O'REILLY®

TURING 图灵程序设计丛书

第2版

Kafka权威指南

Kafka: The Definitive Guide, Second Edition

Kafka核心作者Jay Kreps作序推荐

[美] 格温·沙皮拉
[美] 托德·帕利诺
[英] 拉吉尼·西瓦拉姆
[美] 克里特·佩蒂
著

薛命灯 译

中国工信出版集团

人民邮电出版社 POSTS & TELECOM PRESS

版权信息

书名：Kafka权威指南（第2版）

作者：[美] 格温 • 沙皮拉 托德 • 帕利诺 克里特 • 佩蒂 [英] 拉吉尼 • 西瓦拉姆

译者：薛命灯

ISBN：978-7-115-60142-1

版权声明

O'Reilly Media, Inc.介绍

O'Reilly 以“分享创新知识、改变世界”为己任。40 多年来我们一直向企业、个人提供成功所必需之技能及思想，激励他们创新并做得更好。

O'Reilly 业务的核心是独特的专家及创新者网络，众多专家及创新者通过我们分享知识。我们的在线学习（Online Learning）平台提供独家的直播培训、图书及视频，使客户更容易获取业务成功所需的专业知识。几十年来 O'Reilly 图书一直被视为学习开创未来之技术的权威资料。我们每年举办的诸多会议是活跃的技术人员聚会场所，来自各领域的专业人士在此建立联系，讨论最佳实践并发现可能影响技术行业未来的新趋势。

我们的客户渴望做出推动世界前进的创新之举，我们希望能助他们一臂之力。

业界评论

“O'Reilly Radar 博客有口皆碑。”

——*Wired*

“O'Reilly 凭借一系列非凡想法（真希望当初我也想到了）建立了数百万美元的业务。”

——*Business 2.0*

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O'Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的领域，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，那就走小路。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

本书赞誉

这本书为如何在云端或本地使用好 Kafka 提供了所需的一切。它是开发人员和运维人员必备的读物。格温、托德、拉吉尼和克里特将多年的智慧融汇到这本书中。如果你正在使用 Kafka，那么这本书正是你所需要的。

——**Chris Riccomini，软件工程师、初创公司导师、*TheMissingREADME* 合著者**

这是一部全面的 Kafka 原理和运维指南。

——**Sumant Tambe，LinkedIn 高级软件工程师**

这是 Kafka 开发人员或管理员必备的读物。你既可以从头到尾仔细阅读，也可以把它当作一部参考手册。不管怎样，它的写作清晰度和技术准确性都是一流的。

——**Robin Moffatt，Confluent 高级开发者布道师**

这是为所有对 Kafka 感兴趣的工程师准备的一部基础文献。在 Robinhood，它帮助我们伸缩、升级和调优 Kafka，为我们快速增长用户群提供支撑。

——**Jaren M. Glover，互联网券商 Robinhood 早期工程师、天使投资人**

这是所有从事与 Kafka 相关工作的人士——开发人员或系统管理员、初学者或专家、用户或贡献者——必读的一本书。

——**Matthias J. Sax，Confluent 软件工程师、Apache Kafka 项目管理委员会成员**

对在生产环境中使用 Kafka 的开发团队和分布式系统工程师来说，这是一部非常好的指南。它从入门级内容开始，介绍了 Kafka 的内部原理、如何正确使用 Kafka，同时也指出了 Kafka 的缺陷所在。对于每一个关键特性，作者都清楚地列出了那些只能从 Kafka 老手那里听到的注意要点，这些信息很难从其他地方获得。鉴于这本书一流的清晰度和深度，我甚至想把它推荐给那些没有使用过 Kafka 的工程师，他们可以学习 Kafka 的设计原则、决策，了解运维陷阱，以便在构建其他系统时做出更好的决策。

——**Dmitriy Ryaboy，Zymergen 软件工程副总裁**

第 2 版序

本书第 1 版（英文版）出版于 5 年前（2017 年）。当时，估计有超过 30% 的《财富》世界 500 强公司使用了 Kafka。现在，超过 70% 的《财富》世界 500 强公司正在使用 Kafka。Kafka 仍然是世界上最流行的开源项目，在软件生态系统中备受瞩目。

Kafka 为什么这么流行？我认为主要是因为我们的数据基础设施之间存在巨大差异。传统的数据管理侧重的都是如何存储数据——文件存储或数据库保证了数据安全性，我们可按需查找到想要的数据。人们在这些系统上已经投入大量的精力和金钱。但是，现代化公司不只是拥有一个带有单个数据库的软件系统那么简单，它们的系统可以复杂到令人难以置信的程度，可以由数百甚至数千个自研应用程序、微服务、数据库、SaaS 和分析平台组成。我们所面临的问题逐渐变成了如何将这些“碎片”连接起来，实现实时的协同工作。

这个问题不是关于如何管理静态数据，而是关于如何管理动态数据。Kafka 就是这一运动浪潮的核心，它已经成为所有动态数据平台事实上的基础。

在这段旅程中，Kafka 并没有停滞不前。从最初只是简单地提交日志，到后来加入了连接器和流式处理能力，Kafka 一直在改进架构。为了提升 Kafka 的可用性和稳定性，Kafka 社区不仅在不断地改进已有的 API、配置参数、指标和工具，还加入了新的编程式管理 API、下一代全局复制和数据冗余解决方案 MirrorMaker 2.0、一个新的基于 Raft 的共识协议（通过单个可执行文件就可以运行 Kafka）和分层存储弹性。更重要的是，Kafka 加入了高级安全特性——身份验证、授权和加密，这让企业在关键应用场景中使用 Kafka 变得更加简单。

我们发现，随着 Kafka 的演进，它的应用场景也在发生变化。在本书第 1 版出版时，大部分 Kafka 系统被部署在传统的本地数据中心，使用的是传统的部署脚本。最常见的应用场景是 ETL 和消息传递，那时流式处理才刚刚起步。5 年之后，大部分 Kafka 系统运行在云端，其中有很多被部署在 Kubernetes 集群里。ETL 和消息传递仍然是常见的应用场景，不同的是，现在加入了基于事件驱动的微服务、实时流式处理、物联网、机器学习管道以及数以百计的行业特定应用场景和模式，比如保险公司理赔处理、银行交易系统、实时视频游戏和流媒体服务个性化定制。

虽然 Kafka 被部署到了新的环境中，也有了更多的应用场景，但要使用和部署好 Kafka，仍然需要遵循 Kafka 独有的思想。本书涵盖了 Kafka 应用程序开发人员和站点可靠性工程师需要知道的一切——从最基本的 API 和配置到最新的高级特性。它不仅告诉我们使用 Kafka 能做什么以及如何使用，还告诉我们不能使用 Kafka 做什么以及要避免哪些反模式。不管是新用户还是有经验的老手，这本书都是一部可信赖的指南。

——Jay Kreps，Kafka 核心作者、Confluent 联合创始人兼 CEO

第 1 版序

这是一个激动人心的时刻，成千上万家企业正在使用 Kafka，其中就包括超过三分之一的《财富》世界 500 强公司。Kafka 是成长最快的开源项目之一，它的生态系统也在蓬勃发展。Kafka 正在成为流式数据的管理和处理利器。

Kafka 从何而来？为什么要开发 Kafka？Kafka 到底是什么？

Kafka 最初是 LinkedIn 的一个内部基础设施系统。我们发现，虽然有很多数据库和系统可以用来**存储**数据，但在我们的架构里，刚好缺一个可以处理持续**数据流**的组件。在开发 Kafka 之前，我们实验了各种现成的解决方案，从消息系统到日志聚合系统，再到 ETL 工具，它们都无法满足需求。

最后，我们决定从头开发一个系统。我们不想只是开发能够存储数据的系统（比如传统的关系数据库、键–值存储引擎、搜索引擎或缓存系统），而是希望能够把数据看成持续变化和增长的流，并基于这样的想法构建出一个数据系统，或是一种数据架构。

这个想法的适用范围比我们最初预想的更广。Kafka 一开始被用在社交网络的实时应用程序和数据流中，现在已经成为下一代数据架构的基础。大型零售商正在基于持续数据流改造它们的基础业务流程，汽车公司正在收集和处理来自互联网汽车的实时数据流，银行也正在重新思考如何基于 Kafka 改造它们的基础流程和系统。

那么，Kafka 在这当中充当了怎样的角色？它与现有的系统又有哪些区别？

我们认为 Kafka 是一个**流式平台**：你可以在这个平台上发布和订阅数据流，并保存和处理它们，这就是构建 Kafka 的初衷。以这种方式看待数据可能与人们已有的习惯有所不同，但它确实在构建应用程序和架构方面表现出了强大的抽象能力。Kafka 经常被拿来与现有的技术［比如企业级消息系统、大数据系统（如 Hadoop）和数据集成或 ETL 工具］作比较。这里的每一项比较都有一定的道理，但也有失偏颇。

