper.

- **列出消费者组：**

以下语法用于获取消费者组的列表

```
kafka-consumer-groups --bootstrap-server localhost:9092 --list octopus
```

输出：

```
kafka-consumer-groups --bootstrap-server localhost:9092 --describe --group octopus
GROUP          TOPIC          PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG          OWNER
octopus        test-topic     0          15              15              0            octopus-1/127.0.0.1
octopus        test-topic     1          14              15              1            octopus-2_/127.0.0.1
```

以上输出说明：

- current-offset 是使用者实例最后提交的偏移量
 - log-end-offset 是分区的最高偏移量（因此，求和此列的总和可以得到主题的消息总数）
 - lag 是当前消费者补偿和最高补偿之间的差值
 - owner 是客户端.id（如果未指定，则显示默认值）
- **删除消费者组：**

只有当组元数据存储于 zookeeper（旧使用者 api）中时，删除才可用。使用新的使用者 API，代理处理所有内容，包括元数据删除：当组的最后一个提交偏移量过期时，将自动删除组。

使用以下语法删除使用者组：

```
kafka-consumer-groups --bootstrap-server localhost:9092 --delete --group octopus
```



活动 2.3：基于 CLI 的生产者和消费者

练习题

1. 以下哪个选项是创建主题的正确选项。
 - a. kafkaBroker-topics --create --zookeeper localhost:2181
 - b. kafka-topics --create --zookeeper localhost:2181
 - c. kafka-topic-message --create --zookeeper localhost:2181
 - d. kafka-topics --create --BootsServer localhost:2181
2. 使用 _____ 配置参数设置用于存储代理元数据的 Zookeeper 的位置。
 - a. zookeeper.connect
 - b. log.dirs
 - c. broker.id
 - d. port
3. 使用 _____ 参数设置消息保留值的字节总数。
 - a. log.der.retension
 - b. log.retention.bytes
 - c. retension.der.log
 - d. bytes.retension.log
4. 以下哪一项是显示信息的正确代码。
 - a. kafka-console-consumer --bootstrap-server localhost:9092 --topic test
 - b. kafka-console-consumer --server localhost:9092 --topic test.
 - c. kafka-console-consumer --bootstrap-server localhost:9092 --topics test.
 - d. kafkaBroker-console-consumer --bootstrap-server localhost:9092 --topic test.

5. ____ 是分区的最高偏移量。
- a. log--offset
 - b. log-server-end-offset
 - c. current-offset
 - d. log-end-offset

摘要

在这一章中，你学到了：

- 设置 Kafka 环境.
 - 安装 Java、Kafka 和 Zookeeper 的步骤
- 为什么 Zookeeper 需要 kafka
- 安装 zookeeper 涉及的步骤
- General Broker.
 - 代理.id
 - Port
 - Zookeeper.connect
 - Logs.dir
- Topic 默认值.
 - Num.partition
 - Log.retention.ms
 - Log.segment.bytes.
- Kafka 控制台工具

Kafka 生产者



Kafka 生产者是一个可以充当 Kafka 群集中数据源的应用程序。生产者可以向一个或多个 Kafka topics 发布消息。

Kafka 生产者将记录发送到 topics。这些记录有时被称为消息。生产者为每个 topics 选择要向哪个分区发送记录。生产者可以循环发送记录。生产者可以根据记录的优先级将记录发送到特定分区，从而实现优先级系统。

本章详细论述了 Kafka 生产者

目标

在这一章中，你将学到：

- 📖 Kafka 生产者概述
- 📖 Java 生产者 API
- 📖 创建 Kafka 生产者
- 📖 配置生产者

Kafka 生产者概述

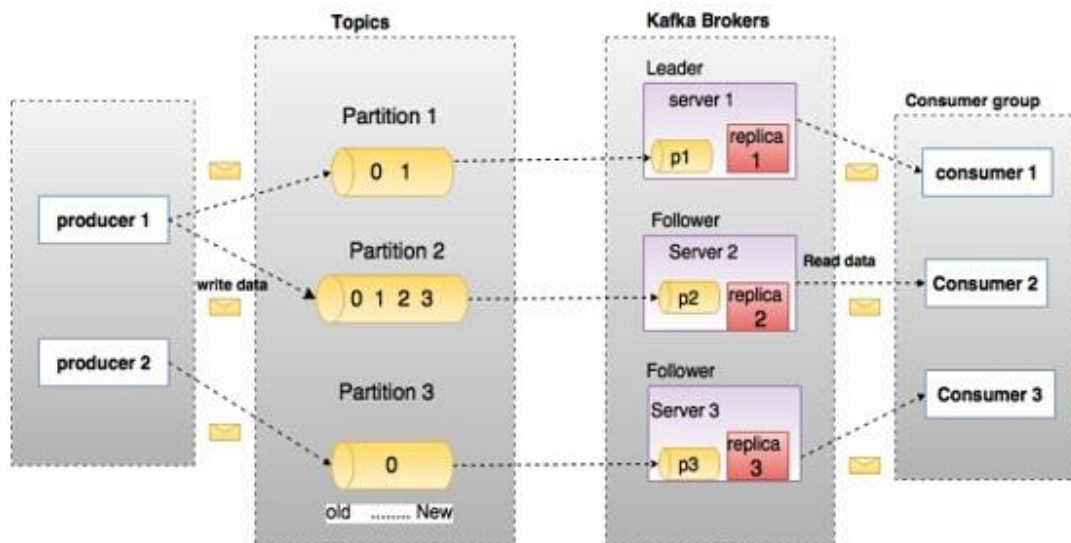
Kafka 生产者用于向卡夫卡写消息。我们需要向 Kafka 写入消息，用于记录用户活动以进行审核或分析、记录指标、存储日志消息、记录来自智能设备的信息、与其他应用程序异步通信、在写入数据库之前缓冲信息等。我们需要在应用程序中确定消息有多重要，我们能否承受丢失消息、重复消息和延迟要求。

Producer 说明

现在，让我们看看 Kafka Producer 是如何工作的，Kafka Producer 的主要角色是获取 Producer 属性，将它们记录为输入，并将它们写入适当的 Kafka 代理。生产者基于分区跨代理序列化、分区、压缩和负载平衡数据。

生产者直接向代理发送数据，代理是分区的领导者，而不需要任何中间的路由层。为了帮助生产者做到这一点，所有的 Kafka 节点都可以在任何给定的时间响应元数据请求，关于哪些服务器是活动的，以及主题分区的领导者在哪里，从而允许生产者适当地指导其请求。

客户机控制它将消息发布到哪个分区。这可以随机进行，实现一种随机的负载平衡，也可以通过某种语义分区函数来实现。例如，如果选择的键是用户 id，那么给定用户的所有数据都将被发送到同一个分区。



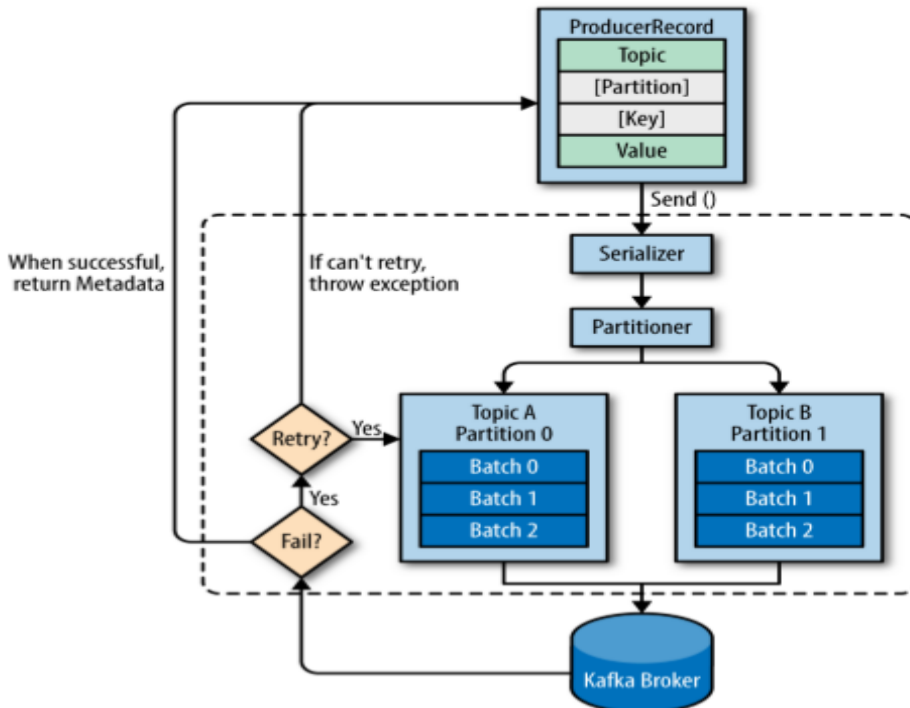
Kafka Producer

Kafka 代理还通过复制为我们提供可靠性和数据保护。如果代理失败，那么分配给该代理的所有分区都将不可用。为了解决这个问题，有一个副本的概念，即每个分区的一个副本。您可以指定一个分区有多少个副本。在给定的时间点上，所有副本都与原始分区相同

Kafka 生产者组件概述

Kafka 生产者组件如下

- **Topic:** 主题是记录发布到的类别或提要名称。Kafka 中的主题始终是多订阅者；也就是说，一个主题可以有零个、一个或多个消费者订阅写入其中的数据。
- **分区:** Kafka 主题被划分为多个分区。分区允许您通过在多个代理之间拆分特定主题中的数据来并行化主题。每个分区都可以放在单独的机器上，以允许多个使用者并行地从一个主题中读取数据。
- **序列化器:** 将对象转换为字节流以进行传输的过程称为序列化。Apache Kafka 提供了我们可以轻松发布和订阅记录流的功能。因此，我们可以灵活地创建我们自己的自定义序列化程序以及重新序列化，这有助于使用它传输不同的数据类型。
- **代理:** 代理是一个无状态的 Kafka 服务器。卡夫卡集群由多个卡夫卡经纪人组成。Kafka 生产者和消费者不直接交互，而是使用 Kafka 服务器作为代理或代理来交换消息服务。Kafka 集群通常由多个代理组成，以保持负载平衡。

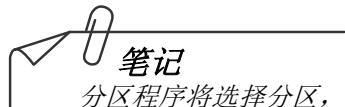


Kafka 生产者组件图

我们通过创建一个生产记录开始向 Kafka 生成消息，其中必须包含我们要将记录发送到的主题和一个值。我们还可以选择指定一个键和/或一个分区。一旦我们发送了生产记录，生产者要做的第一件事就是将 key 和 value 对象序列化到字节数组，这样它们就可以通过网络发送了。接下来，将数据发送到分区程序。如果我们在生产记录中指定了一个分区，那么分区器不会做任何事情，只返

回我们指定的分区。如果我们没有，分区程序将为我们选择一个分区，通常基于生产记录 Key。一旦选择了一个分区，生产者就知道记录将转到哪个主题和分区。然后它将记录添加到一批记录中，这些记录也将被发送到相同的主题和分区。另一个线程负责将这些批记录发送给适当的 Kafka 代理。

当代理收到消息时，它会发送响应。主题、分区和分区内记录的偏移量。如果代理无法写入消息，它将返回错误。当创建者收到错误时，它可以重试再发送几次消息，然后再放弃并返回错误。



笔记

分区程序将选择分区，通常基于生产者记录键。相同的 Key 始终转到同一分区。
如果未提及键，则使用循环方法。

Java 生产者 API

让我们首先了解为 Kafka 集群编写基于 Java 的基本生产者所导入的重要类：

■ KafkaProducer

`KafkaProducer<K,V>`：此类可在 `org.apache.kafka.客户端.生产者包` 中提供。这是一个泛型类，我们需要在其中指定参数的类型；`K` 和 `V` 分别指定分区键和消息值的类型。将记录发布到 Kafka 群集的 Kafka 客户端。

类：

- `KafkaProducer (Map<String,Object> configs)`：通过提供一组键值对作为配置来实例化生产者。
- `KafkaProducer (Map<String,Object> configs, Serializer<K> keySerializer, Serializer<V> valueSerializer)`：通过提供一组键值对作为配置、键和值序列化器来实例化生产者。
- `KafkaProducer(Properties properties)`：通过提供一组键值对作为配置来实例化生产者。
- `KafkaProducer(Properties properties, Serializer<K> keySerializer, Serializer<V> valueSerializer)`：通过提供一组键值对作为配置、键和值序列化器来实例化生产者。

方法：

- `abortTransaction()`：中止正在进行的事务。
- `beginTransaction()`：应在每个新事务开始前调用。
- `close()`：关闭 生产者。
- `close(Duration timeout)`：此方法等待直到超时，以便生产者完成所有未完成请求的发送。
- `commitTransaction()`：提交正在进行的事务。
- `flush()`：调用此方法可使所有缓冲记录立即可用于发送（即使逗留时间.ms 大于 0）并在与这些记录关联的请求完成时阻止。
- `send(ProducerRecord<K,V> record)`：异步向 topic 发送记录。
- `send(ProducerRecord<K,V> record, Callback callback)`：异步向 topic 发送记录，并在确认发送时调用提供的回调。

■ ProducerRecord

`ProducerRecord<K,V>`：要发送到 Kafka 的键/值对。这包括要向其发送记录的主题名称、可选分区编号以及可选键和值。

如果指定了有效的分区号，则在发送记录时将使用该分区。如果未指定分区但存在键，则使用键的哈希选择分区。如果两者都没有键或分区，则将按循环方式分配分区。

记录还具有关联的时间戳。如果用户未提供时间戳，则创建者将用其当前时间标记记录。Kafka 最终使用的时间戳取决于为主题配置的时间戳类型。实际使用的时间戳将在记录气象中返回给用户

- 如果 topic 配置使用 CreateTime，则代理将使用生产记录中的时间戳。
- 如果 topic 配置使用 LogAppendTime，则当代理将消息附加到其日志时，代理将用代理本地时间覆盖生产者记录中的时间戳。

类:

