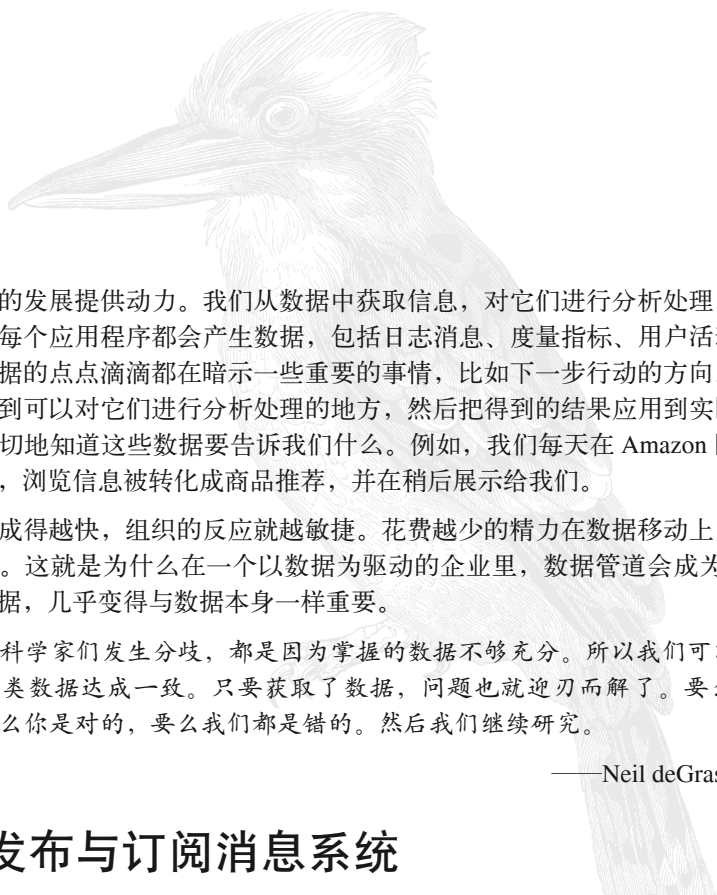


初识Kafka



数据为企业的发展提供动力。我们从数据中获取信息，对它们进行分析处理，然后生成更多的数据。每个应用程序都会产生数据，包括日志消息、度量指标、用户活动记录、响应消息等。数据的点点滴滴都在暗示一些重要的事情，比如下一步行动的方向。我们把数据从源头移动到可以对它们进行分析处理的地方，然后把得到的结果应用到实际场景中，这样才能够确切地知道这些数据要告诉我们什么。例如，我们每天在 Amazon 网站上浏览感兴趣的商品，浏览信息被转化成商品推荐，并在稍后展示给我们。

这个过程完成得越快，组织的反应就越敏捷。花费越少的精力在数据移动上，就越能专注于核心业务。这就是为什么在一个以数据为驱动的企业里，数据管道会成为关键性组件。如何移动数据，几乎变得与数据本身一样重要。

每一次科学家们发生分歧，都是因为掌握的数据不够充分。所以我们可以先就获取哪一类数据达成一致。只要获取了数据，问题也就迎刃而解了。要么我是对的，要么你是对的，要么我们都是错的。然后我们继续研究。

——Neil deGrasse Tyson

1.1 发布与订阅消息系统

在正式讨论 Apache Kafka（以下简称 Kafka）之前，先来了解发布与订阅消息系统的概念，并认识这个系统的重要性。数据（消息）的发送者（发布者）不会直接把消息发送给接收者，这是发布与订阅消息系统的一个特点。发布者以某种方式对消息进行分类，接收者（订阅者）订阅它们，以便接收特定类型的消息。发布与订阅系统一般会有一个 broker，也就是发布消息的中心点。

1.1.1 如何开始

发布与订阅消息系统的大部分应用场景都是从一个简单的消息队列或一个进程间通道开始的。例如，你的应用程序需要往别处发送监控信息，可以直接在你的应用程序和另一个可以在仪表盘上显示度量指标的应用程序之间建立连接，然后通过这个连接推送度量指标，如图 1-1 所示。

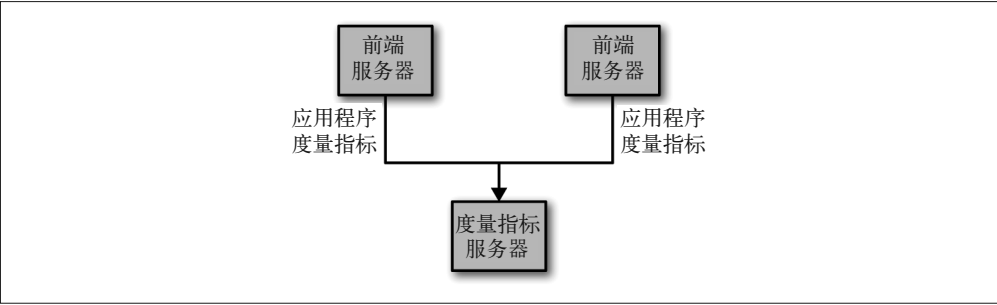


图 1-1：单个直连的度量指标发布者

这是刚接触监控系统时简单问题的应对方案。过了不久，你需要分析更长时间片段的度量指标，而此时的仪表盘程序满足不了需求，于是，你启动了一个新的服务来接收度量指标。该服务把度量指标保存起来，然后进行分析。与此同时，你修改了原来的应用程序，把度量指标同时发送到两个仪表盘系统上。现在，你又多了 3 个可以生成度量指标的应用程序，它们都与这两个服务直接相连。而你的同事认为最好可以对这些服务进行轮询以便获得告警功能，于是你为每一个应用程序增加了一个服务器，用于提供度量指标。再过一阵子，有更多的应用程序出于各自的目的，都从这些服务器获取度量指标。这时的架构看起来就像图 1-2 所示的那样，节点间的连接一团糟。

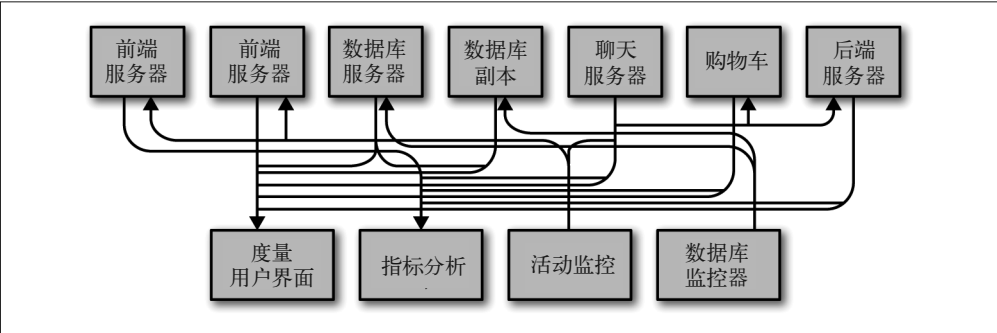


图 1-2：多个直连的度量指标发布者

这时，技术债务开始凸显出来，于是你决定偿还掉一些。你创建了一个独立的应用程序，用于接收来自其他应用程序的度量指标，并为其他系统提供了一个查询服务器。这样，之前架构的复杂度被降低到图 1-3 所示的那样。那么恭喜你，你已经创建了一个基于发布与订阅的消息系统。

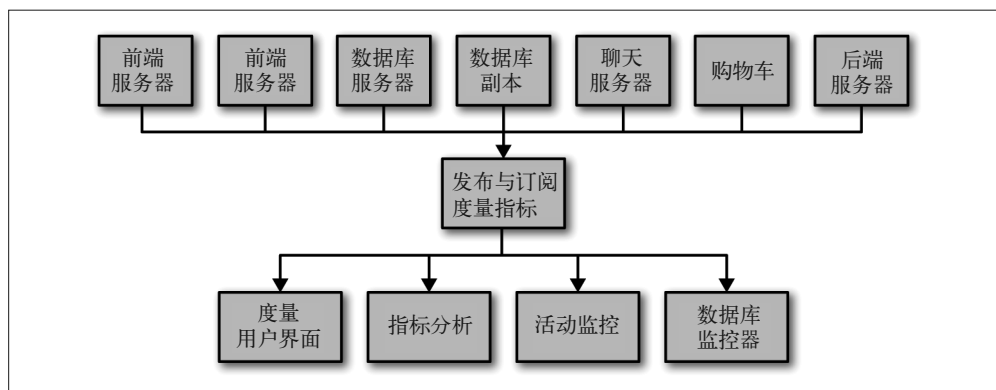


图 1-3: 度量指标发布与订阅系统

1.1.2 独立的队列系统

在你跟度量指标打得不可开交的时候，你的一个同事也正在跟日志消息奋战。还有另一个同事正在跟踪网站用户的行为，为负责机器学习开发的同事提供信息，同时为管理团队生成报告。你和同事们使用相同的方式创建这些系统，解耦信息的发布者和订阅者。图 1-4 所示的架构包含了 3 个独立的发布与订阅系统。

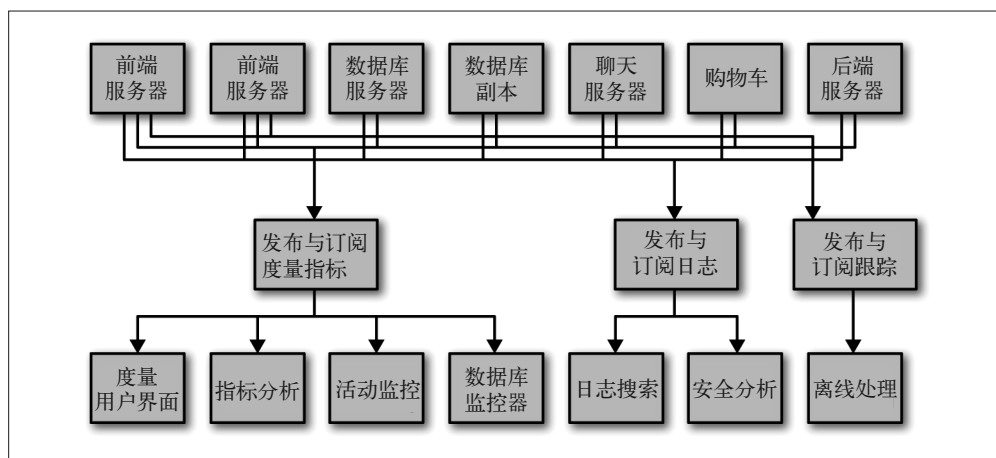


图 1-4: 多个发布与订阅系统

这种方式比直接使用点对点的连接（图 1-2）要好得多，但这里有太多重复的地方。你的公司因此要为数据队列维护多个系统，每个系统又有各自的缺陷和不足。而且，接下来可能会有更多的场景需要用到消息系统。此时，你真正需要的是一个单一的集中式系统，它可以用来发布通用类型的数据，其规模可以随着公司业务的增长而增长。