Kafka 有点儿像消息系统，我们可以用它来发布和订阅消息流。从这一点来看，它和 ActiveMQ、RabbitMQ 或 IBM 的 MQSeries 等产品有些相似。尽管看上去相似，但 Kafka 与这些传统消息系统之间有很多关键的不同。第一，作为现代分布式系统，Kafka 以集群的方式运行，可以自由伸缩，为公司所有的应用程序提供支撑。Kafka 集群并不是一组独立运行的 broker，而是一个可以灵活伸缩的中心平台，能够处理整个公司所有的数据流。第二，Kafka 可以按照要求存储数据，保存多久都可以。作为数据连接层，Kafka 提供了数据传递保证——可复制、持久化、保留多长时间完全由你决定。第三，流式处理将数据处理的层次提升到了新的高度。一般的消息系统只传递消息，而 Kafka 的流式处理能力让你只用很少的代码就可以动态地处理派生数据流和数据集。Kafka 的这些独到之处足以让你对其刮目相看，它不只是“另一个消息队列”。

从另一个角度来看，可以把 Kafka 看成实时版的 Hadoop，这也是我们设计和构建 Kafka 的原始动机之一。Hadoop 可以存储并定期处理大量的数据文件，Kafka 则可以存储并处理大型的持续数据流。从技术角度来看，它们有着惊人的相似之处。很多人将新兴的流式处理看成批处理的超集，但他们忽略了很重要的一点，即持续低延迟处理系统不断开拓的应用场景与批处理系统正在失守的应用场景非常不同。Hadoop 和大数据主要应用在数据分析上，而 Kafka 因其低延迟的特点更适合用在核心的业务应用上。业务事件时刻在发生，Kafka 能够及时对这些事件做出响应，基于 Kafka 构建的服务可以直接为业务运营提供支撑，并提升用户体验。

Kafka 与 ETL 工具或其他数据集成工具之间也可以做一番比较。Kafka 和这些工具都擅长移动数据，但它们最大的不同在于 Kafka 颠覆了传统的思维。Kafka 并非只是把数据从一个系统抽出来再塞进另一个系统那么简单，它其实是一个面向实时数据流的平台。也就是说，Kafka 不仅可以将现有的应用程序和数据系统连接起来，还能用于增强这些触发数据流的应用程序。我们认为这种以数据流为中心的架构是非常重要的。在某种程度上，这些数据流是现代数字科技公司的核心，其重要性可与它们的现金流相提并论。

兼具上述 3 个方面的能力，能够将所有的数据流整合到一起，流式处理平台因此变得极具吸引力。

当然，除了这些不同点，对那些习惯了开发请求与响应风格的应用程序和关系数据库的人来说，要学会基于持续数据流构建应用程序也着实需要一个巨大的思维转变。借助这本书来学习 Kafka 再好不过了，从内部架构到 API，所有内容都由对 Kafka 最了解的人亲手呈现。我希望你能够像我一样喜欢这本书。

——Jay Kreps，Kafka 核心作者、Confluent 联合创始人兼 CEO

前言

给予一位技术图书作者的最佳赞赏莫过于这句话——“如果在刚开始接触这门技术时有这本书就好了”。在开始撰写本书时，我们就是以这句话作为写作目标的。通过开发 Kafka，在生产环境中运行 Kafka，帮助很多公司构建基于 Kafka 的系统并帮助它们管理数据管道，我们积累了很多经验，但也感到困惑：“应该把哪些东西分享给 Kafka 新用户，让他们从新手变成专家？”本书就是我们日常工作最好的写照：运行 Kafka，并帮助其他人更好地使用 Kafka。

我们相信，书中提供的这些内容能够帮助 Kafka 用户在生产环境中更好地运行 Kafka，并基于 Kafka 构建健壮的高性能应用程序。我们列举了一些非常流行的应用场景：基于事件驱动的微服务系统的消息总线、流式处理应用程序以及大规模数据管道。本书通俗易懂，能够帮助每一位 Kafka 用户在任意的架构或应用场景中使用好 Kafka。书中介绍了如何安装和配置 Kafka、如何使用 Kafka API、Kafka 的设计原则和可靠性保证，以及 Kafka 的一些架构细节（比如复制协议、控制器和存储层）。我们相信，Kafka 的设计原理和内部架构不仅是分布式系统构建者的兴趣所在，对那些在生产环境中部署 Kafka 或使用 Kafka 构建应用程序的人来说也非常有用。越是了解 Kafka，你就越是能够更好地做出技术权衡。

在软件工程中，条条道路通罗马，每一个问题通常有多种解决方案。Kafka 为专家级用户提供了巨大的灵活性，而新手则需要克服陡峭的学习曲线才能成为专家。Kafka 通常会告诉你如何使用某个特性，但不会告诉你为什么要用它或为什么不该用它。我们会尽可能详细地解释我们的设计决策和权衡背后的缘由，以及用户在哪些情况下应该或不应该使用 Kafka 提供的特性。

读者对象

本书是为使用 Kafka API 开发应用程序的工程师和在生产环境中安装、配置、调优、监控 Kafka 的运维工程师（也叫站点可靠性工程师、运维人员或系统管理员）而写的。我们也考虑到了为整个公司设计和构建数据基础架构的数据架构师和数据工程师。某些章（特别是第 3 章、第 4 章和第 14 章）主要面向 Java 开发人员，并假设读者已经熟悉基本的 Java 编程，比如异常处理和并发编程。其他章（特别是第 2 章、第 10 章、第 12 章和第 13 章）假设读者在 Linux 的运行、存储和网络配置方面有一定的经验。剩余各章讨论了具有一般性的软件架构，不要求读者具备特定的知识。

还有一类可能对本书感兴趣的人是经理或架构师，他们不直接使用 Kafka，但会与使用 Kafka 的工程师打交道。他们也有必要了解 Kafka 所能提供的保证机制以及他们的同事在构建基于 Kafka 的系统时所做出的技术权衡。本书可以成为企业管理人员的利器，确保他们的工程师在 Kafka 方面训练有素，掌握工作当中需要用到的知识。

排版约定

本书使用下列排版约定。

黑体字

表示新术语或重点强调的内容。

等宽字体（`constant width`）

表示程序片段，以及正文中出现的变量名、函数名、数据库、数据类型、环境变量、语句和关键字等。

等宽粗体（**`constant width bold`**）

表示应该由用户输入的命令或其他文本。

等宽斜体（*`constant width italic`*）

表示应该由用户输入的值或根据上下文确定的值替换的文本。

该图标表示提示或建议。

该图标表示一般注记。

该图标表示警告或警示。

使用代码示例

如果在使用代码示例的过程中遇到任何技术上的问题或疑问，请发邮件至 errata@oreilly.com.cn。

本书旨在帮助你完成工作。一般来说，你可以在自己的程序或文档中使用本书提供的示例代码。除非需要复制大量代码，否则无须联系我们获得许可。比如，使用本书中的几个代码片段编写程序无须获得许可，销售或分发 O'Reilly 图书的示例光盘则需要获得许可；引用本书中的示例代码回答问题无须获得许可，将本书中的大量示例代码放到你的产品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和 ISBN。比如“*Kafka:TheDefinitiveGuide,SecondEdition* by Gwen Shapira, Todd Palino, Rajini Sivaram, and Krit Petty (O'Reilly). Copyright 2022 Chen Shapira, Todd Palino,

Rajini Sivaram, and Krit Petty, 978-1-492-04308-9”。

如果你对示例代码的用法超出了上述的许可范围，欢迎你通过 permissions@oreilly.com 与我们联系。

O'Reilly 在线学习平台（O'Reilly Online Learning）

40 多年来，O'Reilly Media 致力于提供技术和商业培训、知识和卓越见解，来帮助众多公司取得成功。

我们拥有独特的由专家和创新者组成的庞大网络，他们通过图书、文章和我们的在线学习平台分享知识和经验。O'Reilly 的在线学习平台让你能够按需访问现场培训课程、深入的学习路径、交互式编程环境，以及 O'Reilly 和 200 多家其他出版商提供的大量文本资源和视频资源。有关的更多信息，请访问 https://www.oreilly.com。

联系我们

如有与本书有关的评价或问题，请联系出版社。美国：

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）

奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表[1]、示例代码以及其他信息。本书的网页是 https://oreil.ly/kafka-tdg2。

[1]也可以通过图灵社区本书主页提交中文版勘误。——编者注

要了解更多 O'Reilly 图书、培训课程和新闻的信息，请访问以下网站：https://www.oreilly.com。

我们在 Facebook 的地址如下：http://facebook.com/oreilly。

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia。

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia。

致谢

感谢众多为 Kafka 和它的生态系统做出贡献的人。如果没有他们艰辛的工作，就不会有本书的问世。特别感谢 Jay Kreps、Neha Narkhede 和 Jun Rao，以及他们在 LinkedIn 的同事和领导，他们创造了 Kafka，并把它捐献给了 Apache 软件基金会。

很多人对本书第 1 版提供了有价值的反馈，我们非常感激他们为此付出的时间，也很钦佩他们的专业能力，这些人包括 Apurva Mehta、Arseniy Tashoyan、Dylan Scott、Ewen Cheslack-Postava、Grant Henke、Ismael Juma、James Cheng、Jason Gustafson、Jeff Holoman、Joel Koshy、Jonathan Seidman、Jun Rao、Matthias Sax、Michael Noll、Paolo Castagna 和 Jesse Anderson。我们还想感谢众多在网站上留下评论和反馈的读者。