- `ProducerRecord(String topic, Integer partition, K key, V value)`: 创建一个记录发送到指定 topic 和分区
- `ProducerRecord(String topic, Integer partition, K key, V value, Iterable<Header> headers)`: 创建一个记录发送到指定 topic 和分区
- `ProducerRecord(String topic, Integer partition, Long timestamp, K key, V value)` : 创建一个具有指定时间戳的记录以发送到指定的 topic 和分区
- `ProducerRecord(String topic, Integer partition, Long timestamp, K key, V value, Iterable<Header> headers)` : 创建一个具有指定时间戳的记录以发送到指定的 topic 和分区
- `ProducerRecord(String topic, K key, V value)`: 创建一个要发送给 Kafka 的记录
- `ProducerRecord(String topic, V value)`: 创建无键记录

方法:

- `Headers headers()`
- `K key()`: 键 (如果未指定键, 则为 null)
- `V value()`:
- `Integer partition()`: 记录将被发送到的分区 (如果没有指定分区, 则为 null)
- `Long timestamp()`:
- `String topic()` : 将此记录发送到的主题

■ ProducerConfig

此类在 `org.apache.kafka.客户端.生产者` 中提供。它有助于为 Kafka 生产者设置配置。

部分字段列表如下

修改器和类型	字段	说明
<code>static java.lang.String</code>	<code>ACKS_CONFIG</code>	<code>acks</code>
<code>static java.lang.String</code>	<code>BOOTSTRAP_SERVERS_CONFIG</code>	<code>bootstrap.servers</code>
<code>static java.lang.String</code>	<code>BUFFER_MEMORY_CONFIG</code>	<code>buffer.memory</code>
<code>static java.lang.String</code>	<code>KEY_SERIALIZER_CLASS_CONFIG</code>	<code>key.serializer</code>

static java.lang.String	MAX_REQUEST_SIZE_CONFIG	max.request.size
static java.lang.String	PARTITIONER_CLASS_CONFIG	partitioner.class
static java.lang.String	RECEIVE_BUFFER_CONFIG	receive.buffer.bytes
static java.lang.String	REQUEST_TIMEOUT_MS_CONFIG	request.timeout.ms
static java.lang.String	RETRIES_CONFIG	retries
static java.lang.String	RETRY_BACKOFF_MS_CONFIG	retry.backoff.ms
static java.lang.String	SEND_BUFFER_CONFIG	send.buffer.bytes
static java.lang.String	TRANSACTION_TIMEOUT_CONFIG	transaction.timeout.ms
static java.lang.String	VALUE_SERIALIZER_CLASS_CONFIG	value.serializer

创建 Kafka 生产者

要创建 Kafka producer，首先需要设置属性，然后在 `ProducerRecord` 的帮助下发送消息。

Kafka 生产者的属性列表

以下是 Kafka 生产者使用的所有属性的列表

- **key.serializer class:** 实现的键的序列化程序类
`org.apache.kafka.common.serialization.Serializer` 接口。
- **value.serializer class:** 实现的值的序列化程序类
`org.apache.kafka.common.serialization.Serializer` 接口。
- **acks string:** 生产者要求领导者在考虑完成请求之前收到的确认数。这将控制发送的记录持久性。
允许以下设置：
 - **acks=0** 如果设置为零，则生产者将不等待来自服务器的任何确认。记录将立即添加到套接字缓冲区并视为已发送。无法保证服务器在这种情况下已收到记录，并且重报配置不会生效（因为客户端通常不会知道任何故障）。每项记录的偏移量将始终设置为 -1。
 - **acks=1** 这意味着领导者会将记录写入本地日志，但不会等待所有追随者的完全确认。在这种情况下，如果领导者在确认记录后，但在跟随者复制之前立即失败，则记录将丢失。
 - **acks=all** 这意味着领导者将等待所有的同步副本集来确认记录。这可确保只要至少一个同步副本保持活动状态，记录就会丢失。这是最强的保证。这等效于 **acks=-1** 设置。
- **bootstrap.servers list:** 用于建立与 Kafka 群集的初始连接的主机/端口对的列表。客户端将使用所有服务器，而不管此处指定哪些服务器进行引导 - 此列表仅影响用于发现完整服务器集的初始主机。此列表应以 `host1: port1, host2: port2` 格式, 由于这些服务器仅用于初始连接以发现完整的群集成员身份（可能会动态更改），因此此列表不需要包含完整的服务器集（不过，如果服务器关闭，您可能需要多个服务器）
- **buffer.memory:** 生产者可用于缓冲等待发送到服务器的记录的内存总字节数。如果记录的发送速度比将记录传递到服务器的速度要快，则创建者将阻止 `max.block.ms` 之后，它将引发异常。
此设置应大致对应于创建者将使用的总内存，但不是硬绑定，因为并非所有内存使用生产者用于缓冲。一些额外的内存将用于压缩（如果启用压缩）以及维护飞行中的请求。
- **compression.type:** 生产者生成的所有数据的压缩类型。默认值为无（即无压缩）。有效值为无、gzip、snappy、lz4 或 zstd。压缩是全批数据，因此批处理的有效性也会影响压缩比（更多的批处理意味着更好的压缩）。

- `max.block.ms long`:配置控制 `KafkaProducer.send()` 和 `KafkaProducer.partitionsFor()` 将阻止多长时间。这些方法可以被阻止, 因为缓冲区已满或元数据不可用。用户提供的序列化器或分区器中的阻止不会计入此超时。
- `max.request.size int`: 请求的最大大小 (字节)。此设置将限制生产者在单个请求中发送的记录批数, 以避免发送大量请求。这也有效地限制了最大未压缩记录批大小。请注意, 服务器对记录批大小有自己的上限 (如果启用了压缩, 则在压缩之后), 这可能与此不同。
- `request.timeout.ms int` : 配置控制客户机等待请求响应的最长时间。如果在超时时间过去之前没有收到响应, 客户端将在必要时重新发送请求, 如果重试已用尽, 则请求将失败。这个应该大于副本. 滞后时间. 最大毫秒 (代理配置) 以减少由于不必要的生产者重试而导致消息重复的可能性。

向 Kafka 发送消息

消息可以通过两种方式发送, 具体取决于您是否需要确认

- 同步发送消息: 消息发送, 生产者等待第一条消息的确认以发送第二条消息。

```
ProducerRecord<String, String> record = new ProducerRecord<>("Course",
"Kafka", "India");
try { producer.send(record).get(); //line 1}
catch (Exception e)
{ e.printStackTrace(); // line2 }
```

在第 1 行中, 我们使用 `Future.get()` 等待 Kafka 的答复。如果未成功将记录发送到 Kafka, 此方法将引发异常。如果没有错误, 我们将获取一个 `RecordMetadata` 对象, 我们可以用它来检索消息写入的偏移量。

在第 2 行中, 如果在向 Kafka 发送数据之前出现任何错误, 则如果 Kafka 经纪人返回了不可重复的异常, 或者如果我们用尽了可用的重试, 我们将遇到异常。在这种情况下, 我们只是打印我们遇到的任何异常。

- 异步发送消息

有时我们不需要等待回复, 然后发送下一条消息。如果要分析有多少消息完全失败, 那么我们可以引发异常、记录错误或将消息写入“错误”文件。

为了异步发送消息并仍然处理错误方案, 生产者支持在发送记录时添加回调。下面是我们如何使用回调的示例:

```
private class NIITProducerCallback implements Callback { //line1
@Override
public void onCompletion(RecordMetadata recordMetadata, Exception e)
```

```
{
if (e != null) {          e.printStackTrace();    //line 2      }
}
}
ProducerRecord<String, String> record = new ProducerRecord<>("Course",
"Kafka", "India"); //line 3
producer.send(record, new NIITProducerCallback()); //line 4
```

在第 1 行中：要使用回调，您需要一个实现 `org.apache.kafka` 的类。客户端.生产者.回调接口，具有单个功能-完成（）

第 2 行：如果 Kafka 返回错误，则“完成（）”将有一个非努力异常。在这里，我们通过打印“处理”它，但生产代码可能有更强大的错误处理功能。

第 3 行：记录与以前相同。

第 4 行：在发送记录时，我们会传递一个回调对象。

配置 Kafka Producer

以下是配置生产者发送消息的步骤

1. 使用属性创建生产者配置对象

```
Properties props = new Properties();
props.setProperty("bootstrap.servers", "localhost: 9092");
props.put("acks", "all");
props.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
```

或

```
Properties props = new Properties();
props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost: 9092");
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
LongSerializer.class.getName());
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());
```

2. 使用我们刚才提供的设置创建生产者对象:

```
Producer<String, String> producer = new KafkaProducer<>(props);
```

3. 创建要推送到 Kafka 主题的消息。

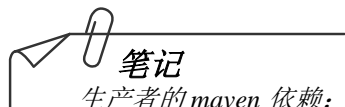
```
ProducerRecord<Integer, String> record = new ProducerRecord<String,
String>("topicName", "key1" "value1");
```

4. 调用发送方法

```
producer.send(record);
```

5. 发送消息后,调用 close 方法

```
producer.close();
```



笔记

生产者的 maven 依赖:

```
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-clients</artifactId>
  <version>2.5.0</version>
</dependency>
```


练习题

1. Kafka 生产者用于向 Kafka 集群写入消息
 - a. 正确
 - b. 错误
2. 在 Kafka 生产者属性中未指定以下哪个选项？
 - a. bootstrap.servers
 - b. key.serializer
 - c. value.serializer
 - d. topic.Name
3. 在“生产者记录”中未指定以下选项之一。
 - a. Topic
 - b. Broker
 - c. Key
 - d. Value
4. 如何异步发送消息？
 - a. `producer.send(record, new NIITProducerCallback());`
 - b. `producer.send(record);`
 - c. `producer.send(record, “topic”);`
 - d. `producer.send(record, “key”);`
5. 如果没有 key 那么使用的是 _____ 方法？
 - a. Hashing
 - b. Round robin
 - c. Alternate partition is used
 - d. Default partition

摘要

在本章中，您学到了：

- Kafka 创建者是一个应用程序中的数据源。
- Kafka 生产者组件如下
 - Topic
 - Partition
 - Serializer
 - Broker
- Java 生产者 API
 - KafkaProducer
 - ProducerRecord
 - ProducerConfig
- Kafka 生产者属性列表
 - acks
 - bootstrap.servers
 - buffer.memory
 - key.serializer
 - max.request.size
 - partitioner.class
 - receive.buffer.bytes
 - request.timeout.ms
 - retries
 - retry.backoff.ms
 - send.buffer.bytes
 - transaction.timeout.ms
- 发送消息
 - 同步
 - 异步： 使用回调
- 配置 Kafka 生产者

Kafka 消费者



Kafka 消费者是一个可以读取 Kafka 集群中数据的应用程序。消费者可以使用来自一个或多个 Kafka topics 的消息。

Kafka 消费者用于使用 Kafka 数据。Kafka 消费者的主要作用是利用 Kafka 连接和消费者属性从相应的 Kafka 代理读取记录。并发应用程序消耗、偏移管理、交互语义等的复杂性由消费者 API 处理

本章详细论述了 Kafka 消费者

目标

在本章中，您将学到：

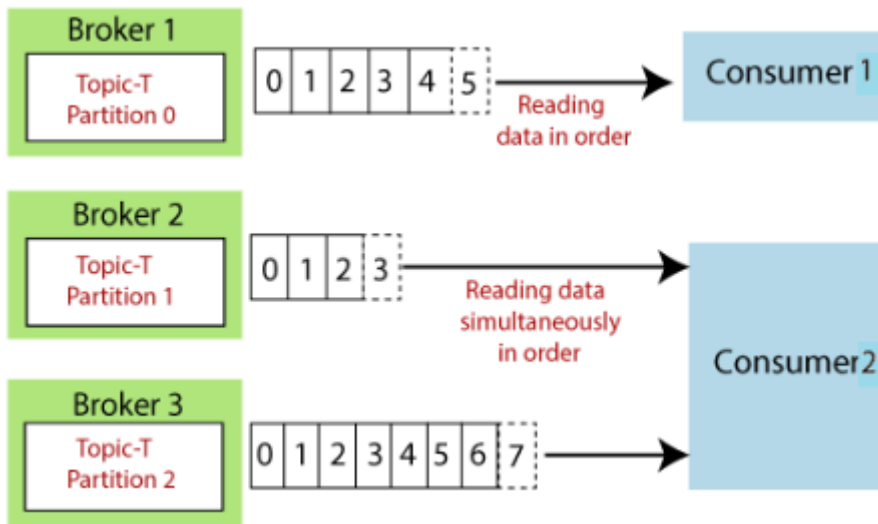
- 📖 Kafka 消费者概述
- 📖 Kafka 消费者 APIs
- 📖 创建 Kafka 消费者
- 📖 配置 消费者
- 📖 读取消息
- 📖 Kafka 消费者属性列表

Kafka 消费者概述


消费者是使用 Kafka 生产者发布的消息并处理从中提取的数据的应用程序。与生产者一样，消费者在本质上也可能有所不同，例如进行实时或近实时分析的应用程序、具有 NoSQL 或数据仓库解决方案的应用程序、后端服务、Hadoop 的消费者或其他基于订户的解决方案。这些消费者也可以用不同的语言实现，比如 Java、C 和 Python。

消费者和消费者组

消费者在 topic 的帮助下使用或读取 Kafka 集群中的数据。消费者知道哪个代理应该读取数据。消费者以有序的方式读取每个分区内的数据。这意味着消费者不应该在从偏移量 0 读取数据之前从偏移量 1 读取数据。此外，消费者可以轻松的同时从多个代理读取数据。i、e. 两个消费者，即消费者 1 和消费者 2 正在读取数据。消费者 1 正在按顺序从代理 1 读取数据。另一方面，消费者 2 按顺序同时从代理 2 和代理 3 读取数据。