1.2 Kafka登场

Kafka 就是为了解决上述问题而设计的一款基于发布与订阅的消息系统。它一般被称为“分布式提交日志”或者“分布式流平台”。文件系统或数据库提交日志用来提供所有事务的持久记录，通过重放这些日志可以重建系统的状态。同样地，Kafka 的数据是按照一定顺序持久化保存的，可以按需读取。此外，Kafka 的数据分布在整个系统里，具备数据故障保护和性能伸缩能力。

1.2.1 消息和批次

Kafka 的数据单元被称为**消息**。如果你在使用 Kafka 之前已经有数据库使用经验，那么可以把消息看成是数据库里的一个“数据行”或一条“记录”。消息由字节数组组成，所以对于 Kafka 来说，消息里的数据没有特别的格式或含义。消息可以有一个可选的元数据，也就是**键**。键也是一个字节数组，与消息一样，对于 Kafka 来说也没有特殊的含义。当消息以一种可控的方式写入不同的分区时，会用到键。最简单的例子就是为键生成一个一致性散列值，然后使用散列值对主题分区数进行取模，为消息选取分区。这样可以保证具有相同键的消息总是被写到相同的分区上。第 3 章将详细介绍键的用法。

为了提高效率，消息被分批次写入 Kafka。**批次**就是一组消息，这些消息属于同一个主题和分区。如果每一个消息都单独穿行于网络，会导致大量的网络开销，把消息分成批次传输可以减少网络开销。不过，这要在时间延迟和吞吐量之间作出权衡：批次越大，单位时间内处理的消息就越多，单个消息的传输时间就越长。批次数据会被压缩，这样可以提升数据的传输和存储能力，但要做更多的计算处理。

1.2.2 模式

对于 Kafka 来说，消息不过是晦涩难懂的字节数组，所以有人建议用一些额外的结构来定义消息内容，让它们更易于理解。根据应用程序的需求，消息**模式**（schema）有许多可用的选项。像 JSON 和 XML 这些简单的系统，不仅易用，而且可读性好。不过，它们缺乏强类型处理能力，不同版本之间的兼容性也不是很好。Kafka 的许多开发者喜欢使用 Apache Avro，它最初是为 Hadoop 开发的一款序列化框架。Avro 提供了一种紧凑的序列化格式，模式和消息体是分开的，当模式发生变化时，不需要重新生成代码；它还支持强类型和模式进化，其版本既向前兼容，也向后兼容。

数据格式的一致性对于 Kafka 来说很重要，它消除了消息读写操作之间的耦合性。如果读写操作紧密地耦合在一起，消息订阅者需要升级应用程序才能同时处理新旧两种数据格式。在消息订阅者升级了之后，消息发布者才能跟着升级，以便使用新的数据格式。新的应用程序如果需要使用数据，就要与消息发布者发生耦合，导致开发者需要做很多繁杂的工作。定义良好的模式，并把它们存放在公共仓库，可以方便我们理解 Kafka 的消息结构。第 3 章将详细讨论模式和序列化。

1.2.3 主题和分区

Kafka 的消息通过**主题**进行分类。主题就好比数据库的表，或者文件系统里的文件夹。主题可以被分为若干个**分区**，一个分区就是一个提交日志。消息以追加的方式写入分区，然后以先入先出的顺序读取。要注意，由于一个主题一般包含几个分区，因此无法在整个主题范围内保证消息的顺序，但可以保证消息在单个分区内的顺序。图 1-5 所示的主题有 4 个分区，消息被追加写入每个分区的尾部。Kafka 通过分区来实现数据冗余和伸缩性。分区可以分布在不同的服务器上，也就是说，一个主题可以横跨多个服务器，以此来提供比单个服务器更强大的性能。

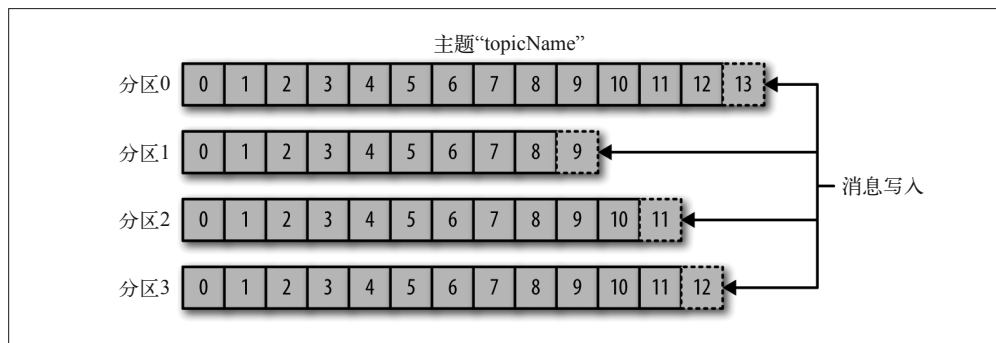


图 1-5: 包含多个分区的主题表示

我们通常会使用**流**这个词来描述 Kafka 这类系统的数据。很多时候，人们把一个主题的数据看成一个流，不管它有多少个分区。流是一组从生产者移动到消费者的数据。当我们讨论流式处理时，一般都是这样描述消息的。Kafka Streams、Apache Samza 和 Storm 这些框架以实时的方式处理消息，也就是所谓的流式处理。我们可以将流式处理与离线处理进行比较，比如 Hadoop 就是被设计用于在稍后某个时刻处理大量的数据。第 11 章将会介绍流式处理。

1.2.4 生产者和消费者

Kafka 的客户端就是 Kafka 系统的用户，它们被分为两种基本类型：生产者和消费者。除此之外，还有其他高级客户端 API——用于数据集成的 Kafka Connect API 和用于流式处理的 Kafka Streams。这些高级客户端 API 使用生产者和消费者作为内部组件，提供了高级的功能。

生产者创建消息。在其他发布与订阅系统中，生产者可能被称为**发布者**或**写入者**。一般情况下，一个消息会被发布到一个特定的主题上。生产者在默认情况下把消息均衡地分布到主题的所有分区上，而并不关心特定消息会被写到哪个分区。不过，在某些情况下，生产者会把消息直接写到指定的分区。这通常是通过消息键和分区器来实现的，分区器为键生成一个散列值，并将其映射到指定的分区上。这样可以保证包含同一个键的消息会被写到同一个分区上。生产者也可以使用自定义的分区器，根据不同的业务规则将消息映射到分区。第 3 章将详细介绍生产者。

消费者读取消息。在其他发布与订阅系统中，消费者可能被称为订阅者或读者。消费者订阅一个或多个主题，并按照消息生成的顺序读取它们。消费者通过检查消息的偏移量来区分已经读取过的消息。偏移量是另一种元数据，它是一个不断递增的整数值，在创建消息时，Kafka 会把它添加到消息里。在给定的分区里，每个消息的偏移量都是唯一的。消费者把每个分区最后读取的消息偏移量保存在 Zookeeper 或 Kafka 上，如果消费者关闭或重启，它的读取状态不会丢失。

消费者是消费者群组的一部分，也就是说，会有一个或多个消费者共同读取一个主题。群组保证每个分区只能被一个消费者使用。图 1-6 所示的群组中，有 3 个消费者同时读取一个主题。其中的两个消费者各自读取一个分区，另外一个消费者读取其他两个分区。消费者与分区之间的映射通常被称为消费者对分区的所有权关系。

通过这种方式，消费者可以消费包含大量消息的主题。而且，如果一个消费者失效，群组里的其他消费者可以接管失效消费者的工作。第 4 章将详细介绍消费者和消费者群组。

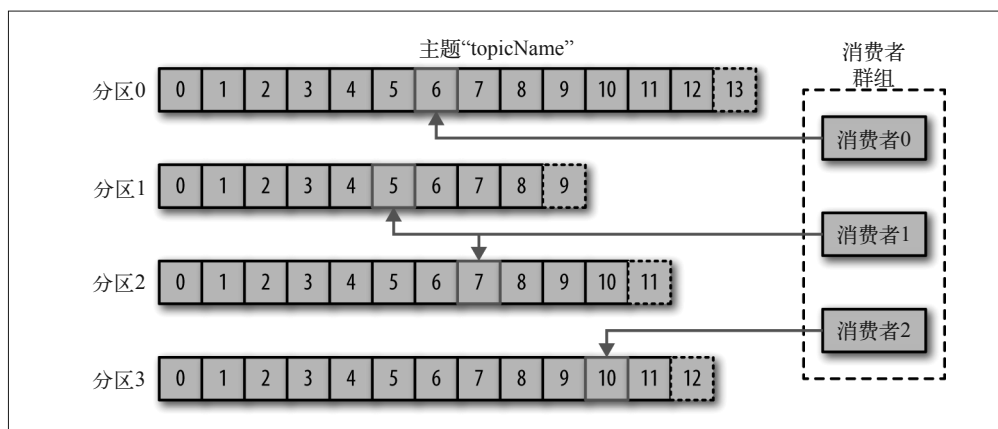


图 1-6: 消费者群组从主题读取消息

1.2.5 broker和集群

一个独立的 Kafka 服务器被称为 broker。broker 接收来自生产者的消息，为消息设置偏移量，并提交消息到磁盘保存。broker 为消费者提供服务，对读取分区的请求作出响应，返回已经提交到磁盘上的消息。根据特定的硬件及其性能特征，单个 broker 可以轻松处理数千个分区以及每秒百万级的消息量。

broker 是集群的组成部分。每个集群都有一个 broker 同时充当了集群控制器的角色（自动从集群的活跃成员中选举出来）。控制器负责管理工作，包括将分区分配给 broker 和监控 broker。在集群中，一个分区从属于一个 broker，该 broker 被称为分区的首领。一个分区可以分配给多个 broker，这个时候会发生分区复制（见图 1-7）。这种复制机制为分区提供了消息冗余，如果有一个 broker 失效，其他 broker 可以接管领导权。不过，相关的消费者和生产者都要重新连接到新的首领。第 6 章将详细介绍集群的操作，包括分区复制。