很多审稿人提供了有价值的意见，极大地提升了本书的质量。书中的遗留错误理应由我们负责。

感谢本书第 1 版的编辑 Shannon Cutt，感谢她的鼓励、耐心和深谋远虑。同时也要感谢第 2 版的编辑 Jess Haberman 和 Gary O'Brien，感谢他们一路和我们一起应对各种挑战。对一位作者来说，与 O'Reilly 合作是一段非凡的经历——他们所提供的支持，从工具到签名售书，都是无可匹敌的。感谢每一位参与了本书相关工作的人，很感激他们愿意与我们一起工作。

另外，还要感谢我们的领导和同事，感谢他们在我们写作本书的过程中给予的帮助和鼓励。

格温要感谢她的丈夫 Omer Shapira，在她写书的几个月时间里，他一直给予她支持和耐心陪伴。她也要感谢她的父亲 Lior Shapira，是他让她学会了如何在困难面前不轻言放弃，尽管这种生活哲学总是让她麻烦不断。她还要感谢她的两只可爱的小猫咪 Luke 和 Lea。

托德要感谢他的妻子 Marcy 以及两个女儿 Bella 和 Kaylee，她们一直在背后默默地支持他。有了她们的支持，他才有更多的时间进行写作，厘清思路，并坚持到最后。

拉吉尼要感谢她的丈夫 Manjunath 和儿子 Tarun，感谢他们坚定的支持和鼓励，以及愿意花周末时间阅读她早期的手稿，也感谢他们一直陪伴在她身边。

克里特要向妻子 Cecilia 以及两个孩子 Lucas 和 Lizabeth 表达他的爱和感激。他们的爱和支持让他每一天都充满了快乐，没有他们，他将无法追求自己的梦想。他还想感谢他的母亲 Cindy Petty，是她鼓励他要永远做最好的自己。

更多信息

扫描下方二维码，即可获取电子书相关信息及读者群通道入口。

第 1 章　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，并生成更多的数据。每个应用程序都会生成数据，包括日志、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东西，比如下一步要做什么。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，然后再把得到的结果应用到实际场景中，这样才能确切地知道这些数据要告诉我们什么。如果每天在亚马逊网站上浏览感兴趣的商品，那么浏览信息就会被转化成商品推荐，展示给我们。

这个过程完成得越快，组织的反应就越敏捷。在数据移动上花费的精力越少，就越能专注于核心业务。这就是为什么在数据驱动型企业中，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。

> 科学家们每一次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，然后继续。
>
> ——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，先来了解什么是发布与订阅消息系统，以及为什么它是数据驱动型应用程序的关键组件。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）通过订阅它们来接收特定类型的消息。发布与订阅系统一般会有一个 broker，也就是发布消息的地方。

1.1.1 如何开始

发布与订阅消息系统的大部分应用场景是从一个简单的消息队列或进程间通道开始的。假设你的应用程序需要往其他地方发送监控信息，那么你就可以直接在这个应用程序和另一个可以在仪表盘上显示指标的应用程序之间建立连接，然后通过该连接推送指标，如图 1-1 所示。

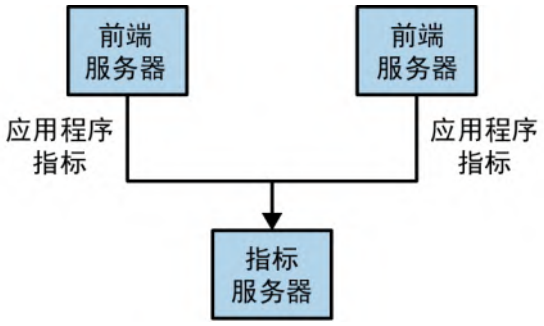

图 1-1：单个直连的指标发布者

这是监控系统在刚开始时针对简单问题的解决方案。不久之后，你需要分析更长时间片段的指标，此时的仪表盘应用程序满足不了需求，于是，你启动了一个新的服务来接收指标。这个服务会把指标保存起来，用于后续的分析。与此同时，你修改了原来的应用程序，把指标同时发送到这两个仪表盘系统中。现在，你又多了 3 个可以生成指标的应用程序，它们都与这两个服务连接。你的同事建议你对这些服务进行轮询，这样就可以实现告警功能了。于是，你为每一个应用程序增加了用于发送指标的服务器。一段时间过后，更多的应用程序出于各自的目的，都从这些服务器获取指标。这时候的架构看起来如图 1-2 所示，节点之间的连接乱糟糟的，难以追踪。

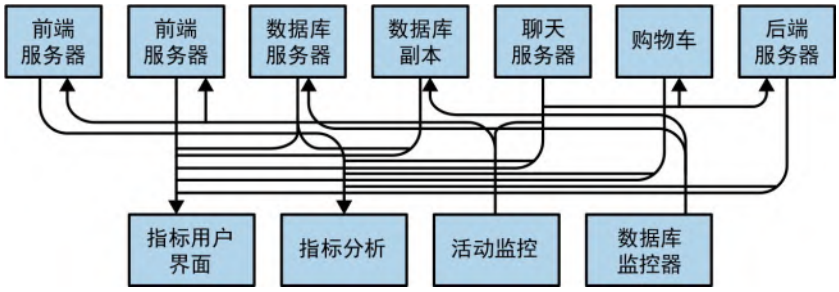

图 1-2：多个直连的指标发布者

很显然，现在出现了技术债务，你决定先解决一些。于是，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，之前架构的复杂性就降低到了图 1-3 所示的样子。恭喜你，你已经创建了一个基于发布与订阅的消息系统。

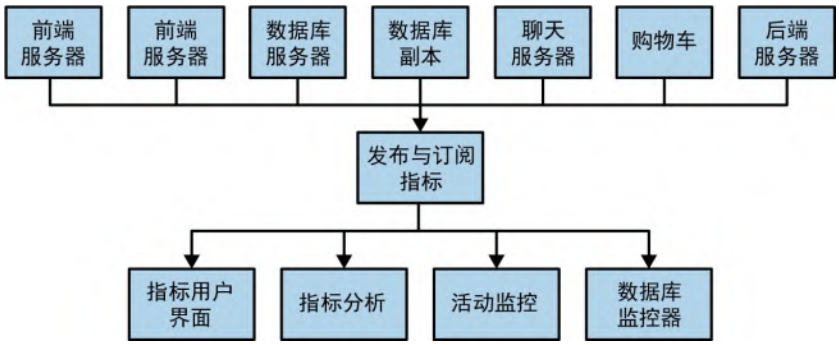

图 1-3：指标发布与订阅系统

1.1.2 独立的队列系统

在你跟指标“打得不可开交”的时候，你的一位同事正在为日志消息忙得焦头烂额。还有一位同事正在跟踪网站用户的行为，为负责开发机器学习模型的同事提供用户活动数据，并为管理团队生成报告。你和同事们使用了同样的方式创建这些系统，将信息的发布者和订阅者解耦。图 1-4 所示的架构包含了 3 个独立的发布与订阅系统。

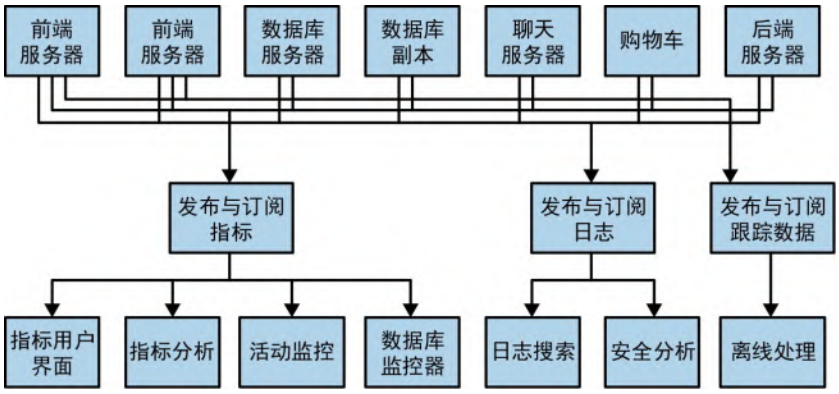

图 1-4：多个发布与订阅系统

这种方式比直接使用点对点连接（参见图 1-2）要好得多，但是它有太多重复的地方。你的公司因此要为数据队列维护多个系统，而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的数据，并且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模式的消息系统。它一般被称为“分布式提交日志”或“分布式流式平台”。文件系统或数据库提交日志旨在保存事务的持久化记录，通过重放这些日志可以重建系统状态。同样，Kafka 的数据是按照一定的顺序持久化保存的，并且可以按需读取。此外，Kafka 的数据分布在整个系统中，具备数据故障恢复能力和性能伸缩能力。

1.2.1 消息和批次

Kafka 的数据单元被称为**消息**。如果在使用 Kafka 之前你已经有使用数据库的经验，那么可以把消息看成数据库中的一个“数据行”或一条“记录”。消息由字节数组组成，对 Kafka 来说，消息里的数据没有特殊的格式或含义。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 Kafka 来说没有特殊含义。当需要以一种可控的方式将消息写入不同的分区时，需要用到键。最简单的例子就是为键生成一个一致性哈希值，然后用哈希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会被写到相同的分区中（前提是分区数量没有发生变化）。