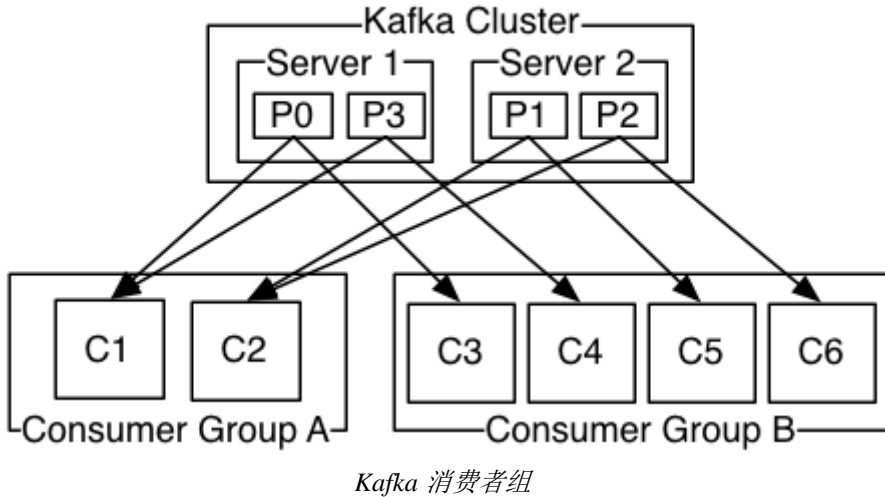
Kafka 消费者

 **笔记**
代理 2 和 3 的偏移 0-n 没有连接。它们是独立的。

消费者记得停止阅读的偏移量。

消费者组

消费者组可以描述为单个逻辑消费者，它订阅了指向应用程序的一组 topic。所有 topic 的分区分配给组中的物理消费者，以便每个分区只分配给一个消费者（单个消费者可以分配多个分区）。属于同一组的单个消费者可以以分布式方式在不同的主机上运行。

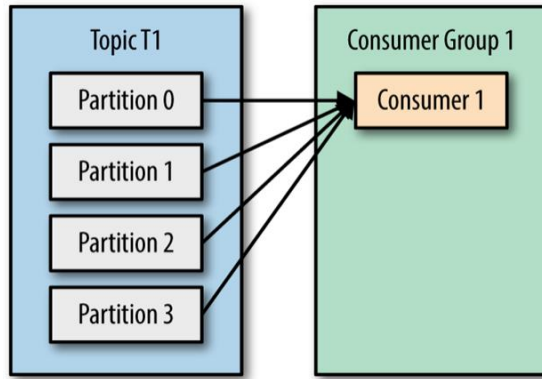


消费者组通过 `group.id`。若要使特定客户端实例成为消费者组的成员，只需分配组即可 `group.id` 对于此客户端，通过客户端的配置：

```
Properties props = new Properties();  
props.put("group.id", "groupName");
```

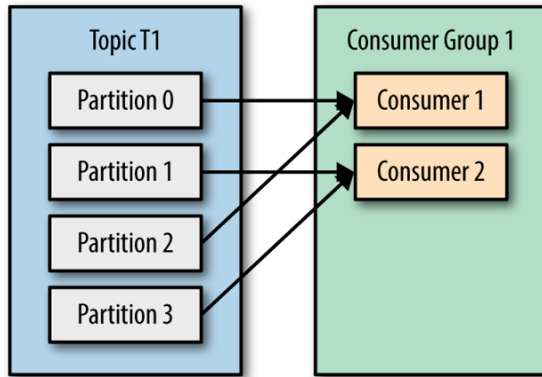
因此，所有连接到同一个 Kafka 集群并使用相同的 `group.id` 形成一个消费组。消费者可以随时离开群组，新的消费者可以随时加入群组。对于这两种情况，都会触发所谓的重新平衡，并将分区重新分配给消费者组，以确保每个分区都由组内的一个消费者处理。一个卡夫卡消费者形成了一个以自己为单一成员的消费组。消费者组中的每个消费者都会处理记录，而该群组中只有一个消费者会取得相同的记录。消费者组负载平衡记录处理中的消费者。消费者组每个分区都有自己的偏移量。

消费者组需要从生产者编写大量消息而单个消费者无法实时读取所有消息的 topic 来扩展消费。让我们借助图表来理解



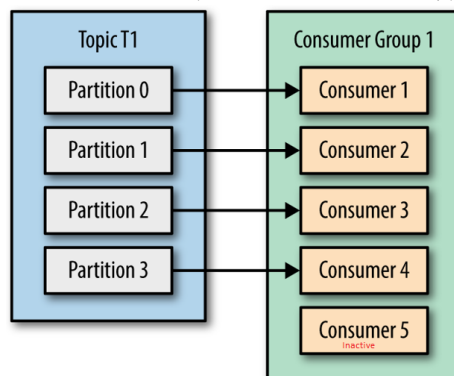
Consumer group Scenario1

一个消费者 C1 从 topicT1 的所有分区读取



消费者组场景 2

现在在 consumer 组中又添加了一个 Consumer2，这样就可以划分工作了。

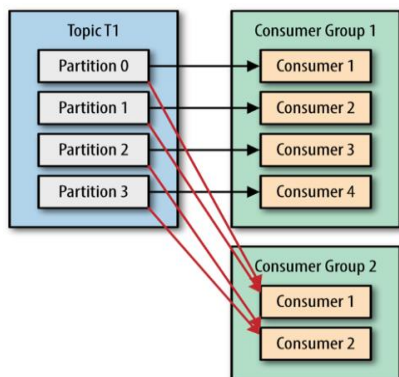


消费者组场景 3

现在我们在消费者组中又增加了 3 个消费者。但是我们只有 4 个分区，所以消费者 5 将不活动。它建议创建具有大量分区的 topic，它允许在负载增加时添加更多的消费者。请记住，在一个 topic 中添加的消费者超过您的分区是没有意义的。

单个消费者无法以数据流入 topic 的速率处理数据，添加更多的消费者可以通过让每个消费者只拥有分区和消息的子集来分担负载，这是我们的主要伸缩方法。

仅仅添加消费者来扩展单个应用程序是不够的，有多个应用程序需要从同一 topic 读取数据是非常常见的。事实上，Kafka 的一个主要设计目标是使为 Kafkatopic 生成的数据可用于整个组织中的许多用例。在这些情况下，我们希望每个应用程序获取所有消息，而不仅仅是一个子集。要确保应用程序获取 topic 中的所有消息，请确保应用程序具有自己的消费者组。卡夫卡在不降低性能的情况下扩展到大量的消费者和消费组。



消费者组场景 4

添加消费者组以确保不遗漏任何消息

总结，以上学习

- 为需要来自一个或多个 topic 的所有消息的每个应用程序创建一个新的消费者组。
- 继续向现有的消费者组添加消费者，以缩放对来自 topic 的讯息的读取和处理，因此群组中的每个额外消费者只会取得讯息的子集。

消费者偏移量

消费者偏移量如下

- Kafka 存储消费者组读取的偏移量
- 在 Kafka 中实时提交的偏移量名为 `__consumer_offsets`
- 当一个组中的消费者处理了从 Kafka 接收到的数据时，它应该提交偏移量
- 如果一个消费者结束，它将能够从它停止的地方而不是从一开始就读回。



Delivery semantics:

提交偏移量的选择取决于消费者。提交偏移量就像读者在阅读书籍或小说时使用的书签。在 Kafka 中，使用了以下三种 Delivery semantics:

- 最多一次：一旦消费者收到消息，就提交偏移量。但是，如果处理错误，消息将丢失，并且消费者将无法再次阅读。
- 至少一次：在处理消息之后提交偏移量。如果处理出错，则消费者将再次读取消息。它会导致消息的重复处理。因此，它需要一个系统成为幂等系统。
- 仅一次：仅使用 Kafka Streams API，就可以实现 Kafka 到 Kafka 工作流的偏移。为了实现 Kafka 对外部系统的偏移，我们需要使用一个幂等消费者。

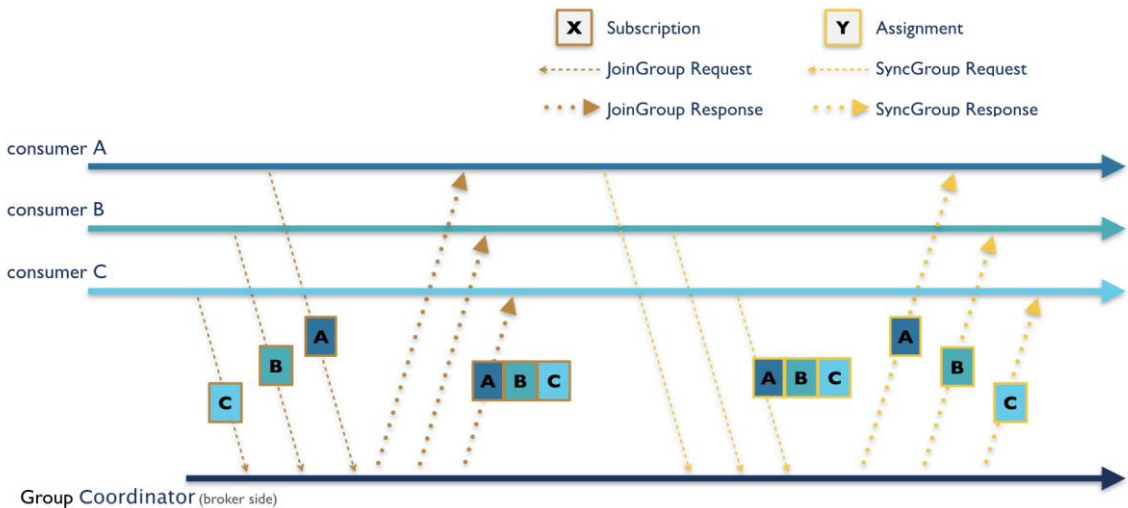
消费者组和分区再平衡

在消费者组中，Kafka 在某些事件将分区的所有权从一个消费者更改到另一个消费者。更改消费者的分区所有权的过程称为重新平衡。在重新平衡期间，消费者不能使用消息，因此再平衡基本上是整个消费者组不可用的一个短窗口。重新平衡很重要，因为它们为消费者组提供了高可用性和可扩展性（使我们能够轻松安全地添加和删除消费者），但在正常情况下，它们是相当不可取的。重新平衡发生在以下事件：

- 扩大：新的消费者被添加到消费者组中。新的消费者开始使用另一个消费者先前使用的分区中的讯息。
- 缩小：从消费者组中删除消费者。另一个消费者开始使用已删除分区中的消息。
- 新分区添加到 topics。
- 应用程序崩溃。
- 消费者关闭：新消费者开始使用另一个消费者先前使用的分区中的讯息。
- 当消费者被组协调器视为 DEAD 时。这可能发生在崩溃之后，或者当消费者忙于长时间运行的处理时，这意味着消费者在配置的会话间隔内没有同时向组协调器发送检测信号。
- 消费者订阅任何 topics。
- 如果您订阅了尚未创建的 topic，则在该 topic 创建后将触发重新平衡。同样，如果你订阅的 topic 被删除。

将分区分配给消费者的过程

当消费者想要加入一个组时，它向组协调器发送 `JoinGroup` 请求。第一个加入该组的消费者成为组长。领导者从组协调器接收组中所有消费者的列表（这将包括最近发送心跳信号的所有消费者，因此这些消费者被认为是活动的），并负责将分区的子集分配给每个消费者。它使用 `PartitionAssignor` 的实现来决定由哪个消费者处理哪些分区。Kafka 有两个内置的分区分配策略，我们将在配置部分更深入地讨论这些策略。在决定分区分配之后，消费者负责人将分配列表发送给 `GroupCoordinator`，`GroupCoordinator` 将此信息发送给所有消费者。每个消费者只能看到自己的分配 `leader` 是唯一拥有组中消费者及其分配的完整列表的客户机进程。每次重新平衡发生时，此过程都会重复。



加入和同步组请求

两个超时属性参与重新平衡的过程。

- Zookeeper 会话超时如果消费者在这段时间内无法检测到 Zookeeper，则认为它已死亡，将发生重新平衡。默认值是 6000 毫秒。这是一直测试的，用于驱逐坏节点。
- 重新平衡 `.backoff.ms * rebalance.max`。重试次数这是再平衡阶段允许的最大窗口，在这个阶段，客户不会从 Kafka 那里读到任何东西。
- 有时会出现 `ConsumerRebalanceFailedException` 异常（消费者不平衡失败异常）。这是由于两个消费者尝试拥有同一 topic 分区时的冲突。
 - 如果您的消费者订阅了许多 topic，而您的 ZK 服务器正忙，这可能是由于消费者没有足够的时间查看同一组中所有消费者的一致视图造成的。如果是这种情况，请尝试增加重新平衡 `.max.rebalance.backoff.ms`。
 - 另一个原因可能是其中一位消费者被硬杀。在再平衡期间，其他消费者不会意识到消费者在一 `zookeeper.session.timeout.ms` 消失。在这种情况下，请确保重新平衡 `.max.rebalance.backoff.ms = zookeeper.session.timeout.ms`。



笔记

在重新平衡的过程中，整个消费者组在短时间内变得不可用。

Java 消费者 API

有两种类型的消费者 API

高级 API

高级消费者 API 是围绕消费者组的逻辑概念构建的，它支持以下内容

- 每个 topic 每个分区的偏移量管理（自动读取 Zookeeper 中消费者组的最后一个偏移量），
- 代理故障转移，以及添加或减去分区和消费者时。
- 负载平衡（当分区和用户增加或减少时，Kafka 自动执行负载平衡）
- 整合 API：新的消费者结合了旧的“simple”和“high-level”消费者客户端的功能，提供了团队协作和较低级别的访问权限，以构建自己的消费策略。
- 减少依赖性：新的消费者是用纯 Java 编写的。它不依赖 Scala 运行时或 Zookeeper，这使得它成为一个更轻量级的库，可以包含在您的项目中。
- 更好的安全性：Kafka 0.9 中实现的安全扩展仅受新消费者的支持。

API 列表

- `KafkaConsumer<K,V>` : 在 `org.apache.kafka.clients.consumer.KafkaConsumer<K,V>` 包里。使用 Kafka 集群中的记录的客户端。这个客户端透明地处理 Kafka 代理的故障，并且透明地适应它在集群中获取的 topic 分区迁移。此客户端还与代理进行交互，以允许用户组使用消费者组来负载平衡消费。消费者不是线程安全的