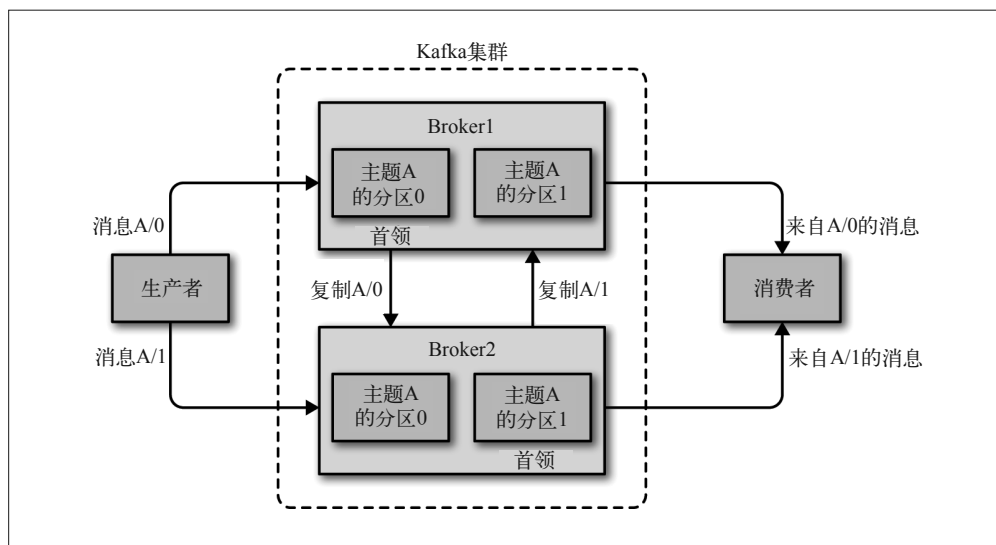


图 1-7：集群里的分区复制

保留消息（在一定期限内）是 Kafka 的一个重要特性。Kafka broker 默认的消息保留策略是这样的：要么保留一段时间（比如 7 天），要么保留到消息达到一定大小的字节数（比如 1GB）。当消息数量达到这些上限时，旧消息就会过期并被删除，所以在任何时刻，可用消息的总量都不会超过配置参数所指定的大小。主题可以配置自己的保留策略，可以将消息保留到不再使用它们为止。例如，用于跟踪用户活动的数据可能需要保留几天，而应用程序的度量指标可能只需要保留几个小时。可以通过配置把主题当作紧凑型日志，只有最后一个带有特定键的消息会被保留下来。这种情况对于变更日志类型的数据来说比较适用，因为人们只关心最后时刻发生的那个变更。

1.2.6 多集群

随着 Kafka 部署数量的增加，基于以下几点原因，最好使用多个集群。

- 数据类型分离
- 安全需求隔离
- 多数据中心（灾难恢复）

如果使用多个数据中心，就需要在它们之间复制消息。这样，在线应用程序才可以访问到多个站点的用户活动信息。例如，如果一个用户修改了他们的资料信息，不管从哪个数据中心都应该能看到这些改动。或者多个站点的监控数据可以被聚集到一个部署了分析程序和告警系统的中心位置。不过，Kafka 的消息复制机制只能在单个集群内进行，不能在多个集群之间进行。

Kafka 提供了一个叫作 **MirrorMaker** 的工具，可以用它来实现集群间的消息复制。MirrorMaker 的核心组件包含了一个生产者和一个消费者，两者之间通过一个队列相连。

消费者从一个集群读取消息，生产者把消息发送到另一个集群上。图 1-8 展示了一个使用 MirrorMaker 的例子，两个“本地”集群的消息被聚集到一个“聚合”集群上，然后将该集群复制到其他数据中心。不过，这种方式在创建复杂的数据管道方面显得有点力不从心。第 7 章将详细讨论这些案例。

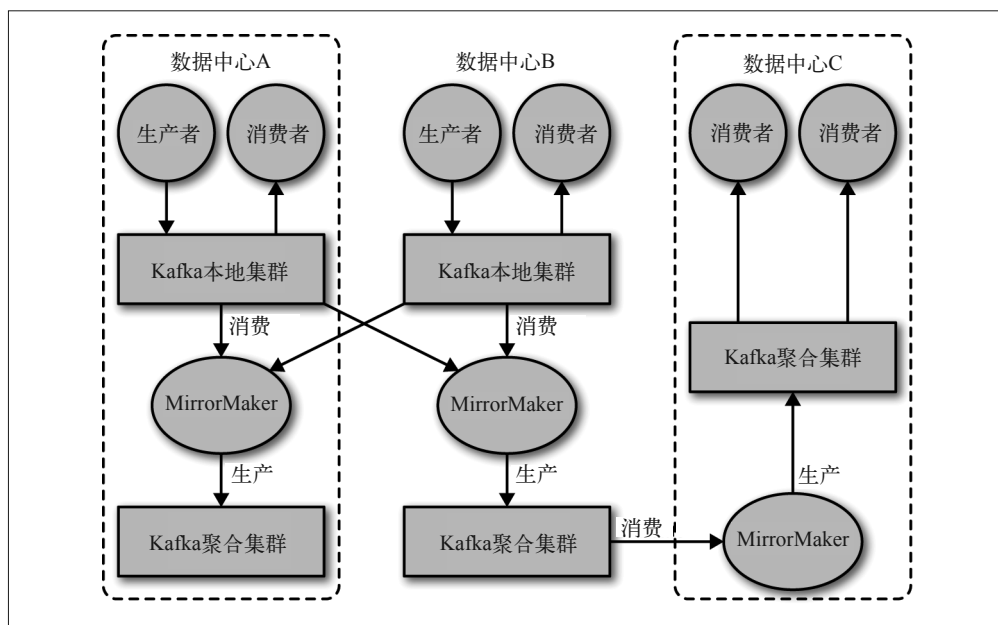


图 1-8: 多数据中心架构

1.3 为什么选择Kafka

基于发布与订阅的消息系统那么多，为什么 Kafka 会是一个更好的选择呢？

1.3.1 多个生产者

Kafka 可以无缝地支持多个生产者，不管客户端在使用单个主题还是多个主题。所以它很适合用来从多个前端系统收集数据，并以统一的格式对外提供数据。例如，一个包含了多个微服务的网站，可以为页面视图创建一个单独的主题，所有服务都以相同的消息格式向该主题写入数据。消费者应用程序会获得统一的页面视图，而无需协调来自不同生产者的数据流。

1.3.2 多个消费者

除了支持多个生产者外，Kafka 也支持多个消费者从一个单独的消息流上读取数据，而且消费者之间互不影响。这与其他队列系统不同，其他队列系统的消息一旦被一个客户端读取，其他客户端就无法再读取它。另外，多个消费者可以组成一个群组，它们共享一个消

息流，并保证整个群组对每个给定的消息只处理一次。

1.3.3 基于磁盘的数据存储

Kafka 不仅支持多个消费者，还允许消费者非实时地读取消息，这要归功于 Kafka 的数据保留特性。消息被提交到磁盘，根据设置的保留规则进行保存。每个主题可以设置单独的保留规则，以便满足不同消费者的需求，各个主题可以保留不同数量的消息。消费者可能会因为处理速度慢或突发的流量高峰导致无法及时读取消息，而持久化数据可以保证数据不会丢失。消费者可以在进行应用程序维护时离线一小段时间，而无需担心消息丢失或堵塞在生产者端。消费者可以被关闭，但消息会继续保留在 Kafka 里。消费者可以从上次中断的地方继续处理消息。

1.3.4 伸缩性

为了能够轻松处理大量数据，Kafka 从一开始就被设计成一个具有灵活伸缩性的系统。用户在开发阶段可以先使用单个 broker，再扩展到包含 3 个 broker 的小型开发集群，然后随着数据量不断增长，部署到生产环境的集群可能包含上百个 broker。对在线集群进行扩展丝毫不影响整体系统的可用性。也就是说，一个包含多个 broker 的集群，即使个别 broker 失效，仍然可以持续地为客户提供服务。要提高集群的容错能力，需要配置较高的复制系数。第 6 章将讨论关于复制的更多细节。

1.3.5 高性能

上面提到的所有特性，让 Kafka 成为了一个高性能的发布与订阅消息系统。通过横向扩展生产者、消费者和 broker，Kafka 可以轻松处理巨大的消息流。在处理大量数据的同时，它还能保证亚秒级的消息延迟。

1.4 数据生态系统

已经有很多应用程序加入到了数据处理的大军中。我们定义了输入和应用程序，负责生成数据或者把数据引入系统。我们定义了输出，它们可以是度量指标、报告或者其他类型的数据。我们创建了一些循环，使用一些组件从系统读取数据，对读取的数据进行处理，然后把它们导到数据基础设施上，以备不时之需。数据类型可以多种多样，每一种数据类型可以有不同的内容、大小和用途。

Kafka 为数据生态系统带来了循环系统，如图 1-9 所示。它在基础设施的各个组件之间传递消息，为所有客户端提供一致的接口。当与提供消息模式的系统集成时，生产者和消费者之间不再有紧密的耦合，也不需要它们之间建立任何类型的直连。我们可以根据业务需要添加或移除组件，因为生产者不再关心谁在使用数据，也不关心有多少个消费者。

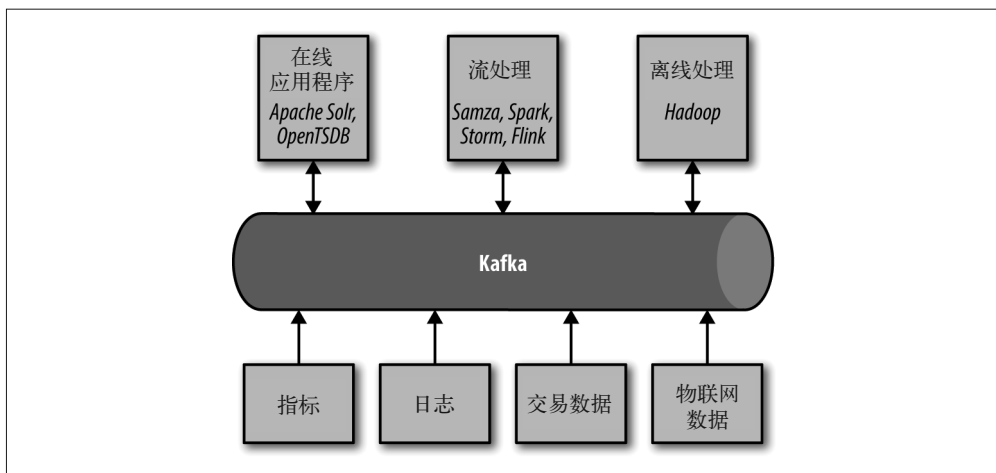


图 1-9: 大数据生态系统

使用场景

1. 活动跟踪

Kafka 最初的使用场景是跟踪用户的活动。网站用户与前端应用程序发生交互，前端应用程序生成用户活动相关的消息。这些消息可以是一些静态的信息，比如页面访问次数和点击量，也可以是一些复杂的操作，比如添加用户资料。这些消息被发布到一个或多个主题上，由后端应用程序负责读取。这样，我们就可以生成报告，为机器学习系统提供数据，更新搜索结果，或者实现其他更多的功能。

2. 传递消息

Kafka 的另一个基本用途是传递消息。应用程序向用户发送通知（比如邮件）就是通过传递消息来实现的。这些应用程序组件可以生成消息，而不需要关心消息的格式，也不需要关心消息是如何被发送的。一个公共应用程序会读取这些消息，对它们进行处理：