为了提高效率，消息会被分成批次写入 Kafka。**批次**包含了一组属于同一个主题和分区的消息。如果每一条消息都单独穿行于网络中，那么就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，这需要在时间延迟和吞吐量之间做出权衡：批次越大，单位时间内处理的消息就越多，对单条消息来说，其传输时间就越长。消息批次会被压缩，这样可以提升数据的传输和存储性能，但需要做更多的计算处理。第 3 章将详细介绍消息的键和批次。

1.2.2 模式

对 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。不同的应用程序有不同的需求，因此消息**模式**（schema）也有很多可选项。一些简单的模式，比如 JavaScript Object Notation（JSON）和 Extensible Markup Language（XML），不仅易用，还具备良好的可读性。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。很多 Kafka 开发者喜欢使用 Apache Avro，其最初是为 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，无须重新生成代码。Avro 还支持强类型和模式演化，既向前兼容，也向后兼容。

数据格式的一致性对 Kafka 来说非常重要，它消除了消息读写操作之间的耦合性。如果读写操作紧密耦合在一起，那么消息订阅者就需要升级应用程序才能同时处理新旧两种数据格式，而消息发布者需要在消息订阅者升级之后跟着升级，然后使用新的数据格式发布消息。将定义好的模式保存在公共库中，更方便我们了解 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文件系统的文件夹。主题可以被分为若干个**分区**，一个分区就是一个提交日志。消息会以追加的方式被写入分区，然后按照先入先出的顺序读取。需要注意的是，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是有序的。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的冗余和伸缩。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台服务器，以此来提供比单台服务器更强大的性能。此外，分区可以被复制，相同分区的多个副本可以保存在多台服务器上，以防其中一台服务器发生故障。

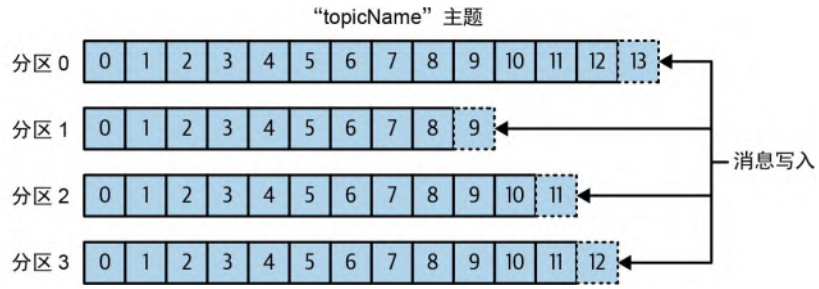

图 1-5：包含多个分区的主题

我们通常会使用流这个词来描述 Kafka 这类系统中的数据。很多时候，人们会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，这就是所谓的流式处理。流式处理有别于离线处理框架（如 Hadoop）处理数据的方式，后者被用于在未来某个时刻处理大量的数据。第 14 章将介绍流式处理。

1.2.4 生产者和消费者

Kafka 的客户端就是 Kafka 系统的用户，其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，提供了更高级的功能。

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**。一条消息会被发布到一个特定的主题上。在默认情况下，生产者会把消息均衡地分布到主题的所有分区中。不过，在某些情况下，生产者会把消息直接写入指定的分区，这通常是通过消息键和分区器来实现的。分区器会为键生成一个哈希值，并将其映射到指定的分区，这样可以保证包含同一个键的消息被写入同一个分区。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不同的分区。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**读取者**。消费者会订阅一个或多个主题，并按照消息写入分区的顺序读取它们。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**（不断递增的整数值）是另一种元数据，在创建消息时，Kafka 会把它添加到消息里。在给定的分区中，每一条消息的偏移量都是唯一的，越往后消息的偏移量越大（但不一定是严格单调递增）。消费者会把每一个分区可能的下一个偏移量保存起来（通常保存在 Kafka 中），如果消费者关闭或重启，则其读取状态不会丢失。

消费者可以是**消费者群组**的一部分，属于同一群组的一个或多个消费者共同读取一个主题。群组可以保证每个分区只被这个群组里的一个消费者读取。在图 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 3 个分区中的 1 个分区，另外一个消费者读取其他 2 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**。

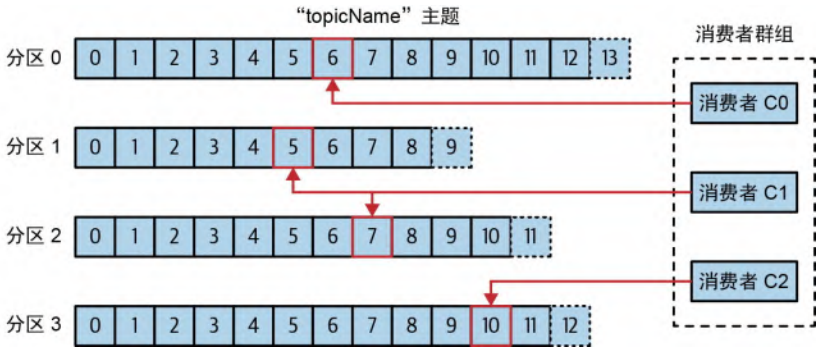

图 1-6：消费者群组从主题读取消息

通过这种方式，消费者可以读取包含大量消息的主题。而且，如果一个消费者失效，那么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。

1.2.5 broker 和集群

一台单独的 Kafka 服务器被称为 broker。broker 会接收来自生产者的消息，为其设置偏移量，并提交到磁盘保存。broker 会为消费者提供服务，对读取分区的请求做出响应，并返回已经发布的消息。根据硬件配置及其性能特征的不同，单个 broker 可以轻松处理数千个分区和每秒百万级的消息量。

broker 组成了**集群**。每个集群都有一个同时充当了**集群控制器**角色的 broker（自动从活动的集群成员中选举出来）。控制器负责管理工作，包括为 broker 分配分区和监控 broker。在集群中，一个分区从属于一个 broker，这个 broker 被称为分区的**首领**。一个被分配给其他 broker 的分区副本（参见图 1-7）叫作这个分区的“跟随者”。分区复制提供了分区的消息冗余，如果一个 broker 发生故障，则其中的一个跟随者可以接管它的领导权。所有想要发布消息的生产者必须连接到首领，但消费者可以从首领或者跟随者那里读取消息。第 7 章将详细介绍如何操作集群（包括复制分区）。

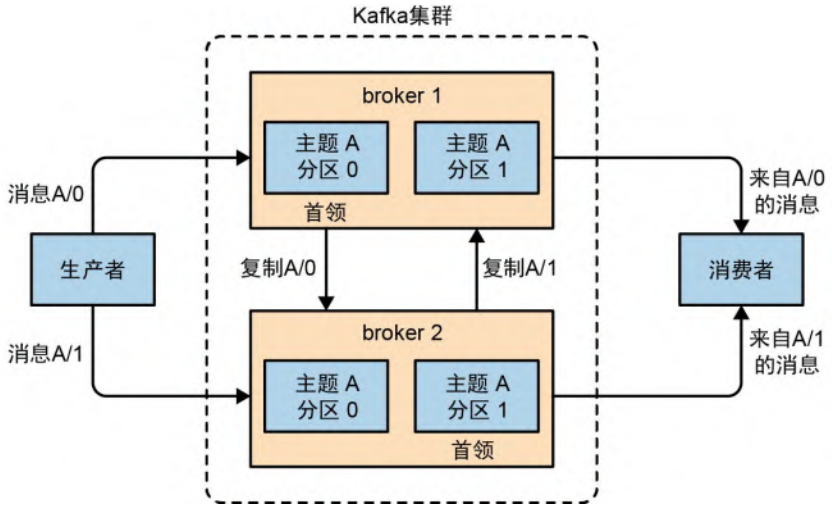

图 1-7：集群中的分区复制

保留消息（在一定期限内）是 Kafka 的一个重要特性。broker 默认的消息保留策略是这样的：要么保留一段时间（如 7 天），要么保留消息总量达到一定的字节数（如 1 GB）。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，在任意时刻，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，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，而应用程序的指标可能只需要保留几小时。我们可以把主题配置成**紧凑型日志**，只有最后一条带有特定键的消息会被保留下来。这比较适用于变更日志类型的数据，因为人们只关心最近一次发生的更新事件。

1.2.6 多集群

随着 broker 数量的增加，最好使用多个集群，原因如下。

- 数据类型分离
- 安全需求隔离
- 多数据中心（灾难恢复）

如果有多个数据中心，则需要在它们之间复制消息，让在线应用程序能够访问到多个站点的用户活动信息。如果一个用户修改了他们的资料，那么不管从哪个数据中心都应该能看到这些更新。或者，可以将多个站点的监控数据聚合到一个部署了分析应用程序和告警系统的中心位置。不过，需要注意的是，Kafka 的消息复制机制只能在单个集群中而不能在多个集群之间进行。

Kafka 提供了一个叫作 **MirrorMaker** 的工具，我们可以用它将数据复制到其他集群中。MirrorMaker 的核心组件包括一个消费者和一个生产者，它们之间通过队列相连。消费者会从一个集群读取消息，生产者则会把消息发送到另一个集群中。图 1-8 是使用 MirrorMaker 的例子，两个“本地”集群的数据被集合到一个“聚合”集群中，然后聚合集群的数据再被复制到其他数据中心。不过，这个应用程序太过简单了，还不足以展示 Kafka 在构建复杂数据管道方面的能力。第 9 章将详细讨论这些复杂的应用场景。