类如下：

- `KafkaConsumer(java.util.Map<java.lang.String, java.lang.Object> configs)` : 通过提供一组键值对作为配置来实例化消费者。
- `KafkaConsumer(java.util.Map<java.lang.String, java.lang.Object> configs, Deserializer<K> keyDeserializer, Deserializer<V> valueDeserializer)` : 通过提供一组键值对作为配置、一个键和一个值反序列化器来实例化消费者。
- `KafkaConsumer(java.util.Properties properties)` : 通过提供属性对象作为配置来实例化消费者。

- `KafkaConsumer(java.util.Properties properties, Deserializer<K> keyDeserializer, Deserializer<V> valueDeserializer)` : 通过提供属性对象作为配置、键和值反序列化器来实例化消费者。

方法:

- `public void seek(TopicPartition partition, long offset)`: 实例化后的消费者在 `poll()` 方法中获取当前偏移量值。
- `public void resume()`: 该方法恢复暂停的分区。
- `public void wakeup()` : 唤醒消费者。
- `public java.util.Set<TopicPartition> assignment()`: 获取消费者当前分配的分区分集。
- `public String subscription()`: 为了订阅给定的 topic 列表以获得动态分配的分区
- `public void subscribe(java.util.List<java.lang.String> topics, ConsumerRebalanceListener listener)`: 订阅给定的 topic 列表以获得动态分配的分区。
- `public void unsubscribe()`: 从给定的分区列表中取消订阅 topic。
- `public void subscribe(java.util.List<java.lang.String> topics)`: 为了订阅给定的 topic 列表以获得动态分配的分区。如果给定的 topic 列表为空, 则与 `unsubscribe()` 相同。
- `public void subscribe(java.util.regex.Pattern pattern, ConsumerRebalanceListener listener)`: 这里参数模式是指正则表达式格式的订阅模式, 侦听器参数从订阅模式中获取通知。
- `public void assign(java.util.List<TopicPartition> partitions)`: 为客户手动分配分区列表。
- `Public ConsumerRecords<K,V> poll(java.time.Duration timeout)` : 获取使用订阅/分配 API 之一指定的 topic 或分区的数据。
- `Public ConsumerRecords<K,V> poll(long timeoutMs)` : 自 2.0 起已弃用。
- `public void commitSync()`: 为了提交在最后一次 `poll()` 中为所有订阅的 topic 和分区列表返回的偏移量。同样的操作也应用于 `commitAsyn()`。

- `ConsumerRecord<K,V>`: 从 Kafka 接收的键/值对。它还包括一个 topic 名和一个从中接收记录的分区号、指向 Kafka 分区中的记录的偏移量以及由相应的 `ProducerRecord` 标记的时间戳。

类如下

- `ConsumerRecord(java.lang.String topic, int partition, long offset, K key, V value)` : 创建要从指定 topic 和分区接收的记录（在消息格式支持的时间戳之前以及在公开序列化元数据之前，提供与 Kafka 0.9 兼容的功能）。
- `ConsumerRecord(java.lang.String topic, int partition, long offset, long timestamp, org.apache.kafka.common.record.TimestampType timestampType, long checksum, int serializedKeySize, int serializedValueSize, K key, V value)` : 创建要从指定 topic 和分区接收的记录（提供此记录是为了与 Kafka 0.10 兼容，然后才支持消息格式的标头）。
- `ConsumerRecord(java.lang.String topic, int partition, long offset, long timestamp, org.apache.kafka.common.record.TimestampType timestampType, java.lang.Long checksum, int serializedKeySize, int serializedValueSize, K key, V value, Headers headers)` : 创建要从指定 topic 和分区接收的记录
- `ConsumerRecord(java.lang.String topic, int partition, long offset, long timestamp, org.apache.kafka.common.record.TimestampType timestampType, java.lang.Long checksum, int serializedKeySize, int serializedValueSize, K key, V value, Headers headers, java.util.Optional<java.lang.Integer> leaderEpoch)` : 创建要从指定 topic 和分区接收的记录。

方法:

- `Headers headers()` : 标题
- `K key()` : 键（如果未指定键，则为 null）
- `java.util.Optional<java.lang.Integer> leaderEpoch()` : 获取记录的 leaderEpoch（如果可用）
- `long offset()` : 记录在相应的 Kafka 分区中的位置。

- `int partition()` : 接收此记录的分区
 - `int serializedKeySize()` : 序列化、未压缩 Key 的大小（以字节为单位）。
 - `int serializedValueSize()` : 序列化、未压缩的 Value 的大小（以字节为单位）。
 - `long timestamp()` : 此记录的时间戳
 - `org.apache.kafka.common.record.TimestampType timestampType()` : 此记录的时间戳类型
 - `V value()` : 获取值
- **ConsumerRecords** : 它是消费者记录的容器。为了保留特定 topic 的每个分区的消费者记录列表，我们使用此 API。

构造函数如下所示

- `public ConsumerRecords(java.util.Map<TopicPartition, java.util.List<ConsumerRecord>K, V>>> records)` : TopicPartition 返回特定 topic 的分区映射。Records 返回 ConsumerRecord 的列表。

方法:

- `public int count()` : 所有 topic 的记录数。
- `public Set partitions()` : 包含此记录集中数据的分区集（如果未返回任何数据，则该集为空）。。
- `public Iterator iterator()` : 使用循环器可以循环访问集合、获取或删除元素。
- `public List records()` : 获取给定分区的记录列表。

- **ConsumerConfig**: 消费者 API 有助于设置配置属性。您可以在第 4.19 页上看到属性列表
- **ConsumerConnector**: Kafka 提供了 **ConsumerConnector** 接口（接口 **ConsumerConnector**），该接口由 **ZookeeperConsumerConnector** 类进一步实现（`kafka.javaapi.consumer.ZookeeperConsumerConnector`）。这个类负责消费者与 ZooKeeper 的所有交互。

方法

- `void commitOffsets()` : 提交此连接器连接的所有代理分区的偏移量。
 - `java.util.Map<java.lang.String, java.util.List<kafka.consumer.KafkaStream<byte[], byte[]>>>`
`createMessageStreams(java.util.Map<java.lang.String, java.lang.Integer> topicCountMap)`
 - `<K,V> java.util.List<kafka.consumer.KafkaStream<K,V>>`
 - `createMessageStreamsByFilter(kafka.consumer.TopicFilter topicFilter, int numStreams, kafka.serializer.Decoder<K> keyDecoder, kafka.serializer.Decoder<V> valueDecoder)` : 创建包含类型为 T 的消息的 **MessageAndTopicStreams** 列表。
 - `void setConsumerRebalanceListener(ConsumerRebalanceListener listener)` : 消费者再平衡时要执行的消费者再平衡侦听器中的线。
 - `void shutdown()` : 关闭连接器
-
- **KafkaStreams** : 包装内提供 `org.apache.kafka.streams`。一种 Kafka 客户机，允许对来自一个或多个输入 topic 的输入执行连续计算，并将输出发送到零个、一个或多个输出 topic。在内部，**KafkaStreams** 实例包含一个普通的 **KafkaProducer** 和 **KafkaConsumer** 实例，用于读取输入和写入输出
-
- `KafkaStreams(Topology topology, Properties props)` : 创建 **KafkaStreams** 实例。
 - `KafkaStreams(Topology topology, Properties props, KafkaClientSupplier clientSupplier)` : 创建一个 **KafkaStreams** 实例。
 - `KafkaStreams(Topology topology, Properties props, KafkaClientSupplier clientSupplier,`

`org.apache.kafka.common.utils.Time time)` : 创建一个 `KafkaStreams` 实例。

- `KafkaStreams(Topology topology, Properties props, org.apache.kafka.common.utils.Time time)` : 创建一个 `KafkaStreams` 实例。

方法:

- `Collection<StreamsMetadata> allMetadata()` : 查找与此实例使用相同应用程序 ID 的所有当前正在运行的 `KafkaStreams` 实例 (可能是远程实例) (即属于同一 `Kafka Streams` 应用程序的所有实例), 并为每个发现的实例返回 `StreamsMetadata`。
- `void cleanUp()` : 清理本地 `StateStore` 目录 (`StreamsConfig.STATE_DIR_配置`) 删除与应用程序 ID 相关的所有数据。
- `void close()` : 关闭这个 `KafkaStreams` 实例, 方法是通知所有线程停止, 然后等待它们加入。
- `boolean close(Duration timeout)` : 通过通知所有线程停止来关闭这个 `KafkaStreams`, 然后等待线程加入超时。
- `Set<ThreadMetadata> localThreadsMetadata()` : 返回有关此 `KafkaStreams` 实例的本地线程的运行时信息。
- `<K> StreamsMetadata metadataForKey(String storeName, K key, Serializer<K> keySerializer)` : 找到当前正在运行的 `KafkaStreams` 实例 (可能是远程的), 该实例使用与该实例相同的应用程序 ID (即, 属于同一 `Kafka Streams` 应用程序的所有实例), 并且包含具有给定 `storeName` 的 `StateStore`, 并且 `StateStore` 包含给定的键, 并返回该实例的 `StreamsMetadata`。
- `void setStateListener(KafkaStreams.StateListener listener)` : 设置单个 `KafkaStreams.StateListener`, 以便当状态更改时通知应用。
- `void start()` : 启动所有线程启动 `KafkaStream` 实例。

- `TopicPartition` : 在 `org.apache.kafka.common` package。它有助于从特定 topic 和 分区 检索消息。它有构造函数 `TopicPartition(java.lang.String topic, int 分区)`

方法:

- `int partition()`
- `java.lang.String topic()`

Low Level API

Low-Level 消费者 API

- 更好地控制 Kafka 消息的过度消耗，例如：
 - 多次读取消息
 - 在一个进程中只使用 topic 中分区的一个子集
 - 管理事务，确保一条消息只处理一次
- 更灵活的控制：
 - 偏移不再透明
 - 需要处理代理自动故障转移
 - 添加消费者、分区和代理需要自己进行负载平衡

low-level API

- `SimpleConsumer` (`kafka.javaapi.consumer.SimpleConsumer`) 类提供到 leadbroker 的连接，用于从 topic 获取消息，以及获取 topic 元数据和偏移列表的方法。
- `FetchRequest` (`kafka.api.FetchRequest`),
- `OffsetRequest` (`kafka.javaapi.OffsetRequest`),
- `OffsetFetchRequest` (`kafka.javaapi.OffsetFetchRequest`),
- `OffsetCommitRequest` (`kafka.javaapi.OffsetCommitRequest`),
- `TopicMetadataRequest` (`kafka.javaapi.TopicMetadataRequest`).

创建 Kafka 消费者

要创建 Kafka 消费者，首先需要设置属性，然后在 ConsumerRecord 的帮助下接收消息。

```
KafkaConsumer<String, String> consumer = new KafkaConsumer<>( props );
```

订阅 topic

subscribe() 方法将 topic 列表作为参数，以订阅：
consumer.subscribe(Collections.singletonList("topicName"));

这里我们只需创建一个包含单个元素的列表。也可以使用正则表达式调用 subscribe。表达式可以匹配多个 topic 名称，如果有人用匹配的名称创建一个新 topic，则几乎立即会发生重新平衡，并且消费者将从新 topic 开始消费。这对于需要从多个 topic 中消费并且可以处理 topic 将包含的不同类型数据的应用程序非常有用。在 Kafka 和另一个系统之间复制数据的应用程序中，使用正则表达式订阅多个 topic 是最常用的。

使用正则表达式订阅

```
consumer.subscribe("regexExp");
```

如果您想订阅多个 topic 并且无法匹配任何正则表达式，则使用 Arrays.asList() 允许消费者订阅多个 topic。用户需要直接指定 topic 名称或通过字符串变量来读取消息。可以有多个 topic 也可以用逗号分隔。

```
consumer.subscribe(Arrays.asList("topicNames"));
```

poll 循环

一旦消费者订阅了 topic，poll 循环将处理协调、分区重新平衡、心跳和数据获取的所有细节，为开发人员留下一个干净的 API，它只需从分配的分区返回可用的数据。消费者的主体将如下所示：

```
while (true) {  
    ConsumerRecords<String, String> records = consumer.poll(100); //line 1  
    for (ConsumerRecord<String, String> record : records) //line 2  
        // print the offset, key and value for the consumer records.  
        System.out.printf("offset = %d, key = %s, value = %s\n",  
            record.offset(), record.key(), record.value()); //line 3  
}
```


第 1 行：poll 方法返回从当前分区的偏移量获取的数据。指定等待数据的持续时间，否则将向消费者返回空的 ConsumerRecord。

第 2 行：使用迭代器逐个迭代记录

第 3 行：可以获取偏移量、键及其值。

配置 Kafka 消费者

以下是配置消费者的步骤

■ 创建消费者属性

```
Properties props = new Properties();
props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, KAFKA_SERVER_URL + ":" +
KAFKA_SERVER_PORT);
props.put(ConsumerConfig.GROUP_ID_CONFIG, CLIENT_ID);
props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "true");
props.put(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG, "1000");
props.put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, "30000");
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
"org.apache.kafka.common.serialization.IntegerDeserializer");
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
"org.apache.kafka.common.serialization.StringDeserializer");
```

■ 创建消费者

```
KafkaConsumer<Integer,String> consumer = new KafkaConsumer<>(props);
```

读取消息

在上一个 topic 中，我们配置了消费者，下一个 topic 是读取消息，步骤如下

- 让消费者订阅特定 topic。

```
consumer.subscribe(Collections.singletonList(this.topic));
```

- 获得一些新数据

```
ConsumerRecords<Integer,String> records = consumer.poll(100);
```

- 消费记录

```
for (ConsumerRecord<Integer, String>record : records)
{
    System.out.println("Received message:
    (" + record.key() + ",      + record.value() + ") at offset
    " + record.offset());
}
```