- 格式化消息（也就是所谓的装饰）；
- 将多个消息放在同一个通知里发送；
- 根据用户配置的首选项来发送数据。

使用公共组件的好处在于，不需要在多个应用程序上开发重复的功能，而且可以在公共组件上做一些有趣的转换，比如把多个消息聚合成一个单独的通知，而这些工作是无法在其他地方完成的。

3. 度量指标和日志记录

Kafka 也可以用于收集应用程序和系统度量指标以及日志。Kafka 支持多个生产者的特性在这个时候就可以派上用场。应用程序定期把度量指标发布到 Kafka 主题上，监控系统或告警系统读取这些消息。Kafka 也可以用在像 Hadoop 这样的离线系统上，进行较长时间片段

的数据分析，比如年度增长走势预测。日志消息也可以被发布到 Kafka 主题上，然后被路由到专门的日志搜索系统（比如 Elasticsearch）或安全分析应用程序。更改目标系统（比如日志存储系统）不会影响到前端应用或聚合方法，这是 Kafka 的另一个优点。

4. 提交日志

Kafka 的基本概念来源于提交日志，所以使用 Kafka 作为提交日志是件顺理成章的事。我们可以把数据库的更新发布到 Kafka 上，应用程序通过监控事件流来接收数据库的实时更新。这种变更日志流也可以用于把数据库的更新复制到远程系统上，或者合并多个应用程序的更新到一个单独的数据库视图上。数据持久化为变更日志提供了缓冲区，也就是说，如果消费者应用程序发生故障，可以通过重放这些日志来恢复系统状态。另外，紧凑型日志主题只为每个键保留一个变更数据，所以可以长时间使用，不需要担心消息过期问题。

5. 流处理

流处理是又一个能提供多种类型应用程序的领域。可以说，它们提供的功能与 Hadoop 里的 map 和 reduce 有点类似，只不过它们操作的是实时数据流，而 Hadoop 则处理更长时间片段的数据，可能是几个小时或者几天，Hadoop 会对这些数据进行批处理。通过使用流式处理框架，用户可以编写小型应用程序来操作 Kafka 消息，比如计算度量指标，为其他应用程序有效地处理消息分区，或者对来自多个数据源的消息进行转换。第 11 章将通过其他案例介绍流处理。

1.5 起源故事

Kafka 是为了解决 LinkedIn 数据管道问题应运而生的。它的设计目的是提供一个高性能的消息系统，可以处理多种数据类型，并能够实时提供纯净且结构化的用户活动数据和系统度量指标。

数据为我们所做的每一件事提供了动力。

——Jeff Weiner, LinkedIn CEO

1.5.1 LinkedIn 的问题

本章开头提到过，LinkedIn 有一个数据收集系统和应用程序指标，它使用自定义的收集器和一些开源工具来保存和展示内部数据。除了跟踪 CPU 使用率和应用性能这些一般性指标外，LinkedIn 还有一个比较复杂的用户请求跟踪功能。它使用了监控系统，可以跟踪单个用户的请求是如何在内部应用间传播的。不过监控系统存在很多不足。它使用的是轮询拉取度量指标的方式，指标之间的时间间隔较长，而且没有自助服务能力。它使用起来不太方便，很多简单的任务需要人工介入才能完成，而且一致性较差，同一个度量指标的名字在不同系统里的叫法不一样。

与此同时，我们还创建了另一个用于收集用户活动信息的系统。这是一个 HTTP 服务，前端的服务器会定期连接进来，在上面发布一些消息（XML 格式）。这些消息文件被转移到线下进行解析和校对。同样，这个系统也存在很多不足。XML 文件的格式无法保持一致，

而且解析 XML 文件非常耗费计算资源。要想更改所创建的活动类型，需要在前端应用和离线处理程序之间做大量的协调工作。即使是这样，在更改数据结构时，仍然经常出现系统崩溃现象。而且批处理时间以小时计算，无法用它完成实时的任务。

监控和用户活动跟踪无法使用同一个后端服务。监控服务太过笨重，数据格式不适用于活动跟踪，而且无法在活动跟踪中使用轮询拉取模型。另一方面，把跟踪服务用在度量指标上也过于脆弱，批处理模型不适用于实时的监控和告警。不过，好在数据间存在很多共性，信息（比如特定类型的用户活动对应用程序性能的影响）之间的关联度还是很高的。特定类型用户活动数量的下降说明相关应用程序存在问题，不过批处理的长时间延迟意味着无法对这类问题作出及时的反馈。

最开始，我们调研了一些现成的开源解决方案，希望能够找到一个系统，可以实时访问数据，并通过横向扩展来处理大量的消息。我们使用 ActiveMQ 创建了一个原型系统，但它当时还无法满足横向扩展的需求。LinkedIn 不得不使用这种脆弱的解决方案，虽然 ActiveMQ 有很多缺陷会导致 broker 暂停服务。客户端的连接因此被阻塞，处理用户请求的能力也受到影响。于是我们最后决定构建自己的基础设施。

1.5.2 Kafka的诞生

LinkedIn 的开发团队由 Jay Kreps 领导。Jay Kreps 是 LinkedIn 的首席工程师，之前负责分布式键值存储系统 Voldemort 的开发。初建团队成员还包括 Neha Narkhede，不久之后，Jun Rao 也加入了进来。他们一起着手创建一个消息系统，可以同时满足上述的两种需求，并且可以在未来进行横向扩展。他们的主要目标如下：

- 使用推送和拉取模型解耦生产者和消费者；
- 为消息传递系统中的消息提供数据持久化，以便支持多个消费者；
- 通过系统优化实现高吞吐量；
- 系统可以随着数据流的增长进行横向扩展。

最后我们看到的这个发布与订阅消息系统具有典型的消息系统接口，但从存储层来看，它更像是一个日志聚合系统。Kafka 使用 Avro 作为消息序列化框架，每天高效地处理数十亿级别的度量指标和用户活动跟踪信息。LinkedIn 已经拥有超过万亿级别的消息使用量（截止到 2015 年 8 月），而且每天仍然需要处理超过千万亿字节的数据。

1.5.3 走向开源

2010 年底，Kafka 作为开源项目在 GitHub 上发布。2011 年 7 月，因为倍受开源社区的关注，它成为 Apache 软件基金会的孵化器项目。2012 年 10 月，Kafka 从孵化器项目毕业。从那时起，来自 LinkedIn 内部的开发团队一直为 Kafka 提供大力支持，而且吸引了大批来自 LinkedIn 外部的贡献者和参与者。现在，Kafka 被很多组织用在一些大型的数据管道上。2014 年秋天，Jay Kreps、Neha Narkhede 和 Jun Rao 离开 LinkedIn，创办了 Confluent。Confluent 是一个致力于为企业开发提供支持、为 Kafka 提供培训的公司。这两家公司连同来自开源社区持续增长的贡献力量，一直在开发和维护 Kafka，让 Kafka 成为大数据管道的不二之选。

1.5.4 命名

关于 Kafka 的历史，人们经常会问到的一个问题就是，Kafka 这个名字是怎么想出来的，以及这个名字和这个项目之间有着怎样的联系。对于这个问题，Jay Kreps 解释如下：

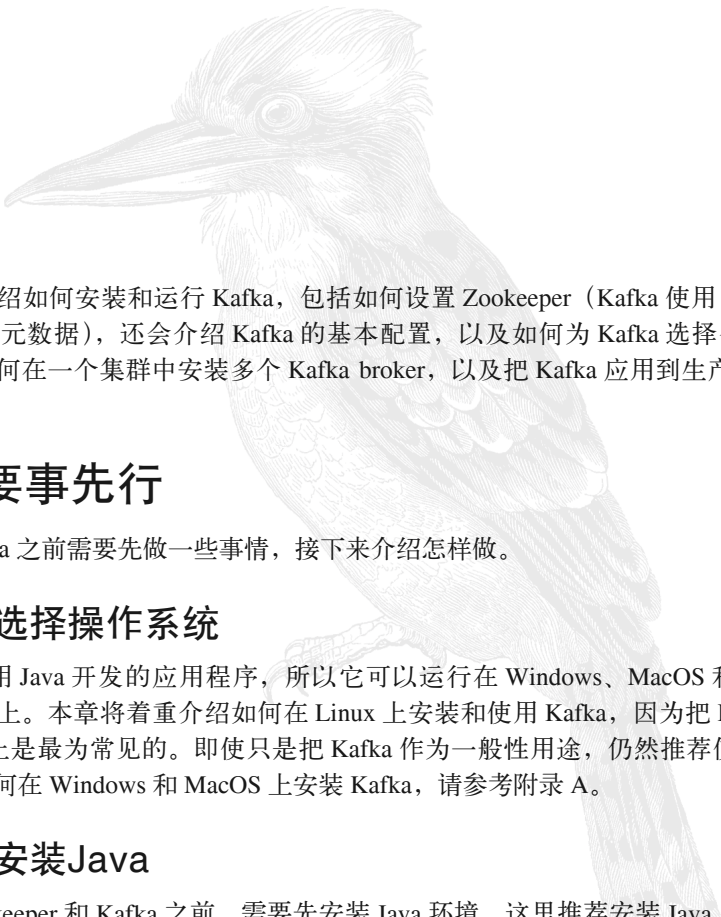
我想既然 Kafka 是为了写数据而产生的，那么用作家的名字来命名会显得更有意义。我在大学时期上过很多文学课程，很喜欢 Franz Kafka。况且，对于开源项目来说，这个名字听起来很酷。因此，名字和应用本身基本没有太多联系。

1.6 开始Kafka之旅

现在我们对 Kafka 已经有了一个大体的了解，还知道了一些常见的术语，接下来可以开始使用 Kafka 来创建数据管道了。在下一章，我们将探究如何安装和配置 Kafka，还会讨论如何选择合适的硬件来运行 Kafka，以及把 Kafka 应用到生产环境需要注意的事项。

第2章

安装Kafka



这一章将介绍如何安装和运行 Kafka，包括如何设置 Zookeeper（Kafka 使用 Zookeeper 保存 Broker 的元数据），还会介绍 Kafka 的基本配置，以及如何为 Kafka 选择合适的硬件，最后介绍如何在一个集群中安装多个 Kafka broker，以及把 Kafka 应用到生产环境需要注意的事项。

2.1 要事先行

在使用 Kafka 之前需要先做一些事情，接下来介绍怎样做。

2.1.1 选择操作系统

Kafka 是使用 Java 开发的应用程序，所以它可以运行在 Windows、MacOS 和 Linux 等多种操作系统上。本章将着重介绍如何在 Linux 上安装和使用 Kafka，因为把 Kafka 安装在 Linux 系统上是最为常见的。即使只是把 Kafka 作为一般性用途，仍然推荐使用 Linux 系统。关于如何在 Windows 和 MacOS 上安装 Kafka，请参考附录 A。