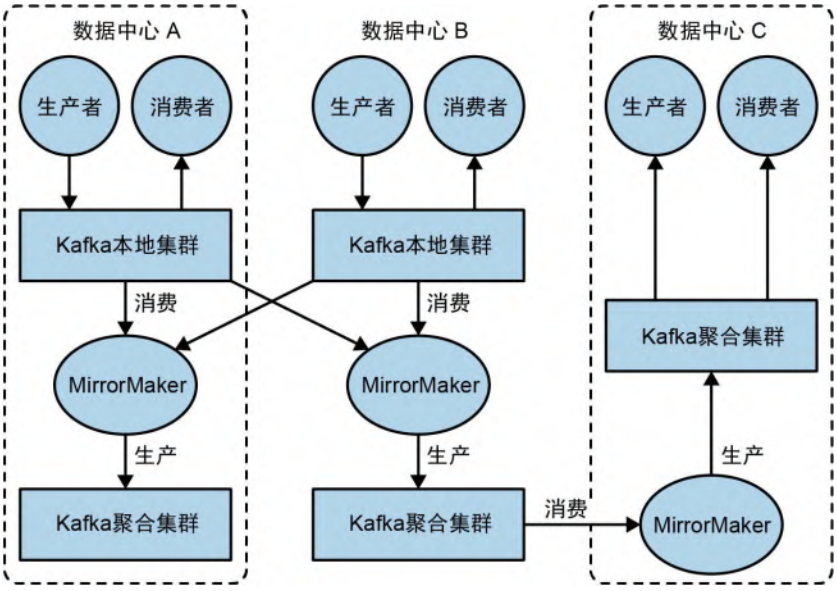

图 1-8：多数据中心架构

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 是更好的选择呢？

1.3.1 多个生产者

Kafka 可以无缝支持多个生产者，不管客户端消费的是单个主题还是多个主题。所以，它很适用于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，对于一个包含了多个微服务的网站，我们可以为页面视图创建一个主题，所有服务都以相同的消息格式向这个主题写入数据。消费者应用程序将获得统一的页面视图，不需要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，Kafka 也支持多个消费者从一个单独的消息流读取数据，而且消费者之间互不影响。这与其他队列系统不同。在其他队列系统中，消息一旦被一个客户端读取，就无法再被其他客户端读取。多个消费者还可以组成一个群组，它们共享一个消息流，并保证整个群组只处理一次给定的消息。

1.3.3 基于磁盘的数据保留

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。消息会被提交到磁盘，并根据设置的保留策略进行保存。每个主题可以设置单独的保留策略，以满足不同消费者的需求。各个主题还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而无法及时读取消息，在这种情况下，持久化的数据可以保证数据不会丢失。消费者可以在应用程序维护期间离线一小段时间，无须担心消息丢失或被堵塞在生产者端。消费者也可以被关闭，但消息会继续保留在 Kafka 中。消费者被重启之后，可以从上次中断的地方继续处理消息。

1.3.4 伸缩性

为了能够轻松地处理大量数据，Kafka 从一开始就被设计成一个具备灵活伸缩性的系统。用户可以在开发阶段使用单个 broker，然后再扩展到包含 3 个 broker 的小型开发集群。随着数据量不断增长，在部署到生产环境时，集群可以包含上百个 broker。对在线集群进行扩展丝毫不影响系统的整体可用性。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端提供服务。要提高集群的容错能力，需要配置较高的复制系数。第 7 章将介绍更多的有关复制的内容。

1.3.5 高性能

上面提到的所有特性让 Kafka 成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。

1.3.6 平台特性

Kafka 核心项目还加入了一些流式平台特性，从而使开发人员能够更容易执行一些常见的任务。虽然 Kafka 算不上是一个完整的平台（完整的平台通常包含像 YARN 这样的结构化运行时环境），但这些特性会以 API 和开发库的形式存在，为开发者构建其他功能提供坚实的基础和非常好的部署灵活性。我们可以用 Kafka Connect 从一个源数据抽取数据并将其推送到 Kafka，或者从 Kafka 抽取数据并将其推送到另一个接收数据的系统。Kafka Streams 提供了一个开发库，开发人员可以用它开发具备伸缩性和容错能力的流式处理应用程序。第 9 章将更详细地介绍 Apache Connect，而 Apache Streams 将在第 14 章进行介绍。

1.4 数据生态系统

现在已经有很多应用程序加入到我们为数据处理而构建的生态系统中。我们通过应用程序的形式定义了输入，这些应用程序会生成数据或者把数据引入系统中。我们也定义了输出，它们可以是指标、报告或者其他类型的数据产品。我们创建了一个闭环，用组件从系统中读取数据，再用来自其他数据源的数据对已读取的数据进行转换，然后再将其写回数据基础设施。数据类型可以多种多样，每一种可以有不同的内容、大小和用途。

Kafka 为数据生态系统带来了循环能力，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。如果系统提供了消息模式，那么生产者和消费者之间将不再紧密耦合，我们也不需要在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。

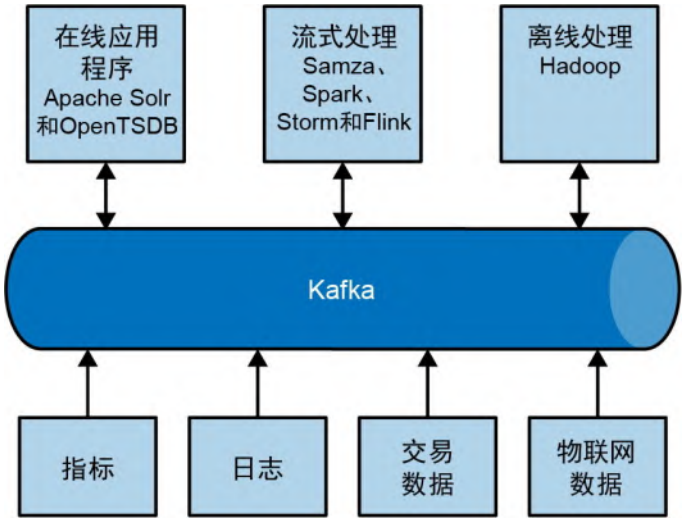

图 1-9：大型数据生态系统

应用场景

01. 活动跟踪

Kafka 最初的应用场景是跟踪网站用户的活动。网站用户与前端应用程序发生交互，前端应用程序再生成与用户活动相关的消息。这些消息既可以是一些静态信息，比如页面访问次数和点击量，也可以是一些复杂的操作，比如修改用户资料。这些消息会被发布到一个或多个主题上，并会被后端应用程序读取。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，或者实现更多其他的功能。

02. 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（如邮件）就是通过消息传递来实现的。这些应用程序组件可以生成消息，而无须关心消息的格式以及消息是如何被发送出去的。一个公共应用程序会负责读取并处理如下这些消息。

- 格式化消息（也就是所谓的**装饰**）。
- 将多条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消息。

使用公共组件的好处在于，无须在多个应用程序中开发重复的功能，并且可以在公共组件中做一些有趣的转换，比如把多条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。

03. 指标和日志记录

Kafka 也可以用于收集应用程序以及系统的指标和日志。Kafka 的多生产者特性在这个时候就派上用场了。应用程序定期把指标发布到 Kafka 主题上，监控系统或告警系统会读取这些消息。Kafka 也可以被用在离线处理系统（如 Hadoop）中，进行较长时间片段的数据分析，比如年度增长走势预测。我们也可以把日志消息发布到 Kafka 主题上，然后再路由给专门的日志搜索系统（如 Elasticsearch）或安全分析应用程序。更改目标系统（如日志存储系统）不会影响前端应用程序或聚合方法，这是 Kafka 的另一个优点。

04. 提交日志

Kafka 的基本概念源自提交日志，所以将 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，然后应用程序会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，或者将多个应用程序的更新合并到一个单独的数据库。持久化的数据为变更日志提供了缓冲，也就是说，如果消费者应用程序发生故障，则可以通过重放这些日志来恢复系统状态。另外，可以用紧凑型主题更长时间地保留数据，因为我们只为一个键保留了一条最新的变更数据。

05. 流式处理

流式处理是另一个包含多种类型应用程序的领域。虽然可以认为大部分 Kafka 应用程序是基于流式处理，但真正的流式处理通常是指提供了类似 map/reduce（Hadoop）处理功能的应用程序。Hadoop 通常依赖较长时间片段的数据聚合，可以是几小时或几天。流式处理采用实时的方式处理消息，速度几乎与生成消息一样快。开发人员可以通过用流式处理框架开发小型应用程序来处理 Kafka 消息，执行一些常见的任务，比如指标计数、对消息进行分区或使用多个数据源的数据来转换消息，等等。第 14 章将通过其他案例来介绍流式处理。

1.5 起源故事

Kafka 是为了解决 LinkedIn 数据管道问题应运而生的。它的设计目标是提供一个高性能的消息系统，该系统可以处理多种数据类型，并实时提供纯净、结构化的用户活动数据和系统指标。