Kafka 消费者属性列表

用于配置 Kafka 消费者的一些常见属性的列表

数据类型	字段	说明
static java.lang.String	AUTO_COMMIT_INTERVAL_MS_CONFIG	auto.commit.interval.ms
static java.lang.String	AUTO_OFFSET_RESET_CONFIG	auto.offset.reset
static java.lang.String	BOOTSTRAP_SERVERS_CONFIG	bootstrap.servers
static java.lang.String	CONNECTIONS_MAX_IDLE_MS_CONFIG	connections.max.idle.ms
static java.lang.String	DEFAULT_API_TIMEOUT_MS_CONFIG	default.api.timeout.ms
static java.lang.String	ENABLE_AUTO_COMMIT_CONFIG	enable.auto.commit
static java.lang.String	GROUP_ID_CONFIG	group.id
static java.lang.String	HEARTBEAT_INTERVAL_MS_CONFIG	heartbeat.interval.ms
static java.lang.String	ISOLATION_LEVEL_CONFIG	isolation.level
static java.lang.String	KEY_DESERIALIZER_CLASS_CONFIG	key.deserializer
static java.lang.String	MAX_POLL_INTERVAL_MS_CONFIG	max.poll.interval.ms
static java.lang.String	MAX_POLL_RECORDS_CONFIG	max.poll.records
static java.lang.String	RECEIVE_BUFFER_CONFIG	receive.buffer.bytes
static java.lang.String	REQUEST_TIMEOUT_MS_CONFIG	request.timeout.ms
static java.lang.String	RETRY_BACKOFF_MS_CONFIG	retry.backoff.ms
static java.lang.String	SEND_BUFFER_CONFIG	send.buffer.bytes
static java.lang.String	SESSION_TIMEOUT_MS_CONFIG	session.timeout.ms
static java.lang.String	VALUE_DESERIALIZER_CLASS_CONFIG	value.deserializer



笔记

消费者的 maven 依赖:

```
<dependency>  
  <groupId>org.apache.kafka</groupId>  
  <artifactId>kafka-clients</artifactId>  
  <version>2.5.0</version>  
</dependency>
```



活动 4.1：创建 java 消费者



活动 4.2:创建自定义序列化程序

练习题

1. 以下传递语义 消息可能会丢失，但永远不会重新传递的是。
 - a. 最多一次
 - b. 至少一次
 - c. 就一次
 - d. 以上都不是
2. 这些消费者也可以用不同的语言实现，例如
 - a. Java
 - b. C
 - c. Python.
 - d. 以上都是
3. 哪些是 Kafka 的元素
 - a. Topic
 - b. Producer
 - c. Consumer
 - d. 以上都是
4. 哪些属性有助于设置消费者组 ID?
 - a. `consumergroup.id`
 - b. `groupid`
 - c. `group.id`
 - d. `consumergroupid`
5. `Subscribe()` 方法接受____
 - a. 单个 topic name
 - b. 多个 topic names
 - c. 正则表达式
 - d. 以上都是

摘要

在本章中，你学到了：

- Kafka 消费者是一个可以读取 Kafka 集群中数据的应用程序。
- 消费者是使用 Kafka 生产者发布的信息并处理从中提取的数据的应用程序。
- 消费者组可以被描述为一个单一的逻辑消费者，它订阅了一组指向应用程序的 topic。它可以在两种情况下使用
 - 为每个应用程序创建一个新的消费者组，该用户组需要来自一个或多个 topic 的所有消息。
 - 继续将消费者添加到现有消费者组，以缩放来自 topic 的消息的读取和处理，因此组中的每个其他消费者将只获取消息的子集。
- Kafka 存储消费者组读取的偏移量。 `__consumer_offsets`
- Delivery semantics
 - 最多一次一消息可能丢失，但永远不会重发。
 - 至少有一次一消息不会丢失，但可以重新发送。
 - 就一次一这就是人们真正想要的，每一条信息只传递一次。
- 在消费者组中，Kafka 在某些事件将分区的所有权从一个消费者更改到另一个消费者。更改消费者的分区所有权的过程称为重新平衡。
- High Level API
 - `KafkaConsumer<K,V>`
 - `ConsumerRecord<K,V>`：
 - `ConsumerRecords`
 - `ConsumerConfig`
 - `ConsumerConnector`
 - `KafkaStreams`
 - `TopicPartition`
- Low level API:更灵活的控制
- `subscribe ()` 方法将 topic 列表作为参数订阅：
- 您可以一次订阅一个或多个 topic。
- `poll` 循环处理协调、分区重新平衡、检测信号和数据提取的所有详细信息，使开发人员拥有一个干净的 API，该 API 仅从分配的分区返回可用数据。
- 配置 Kafka 消费者

Kafka 监控和管理

本章讲解 Kafka 集群的监控和管理办法。

Kafka 提供了几个命令行工具，我们可以使用它们轻松地对 Kafka 集群进行操作。在本章中，我们将介绍如何使用它们来操作主题和消费者组，以及在不停止集群的情况下更改配置。

第二部分是 Kafka 的监控。在 Kafka 应用中，有数量相当多的指标（Metrics）用于监控 Kafka 集群中的各种操作状态，所以我们需要找出哪些指标是重要的，哪些又是不重要的。所有由 Kafka 暴露出来的指标都可以通过 Java Management Extensions (JMX)接口访问。我们将学习使用 Jconsole 来监视指标。

我们还将介绍另一个工具 Kafka Tool，这是一个用于管理和操作 Kafka 集群的 GUI 应用程序。

目标

在本章中, 您将学习:

- 主题操作
- 消费者操作
- 动态配置修改
- Kafka 监控



主题操作

在做操作之前，需要先启动并运行 **kafka** 集群。然后就可以在上面创建主题了。**Kafka** 提供了一个命令行实用程序来操作 **Kafka** 服务器上的主题。

创建主题

让我们使用这个实用工具创建一个名为 **my-topic** 的主题，它只有 1 个分区和 1 个副本：

```
> bin/kafka-topics.sh --zookeeper localhost:2181 --create --topic my-topic --
replication-factor 1 --partitions 1
Created topic "my-topic".
```

- **--zookeeper** <String: hosts> (从 **Kafka 2.2**+起废弃)，用 **host:port** 格式表示的 **zookeeper** 的连接字符串，可以指定多个主机避免单点故障。
- **--create** 创建一个主题。
- **--partitions** <Integer: # of partitions> 创建或者修改操作时所指定的主题分区数。
- **--replication-factor** <Integer: replication factor> 即将创建主题的每个分区的副本因子。如果没有指定则采用集群默认配置。
- **--topic** <String: topic> 指定创建、修改、描述或者删除操作的主题。



Note

--topic 也可以接受正则表达式，用 **--create** 时除外。主题名字中如果包含特殊字符可以在两端加上双引号转义，例如 **"test.topic"**。

kafka-topics.sh 工具将会创建一个新主题，默认分区数 2 被覆盖，新主题的分区数指定为 1，并显示一个成功创建的消息。这个命令还需要获取 **Zookeeper** 服务器信息，在本例中为 **localhost:2181**。

为了获取特定主题的详细信息，可以执行下面的命令：

```
> bin/kafka-topics.sh --zookeeper localhost:2181 --describe --topic mytopic

Topic: my-topic PartitionCount: 1    ReplicationFactor: 1    Configs:      Topic:
my-topic Partition: 0    Leader: 0    Replicas: 0 Isr: 0
```

--describe 选项可以列出给定主题的详情, 对输出的解释如下:

- **PartitionCount** 现存的 topic 分区 数量
- **ReplicationFactor** 现存的 topic 副本因子
- **Replicas** Kafka 中复制数据的节点列表
- **Isr** 用于当前同步副本之间数据的节点列表

为主题添加分区

现在为上一步骤中创建的主题添加一个分区:

```
> kafka-topics.sh --bootstrap-server localhost:2181 --alter --topic mytopic --partitions 2
```

现在可以看到原来主题的分区数已经变成 2 个了:

```
> kafka-topics.sh --bootstrap-server localhost:2181 --describe --topic my-topic

Topic: my-topic PartitionCount: 2    ReplicationFactor: 1    Configs:
Topic: my-topic Partition: 0        Leader: 0    Replicas: 0 Isr: 0    Topic:
my-topic Partition: 1        Leader: 0    Replicas: 0 Isr: 0
```

正如你所看到的那样, --alter 选项可以修改指定主题的分区数, 副本数, 以及配置信息.



Note

目前的版本 Kafka 不允许减少分区的操作, 如果那样做会导致 *InvalidPartitionsException*. 异常发生.

删除主题

可以使用下面的命令完成删除主题的操作:

```
> bin/kafka-topics.sh --zookeeper localhost:2181 --delete --topic my-topic
```

**Note**

默认情况 `-delete` 只是将主题标记为删除状态，并非真正意义上的删除。如果想要真正将其删除，需要在 `server.properties` 里面加上一行配置如下：

```
delete.topic.enable=true
```

列出集群中的主题

要获得一个 Kafka 集群中某台服务器上所有可用的主题列表，可在控制台使用如下命令：

```
> bin/kafka-topics.sh --zookeeper localhost:2181 --list
my-topic-1
my-topic-2
my-topic-3
```

消费者操作

使用 `ConsumerGroupCommand` 工具，我们可以列出、描述或删除消费者组。消费者组可以被手动删除，也可以在该组最后一次提交的偏移量到期时被自动删除。手动删除仅在组中没有任何活动成员时有效。

列出和描述消费者群组

如果您使用的是老客户端，那么应该使用 `-zookeeper` 和 `-list` 选项。但是，如果使用新的消费者客户端，要列出消费者组，可以使用 `-bootstrap-server` 和 `-list` 选项。因为新的客户端已经删除了 `-zookeeper` 选项。

```
> bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --list

test-consumer-group-1
test-consumer-group-2
test-consumer-group-3
```

■ **--bootstrap-server** <String: server to connect to> 必须指定: 要连接的服务器的主机:端口号

您可以通过将 `--list` 更改为 `--describe` 并添加 `--group` 参数来获得更多详情信息，比如偏移量。这将列出指定消费者组正在使用的所有主题，以及每个主题分区的偏移量。

```
> bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --describe --group my-group
```

TOPIC	PARTITION	CURRENT-OFFSET	LOG-END-OFFSET	LAG	CONSUMER-ID	HOST	CLIENTID
topic1	0	854144	855809	1665	consumer1-3fc8d6f1-581a-4472-bdf3-3515b4aee8c1	/127.0.0.1	consumer1
topic2	0	460537	803290	342753	consumer1-3fc8d6f1-581a-4472-bdf3-3515b4aee8c1	/127.0.0.1	consumer1
topic3	2	243655	398812	155157	consumer4-117fe4d3-c6c1-4178-8ee9-eb4a3954bee0	/127.0.0.1	consumer4

删除消费者群组

要手动删除一个或多个消费者组，可以使用 `--delete` 选项：

```
> bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 -delete --group my-group --group my-other-group

Deletion of requested consumer groups('my-group', 'my-other-group') was successful.
```

当要删除的消费者组不为空时，执行上面命令你将得到以下错误: `GroupNotEmptyException`。

偏移量管理

除了使用`--describe` 选项显示偏移量之外，我们还可以使用`--delete-offsets` 选项删除消费者组的偏移量。此选项同时支持一个消费者组和一个或多个主题。

例如，将主题“my-topic-1”和“my-topic-2”从“my-group”消费组中删除：

```
> kafka-consumer-groups.sh --bootstrap-server localhost:9092 --delete-offsets --group my-group --topic my-topic-1 --topic my-topic-2
```

TOPIC	PARTITION	STATUS
my-topic-1	0	Successful
my-topic-2	0	Successful

还可以重设偏移量。这在需要重新读取消息为消费者重置偏移量是非常有用的，或者消费者在消费消息出现问题时可以向后推进偏移量(例如，如果存在使用者无法处理的格式化错误的消息)。

要重置使用者组的偏移量，可以使用“`--reset-offsets`”选项。此选项一次只支持一个消费者组。它需要定义以下范围：`--all-topic` 或 `--topic`。必须选择一个作用域，除非使用“`--from-file`”的方式从文件导入。

例如，要将消费者组的偏移量重置为最新偏移量：

如果您使用的是旧的高级消费者，也就是消费者组元数据是存储在 ZooKeeper 中(即配置了 `offset.storage=ZooKeeper`)，则需要使用`--zooKeeper` 而不是`--bootstrap-server`：

```
> bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --reset-offsets --group consumergroup1 --topic topic1 --to-latest
```

TOPIC	PARTITION	NEW-OFFSET
topic1		

- `-to-datetime <String: datetime>` : Reset offsets to offsets from datetime. Format: 'YYYY-MM-DDTHH:mm:SS.sss'
- `--to-earliest` : 重置为最早的偏移量
- `--to-latest` : 重置为最近的偏移量
- `--shift-by <Long: number-of-offsets>` : 将当前的偏移量偏移 `n` 个单位，`n` 可以为正数也可以是负数
- `--from-file` : 利用 CSV 文件中的数据重置偏移量
- `--to-current` : 将偏移量重置为当前值
- `--by-duration <String: duration>` : 重置偏移量为从当前时间戳开始的时长，格式: 'PnDTnHnMnS'
- `--to-offset` : 重置偏移量为指定的值

动态配置修改

当我们想要更改服务器中的 `server.properties` 文件中的配置，我们需要停止 **Broker**(Kafka 服务器)，更改完成后还要重新启动。显示这不适合在运行中的生产集群上操作。因此，从 **Kafka 1.1.0** 开始有了动态配置修改的新特性。动态意味着在修改了 **Broker** 的配置后，我们不需要重新启动 **Broker** 来使其生效。

这个新特性包含在一个名为 `kafka-config.sh` 的命令行工具脚本中。一旦使用这个工具设置了配置参数，新的更改将永久存储在 **zookeeper** 集群中。

覆盖主题默认配置

有许多应用于主题的配置，可以针对单个主题更改这些配置，以适应单个集群中的不同用例。大多数配置都在代理配置中指定了缺省值，除非设置了覆盖(Overriding Topic Configuration Defaults)，否则将应用该缺省值。

更改主题配置命令的格式为：

```
kafka-configs.sh --zookeeper zoo1.example.com:2181/kafka-cluster --alter --entity-type topics --entity-name <topic name> --add-config <key>=<value>[,<key>=<value>...]
```