2.1.2 安装Java

在安装 Zookeeper 和 Kafka 之前，需要先安装 Java 环境。这里推荐安装 Java 8，可以使用系统自带的安装包，也可以直接从 java.com 网站下载。虽然运行 Zookeeper 和 Kafka 只需要 Java 运行时版本，但也可以安装完整的 JDK，以备不时之需。假设 JDK 8 update 51 已经安装在 /usr/java/jdk1.8.0_51 目录下，其他软件的安装都是基于这个前提进行的。

2.1.3 安装Zookeeper

Kafka 使用 Zookeeper 保存集群的元数据信息和消费者信息。Kafka 发行版自带了 Zookeeper，可以直接从脚本启动，不过安装一个完整版的 Zookeeper 也并不费劲。

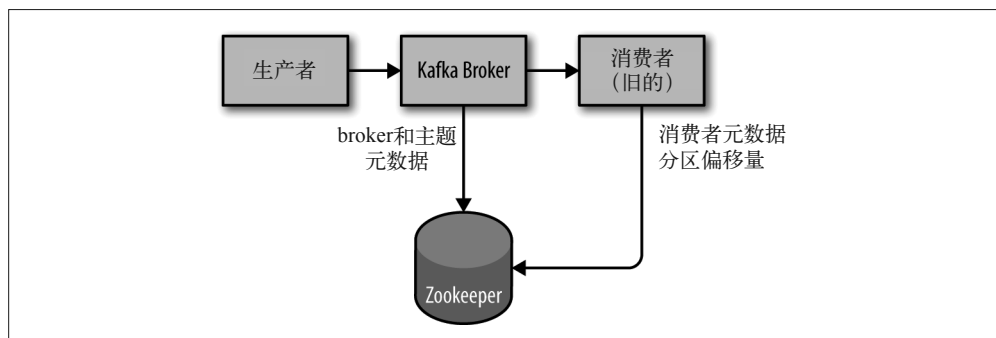


图 2-1: Kafka 和 Zookeeper

Zookeeper 的 3.4.6 稳定版已经在 Kafka 上做过全面测试，可以从 [apache.org](http://bit.ly/2sDWSgJ) 下载该版本的 Zookeeper：http://bit.ly/2sDWSgJ。

1. 单机服务

下面的例子演示了如何使用基本的配置安装 Zookeeper，安装目录为 `/usr/local/zookeeper`，数据目录为 `/var/lib/zookeeper`。

```
# tar -zxf 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
#
```

现在可以连到 Zookeeper 端口上，通过发送四字命令 `srvr` 来验证 Zookeeper 是否安装正确。

```
# 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.
#
```

2. Zookeeper 群组 (Ensemble)

Zookeeper 集群被称为**群组**。Zookeeper 使用的是一致性协议，所以建议每个群组里应该包含奇数个节点（比如 3 个、5 个等），因为只有当群组里的大多数节点（也就是法定人数）处于可用状态，Zookeeper 才能处理外部的请求。也就是说，如果你有一个包含 3 个节点的群组，那么它允许一个节点失效。如果群组包含 5 个节点，那么它允许 2 个节点失效。



群组节点个数的选择

假设有一个包含 5 个节点的群组，如果要对群组做一些包括更换节点在内的配置更改，需要依次重启每一个节点。如果你的群组无法容忍多个节点失效，那么在群组维护时就会存在风险。不过，也不建议一个群组包含超过 7 个节点，因为 Zookeeper 使用了一致性协议，节点过多会降低整个群组的性能。

群组需要有一些公共配置，上面列出了所有服务器的清单，并且每个服务器还要在数据目录中创建一个 myid 文件，用于指明自己的 ID。如果群组里服务器的机器名是 zoo1.example.com、zoo2.example.com、zoo3.example.com，那么配置文件可能是这样的：

```
tickTime=2000
dataDir=/var/lib/zookeeper
clientPort=2181
initLimit=20
syncLimit=5
server.1=zoo1.example.com:2888:3888
server.2=zoo2.example.com:2888:3888
server.3=zoo3.example.com:2888:3888
```

在这个配置中，initLimit 表示用于在从节点与主节点之间建立初始化连接的时间上限，syncLimit 表示允许从节点与主节点处于不同步状态的时间上限。这两个值都是 tickTime 的倍数，所以 initLimit 是 20*2000ms，也就是 40s。配置里还列出了群组中所有服务器的地址。服务器地址遵循 server.X=hostname:peerPort:leaderPort 格式，各个参数说明如下：

X

服务器的 ID，它必须是一个整数，不过不一定要从 0 开始，也不要求是连续的；

hostname

服务器的机器名或 IP 地址；

peerPort

用于节点间通信的 TCP 端口；

leaderPort

用于首领选举的 TCP 端口。

客户端只需要通过 clientPort 就能连接到群组，而群组节点间的通信则需要同时用到这 3 个端口（peerPort、leaderPort、clientPort）。

除了公共的配置文件外，每个服务器都必须在 data Dir 目录中创建一个叫作 myid 的文件，文件里要包含服务器 ID，这个 ID 要与配置文件里配置的 ID 保持一致。完成这些步骤后，就可以启动服务器，让它们彼此间进行通信了。

2.2 安装Kafka Broker

配置好 Java 和 Zookeeper 之后，接下来就可以安装 Kafka 了。可以从 <http://kafka.apache.org/downloads.html> 下载最新版本的 Kafka。截至本书写作时，Kafka 的版本是 0.9.0.1，对应的 Scala 版本是 2.11.0。

下面的例子将 Kafka 安装在 /usr/local/kafka 目录下，使用之前配置好的 Zookeeper，并把消息日志保存在 /tmp/kafka-logs 目录下。

```
# tar -zxf 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
#
```

一旦 Kafka 创建完毕，就可以对这个集群做一些简单的操作来验证它是否安装正确，比如创建一个测试主题，发布一些消息，然后读取它们。

创建并验证主题：

```
# /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
#
```

往测试主题上发布消息：

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

从测试主题上读取消息：

```
# /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
#
```

2.3 broker配置

Kafka 发行包里自带的配置样本可以用来安装单机服务，但并不能满足大多数安装场景的要求。Kafka 有很多配置选项，涉及安装和调优的方方面面。不过大多数调优选项可以使用默认配置，除非你对调优有特别的要求。

2.3.1 常规配置

有一些配置选项，在单机安装时可以直接使用默认值，但在部署到其他环境时要格外小心。这些参数是单个服务器最基本的配置，它们中的大部分需要经过修改后才能用在集群里。

1. broker.id

每个 broker 都需要有一个标识符，使用 `broker.id` 来表示。它的默认值是 0，也可以被设置成其他任意整数。这个值在整个 Kafka 集群里必须是唯一的。这个值可以任意选定，如果出于维护的需要，可以在服务器节点间交换使用这些 ID。建议把它们设置成与机器名具有相关性的整数，这样在进行维护时，将 ID 号映射到机器名就没那么麻烦了。例如，如果机器名包含唯一性的数字（比如 `host1.example.com`、`host2.example.com`），那么用这些数字来设置 `broker.id` 就再好不过了。

2. port

如果使用配置样本来启动 Kafka，它会监听 9092 端口。修改 `port` 配置参数可以把它设置成其他任意可用的端口。要注意，如果使用 1024 以下的端口，需要使用 root 权限启动 Kafka，不过不建议这么做。

3. zookeeper.connect

用于保存 broker 元数据的 Zookeeper 地址是通过 `zookeeper.connect` 来指定的。`localhost:2181` 表示这个 Zookeeper 是运行在本地的 2181 端口上。该配置参数是用冒号分隔的一组 `hostname:port/path` 列表，每一部分的含义如下：

- `hostname` 是 Zookeeper 服务器的机器名或 IP 地址；
- `port` 是 Zookeeper 的客户端连接端口；
- `/path` 是可选的 Zookeeper 路径，作为 Kafka 集群的 chroot 环境。如果不指定，默认使用根路径。

如果指定的 chroot 路径不存在，broker 会在启动的时候创建它。



为什么使用 chroot 路径

在 Kafka 集群里使用 chroot 路径是一种最佳实践。Zookeeper 群组可以共享给其他应用程序，即使还有其他 Kafka 集群存在，也不会产生冲突。最好是在配置文件里指定一组 Zookeeper 服务器，用分号把它们隔开。一旦有一个 Zookeeper 服务器宕机，broker 可以连接到 Zookeeper 群组的另一个节点上。

4. log.dirs

Kafka 把所有消息都保存在磁盘上，存放这些日志片段的目录是通过 log.dirs 指定的。它是一组用逗号分隔的本地文件系统路径。如果指定了多个路径，那么 broker 会根据“最少使用”原则，把同一个分区的日志片段保存到同一个路径下。要注意，broker 会往拥有最少数目分区的路径新增分区，而不是往拥有最小磁盘空间的路径新增分区。

5. num.recovery.threads.per.data.dir

对于如下 3 种情况，Kafka 会使用可配置的线程池来处理日志片段：

- 服务器正常启动，用于打开每个分区的日志片段；
- 服务器崩溃后重启，用于检查和截短每个分区的日志片段；
- 服务器正常关闭，用于关闭日志片段。

默认情况下，每个日志目录只使用一个线程。因为这些线程只是在服务器启动和关闭时会用到，所以完全可以设置大量的线程来达到并行操作的目的。特别是对于包含大量分区的服务器来说，一旦发生崩溃，在进行恢复时使用并行操作可能会省下数小时的时间。设置此参数时需要注意，所配置的数字对应的是 log.dirs 指定的单个日志目录。也就是说，如果 num.recovery.threads.per.data.dir 被设为 8，并且 log.dir 指定了 3 个路径，那么总共需要 24 个线程。

6. auto.create.topics.enable

默认情况下，Kafka 会在如下几种情形下自动创建主题：

- 当一个生产者开始往主题写入消息时；
- 当一个消费者开始从主题读取消息时；
- 当任意一个客户端向主题发送元数据请求时。

很多时候，这些行为都是非预期的。而且，根据 Kafka 协议，如果一个主题不先被创建，根本无法知道它是否已经存在。如果显式地创建主题，不管是手动创建还是通过其他配置系统来创建，都可以把 auto.create.topics.enable 设为 false。

2.3.2 主题的默认配置

Kafka 为新创建的主题提供了很多默认配置参数。可以通过管理工具（将在第 9 章介绍）为每个主题单独配置一部分参数，比如分区个数和数据保留策略。服务器提供的默认配置可以作为基准，它们适用于大部分主题。



使用主题配置覆盖 (override)

之前的 Kafka 版本允许主题覆盖服务器的默认配置, 包括 `log.retention.hours.per.topic`、`log.retention.bytes.per.topic` 和 `log.segment.bytes.per.topic` 这几个参数。新版本不再支持这些参数, 而且如果要对参数进行覆盖, 需要使用管理工具。