> 数据为我们所做的每一件事提供了动力。
>
> ——Jeff Weiner，LinkedIn 前 CEO

1.5.1 LinkedIn 的问题

本章在开头提到过，LinkedIn 有一个用来收集应用程序指标的系统，该系统使用自研的收集器和开源工具来保存和展示数据。除了跟踪 CPU 使用率和应用程序性能这些一般性指标，还有一个比较复杂的用户请求跟踪功能，这个功能使用了监控系统，可以跟踪用户的请求是如何在内部应用程序之间流转的。不过，这个监控系统有很多不足，它使用的是轮询拉取指标的方式，指标之间的时间间隔较长，应用程序所有者无法管理属于自己的指标。另外，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，同一个指标在不同系统中的叫法都不一样。

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 XML 格式的消息。这些消息文件会被转移到线下批处理平台进行解析和校对。同样，这个系统也有很多不足。XML 文件格式无法保持一致，解析这种文件非常耗费计算资源。如果要修改已经创建好的活动类型，则需要在前端应用程序和离线处理程序之间做大量的协调工作。即使是这样，当数据结构发生变化时，系统仍然会崩溃。另外，批处理以小时为单位，无法实现实时处理。

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，而且监控使用的轮询拉取模式与活动跟踪使用的推送模式不兼容。另外，将跟踪服务提供的数据作为指标太过脆弱，并且批处理模型不适用于实时的监控和告警。不过，监控数据和活动跟踪数据之间存在很多共性，信息之间的关联度（比如特定类型用户活动对应用程序性能的影响）还是很高的。特定类型用户活动数量的下降说明相关的应用程序出现了问题，批处理模型数小时的延迟说明系统无法对这类问题做出及时的响应。

最开始，我们调研了一些现成的开源解决方案，希望找到一个能够实时访问数据并可以通过横向扩展来处理大量消息的系统。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。对 LinkedIn 来说，这是一种脆弱的解决方案。ActiveMQ 的很多缺陷会导致 broker 暂停服务，进而导致客户端的连接被阻塞，应用程序处理用户请求的能力也会受到影响。于是，我们最后决定自己构建数据管道基础设施。

1.5.2 Kafka 的诞生

LinkedIn 的开发团队由 Jay Kreps 领导。Jay Kreps 是 LinkedIn 的首席软件工程师，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。他们一起着手创建了一个消息系统，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伸缩。

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，每天可以高效处理数十亿级别的指标和用户活动跟踪信息。借助 Kafka 的伸缩能力，LinkedIn 的消息处理量已经超过 7 万亿条（截至 2020 年 2 月），每天处理超过 5000 万亿字节的数据。

1.5.3 走向开源

2010 年年底，Kafka 作为开源项目在 GitHub 网站上发布。2011 年 7 月，因为倍受开源社区的关注，所以 Kafka 成了 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，同时它还吸引了大批来自 LinkedIn 以外的贡献者和参与者。现在，Kafka 被很多公司（Netflix、Uber 等）用在一些大型的数据管道中。

Kafka 的广泛采用也为 Kafka 核心项目创造了一个健康的生态环境。全世界有几十个国家活跃着 Kafka 讨论小组，他们讨论流式处理，并为其提供支持。除此之外，还有很多与 Kafka 相关的开源项目。LinkedIn 一直在维护其中的几个项目，包括 Cruise Control、Kafka Monitor 和 Burrow。除了提供商业产品，Confluent 还发布了一些项目，包括 ksqlDB、一个模式注册中心和社区版的 REST 代理（严格来说它不算是开源的，因为存在一些使用上的限制）。附录 B 列出了一些最为流行的项目。

1.5.4 商业化

2014 年秋天，Jay Kreps、Neha Narkhede 和 Jun Rao 离开 LinkedIn，成立了 Confluent 公司，主要从事与 Kafka 相关的开发、企业支持和培训业务。他们还与其他公司（如 Heroku）合作，提供 Kafka 云服务。Confluent 与谷歌合作，在谷歌云平台上提供 Kafka 托管集群，同时也在 Amazon Web Services 和 Azure 上提供了类似的服务。Confluent 的另一个主要职责是组织 Kafka 系列峰会。Kafka 峰会始于 2016 年，每年都会在美国的一些城市和英国伦敦举行，让来自全球社区的同人们聚集在一起，分享 Kafka 及相关项目的知识。

1.5.5 命名

人们经常会问到有关 Kafka 历史的问题，即 Kafka 这个名字是怎么想出来的，以及它和项目之间有着怎样的联系。Jay Kreps 对此解释如下。

> 我想，既然 Kafka 是为写数据而生的，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，对开源项目来说，这个名字听起来很酷。因此，Kafka 这个名字和应用程序本身并没有太多联系。

1.6 开始 Kafka 之旅

现在，我们对 Kafka 已经有了一个大体的了解，还知道了一些常见术语，接下来就可以用 Kafka 创建数据管道了。第 2 章将介绍如何安装和配置 Kafka，还会讨论如何为 Kafka 选择合适的硬件，以及把 Kafka 应用到生产环境中时需要注意的事项。

第 2 章　安装 Kafka

本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），还将介绍 Kafka 的基本配置，以及如何为 Kafka 选择合适的硬件，最后将介绍如何在一个集群中安装多个 broker，以及把 Kafka 应用到生产环境需要注意的事项。

2.1 环境配置

在使用 Kafka 之前，需要准备好满足先决条件的运行环境。以下内容将指导你完成这个过程。

2.1.1 选择操作系统

Kafka 是用 Java 开发的应用程序，可以运行在多种操作系统中，比如 Windows、macOS、Linux，等等，但在一般情况下，我们推荐 Linux。本章介绍的安装步骤着重关注如何在 Linux 环境中配置和使用 Kafka。关于如何在 Windows 和 MacOS 中安装 Kafka，请参考附录 A。

2.1.2 安装 Java

在安装 ZooKeeper 或 Kafka 之前，需要有一个 Java 运行时环境。Kafka 和 ZooKeeper 可以运行在所有基于 OpenJDK 的 Java 运行时中，包括 Oracle JDK。最新版本的 Kafka 支持 Java 8 和 Java 11（在撰写本书时）。你既可以使用操作系统提供的 Java 运行时，也可以从网上下载，例如，从 Oracle 网站上下载 Oracle 的版本。虽然 ZooKeeper 和 Kafka 只需要 Java 运行时，但如果要开发应用程序，则需要安装完整的 Java 开发工具包（JDK）。建议安装打过最新补丁的版本，因为旧版本可能存在安全漏洞。在接下来的安装步骤中，假设你已经安装了 /usr/java/jdk-11.0.10 目录下的 JDK 11 Update 10。

2.1.3 安装 ZooKeeper

Kafka 用 ZooKeeper 来保存集群元数据和消费者信息，如图 2-1 所示。ZooKeeper 是一种集中式服务，用于维护配置信息、命名、提供分布式同步和组服务。本书不会深入介绍 ZooKeeper，只会解释与运行 Kafka 相关的部分。你可以直接用 Kafka 发行版提供的脚本运行 ZooKeeper，不过安装完整版的 ZooKeeper 其实也很简单。

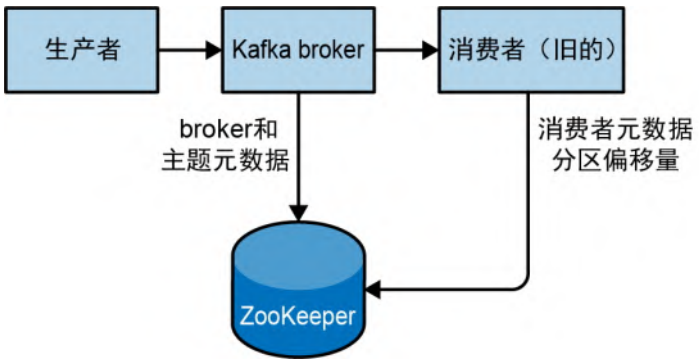

图 2-1：Kafka 和 ZooKeeper

Kafka 已经针对 ZooKeeper 3.5 稳定版进行了广泛的测试，并会定期更新，以便兼容最新的版本。本书将使用 ZooKeeper 3.5.9，安装包可以从 ZooKeeper 网站下载。

01. 单机服务

ZooKeeper 安装目录下的 /usr/local/zookeeper/config/zoo_sample.cfg 文件提供了基本的配置示例，可以被用在大多数场景中。但为了便于演示，本书将使用我们自己创建的配置文件。在下面的示例中，ZooKeeper 的配置文件位于 /usr/local/zookeeper 目录下，数据文件位于 /var/lib/zookeeper 目录下。

```
# tar -zxf apache-zookeeper-3.5.9-bin.tar.gz
# mv apache-zookeeper-3.5.9-bin /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/jdk-11.0.10
# /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。它将返回 Kafka 的基本信息。