下面显示了主题的部分有效配置。

Configuration Key	说明
<code>cleanup.policy</code>	如果设置为 <i>compact</i> ，则 <i>topic</i> 中的消息将被丢弃，仅保留具有给定 <i>key</i> 的最新消息（日志压缩）。
<code>compression.type</code>	<i>broker</i> 将消息写入磁盘时使用的压缩类型，可以用 <i>gzip</i> 、 <i>snappy</i> 和 <i>lz4</i> 。
<code>delete.retention.ms</code>	删除墓碑，将为这个 <i>topic</i> 保留多长时间。仅仅对日志压缩的 <i>topic</i> 有效。
<code>file.delete.delay.ms</code>	从磁盘中删除此 <i>topic</i> 的日志段和索引之前需要等待的多长时间
<code>flush.messages</code>	在强制将此 <i>topic</i> 的消息刷到磁盘之前接收的消息数
<code>flush.ms</code>	在强制将此 <i>topic</i> 的消息刷到磁盘之前需要的时间，单位是 <i>ms</i>
<code>index.interval.bytes</code>	日志段索引中的条目之间可以产生多少字节的消息

<code>max.message.bytes</code>	此 topic 中单个消息的大小
<code>retention.bytes</code>	为 topic 保留的消息量的总字节数
<code>retention.ms</code>	topic 中消息保留的最长时间 毫秒

下面的示例将 `my-topic` 主题的留存时间设置为 1 小时，即 `3600000ms`：

```
> kafka-configs.sh --zookeeper localhost:2181 --alter --entity-type topics --entity-name my-topic --add-config retention.ms=3600000
Warning: --zookeeper is deprecated and will be removed in a future version of Kafka.
Use --bootstrap-server instead to specify a broker to connect to.
Completed updating config for entity: topic 'my-topic'.
```

覆盖客户端默认配置

Kafka 客户端唯一可以覆盖的配置是生产者和消费者的配额，即允许具有指定客户端 ID 的所有客户端在每个 broker 上每秒生产或者消费的字节数。这意味着，如果集群中有 5 个 broker，并且为一个客户端指定了 10M/s 的生产者配额，那么该客户端将被允许在每个 broker 上同时生产 10MB/s，总量为 50MB/s。

覆盖客户端默认配置（Overriding Client Configuration Defaults）的命令格式为：

```
> kafka-configs.sh --zookeeper zoo1.example.com:2181/kafka-cluster --alter --entity-type clients --entity-name <client ID> --add-config <key>=<value>[,<key>=<value>...]
```

下面显示了客户端的全部可覆盖配置。

Configuration Key	说明
<code>producer_bytes_rate</code>	允许单个客户端 ID 在一秒内生成给单个 broker 的消息量。以字节为单位
<code>consumer_bytes_rate</code>	允许单个消费者 ID 在一秒内单个 broker 中消费的消息量，以字节为单位

显示覆盖的配置

可以使用命令行工具 `kafka-configs.sh` 来检查主题或客户机的特定配置。显示覆盖的配置 (Describing Configuration Overrides) 需要使用 `--describe` 选项。例如，显示名为 “my-topic” 的主题的所有覆盖过的配置:

```
[hadoop@hadoop000 bin]$ kafka-configs.sh --zookeeper localhost:2181 --describe
--entity-type topics --entity-name my-topic

Warning: --zookeeper is deprecated and will be removed in a future version of
Kafka.

Use --bootstrap-server instead to specify a broker to connect to.

Configs for topic 'my-topic' are retention.ms=3600000
```

删除覆盖的配置

可以完全删除动态配置，这将导致集群配置恢复到默认值，要删除配置覆盖，请使用 `--alter` 命令以及 `--delete-config` 命令。

下面的示例可以删除一个名为 “my-topic” 的主题的覆盖后的 `retention.ms` 配置，删除后 `retention.ms` 将恢复为默认值:

```
kafka-configs.sh --zookeeper localhost:2181 --alter --entity-type topics --
entity-name my-topic --delete-config retention.ms

Warning: --zookeeper is deprecated and will be removed in a future version of
Kafka.

Use --bootstrap-server instead to specify a broker to connect to.

Completed updating config for entity: topic 'my-topic'.
```



活动 5.1: 搜索各种管理和监控工具

监控 Kafka

Kafka 在服务器和客户端都使用 Yammer Metrics 来测量系统的运行状态。查看可用指标(Metrics)的最简单方法是启动 Java 自带的 Jconsole 工具, 连接到运行中的 kafka 客户端或服务; Jconsole 是基于 JMX(Java 管理扩展)的, 通过 JMX 就可以查看连接到的客户端或者服务器上的所有指标。Kafka 默认禁用远程 JMX, 但我们通过为执行 CLI 命令所在的进程中设置环境变量 JMX_PORT, 或者设置标准 Java 系统属性, 就可以通过编程方式启用远程 JMX。例如, 您可以在启动 kafka 时设置 JMX_PORT:

```
JMX_PORT=9988 && bin/kafka-server-start.sh -daemon config/server.properties
```

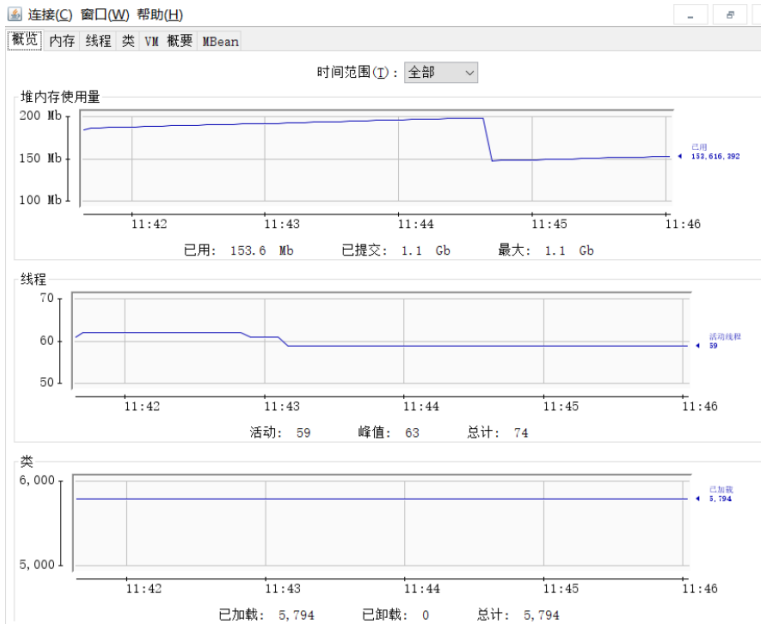
Jconsole 是 JAVA_HOME/bin 中的一个工具, 用于监视正在运行的 Java 程序的状态。



选择远程进程, 输入要连接的主机名和端口号, 以及远程系统的用户名和密码, 在点击连接按钮:



通过 Jconsole 界面，我们可以监控远程系统的内存、线程、Java 类、虚拟机基本信息还有 Mbeans 信息。



打开 Mbean 标签页，可以图形化展示所有监控指标，
查看 <http://kafka.apache.org/documentation/#security> 网站可以找到所有指标的说明信息：

The figure shows the JConsole 'MBeanInfo' tab. The left pane lists the MBean hierarchy, with 'kafka.server.BrokerTopicMetrics.BytesInPerSec' selected. The right pane displays the details for this MBean.

名称	值
信息:	
ObjectName	kafka.server:type=BrokerTopicMetrics...
ClassName	com.yammer.metrics.reporting.JmxRepor...
说明	Information on the management interfa...

名称	值
描述符	
信息:	
immutableInfo	true
interfaceClass...	com.yammer.metrics.reporting.JmxRepor...
mbean	false

监控服务器状态

下面这些指标可用于服务器实例。

说明	Mbean 名称
<code>Message in rate</code> 消息比率	<code>kafka.server:type=BrokerTopicMetrics, name=MessagesInPerSec</code>
<code>Byte in rate</code> 字节比率	<code>kafka.server:type=BrokerTopicMetrics, name=BytesInPerSec</code>
<code>Request rate</code> 请求比率	<code>kafka.network:type=RequestMetrics, name=RequestsPerSec, request={Produce FetchConsumer FetchFollower}</code>
<code>Byte out rate</code> 字节输出比率	<code>kafka.server:type=BrokerTopicMetrics, name=BytesOutPerSec</code>
<code>Log flush rate and time</code> 日志冲洗比率和时间	<code>kafka.log:type=LogFlushStats, name=LogFlushRateAndTimeMs</code>
<code># of under replicated partitions (ISR < all replicas)</code> 失效副本分区个数	<code>kafka.server:type=ReplicaManager, name=UnderReplicatedPartitions</code>
<code>Is controller active on broker</code> Broker 上的控制器是否活跃	<code>kafka.controller:type=KafkaController, name=ActiveControllerCount</code>
<code>Leader election rate</code> leader 选举比率	<code>kafka.controller:type=ControllerStats, name=LeaderElectionRateAndTimeMs</code>
<code>Unclean leader election rate</code> 争议 leader 选举比率	<code>kafka.controller:type=ControllerStats, name=UncleanLeaderElectionsPerSec</code>

上面只列出了部分服务器可用的 Mbean 说明信息。

监控生产者状态

下面这些指标可用于生产者实例。

说明	Mbean 名称
The number of user threads blocked waiting for buffer memory to enqueue their records. 阻塞等待缓冲内存消息入队的用户线程数	<code>kafka.producer:type=producer-metrics,client-id={client-id}</code>
The maximum amount of buffer memory the client can use (whether or not it is currently used). 客户端可以使用的最大缓冲区内存（无论目前是否使用）	<code>kafka.producer:type=producer-metrics,client-id={client-id}</code>
The total amount of buffer memory that is not being used (either unallocated or in the free list). 未使用的缓冲内存总量（未分配或在空闲列表中）。	<code>kafka.producer:type=producer-metrics,client-id={client-id}</code>
The fraction of time an appender waits for space allocation. appender 等待空间分配的时间比率。	<code>kafka.producer:type=producer-metrics,client-id={client-id}</code>
The average number of bytes sent per partition per-request. 每个分区每个请求发送的平均字节数	<code>kafka.producer:type=producer-metrics,client-id={client-id}</code>
The max number of bytes sent per partition per-request. 每个分区每个请求发送的最大字节数	<code>kafka.producer:type=producer-metrics,client-id={client-id}</code>
The average number of batch splits per second 每秒批次切分的平均数量	<code>kafka.producer:type=producer-metrics,client-id={client-id}</code>
The total number of batch splits 批次切分的总数	<code>kafka.producer:type=producer-metrics,client-id={client-id}</code>
The average number of bytes sent per second for a topic 一个主题每秒发送的字节数	<code>kafka.producer:type=producer-topic-metrics,client-id={client-id},topic={topic}</code>
The total number of bytes sent for a topic. 一个主题发送的字节总数	<code>kafka.producer:type=producer-topic-metrics,client-id={client-id},topic={topic}</code>

上面只列出了部分生产者可用的 Mbean 说明信息。

监控消费者状态

下面这些指标可用于消费者实例。

说明	Mbean 名称
The average delay between invocations of poll(). 调用 poll() 之间的平均延迟。	kafka.consumer:type=consumer-metrics,client-id=([-.\w]+)
The max delay between invocations of poll(). 调用 poll() 之间的最大延迟。	kafka.consumer:type=consumer-metrics,client-id=([-.\w]+)
The average time taken for a commit request 提交请求所花费的平均时间	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)
The number of commit calls per second 每秒提交调用的次数	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)
The total number of commit calls 提交调用的总数	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)
The average number of bytes consumed per second	kafka.consumer:type=consumer-fetch-manager-metrics,client-id="{client-id}"
The total number of bytes consumed 每秒消费的平均字节数	kafka.consumer:type=consumer-fetch-manager-metrics,client-id="{client-id}"
The average time taken for a fetch request 消费的总字节数	kafka.consumer:type=consumer-fetch-manager-metrics,client-id="{client-id}"
The max time taken for any fetch request. 任意 fetch 请求所花费的最大时间。	kafka.consumer:type=consumer-fetch-manager-metrics,client-id="{client-id}"

上面只列出了部分消费者可用的 Mbean 说明信息。



活动 5.2: 使用 Jconsole 进行监控



活动 5.3: 使用 Kafka Tool 进行监控

练习问题

1. _____ 可以用来创建主题?
 - a. kafka-topics.sh
 - b. kafka-topic
 - c. kafka-topics
 - d. kafka-topic.sh
2. my-topic 本来有 10 个分区, 如何使用命令为它再添加 20 个分区?
 - a. kafka-topics.sh --bootstrap-server localhost:2181 --alter --topic my-topic --partitions 20
 - b. kafka-topics.sh --bootstrap-server localhost:2181 --topic my-topic --partitions 30
 - c. kafka-topics.sh --bootstrap-server localhost:2181 --modify --topic my-topic --partitions 20
 - d. kafka-topics.sh --bootstrap-server localhost:2181 --alter --topic my-topic --partitions 30
3. 删除一个或者多个消费者群组, 可以使用 ____ ?
 - a. --truncate
 - b. --remove
 - c. --delete
 - d. --deleteGroup
4. 在消费者示例的监控指标中, 哪个属性是表示调用 poll()之间的平均延迟 ?
 - a. time-poll-between-avg
 - b. time-between-poll-max
 - c. poll-idle-ratio-avg
 - d. time-between-poll-avg
5. 下面哪一个说法不为真?
 - a. Jconsole 是基于 JMX 的
 - b. Jconsole 可以用来监控 Kafka 集群的状态
 - c. Kafka 使用 Yammer Metrics 来测量系统的运行状态
 - d. Kafka 默认情况下远程 JMX 是开启的

小结

您在这个章节学习了:

- 主题操作:
 - 1. 创建新主题
 - 2. 为主题添加分区
 - 3. 删除主题
 - 4. 列出集群中所有主题
- 消费者操作:
 - 1. 列出和描述消费者群组.
 - 2. 删除消费者群组
 - 3. 偏移量管理
- 动态配置修改:
 - 1. 覆盖主题配置默认值
 - 2. 覆盖客户端配置默认值
 - 3. 显示覆盖配置信息
 - 4. 删除覆盖配置信息
- 监控 Kafka
 - 1. 服务器状态监控
 - 2. 生产者状态监控
 - 3. 消费者状态监控

流处理



行业分析师有时声称，所有这些流处理系统都像复杂的事件处理（CEP）系统，这些系统已经存在了 20 年。我们认为流处理变得更加流行，因为它是在 Kafka 之后创建的，因此可以使用 Kafka 作为处理事件流的可靠来源。