1. num.partitions

`num.partitions` 参数指定了新创建的主题将包含多少个分区。如果启用了主题自动创建功能 (该功能默认是启用的), 主题分区的个数就是该参数指定的值。该参数的默认值是 1。要注意, 我们可以增加主题分区的个数, 但不能减少分区的个数。所以, 如果要让一个主题的分区个数少于 `num.partitions` 指定的值, 需要手动创建该主题 (将在第 9 章讨论)。

第 1 章里已经提到, Kafka 集群通过分区对主题进行横向扩展, 所以当有新的 broker 加入集群时, 可以通过分区个数来实现集群的负载均衡。当然, 这并不是说, 在存在多个主题的情况下 (它们分布在多个 broker 上), 为了能让分区分布到所有 broker 上, 主题分区的个数必须要大于 broker 的个数。不过, 拥有大量消息的主题如果要进行负载分散, 就需要大量的分区。



如何选定分区数量

为主题选定分区数量并不是一件可有可无的事情, 在进行数量选择时, 需要考虑如下几个因素。

- 主题需要达到多大的吞吐量? 例如, 是希望每秒钟写入 100KB 还是 1GB?
- 从单个分区读取数据的最大吞吐量是多少? 每个分区一般都会有一个消费者, 如果你知道消费者将数据写入数据库的速度不会超过每秒 50MB, 那么你也该知道, 从一个分区读取数据的吞吐量不需要超过每秒 50MB。
- 可以通过类似的方法估算生产者向单个分区写入数据的吞吐量, 不过生产者的速度一般比消费者快得多, 所以最好为生产者多估算一些吞吐量。
- 每个 broker 包含的分区个数、可用的磁盘空间和网络带宽。
- 如果消息是按照不同的键来写入分区的, 那么为已有的主题新增分区就会很困难。
- 单个 broker 对分区个数是有限制的, 因为分区越多, 占用的内存越多, 完成首选举需要的时间也越长。

很显然, 综合考虑以上几个因素, 你需要很多分区, 但不能太多。如果你估算出主题的吞吐量和消费者吞吐量, 可以用主题吞吐量除以消费者吞吐量算出分区的个数。也就是说, 如果每秒钟要从主题上写入和读取 1GB 的数据, 并且每个消费者每秒钟可以处理 50MB 的数据, 那么至少需要 20 个分区。这样就可以让 20 个消费者同时读取这些分区, 从而达到每秒钟 1GB 的吞吐量。

如果不知道这些信息，那么根据经验，把分区的大小限制在 25GB 以内可以得到比较理想的效果。

2. log.retention.ms

Kafka 通常根据时间来决定数据可以被保留多久。默认使用 `log.retention.hours` 参数来配置时间，默认值为 168 小时，也就是一周。除此以外，还有其他两个参数 `log.retention.minutes` 和 `log.retention.ms`。这 3 个参数的作用是一样的，都是决定消息多久以后会被删除，不过还是推荐使用 `log.retention.ms`。如果指定了不止一个参数，Kafka 会优先使用具有最小值的那个参数。



根据时间保留数据和最后修改时间

根据时间保留数据是通过检查磁盘上日志片段文件的最后修改时间来实现的。一般来说，最后修改时间指的就是日志片段的关闭时间，也就是文件里最后一个消息的时间戳。不过，如果使用管理工具在服务器间移动分区，最后修改时间就不准确了。时间误差可能导致这些分区过多地保留数据。在第 9 章讨论分区移动时会提到更多这方面的内容。

3. log.retention.bytes

另一种方式是通过保留的消息字节数来判断消息是否过期。它的值通过参数 `log.retention.bytes` 来指定，作用在每一个分区上。也就是说，如果有一个包含 8 个分区的主题，并且 `log.retention.bytes` 被设为 1GB，那么这个主题最多可以保留 8GB 的数据。所以，当主题的分区个数增加时，整个主题可以保留的数据也随之增加。



根据字节大小和时间保留数据

如果同时指定了 `log.retention.bytes` 和 `log.retention.ms`（或者另一个时间参数），只要任意一个条件得到满足，消息就会被删除。例如，假设 `log.retention.ms` 设置为 86 400 000（也就是 1 天），`log.retention.bytes` 设置为 1 000 000 000（也就是 1GB），如果消息字节总数在不到一天的时间就超过了 1GB，那么多出来的部分就会被删除。相反，如果消息字节总数小于 1GB，那么一天之后这些消息也会被删除，尽管分区的数据总量小于 1GB。

4. log.segment.bytes

以上的设置都作用在日志片段上，而不是作用在单个消息上。当消息到达 broker 时，它们被追加到分区的当前日志片段上。当日志片段大小达到 `log.segment.bytes` 指定的上限（默认是 1GB）时，当前日志片段就会被关闭，一个新的日志片段被打开。如果一个日志片段被关闭，就开始等待过期。这个参数的值越小，就会越频繁地关闭和分配新文件，从而降低磁盘写入的整体效率。

如果主题的消息量不大，那么如何调整这个参数的大小就变得尤为重要。例如，如果一个主题每天只接收 100MB 的消息，而 `log.segment.bytes` 使用默认设置，那么需要 10 天时

间才能填满一个日志片段。因为在日志片段被关闭之前消息是不会过期的，所以如果 `log.retention.ms` 被设为 604 800 000（也就是 1 周），那么日志片段最多需要 17 天才会过期。这是因为关闭日志片段需要 10 天的时间，而根据配置的过期时间，还需要再保留 7 天时间（要等到日志片段里的最后一个消息过期才能被删除）。



使用时间戳获取偏移量

日志片段的大小会影响使用时间戳获取偏移量。在使用时间戳获取日志偏移量时，Kafka 会检查分区里最后修改时间大于指定时间戳的日志片段（已经被关闭的），该日志片段的前一个文件的最后修改时间小于指定时间戳。然后，Kafka 返回该日志片段（也就是文件名）开头的偏移量。对于使用时间戳获取偏移量的操作来说，日志片段越小，结果越准确。

5. log.segment.ms

另一个可以控制日志片段关闭时间的参数是 `log.segment.ms`，它指定了多长时间之后日志片段会被关闭。就像 `log.retention.bytes` 和 `log.retention.ms` 这两个参数一样，`log.segment.bytes` 和 `log.retention.ms` 这两个参数之间也不存在互斥问题。日志片段会在大小或时间达到上限时被关闭，就看哪个条件先得到满足。默认情况下，`log.segment.ms` 没有设定值，所以只根据大小来关闭日志片段。



基于时间的日志片段对磁盘性能的影响

在使用基于时间的日志片段时，要着重考虑并行关闭多个日志片段对磁盘性能的影响。如果多个分区的日志片段永远不能达到大小的上限，就会发生这种情况，因为 broker 在启动之后就开始计算日志片段的过期时间，对于那些数据量小的分区来说，日志片段的关闭操作总是同时发生。

6. message.max.bytes

broker 通过设置 `message.max.bytes` 参数来限制单个消息的大小，默认值是 1 000 000，也就是 1MB。如果生产者尝试发送的消息超过这个大小，不仅消息不会被接收，还会收到 broker 返回的错误信息。跟其他与字节相关的配置参数一样，该参数指的是压缩后的消息大小，也就是说，只要压缩后的消息小于 `message.max.bytes` 指定的值，消息的实际大小可以远大于这个值。

这个值对性能有显著的影响。值越大，那么负责处理网络连接和请求的线程就需要花越多的时间来处理这些请求。它还会增加磁盘写入块的大小，从而影响 IO 吞吐量。



在服务端和客户端之间协调消息大小的配置

消费者客户端设置的 `fetch.message.max.bytes` 必须与服务器端设置的消息大小进行协调。如果这个值比 `message.max.bytes` 小，那么消费者就无法读取比较大的消息，导致出现消费者被阻塞的情况。在为集群里的 broker 配置 `replica.fetch.max.bytes` 参数时，也遵循同样的原则。

2.4 硬件的选择

为 Kafka 选择合适的硬件更像是一门艺术。Kafka 本身对硬件没有特别的要求，它可以运行在任何系统上。不过，如果比较关注性能，那么就需要考虑几个会影响整体性能的因素：磁盘吞吐量和容量、内存、网络和 CPU。在确定了性能关注点之后，就可以在预算范围内选择最优化的硬件配置。

2.4.1 磁盘吞吐量

生产者客户端的性能直接受到服务器端磁盘吞吐量的影响。生产者生成的消息必须被提交到服务器保存，大多数客户端在发送消息之后会一直等待，直到至少有一个服务器确认消息已经成功提交为止。也就是说，磁盘写入速度越快，生成消息的延迟就越低。

在考虑硬盘类型对磁盘吞吐量的影响时，是选择传统的机械硬盘（HDD）还是固态硬盘（SSD），我们可以很容易地作出决定。固态硬盘的查找和访问速度都很快，提供了最好的性能。机械硬盘更便宜，单块硬盘容量也更大。在同一个服务器上使用多个机械硬盘，可以设置多个数据目录，或者把它们设置成磁盘阵列，这样可以提升机械硬盘的性能。其他方面的因素，比如磁盘特定的技术（串行连接存储技术或 SATA），或者磁盘控制器的质量，都会影响吞吐量。

2.4.2 磁盘容量

磁盘容量是另一个值得讨论的话题。需要多大的磁盘容量取决于需要保留的消息数量。如果服务器每天会收到 1TB 消息，并且保留 7 天，那么就需要 7TB 的存储空间，而且还要为其他文件提供至少 10% 的额外空间。除此之外，还需要提供缓冲区，用于应付消息流量的增长和波动。

在决定扩展 Kafka 集群规模时，存储容量是一个需要考虑的因素。通过让主题拥有多个分区，集群的总流量可以被均衡到整个集群，而且如果单个 broker 无法支撑全部容量，可以让其他 broker 提供可用的容量。存储容量的选择同时受到集群复制策略的影响（将在第 6 章讨论更多的细节）。

2.4.3 内存

除了磁盘性能外，服务器端可用的内存容量是影响客户端性能的主要因素。磁盘性能影响生产者，而内存影响消费者。消费者一般从分区尾部读取消息，如果有生产者存在，就紧跟在生产者后面。在这种情况下，消费者读取的消息会直接存放在系统的页面缓存里，这比从磁盘上重新读取要快得多。

运行 Kafka 的 JVM 不需要太大的内存，剩余的系统内存可以用作页面缓存，或者用来缓存正在使用中的日志片段。这也就是为什么不建议把 Kafka 同其他重要的应用程序部署在一起的原因，它们需要共享页面缓存，最终会降低 Kafka 消费者的性能。

2.4.4 网络

网络吞吐量决定了 Kafka 能够处理的最大数据流量。它和磁盘存储是制约 Kafka 扩展规模的主要因素。Kafka 支持多个消费者，造成流入和流出的网络流量不平衡，从而让情况变得更加复杂。对于给定的主题，一个生产者可能每秒钟写入 1MB 数据，但可能同时有多个消费者瓜分网络流量。其他的操作，如集群复制（在第 6 章介绍）和镜像（在第 8 章介绍）也会占用网络流量。如果网络接口出现饱和，那么集群的复制出现延时就在所难免，从而让集群不堪一击。