```
# telnet localhost 2181
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
srvr
ZooKeeper version: 3.5.9-83df9301aa5c2a5d284a9940177808c01bc35cef, built on
01/06/2021 19:49 GMT
Latency min/avg/max: 0/0/0
Received: 1
Sent: 0
Connections: 1
Outstanding: 0
Zxid: 0x0
Mode: standalone
Node count: 5
Connection closed by foreign host.
#
```

02. ZooKeeper 群组

为了保证高可用，ZooKeeper 以集群（被称为**群组**）的方式运行。由于使用了再均衡算法，建议一个 ZooKeeper 集群应该包含奇数个节点（比如 3 个、5 个等）。只有当群组中的大多数节点（也就是所谓的**仲裁**）处于可用状态时，ZooKeeper 才能处理外部请求。也就是说，一个包含 3 个节点的群组允许 1 个节点失效，而一个包含 5 个节点的群组允许 2 个节点失效。

群组节点个数的选择

假设我们有一个包含 5 个节点的群组，如果要对群组做一些如更换节点的修改，那么就需要依次重启每个节点。如果你的群组无法容忍多个节点失效，那么在维护过程中就存在风险。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，节点过多则会降低整个群组的性能。

此外，如果由于客户端连接太多，5 个或 7 个节点仍然无法支撑负载，则可以考虑增加额外的观察者节点来分摊只读流量。

群组需要有一些公共配置，其中包含了所有服务器的地址，每台服务器还要在自己的数据目录下创建一个 myid 文件，用于指明自己的 ID。如果群组中服务器的主机名是 zoo1.example.com、zoo2.example.com 和 zoo3.example.com，那么配置文件可能如下所示。

```
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 实际的时间是 20×2000 毫秒，也就是 40 秒。配置中还列出了群组中每台服务器的地址，格式为 *server.X=hostname:peerPort:leaderPort*，各个参数说明如下。

X

服务器的 ID，必须是一个整数，不过不一定要从零开始，也不要求是连续的。

hostname

服务器的主机名或 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。

客户端只需通过 *clientPort* 就能连接到群组，但群组的节点必须通过 3 个端口（peerPort、leaderPort、clientPort）进行节点间通信。

除了公共的配置文件，每台服务器必须在 dataDir 指定的目录下创建一个叫作 myid 的文件，文件中要包含服务器 ID，这个 ID 要与配置文件中的 ID 一致。在完成这些步骤之后，就可以启动服务器，让它们之间相互通信了。

在单台机器上测试群组

在配置文件中将所有主机名指定为 localhost，并为每个实例的 *peerPort* 和 *leaderPort* 指定不一样的端口，这样就可以在一台机器上测试和运行 ZooKeeper 群组。此外，还需要为每个实例创建一个单独的 zoo.cfg 文件，并在文件中为每个实例配置唯一的 dataDir 和 *clientPort*。不过，这种方式只建议用于测试目的，不建议用于生产环境中。

2.2 安装 broker

配置好 Java 和 ZooKeeper 之后，接下来就可以安装 Kafka 了。可以从 Kafka 网站下载最新版本的 Kafka。在撰写本书时，Kafka 的版本是 2.8.0，对应的 Scala 版本是 2.13.0。本章的示例使用的是 2.7.0。

下面将 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_2.13-2.7.0.tgz
# mv kafka_2.13-2.7.0 /usr/local/kafka
# mkdir /tmp/kafka-logs
# export JAVA_HOME=/usr/java/jdk-11.0.10
# /usr/local/kafka/bin/kafka-server-start.sh -daemon
/usr/local/kafka/config/server.properties
#
```

创建好 Kafka 之后，可以对这个集群做一些简单的操作，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，然后再读取这些消息。

创建并验证主题。

```
# /usr/local/kafka/bin/kafka-topics.sh --bootstrap-server localhost:9092 --
create
--replication-factor 1 --partitions 1 --topic test
Created topic "test".
# /usr/local/kafka/bin/kafka-topics.sh --bootstrap-server localhost:9092
--describe --topic test
Topic:test    PartitionCount:1    ReplicationFactor:1    Configs:
    Topic: test    Partition: 0    Leader: 0    Replicas: 0    Isr: 0
#
```

往测试主题中发布消息（可以在任意时候按下 Ctrl-C 停止发送消息）。

```
# /usr/local/kafka/bin/kafka-console-producer.sh --bootstrap-server
localhost:9092 --topic test
Test Message 1
Test Message 2
^C
#
```

从测试主题中读取消息。