随着 Apache-Kafka 的日益普及，最初是作为一个简单的消息总线，后来作为一个数据集成系统，许多公司都有一个包含许多有趣数据流的系统，这些数据流被存储了很长时间，并且秩序井然，只是在等待某个流处理框架出现并进行处理。

本章重点介绍流处理、概念、设计模式和体系结构概述。

目标

在本章中，你将学到：

- 📖 什么 流处理？
- 📖 流处理概念
- 📖 流处理设计模式
- 📖 Kafka Streams: 架构概述
- 📖 Stream 处理用例

什么是流处理？

流处理的世界仍在发展，仅仅因为特定流行实现以特定方式执行操作或具有特定限制并不意味着这些详细信息是处理数据流的内在部分。

数据流（也称为事件流或流数据）是表示无限数据集的抽象。无界意味着无限和不断成长。数据集是无界的，因为随着时间的推移，新记录不断到达。

流处理是指对一个或多个事件流进行的处理。流处理与请求-响应和批处理一样是一种编程范式。

事件流模型的其他属性，除了它们的无限性之外：

- 对事件流进行排序
- 数据记录不可变
- 事件流可重播
- 请求-响应
- 批处理
- 流处理

■ 对事件流进行排序

有这么一个概念，即哪些事件发生在其他事件之前或之后。例如，在研究金融事件时，这一点很明显。我先把钱放在我的账户里，然后再把钱花掉的顺序和我先花钱，然后再存钱来还债的顺序是完全不同的。后者将产生透支费用，而前者则不会。这是事件流和数据库表之间的区别之一，表中的记录总是被认为是无序的，SQL 的“order by”子句不是关系模型的一部分；添加它是为了帮助报告。

■ 数据记录不可变

事件一旦发生，就永远无法修改。例如，取消的金融交易不会消失。相反，一个额外的事件被写入流，记录前一个事务的取消。当顾客将商品退回商店时，我们不会删除商品先前卖给他的事实，而是将退货记录为附加事件。这是数据流和数据库表之间的另一个区别。我们可以删除或更新表中的记录，但这些都是发生在数据库中的附加事务，因此可以记录在记录所有事务的事件流中。如果您熟悉数据库中的 binlogs、WALs 或 redo 日志，您可以看到，如果我们在表中插入一条记录，然后删除它，那么表将不再包含该记录，但是 redo log 将包含两个事务：insert 和 delete。

■ 事件流可重播

通过套接字流式传输的 TCP 数据包通常是不可播放的，对于大多数业务应用程序来说，能够重放发生在几个月（有时甚至几年）之前的原始事件流是至关重要的。这是为了纠正错误、尝试新的分析方法或执行审计所必需的。这就是我们相信卡夫卡使流处理在现代企业中如此成功的原因。它允许捕捉和回放事件流。

■ 请求-响应

处理模式通常阻止发送请求并等待处理系统响应的应用。在数据库中，此范例称为在线事务处理（OLTP）。销售点系统（如信用卡处理）和时间跟踪系统通常以“请求 + 响应”模式工作。

Kafka 是延迟最低的范例，其响应时间从亚毫秒到几毫秒不等，通常期望响应时间高度一致。

■ 批处理

在数据库中，这些都是数据仓库和商业智能系统，每天都会大量地加载数据，生成报告，用户查看相同的报告，直到下一次数据加载发生。这种模式通常具有极大的效率和规模经济性，但近年来，企业需要在较短的时间内获得可用的数据，以使决策更及时、更高效。这给那些开发规模经济而不是提供低延迟报告的系统带来了巨大的压力。

■ Stream 处理

流处理是一种计算机编程范式，相当于数据流编程、事件流处理和反应式编程，[1]允许一些应用程序更容易地利用有限形式的并行处理。这种应用可以使用多个计算单元，例如图形处理单元上的浮点单元。没有显式地管理这些对象之间的分配、同步或通信单位流处理范式通过限制可执行的并行计算来简化并行软件和硬件。给定一个数据序列（流），一系列操作（内核函数）应用于流中的每个元素。内核函数通常是流水线式的，并尝试优化本地片上内存重用，以尽量减少与外部内存交互相关的带宽损失。统一流，其中一个内核函数应用于流中的所有元素

流处理概念

流处理与任何类型的数据处理非常相似，您编写接收数据的代码，对数据执行一些转换、聚合、充实等操作，然后将结果放在某个地方。下面列出了流处理的一些关键概念。

- 时间
- 状态
- 流和表的对偶性
- 时间窗口

■ 时间

时间可能是流处理中最重要的概念，而且常常是最令人困惑的。为了了解在讨论分布式系统时时间有多复杂，我们推荐 justinsheehy 的优秀论文“现在没有”。在流处理的上下文中，有一个共同的时间概念是至关重要的，因为大多数流应用程序在时间窗口上执行操作。例如，我们的 stream 应用程序可能会计算股票价格的 5 分钟移动平均值。在这种情况下，我们需要知道当我们有一个生产商由于网络问题离线两个小时，并返回两个小时的数据时该怎么办大多数数据将与 5 分钟的时间窗口相关，而这些窗口已经过了很长一段时间，并且结果已经被计算和存储。流处理系统通常指以下时间概念：

■ 事件时间

这是我们跟踪的事件发生和记录创建的时间，采取测量，在商店出售的项目，用户查看了我们网站上的网页，等等。在版本 0.10.0 及更晚版本中，Kafka 会在创建时自动将当前时间添加到创建者记录。如果这与应用程序的事件时间概念不匹配，例如在事件发生后的某个时间根据数据库记录创建 Kafka 记录的情况下，您应该将事件时间添加为记录本身中的字段。事件时间通常是处理流数据时最重要的时间。

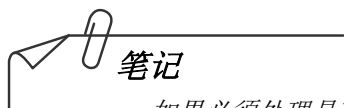
■ 日志附加时间

这是事件到达 Kafka 代理并存储在那里的时间。在 0.10.0 及更高版本中，如果 Kafka 配置为这样做，或者如果记录来自旧的生产商并且不包含时间戳，Kafka 代理将自动将此时间添加到他们接收的记录中。这种时间概念通常与流处理不太相关，因为我们通常对事件发生的时间感兴趣。例如，如果我们计算每天生产的设备数量，我们要计算当天实际生

产的设备，即使存在网络问题并且事件只在第二天到达 Kafka。但是，在没有记录实际事件时间的情况下，仍然可以一致地使用 `log append time`，因为它在创建记录之后不会更改。

■ 处理时间

这是流处理应用程序接收事件以执行某些计算的时间。此时间可以是事件发生后的毫秒、小时或天。这个时间概念根据每个流处理应用程序读取事件的确切时间，为同一事件分配不同的时间戳。对于同一个应用程序中的两个线程，它甚至可能不同。因此，这种时间概念非常不可靠，最好避免。



笔记

如果必须处理具有不同时区的数据流，则需要确保在对时间窗口执行操作之前可以将事件转换为单个时区。这通常意味着将时区存储在记录本身中。

■ 状态

只要您只需要单独处理每个事件，流处理就是一个非常简单的活动。例如，如果您只需从 Kafka 读取一个在线购物交易流，找到超过 10000 美元的交易，并将电子邮件发送给相关销售人员，那么您可能只需几行代码就可以使用 Kafka 消费者和 SMTP 库编写这些内容。

当操作涉及多个事件时，流处理变得非常有趣：按类型计算事件数、移动平均数、将两个流合并以创建丰富的信息流等。在这些情况下，单独查看每个事件是不够的；你需要记录更多的信息每种类型的事件有多少，所有需要连接的事件，求和，平均值等等。我们把存储在事件之间的信息称为状态。

通常，将状态存储在 `streamprocessing` 应用程序本地的变量中，例如存储移动计数的简单哈希表。事实上，我们在本书的许多例子中都是这样做的。然而，这不是一个可靠的方法来管理流处理中的状态，因为当流处理应用程序停止时，状态将丢失，从而改变结果。这通常不是期望的结果，因此在启动应用程序时，应该注意保持最近的状态并恢复它。流处理是指以下类型的状态：

■ 本地或内部状态

只能由流处理应用程序的特定实例访问的状态。此状态通常使用应用程序中运行的内存数据库中的嵌入式内存数据库进行维护和管理。地方州的优势在于它非常快。缺点是您只能使用可用的内存量。因此，流处理中的许多设计模式侧重于将数据分区到子流中的方法，这些子流可以使用有限的本地状态进行处理。

■ 外部状态

在外部数据存储中维护的状态，通常是像 Cassandra 这样的 NoSQL 系统。外部状态的优点是它的大小几乎是无限的，并且可以从应用程序的多个实例甚至不同的应用程序访问它。缺点是额外的系统会带来额外的延迟和复杂性。大多数流处理应用程序尽量避免处理外部存储，或者至少通过在本地图态缓存信息并尽可能少地与外部存储通信来限制延迟开销。这通常会给保持内部和外部状态之间的一致性带来挑战。

■ 流和表的对偶性

我们都熟悉数据库表。表是记录的集合，每个记录由其主键标识，并包含由模式定义的一组属性。表记录是可变的（即，表允许更新和删除操作）。查询表允许检查特定时间点的数据状态。例如，通过查询数据库中的 CUSTOMERS_CONTACTS 表，我们期望找到所有客户的当前联系人详细信息。除非这个表是专门设计来包括历史的，否则我们不会在表中找到他们过去的联系人。

与表不同，流包含更改的历史记录。流是一系列事件，其中每个事件都会导致更改。表包含世界的当前状态，这是许多更改的结果。从这个描述中，很明显，流和表是同一个硬币的两面，世界总是在变化，有时我们对引起这些变化的事件感兴趣，而有时我们对世界的现状感兴趣。允许您在两种查看数据的方式之间来回转换的系统比只支持一种方式的系统更强大。

为了将表转换为流，我们需要捕获修改表的更改。获取所有这些 insert、update 和 delete 事件并将它们存储在流中。大多数数据库提供更更改数据捕获（CDC）解决方案来捕获这些更改，并且有许多 Kafka 连接器可以将这些更改通过管道传输到 Kafka 中，以便在那里进行流处理。

为了将流转换为表，我们需要应用流包含的所有更改。这也叫物化流。我们在内存、内部状态存储或外部数据库中创建一个表，并开始从头到尾检

查流中的所有事件，同时更改状态。当我们完成时，我们有一个表，表示在特定时间的状态，我们可以使用。假设我们有一家卖鞋的商店。我们零售活动的流表示可以是事件流：

“货物到达时，有红、蓝、绿三种颜色的鞋子”

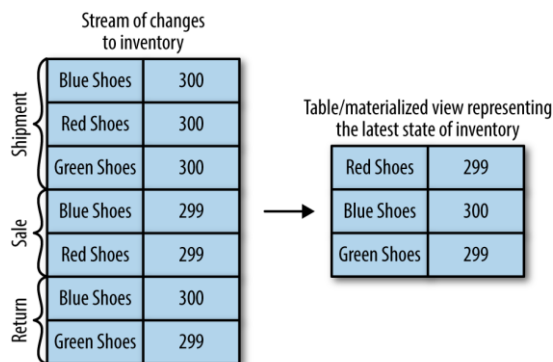
“出售蓝色鞋子”

“出售红色鞋子”

“蓝鞋子回来了”

“出售绿色鞋子”

如果我们想知道我们的库存现在包括什么，或者我们到现在为止赚了多少钱，我们需要具体化这个观点。下图显示，我们目前有蓝色和黄色的鞋子和 170 美元的银行存款。如果我们想知道商店有多忙，我们可以查看整个流程，发现有五个事务。我们可能还想调查为什么蓝鞋被退回。



库存变化

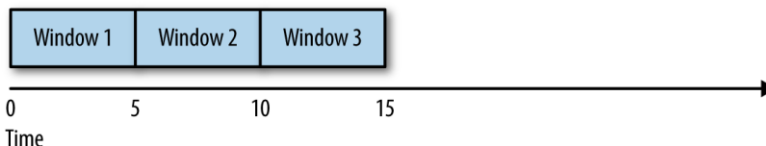
■ 时间窗口

流上的大多数操作都是在时间片上操作的窗口操作：移动平均值、本周销售的顶级产品、系统上的第 99 个百分位负载等。两个流上的连接操作也是窗口化的，我们连接在同一时间段发生的事件。很少有人停下来思考他们想要的操作窗口类型。例如，在计算移动平均值时，我们想知道：

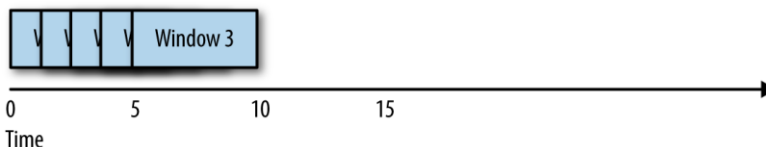
- **窗口大小：**是否要计算每个五分钟窗口中所有事件的平均值？每 15 分钟一次？还是一整天？较大的窗口比较平滑，但是如果价格上涨，则延迟时间会更长，与较小的窗口相比，需要更长的时间来注意。
- **窗口移动的频率（提前间隔）：**5 分钟的平均值可以每分钟、每秒钟或每次发生新事件时更新。当提前间隔等于窗口大小时，这有时称为翻滚窗口。当窗口在每个记录上移动时，这有时称为滑动窗口。
- **窗口保持可更新的时间：**我们的五分钟移动平均值计算了 00:00–00:05 窗口的平均值。现在一个小时后，我们得到更多的结果，他们的活动时间显示 00:02。我们是否更新 00:00–00:05 期间的结果？还是让过去的事情过去？理想情况下，我们将能够定义一个特定的时间段，在此期间事件将被添加到各自的时间片中。例如，如果事件延迟了四个小时，我们应该重新计算结果并更新。如果事件比那晚到达，我们可以忽略它们。

窗口可以与时钟时间对齐，即每分钟移动一次的五分钟窗口的第一个切片为 00:00–00:05，第二个切片为 00:01–00:06。或者它可以是不对齐的，只要启动应用程序就可以启动，然后第一个切片可以是 03:17–03:22。滑动窗口永远不会对齐，因为只要有新记录，它们就会移动。这两种类型的窗户的区别见下图。

Tumbling Window: 5-minute window, every 5 minutes.