2.4.5 CPU

与磁盘和内存相比，Kafka 对计算处理能力的要求相对较低，不过它在一定程度上还是会影响整体的性能。客户端为了优化网络和磁盘空间，会对消息进行压缩。服务器需要对消息进行批量解压，设置偏移量，然后重新进行批量压缩，再保存到磁盘上。这就是 Kafka 对计算处理能力有所要求的地方。不过不管怎样，这都不应该成为选择硬件的主要考虑因素。

2.5 云端的Kafka

Kafka 一般被安装在云端，比如亚马逊网络服务（Amazon Web Services, AWS）。AWS 提供了很多不同配置的实例，我们要根据 Kafka 的性能优先级来选择合适的实例。可以先从要保留数据的大小开始考虑，然后考虑生产者方面的性能。如果要求低延迟，那么就需要专门为 I/O 优化过的使用固态硬盘的实例，否则，使用配备了临时存储的实例就可以了。选好存储类型之后，再选择 CPU 和内存就容易得多。

实际上，如果使用 AWS，一般会选择 m4 实例或 r3 实例。m4 实例允许较长时间地保留数据，不过磁盘吞吐量会小一些，因为它使用的是弹性块存储。r3 实例使用固态硬盘，具有较高的吞吐量，但保留的数据量会有所限制。如果想两者兼顾，那么需要升级成 i2 实例或 d2 实例，不过它们的成本要高得多。

2.6 Kafka集群

单个 Kafka 服务器足以满足本地开发或 POC 要求，不过集群也有它的强大之处。使用集群最大的好处是可以跨服务器进行负载均衡，再则就是可以使用复制功能来避免因单点故障造成的数据丢失。在维护 Kafka 或底层系统时，使用集群可以确保为客户端提供高可用性。本节只是介绍如何配置 Kafka 集群，第 6 章将介绍更多关于数据复制的内容。

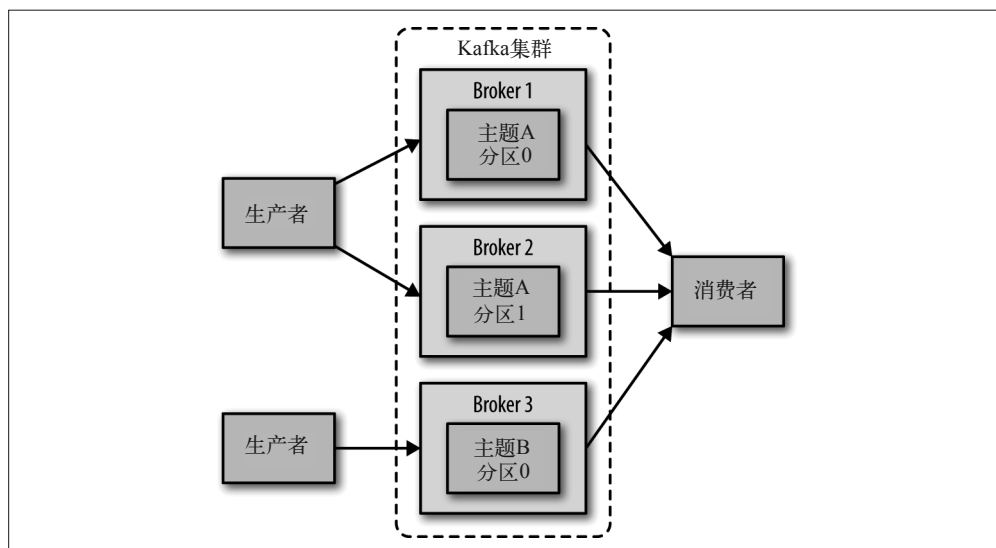


图 2-2：一个简单的 Kafka 集群

2.6.1 需要多少个broker

一个 Kafka 集群需要多少个 broker 取决于以下几个因素。首先，需要多少磁盘空间来保留数据，以及单个 broker 有多少空间可用。如果整个集群需要保留 10TB 的数据，每个 broker 可以存储 2TB，那么至少需要 5 个 broker。如果启用了数据复制，那么至少还需要一倍的空间，不过这要取决于配置的复制系数是多少（将在第 6 章介绍）。也就是说，如果启用了数据复制，那么这个集群至少需要 10 个 broker。

第二个要考虑的因素是集群处理请求的能力。这通常与网络接口处理客户端流量的能力有关，特别是当有多个消费者存在或者在数据保留期间流量发生波动（比如高峰时段的流量爆发）时。如果单个 broker 的网络接口在高峰时段可以达到 80% 的使用量，并且有两个消费者，那么消费者就无法保持峰值，除非有两个 broker。如果集群启用了复制功能，则要把这个额外的消费者考虑在内。因磁盘吞吐量低和系统内存不足造成的性能问题，也可以通过扩展多个 broker 来解决。

2.6.2 broker配置

要把一个 broker 加入到集群里，只需要修改两个配置参数。首先，所有 broker 都必须配置相同的 `zookeeper.connect`，该参数指定了用于保存元数据的 Zookeeper 群组 and 路径。其次，每个 broker 都必须为 `broker.id` 参数设置唯一的值。如果两个 broker 使用相同的 `broker.id`，那么第二个 broker 就无法启动。在运行集群时，还可以配置其他一些参数，特别是那些用于控制数据复制的参数，这些将在后续的章节介绍。

2.6.3 操作系统调优

大部分 Linux 发行版默认的内核调优参数配置已经能够满足大多数应用程序的运行需求，不过还是可以通过调整一些参数来进一步提升 Kafka 的性能。这些参数主要与虚拟内存、网络子系统和用来存储日志片段的磁盘挂载点有关。这些参数一般配置在 `/etc/sysctl.conf` 文件里，不过在对内核参数进行调整时，最好参考操作系统的文档。

1. 虚拟内存

一般来说，Linux 的虚拟内存会根据系统的工作负荷进行自动调整。我们可以对交换分区的处理方式和内存脏页进行调整，从而让 Kafka 更好地处理工作负载。

对于大多数依赖吞吐量的应用程序来说，要尽量避免内存交换。内存页和磁盘之间的交换对 Kafka 各方面的性能都有重大影响。Kafka 大量地使用系统页面缓存，如果虚拟内存被交换到磁盘，说明已经没有任何内存可以分配给页面缓存了。

一种避免内存交换的方法是不设置任何交换分区。内存交换不是必需的，不过它确实能够在系统发生灾难性错误时提供一些帮助。进行内存交换可以防止操作系统由于内存不足而突然终止进程。基于上述原因，建议把 `vm.swappiness` 参数的值设置得小一点，比如 1。该参数指明了虚拟机的子系统将如何使用交换分区，而不是只把内存页从页面缓存里移除。要优先考虑减小页面缓存，而不是进行内存交换。



为什么不把 `vm.swappiness` 设为零

先前，人们建议尽量把 `vm.swappiness` 设为 0，它意味着“除非发生内存溢出，否则不要进行内存交换”。直到 Linux 内核 3.5-rc1 版本发布，这个值的意义才发生了变化。这个变化被移植到其他的发行版上，包括 Red Hat 企业版内核 2.6.32-303。在发生变化之后，0 意味着“在任何情况下都不要发生交换”。所以现在建议把这个值设为 1。

脏页会被冲刷到磁盘上，调整内核对脏页的处理方式可以让我们从中获益。Kafka 依赖 I/O 性能为生产者提供快速的响应。这就是为什么日志片段一般要保存在快速磁盘上，不管是单个快速磁盘（如 SSD）还是具有 NVRAM 缓存的磁盘子系统（如 RAID）。这样一来，在后台刷新进程将脏页写入磁盘之前，可以减少脏页的数量，这个可以通过将 `vm.dirty_background_ratio` 设为小于 10 的值来实现。该值指的是系统内存的百分比，大部分情况下设为 5 就可以了。它不应该被设为 0，因为那样会促使内核频繁地刷新页面，从而降低内核为底层设备的磁盘写入提供缓冲的能力。

通过设置 `vm.dirty_ratio` 参数可以增加被内核进程刷新到磁盘之前的脏页数量，可以将它设为大于 20 的值（这也是系统内存的百分比）。这个值可设置的范围很广，60~80 是个比较合理的区间。不过调整这个参数会带来一些风险，包括未刷新磁盘操作的数量和同步刷新引起的长时间 I/O 等待。如果该参数设置了较高的值，建议启用 Kafka 的复制功能，避免因系统崩溃造成数据丢失。

为了给这些参数设置合适的值，最好是在 Kafka 集群运行期间检查脏页的数量，不管是在生存环境还是模拟环境。可以在 `/proc/vmstat` 文件里查看当前脏页数量。

```
# cat /proc/vmstat | egrep "dirty|writeback"
nr_dirty 3875
nr_writeback 29
nr_writeback_temp 0
#
```

2. 磁盘

除了选择合适的磁盘硬件设备和使用 RAID 外，文件系统是影响性能的另一个重要因素。有很多种文件系统可供选择，不过对于本地文件系统来说，EXT4（第四代可扩展文件系统）和 XFS 最为常见。近来，XFS 成为很多 Linux 发行版默认的文件系统，因为它只需要做少量调优就可以承担大部分的工作负荷，比 EXT4 具有更好的表现。EXT4 也可以做得很好，但需要做更多的调优，存在较大的风险。其中就包括设置更长的提交间隔（默认是 5），以便降低刷新的频率。EXT4 还引入了块分配延迟，一旦系统崩溃，更容易造成数据丢失和文件系统毁坏。XFS 也使用了分配延迟算法，不过比 EXT4 的要安全些。XFS 为 Kafka 提供了更好的性能，除了由文件系统提供的自动调优之外，无需额外的调优。批量磁盘写入具有更高的效率，可以提升整体的 I/O 吞吐量。

不管使用哪一种文件系统来存储日志片段，最好要对挂载点的 `noatime` 参数进行合理的设置。文件元数据包含 3 个时间戳：创建时间（`ctime`）、最后修改时间（`mtime`）以及最后访问时间（`atime`）。默认情况下，每次文件被读取后都会更新 `atime`，这会导致大量的磁盘写操作，而且 `atime` 属性的用处不大，除非某些应用程序想要知道某个文件在最近一次修改后有没有被访问过（这种情况可以使用 `realtime`）。Kafka 用不到该属性，所以完全可以把它禁用掉。为挂载点设置 `noatime` 参数可以防止更新 `atime`，但不会影响 `ctime` 和 `mtime`。