Hopping Window: 5-minute window, every 1 minute.
Windows overlap, so events belong to multiple windows.



翻窗与跳窗



_____处理 Kafka 主题的推送消息。

1. *producer*
2. *consumer*
3. *Kafka broker*
4. *partition*

Answer:

1. *producer*

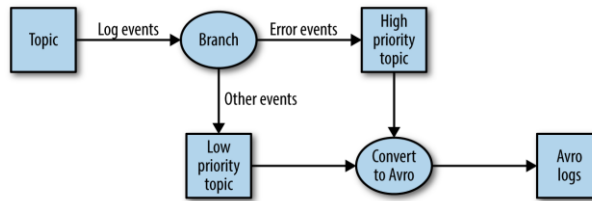
流处理设计模式

每一个流处理系统都是不同的，从消费者、处理逻辑和生产者的基本组合到涉及的集群，比如 Spark Streaming 及其机器学习库，等等。但是有一些基本的设计模式，它们是流处理架构的常见需求的已知解决方案。我们将回顾其中一些众所周知的模式，并通过几个示例说明如何使用它们。

■ 单事件处理

流处理的最基本模式是单独地处理每个事件。这也被称为 map/filter 模式，因为它通常用于从流中过滤不必要的事件或转换每个事件。（术语“map”基于 map/reduce 模式，其中 map stage 转换事件，reduce stage 聚合事件。）

在这种模式中，流处理应用程序使用流中的事件，修改每个事件，然后将事件生成到另一个流。例如，一个应用程序从流中读取日志消息并将错误事件写入高优先级流，将其余事件写入低优先级流。另一个例子是从流中读取事件并将其从 JSON 修改为 Avro 的应用程序。这样的应用程序需要在应用程序中保持状态，因为每个事件都可以独立处理。这意味着从应用程序故障或负载平衡中恢复非常容易，因为不需要恢复状态；您只需将事件交给应用程序的另一个实例来处理。使用简单的生产者和消费者可以很容易地处理这种模式，如图所示

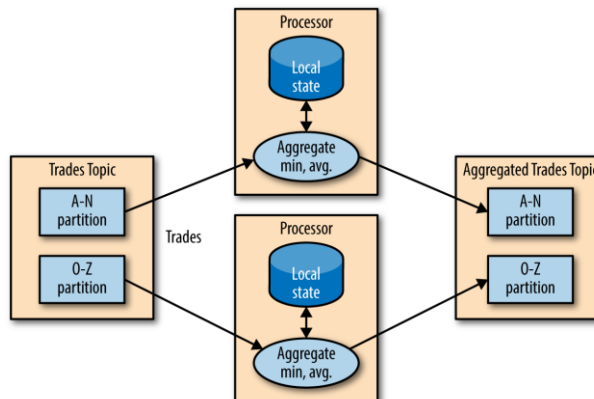


单事件处理

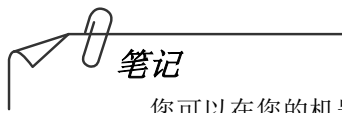
■ 本地处理状态

大多数流处理应用程序都关注聚合信息，尤其是时间窗口聚合。这方面的一个例子比如查找每天交易的最小和最高股票价格，并计算移动平均线。

这些聚合需要维护流的状态。在我们的示例中，为了计算每天的最低和平均价格，我们需要存储到当前时间之前看到的最小值和最大值，并将流中的每个新值与存储的最小值和最大值进行比较。所有这些都可以使用本地状态（而不是共享状态）来完成，因为我们示例中的每个操作都是一个分组聚合。也就是说，我们对每个股票符号进行聚合，而不是在整个股票市场上。我们使用 Kafka 分区器来确保具有相同存储码的所有事件都写入同一分区。然后，应用程序的每个实例将从分配给它的分区中获取所有事件（这是 Kafka 消费者保证）。这意味着应用程序的每个实例都可以维护写入分配给的分区的股票符号子集的状态它如图所示。



具有局部状态的事件处理



笔记

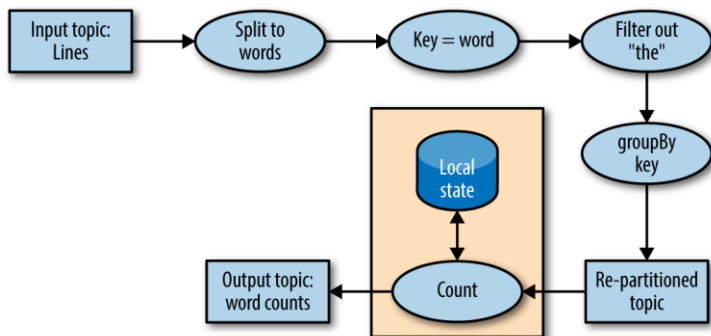
您可以在您的机器上运行整个示例，而无需安装除 Apache Kafka 之外的任何东西。

Kafka Streams: 架构概述

我们了解了如何使用 Kafka Streams API 来实现一些众所周知的流处理设计模式。但是为了更好地理解 Kafka 的 Streams 库是如何工作和扩展的，我们需要深入了解 API 背后的一些设计原则。

■ Building a Topology (构建拓扑)

每个 streams 应用程序至少实现和执行一个拓扑。拓扑（在其他流处理框架中也称为 DAG 或有向无环图）是一组操作和转换，每个事件都要经过这些操作和转换从输入到输出。图中显示了“字数统计”中的拓扑结构



字计数流处理示例的拓扑结构

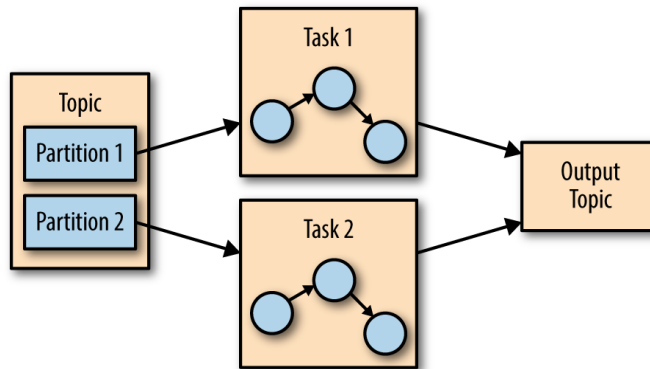
即使是一个简单的应用程序也有非平凡的拓扑结构。拓扑由处理器组成，这些处理器是拓扑图中的节点（在我们的图中用圆圈表示）。大多数处理器实现数据过滤、映射、聚合等操作。还有一些源处理器，它们使用来自主题的数据并将其传递给另一个主题；还有 sink 处理器，它们从较早的处理

器获取数据并将其生成到主题中。拓扑总是以一个或多个源处理器开始，以一个或多个接收器处理器结束。

■ Scaling the Topology (扩展拓扑)

Kafka Streams 通过允许在应用程序的一个实例中执行多个线程以及支持应用程序的分布式实例之间的负载平衡来进行扩展。您可以在一台具有多个线程的计算机上运行 Streams 应用程序，也可以在多台计算机上运行；无论哪种情况，应用程序中的所有活动线程都将平衡数据处理中涉及的工作。

Streams 引擎通过将拓扑拆分为任务来并行执行拓扑。任务数由 Streams 引擎确定，并取决于应用程序处理的主题中的分区数。每个任务负责分区的一个子集：任务将订阅这些分区并使用其中的事件。对于它使用的每个事件，该任务将按顺序执行应用于该分区的所有处理步骤，然后最终将结果写入接收器。这些任务是 Kafka 流中并行性的基本单位，因为每个任务都可以独立于其他任务执行。如下图所示。



运行相同拓扑的两个任务 — 输入主题中每个分区的一个

■ Surviving Failures (故障容错)

允许我们扩展应用程序的同一个模型也允许我们优雅地处理失败。首先，卡夫卡是高度可用的，因此我们保存到卡夫卡的数据也是高度可用的。因此，如果应用程序失败并需要重新启动，它可以从 Kafka 查找其在流中的最后一个位置，并从失败前提交的最后一个偏移量继续处理。注意，如果本地状态存储丢失（例如，因为我们需要替换存储它的服务器），streams 应用程序总是可以从它存储在 Kafka 中的更改日志重新创建它。

Kafka Streams 还利用 Kafka 的消费者协调功能为任务提供高可用性。如果任务失败，但存在活动的线程或 streams 应用程序的其他实例，则该任务将在其中一个可用线程上重新启动。这类似于使用者组如何处理组中某个使用者的故障，方法是将分区指派给其他使用者之一。

Stream 处理用例

流处理或连续处理在您希望以快速顺序处理事件而不是等待数小时直到下一批处理，但您不希望响应以毫秒为单位到达的情况下非常有用。这一切都是真的，但也是非常抽象的。让我们看看一些可以通过流处理解决的实际方案：

■ Customer Service（客户服务）

假设您刚刚在一家大型连锁酒店预订了一个房间，您希望收到电子邮件确认和收据。预订几分钟后，当确认书还没到时，你打电话给客服确认你的预订。假设客户服务台告诉您“我在我们的系统中看不到订单，但是将数据从预订系统加载到酒店和客户服务台的批处理作业每天只运行一次，所以请明天再打电话。你应该在 2-3 个工作日内看到这封邮件，“这听起来不是很好的服务，但我已经和一家大型连锁酒店进行过多次交谈。我们真正想要的是连锁酒店中的每一个系统在预订后的几秒钟或几分钟内都能得到有关新预订的更新，包括客户服务中心、酒店、发送电子邮件确认的系统、网站等。您还希望客户服务中心能够立即提供所有详细信息关于您过去访问过连锁酒店的任何一次，以及酒店的前台，以了解您是一位忠诚的客户，以便他们可以为您提供升级服务。使用流处理应用程序构建所有这些系统，使它们能够近实时地接收和处理更新，这有助于获得更好的客户体验。有了这样一个系统，我会在几分钟内收到一封确认邮件，我的信用卡将按时收费，收据将被发送，服务台可以立即回答我有关预订的问题。

■ Internet of Things（物联网）

物联网可以意味着很多事情，从一个用于调节温度和订购洗衣粉的家用设备到制药生产的实时质量控制。将流处理应用于传感器和设备时，一个非常常见的用例是尝试预测何时需要预防性维护。这与应用

程序监控类似，但应用于硬件，在许多行业都很常见，包括制造业、电信业（识别故障的手机塔）、有线电视（在用户投诉前识别出故障的机顶盒设备）等等。每个案例都有自己的模式，但目标是相似的：大规模处理来自设备的事件，并识别出设备需要维护的信号模式。这些模式可以是交换机丢下数据包，在制造过程中需要更大的力来拧紧螺丝，或者用户为了有线电视更频繁地重新启动盒子。

■ Fraud Detection（欺诈检测）：

也被称为异常检测，是一个非常广泛的领域，重点是捕捉系统中的“作弊者”或坏人。欺诈检测应用程序的示例包括检测信用卡欺诈、股票交易欺诈、视频游戏作弊和网络安全风险。在所有这些领域，尽早发现欺诈行为有很大的好处，因此，一个能够快速响应事件的近实时系统（可能在交易被批准前就阻止了一笔不良交易）比在清理工作复杂得多的情况下在三天后发现欺诈行为的批处理作业更为可取。这又是一个在大规模事件流中识别模式的问题。



Just a minute:

Streams 引擎通过将拓扑拆分为_____来并行化拓扑的执行。

1. *Stream*
2. 日志
3. 任务
4. 信息

Answer:

3. 任务



活动 6.1：字数统计

练习题

1. 流处理是指对一个或多个事件流进行的处理。
 - a. 该描述不对。
 - b. 该描述有时是对的。
 - c. 该描述是对的。
 - d. 该描述无论什么情况下都是错的
2. 事件一旦发生，就永远不会_____。
 - a. 更新
 - b. 修改
 - c. 无法修改
 - d. 以上都不是
3. _____只能由流处理应用程序的特定实例访问。
 - a. 本地状态
 - b. 外部状态
 - c. 内部状态
 - d. 本地或内部状态
4. _____用于确保具有相同存储码的所有事件都写入同一分区。
 - a. Kafka 分区
 - b. Kafka 生产者
 - c. Kafka 消费者
 - d. Kafka 成员
5. 以下哪个陈述是正确的？
 1. Kafka Streams 通过允许在应用程序的一个实例中执行多个线程，并支持应用程序分布式实例之间的负载平衡来进行扩展。
 2. Kafka Streams 没有利用 Kafka 的消费者协调来为任务提供高可用性。
 - a. 1 正确，2 错误
 - b. 1 错误，2 正确
 - c. 两个都正确
 - d. 两个都错误

摘要

在本章中，你学到了：

- 流处理是指一个或多个事件流的持续处理。Stream
- 流处理与请求-响应和批处理一样是一种编程范式。
- 事件流模型的其他属性，除了它们的无限性之外：
 1. 对事件流进行排序
 2. 数据记录不可变
 3. 事件流可重播
 4. 请求-响应
 5. 批处理
 6. 流处理
- 流处理与任何类型的数据处理非常相似，您编写代码接收数据，对数据进行一些转换、聚合、充实等操作。将结果放在某个地方。
- 下面列出了流处理的一些关键概念。
 1. 时间
 2. 状态
 3. 流和表的对偶性
 4. 时间窗口
- 每个流处理系统都不同于使用者、处理逻辑和生产者的基本组合，它们与 Spark Stream 及其机器学习库等涉及的群集不同，而且介于两者之间。
- 但也有一些基本的设计模式，这些模式是流处理体系结构的常见要求的已知解决方案。
- 我们将回顾一些众所周知的模式，并展示如何使用它们与几个示例。
 1. 单事件处理
 2. 本地处理状态
- Kafka Streams 架构概述
 3. 构建拓扑
 4. 扩展拓扑
 5. 故障容错

- 流处理或连续处理在以下情况下非常有用：您希望快速处理事件，而不是等待数小时后下一批处理，而且您不希望响应在毫秒内到达。