3. 网络

默认情况下，系统内核没有针对快速的大流量网络传输进行优化，所以对于应用程序来说，一般需要对 Linux 系统的网络栈进行调优，以实现支持大流量的支持。实际上，调整 Kafka 的网络配置与调整其他大部分 Web 服务器和网络应用程序的网络配置是一样的。首先可以对分配给 socket 读写缓冲区的内存大小作出调整，这样可以显著提升网络的传输性能。socket 读写缓冲区对应的参数分别是 `net.core.wmem_default` 和 `net.core.rmem_default`，合理的值是 131 072（也就是 128KB）。读写缓冲区最大值对应的参数分别是 `net.core.wmem_max` 和 `net.core.rmem_max`，合理的值是 2 097 152（也就是 2MB）。要注意，最大值并不意味着每个 socket 一定要有这么大的缓冲空间，只是说在必要的情况下才会达到这个值。

除了设置 socket 外，还需要设置 TCP socket 的读写缓冲区，它们的参数分别是 `net.ipv4.tcp_wmem` 和 `net.ipv4.tcp_rmem`。这些参数的值由 3 个整数组成，它们使用空格分隔，分别表示最小值、默认值和最大值。最大值不能大于 `net.core.wmem_max` 和 `net.core.rmem_max` 指定的大小。例如，“4096 65536 2048000”表示最小值是 4KB、默认值是 64KB、最大值是 2MB。根据 Kafka 服务器接收流量的实际情况，可能需要设置更高的最大值，为网络连接提供更大的缓冲空间。

还有其他一些有用的网络参数。例如，把 `net.ipv4.tcp_window_scaling` 设为 1，启用 TCP 时间窗扩展，可以提升客户端传输数据的效率，传输的数据可以在服务器端进行缓冲。把

`net.ipv4.tcp_max_syn_backlog` 设为比默认值 1024 更大的值，可以接受更多的并发连接。把 `net.core.netdev_max_backlog` 设为比默认值 1000 更大的值，有助于应对网络流量的爆发，特别是在使用千兆网络的情况下，允许更多的数据包排队等待内核处理。

2.7 生产环境的注意事项

当你准备把 Kafka 从测试环境部署到生产环境时，需要注意一些事项，以便创建更可靠的消息服务。

2.7.1 垃圾回收器选项

为应用程序调整 Java 垃圾回收参数就像是一门艺术，我们需要知道应用程序是如何使用内存的，还需要大量的观察和试错。幸运的是，Java 7 为我们带来了 G1 垃圾回收器，让这种状况有所改观。在应用程序的整个生命周期，G1 会自动根据工作负载情况进行自我调节，而且它的停顿时间是恒定的。它可以轻松地处理大块的堆内存，把堆内存分为若干小块的区域，每次停顿时并不会对整个堆空间进行回收。

正常情况下，G1 只需要很少的配置就能完成这些工作。以下是 G1 的两个调整参数。

MaxGCPauseMillis:

该参数指定每次垃圾回收默认的停顿时间。该值不是固定的，G1 可以根据需要使用更长的时间。它的默认值是 200ms。也就是说，G1 会决定垃圾回收的频率以及每一轮需要回收多少个区域，这样算下来，每一轮垃圾回收大概需要 200ms 的时间。

InitiatingHeapOccupancyPercent:

该参数指定了在 G1 启动新一轮垃圾回收之前可以使用的堆内存百分比，默认值是 45。也就是说，在堆内存的使用率达到 45% 之前，G1 不会启动垃圾回收。这个百分比包括新生代和老年代的内存。

Kafka 对堆内存的使用率非常高，容易产生垃圾对象，所以可以把这些值设得小一些。如果一台服务器有 64GB 内存，并且使用 5GB 堆内存来运行 Kafka，那么可以参考以下的配置：`MaxGCPauseMillis` 可以设为 20ms；`InitiatingHeapOccupancyPercent` 可以设为 35，这样可以让垃圾回收比默认的要早一些启动。

Kafka 的启动脚本并没有启用 G1 回收器，而是使用了 Parallel New 和 CMS（Concurrent Mark-Sweep，并发标记和清除）垃圾回收器。不过它可以通过环境变量来修改。本章前面的内容使用 `start` 命令来修改它：

```
# export JAVA_HOME=/usr/java/jdk1.8.0_51
# export KAFKA_JVM_PERFORMANCE_OPTS="-server -XX:+UseG1GC
-XX:MaxGCPauseMillis=20 -XX:InitiatingHeapOccupancyPercent=35
-XX:+DisableExplicitGC -Djava.awt.headless=true"
# /usr/local/kafka/bin/kafka-server-start.sh -daemon
/usr/local/kafka/config/server.properties
#
```

2.7.2 数据中心布局

在开发阶段，人们并不会太关心 Kafka 服务器在数据中心所处的物理位置，因为即使集群在短时间内出现局部或完全不可用，也不会造成太大影响。但是，在生产环境，服务不可用意味着金钱的损失，具体表现为无法为用户提供服务或者不知道用户正在做什么。这个时候，使用 Kafka 集群的复制功能就变得尤为重要（请参考第 6 章），而服务器在数据中心所处的物理位置也变得重要起来。如果在部署 Kafka 之前没有考虑好这个问题，那么在后续的维护过程中，移动服务器需要耗费更高的成本。

在为 broker 增加新的分区时，broker 并无法获知机架的信息。也就是说，两个 broker 有可能是在同一个机架上，或者在同一个可用区域里（如果运行在像 AWS 这样的云服务上），所以，在为分区添加副本的时候，这些副本很可能被分配给同一个机架上的 broker，它们使用相同的电源和网络连接。如果该机架出了问题，这些分区就会离线，客户端就无法访问到它们。更糟糕的是，如果发生不完整的主节点选举，那么在恢复时就有可能丢失数据（第 6 章将介绍更多细节）。

所以，最好把集群的 broker 安装在不同的机架上，至少不要让它们共享可能出现单点故障的基础设施，比如电源和网络。也就是说，部署服务器需要至少两个电源连接（两个不同的回路）和两个网络交换器（保证可以进行无缝的故障切换）。除了这些以外，最好还要把 broker 安放在不同的机架上。因为随着时间的推移，机架也需要进行维护，而这会导致机器离线（比如移动机器或者重新连接电源）。

2.7.3 共享 Zookeeper

Kafka 使用 Zookeeper 来保存 broker、主题和分区的元数据信息。对于一个包含多个节点的 Zookeeper 群组来说，Kafka 集群的这些流量并不算多，那些写操作只是用于构造消费者群组或集群本身。实际上，在很多部署环境里，会让多个 Kafka 集群共享一个 Zookeeper 群组（每个集群使用一个 chroot 路径）。



Kafka 消费者和 Zookeeper

在 Kafka 0.9.0.0 版本之前，除了 broker 之外，消费者也会使用 Zookeeper 来保存一些信息，比如消费者群组的信息、主题信息、消费分区的偏移量（在消费者群组里发生失效转移时会用到）。到了 0.9.0.0 版本，Kafka 引入了一个新的消费者接口，允许 broker 直接维护这些信息。这个新的消费者接口将在第 4 章介绍。

不过，消费者和 Zookeeper 之间还是有一个值得注意的地方，消费者可以选择将偏移量提交到 Zookeeper 或 Kafka，还可以选择提交偏移量的时间间隔。如果消费者将偏移量提交到 Zookeeper，那么在每个提交时间点上，消费者将会为每一个消费的分区分往 Zookeeper 写入一次偏移量。合理的提交间隔是 1 分钟，因为这刚好是消费者群组的某个消费者发生失效时能够读取到重复消息的时间。值得注意的是，这些提交对于 Zookeeper 来说流量不算小，特别是当集群里有多个消费者的时候。如果 Zookeeper 群组无法处理太大的流量，就

有必要使用长一点的提交时间间隔。不过不管怎样，还是建议使用最新版本的 Kafka，让消费者把偏移量提交到 Kafka 服务器上，消除对 Zookeeper 的依赖。

虽然多个 Kafka 集群可以共享一个 Zookeeper 群组，但如果有可能的话，不建议把 Zookeeper 共享给其他应用程序。Kafka 对 Zookeeper 的延迟和超时比较敏感，与 Zookeeper 群组之间的一个通信异常就可能导致 Kafka 服务器出现无法预测的行为。这样很容易让多个 broker 同时离线，如果它们与 Zookeeper 之间断开连接，也会导致分区离线。这也会给集群控制器带来压力，在服务器离线一段时间之后，当控制器尝试关闭一个服务器时，会表现出一些细小的错误。其他的应用程序因重度使用或进行不恰当的操作给 Zookeeper 群组带来压力，所以最好让它们使用自己的 Zookeeper 群组。

2.8 总结

在这一章，我们学习了如何运行 Kafka，同时也讨论了如何为 Kafka 选择合适的硬件，以及在生产环境中使用 Kafka 需要注意的事项。有了 Kafka 集群之后，接下来要介绍基本的客户端应用程序。后面两章将介绍如何创建客户端，并用它们向 Kafka 生产消息（第 3 章）以及从 Kafka 读取这些消息（第 4 章）。

Kafka生产者——向Kafka写入数据

不管是把 Kafka 作为消息队列、消息总线还是数据存储平台来使用，总是需要有一个可以往 Kafka 写入数据的生产者和一个可以从 Kafka 读取数据的消费者，或者一个兼具两种角色的应用程序。

例如，在一个信用卡事务处理系统里，有一个客户端应用程序，它可能是一个在线商店，每当有支付行为发生时，它负责把事务发送到 Kafka 上。另一个应用程序根据规则引擎检查这个事务，决定是批准还是拒绝。批准或拒绝的响应消息被写回 Kafka，然后发送给发起事务的在线商店。第三个应用程序从 Kafka 上读取事务和审核状态，把它们保存到数据库，随后分析师可以对这些结果进行分析，或许还能借此改进规则引擎。

开发者们可以使用 Kafka 内置的客户端 API 开发 Kafka 应用程序。

在这一章，我们将从 Kafka 生产者的设计和组件讲起，学习如何使用 Kafka 生产者。我们将演示如何创建 `KafkaProducer` 和 `ProducerRecords` 对象、如何将记录发送给 Kafka，以及如何处理从 Kafka 返回的错误，然后介绍用于控制生产者行为的重要配置选项，最后深入探讨如何使用不同的分区方法和序列化器，以及如何自定义序列化器和分区器。

在第 4 章，我们将会介绍 Kafka 的消费者客户端，以及如何从 Kafka 读取消息。



第三方客户端

除了内置的客户端外，Kafka 还提供了二进制连接协议，也就是说，我们直接向 Kafka 网络端口发送适当的字节序列，就可以实现从 Kafka 读取消息或往 Kafka 写入消息。还有很多用其他语言实现的 Kafka 客户端，比如 C++、Python、Go 语言等，它们都实现了 Kafka 的连接协议，使得 Kafka 不仅仅局限于在 Java 里使用。这些客户端不属于 Kafka 项目，不过 Kafka 项目 wiki 上提供了一个清单，列出了所有可用的客户端。连接协议和第三方客户端超出了本章的讨论范围。