```
# /usr/local/kafka/bin/kafka-console-consumer.sh --bootstrap-server
localhost:9092 --topic test --from-beginning
Test Message 1
Test Message 2
^C
Processed a total of 2 messages
#
```

弃用 Kafka 命令行中的 ZooKeeper 连接串

如果对旧版本的 Kafka 命令行很熟悉，那么你可能已经习惯使用 `--zookeeper` 连接串。这种方式已经被弃用了，现在建议直接使用新的 `--bootstrap-server` 连接串。如果是在集群内使用命令行，那么你可以为它指定集群内任意一个 broker 的主机地址和端口。

2.3 配置 broker

Kafka 发行版中自带的配置示例可以用来安装单机服务，主要用于概念验证，并不能满足大型集群的配置需求。Kafka 有很多配置参数，涉及安装和调优的方方面面，其中的大多数参数可以使用默认值，除非你对调优有特别的需求。

2.3.1 常规配置参数

一些配置参数在单机安装时可以使用默认值，但部署到其他环境中时要格外小心。这些参数是针对单台服务器最基本的配置，其中大多数参数需要经过修改后才能用在集群中。

01. `broker.id`

    每个 broker 都需要有一个整数标识符，该标识符是使用 `broker.id` 指定的。它的默认值是 `0`，但可以被设置成其他任意整数。这个值在整个 Kafka 集群中必须是唯一的，并且可以在服务器节点间移动。建议把 ID 设置成与主机名具有相关性的整数，这样就可以很容易地将 ID 与主机名映射起来。如果主机名包含唯一性的数字（比如 host1.example.com、host2.example.com 等），那么用 1、2 这些数字来设置 `broker.id` 就再好不过了。

02. `listeners`

    旧版 Kafka 使用的是简单的端口配置。对于简单的应用场景可以继续使用这种配置，但这种方式已经被弃用了。如果使用了 Kafka 自带的示例配置，那么 Kafka 在启动时会在 TCP 端口 9092 上设置一个监听器。新的 `listeners` 配置参数是一个用逗号分隔的 URI 列表，也就是要监听的地址和端口。如果没有为监听器指定安全协议，则还需要额外配置 `listener.security.protocol.map` 参数。监听器的格式为 *`<protocol>://<hostname>:<port>`*，例如，`PLAINTEXT://localhost:9092,SSL://:9091` 就是一个合法的配置。如果主机名是 `0.0.0.0`，那么将绑定所有的网络接口地址。如果主机名为空，那么将绑定默认的网络接口地址。需要注意的是，如果指定的端口号小于 1024，则必须用 root 权限启动 Kafka，但不建议这么配置。

03. `zookeeper.connect`

    用于保存 broker 元数据的 ZooKeeper 地址是通过 `zookeeper.connect` 来指定的。示例配置使用了一个运行在 2181 端口上的 ZooKeeper，所以指定了 `localhost:2181`。这个参数的值是用逗号分隔的一组 `hostname:port/path`，每一部分的含义如下。

    `hostname`

    ZooKeeper 服务器的主机名或 IP 地址。

    `port`

    ZooKeeper 的客户端连接端口。

    `/path`

    可选的 ZooKeeper 路径，以作为 Kafka 集群的 chroot。如果不指定，则默认使用根路径。

    如果指定的 chroot 路径（一般作为应用程序的根目录）不存在，那么 broker 会在启动时创建它。

为什么使用 chroot 路径？

使用 chroot 路径是一种最佳实践，因为这样可以在不发生冲突的情况下将 ZooKeeper 群组共享给其他应用程序（包括其他 Kafka 集群）。另外，最好在配置文件中指定属于同一群组的所有 ZooKeeper 服务器地址，因为一旦有节点停机，broker 就可以连到其他节点上。

04. `log.dirs`

Kafka 把所有消息都保存在磁盘上，存放日志片段的目录是通过 `log.dir` 来指定的。如果有多个目录，则可以用 `log.dirs` 来指定。如果没有指定这个参数，则默认使用 `log.dir`。`log.dirs` 是一组用逗号分隔的本地文件系统路径。如果指定了多条路径，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条路径下。需要注意的是，broker 会向分区数量最少的目录新增分区，而不是向可用磁盘空间最小的目录新增分区，所以并不能保证数据会被均匀地分布在多个目录中。

05. `num.recovery.threads.per.data.dir`

Kafka 使用线程池来处理日志片段。目前，线程池被用于以下 3 种情形。

- 当服务器正常启动时，用于打开每个分区的日志片段。
- 当服务器发生崩溃并重启时，用于检查和截短每个分区的日志片段。
- 当服务器正常关闭时，用于关闭日志片段。

在默认情况下，每个日志目录只使用一个线程。因为这些线程只在服务器启动和关闭时使用，所以可以多设置一些线程来实现并行操作。特别是对包含大量分区的服务器来说，一旦发生崩溃，在从错误中恢复时可以通过并行操作省下数小时的时间。需要注意的是，这个参数对应的是 `log.dirs` 中的一个目录，也就是说，如果 `num.recovery.threads.per.data.dir` 被设为 8，并且 `log.dirs` 指定了 3 条路径，那么总共需要 24 个线程。

06. `auto.create.topics.enable`

在默认情况下，Kafka 会在如下几种情形中自动创建主题。

- 当一个生产者开始向主题写入消息时。
- 当一个消费者开始从主题读取消息时。
- 当客户端向主题发送获取元数据的请求时。

很多时候，我们并不期望这些行为的发生。根据 Kafka 协议，如果一个主题不提前被创建，则无法知道它是否存在。如果你选择通过手动的方式或其他配置系统来显式地创建主题，那么可以把 `auto.create.topics.enable` 设为 `false`。

07. `auto.leader.rebalance.enable`

为了确保主题的所有权不会集中在一台 broker 上，可以将这个参数设置为 `true`，让主题的所有权尽可能地在集群中保持均衡。如果启用了这个功能，那么就会有一个后台线程定期检查分区的分布情况（这个时间间隔可以通过 `leader.imbalance.check.interval.seconds` 来配置）。如果不均衡的所有权超出了 `leader.imbalance.per.broker.percentage` 指定的百分比，则会启动一次分区首领再均衡。

08. `delete.topic.enable`

根据应用场景和数据保留策略的不同，你可能希望将集群锁定，以防主题被随意删除。把这个参数设置为 `false` 就可以禁用主题删除功能。

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的配置参数。可以通过管理工具（参见第 12 章）为每个主题单独配置一些参数，比如分区数和数据保留策略。还可以将服务器提供的默认配置作为基准，应用于集群内的大部分主题。

个体主题的配置覆盖

旧版本 Kafka 允许个体主题使用 `log.retention.hours.per.topic`、`log.retention.bytes.per.topic` 和 `log.segment.bytes.per.topic` 覆盖默认配置。但现在不再支持这些参数，而且如果想覆盖参数配置，则必须使用管理工具。

01. `num.partitions`

`num.partitions` 参数指定了新创建的主题将包含多少个分区，特别是如果启用了主题自动创建功能（默认是启用的），那么主题的分区数就是这个参数指定的值。默认是 1 个分区。需要注意的是，可以增加主题的分区数，但不能减少。所以，如果要让一个主题的分区数小于 `num.partitions` 指定的值，则需要手动创建主题（参见第 12 章）。

第 1 章中提到过，Kafka 集群通过分区来实现主题的横向伸缩，当不断有新 broker 加入集群时，通过分区数来实现集群的负载均衡就变得十分重要。很多用户将主题的分区数设置为集群 broker 的数量，或者是它的倍数，这样可以让分区均衡地分布到 broker 上，进而均衡地分布消息负载。例如，在一个包含 10 台主机的 Kafka 集群中，有一个主题包含了 10 个分区，这 10 个分区的所有权均衡地分布在这 10 台主机上，这样可以获得最佳的吞吐性能。但并不是必须这样做，因为还可以通过其他方式来实现消息负载均衡，比如使用多个主题。

如何选择分区数量？

在选择主题的分区数量时，需要考虑如下因素。

- 主题需要达到多大的吞吐量？例如，是希望写入 100 KBps 还是 1 GBps？
- 从单个分区读取数据的最大吞吐量是多少？通常每个分区都会有一个消费者读取。（即使不使用消费者群组，消费者也必须读取分区中的所有消息。）如果你知道消费者将数据写入数据库的速率不会超过 50 MBps，那么也就知道了从一个分区读取数据的吞吐量不需要超过 50 MBps。
- 可以通过与上面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，生产者的速度通常比消费者快得多，所以最好为生产者多估算一些吞吐量。
- 如果消息是按照不同的键写入分区，那么就很难在未来为已有的主题新增分区，所以要根据未来的预期使用量而不是当前的使用量来估算吞吐量。
- 每个 broker 包含的分区数、可用的磁盘空间和网络带宽。
- 避免使用太多分区，因为每个分区都会占用 broker 的内存和其他资源，还会增加元数据更新和首领选举的时间。
- 是否需要镜像数据？如果是，那么你可能还需要考虑镜像吞吐量。大型分区可能会成为镜像的瓶颈。
- 如果你使用的是云服务，那么你的虚拟机或磁盘有 IOPS（每秒输入 / 输出操作）限制吗？云服务和虚拟机的配额可能会硬性限制 IOPS 的数量。因为涉及并行操作，所以分区数量太大可能会导致 IOPS 数量增加。

很显然，综合考虑以上因素，你可能需要很多分区，但又不能太多。如果你已经估算出主题吞吐量和消费者吞吐量，那么可以用主题吞吐量除以消费者吞吐量来计算分区数。如果要向主题写入和从主题读取 1 GBps 的数据，并且每个消费者可以处理 50 MBps 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 1 GBps 的吞吐量。

如果你无法获得这些信息，那么根据经验，将分区每天保留的数据限制在 6 GB 以内可以获得比较理想的效果。先从小容量开始，再根据需要进行扩展，这比一开始就使用大容量要容易得多。

02. `default.replication.factor`

如果启用了自动创建主题功能，那么这个参数的值就是新创建主题的复制系数。不同的集群持久性或可用性需求需要不同的复制策略，后面的章节会深入讨论这方面的内容。下面这个简单的建议可用于防止由 Kafka 外部因素（如硬件故障）导致的中断。

建议将复制系数设置为至少比 `min.insync.replicas` 大 1 的数。为了提升故障对抗能力，如果你有足够大的集群和足够多的硬件资源，则可以将复制系数设置为比 `min.insync.replicas` 大 2 的数（简写为 RF++）。RF++ 让集群维护变得更加容易，也能更好地防止停机，因为它允许集群内同时发生一次计划内停机和一次计划外停机。对于典型的集群，这意味着每个分区至少要有 3 个副本。如果在滚动部署或升级 Kafka 或底层操作系统期间出现网络交换机中断、磁盘故障或其他计划外的问题，你可以保证仍然有 1 个副本可用。第 7 章将讨论更多的细节。

03. `log.retention.ms`

Kafka 通常根据配置的时间长短来决定数据可以被保留多久。我们使用 `log.retention.hours` 参数来配置时间，默认为 168 小时，也就是 1 周。除此以外，还有另外两个参数 `log.retention.minutes` 和 `log.retention.ms`。这 3 个参数的作用是一样的（都是用于确定消息将在多久以后被删除），不过还是推荐使用 `log.retention.ms`，因为如果指定了不止一个参数，那么 Kafka 会优先使用具有最小单位值的那个。

根据时间保留数据与日志文件的最后修改时间

根据时间保留数据是通过检查日志片段文件的最后修改时间来实现的。一般来说，最后修改时间就是日志片段的关闭时间，也就是文件中最后一条消息的时间戳。不过，如果使用管理工具在服务器间移动分区，那么最后修改时间就不准确了，这种误差可能会导致这些分区过多地保留数据。第 12 章在讨论分区移动时将更详细地介绍这方面的内容。

04. `log.retention.bytes`

另一种数据保留策略是通过计算已保留的消息的字节总数来判断旧消息是否过期。这个字节总数阈值通过参数 `log.retention.bytes` 来指定，对应的是每一个分区。也就是说，如果一个主题包含 8 个分区，并且 `log.retention.bytes` 被设置为 1 GB，那么这个主题最多可以保留 8 GB 的数据。需要注意的是，所有保留都是针对单个分区而不是主题执行的。

所以，如果配置了这个参数，那么当主题增加了新分区，整个主题可以保留的数据也会随之增加。如果这个值被设置为–1，那么分区就可以无限期地保留数据。

同时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（或另一个按时间保留的参数），那么只要任意一个条件得到满足，消息就会被删除。假设 log.retention.ms 被设置为 86 400 000（也就是 1 天），log.retention.bytes 被设置为 1 000 000 000（也就是 1 GB），如果消息字节总数不到一天就超过了 1 GB，那么旧数据就会被删除。相反，如果消息字节总数小于 1 GB，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 1 GB。为简单起见，建议只选择其中的一种保留策略，要么基于数据大小，要么基于时间，或者两种都不选择，以防发生意外的数据丢失。不过，对于复杂的场景，可以两种都使用。

05. log.segment.bytes

上面介绍的参数都作用在日志片段而不是单条消息上。当消息到达 broker 时，它们会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 1 GB）时，当前日志片段会被关闭，一个新的日志片段会被打开。一旦日志片段被关闭，就可以开始进入过期倒计时。这个参数的值越小，关闭和分配新文件就会越频繁，从而降低整体的磁盘写入效率。

如果主题的消息量不是很大，那么如何设置这个参数就变得尤为重要。如果一个主题每天只接收 100 MB 的消息，并且 log.segment.bytes 使用了默认设置，那么填满一个日志片段将需要 10 天。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，而根据配置的过期时间，还需要再保留数据 7 天（要等到日志片段的最后一条消息过期才能将其删除）。

使用时间戳获取偏移量

日志片段的大小也会影响使用时间戳获取偏移量的行为。当使用时间戳获取日志偏移量时，Kafka 会查找在指定时间戳写入的日志片段文件，也就是创建时间在指定时间戳之前且最后修改时间在指定时间戳之后的文件。然后，Kafka 会返回这个日志片段开头的偏移量（也就是文件名）。

06. log.roll.ms

另一个可用于控制日志片段关闭时间的参数是 log.roll.ms，它指定了多长时间之后日志片段可以被关闭。就像 log.retention.bytes 和 log.retention.ms 一样，log.segment.bytes 和 log.roll.ms 并不互斥。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。在默认情况下，log.roll.ms 没有设定值，所以使用 log.roll.hours 设定的默认值——168 小时，也就是 7 天。

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，需要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一直未达到大小的上限，就会出现这种情况。这是因为 broker 在启动时会计算日志片段的过期时间，一旦满足条件，就会并行关闭它们，尽管它们的数据量可能很少。

07. min.insync.replicas

为了提升集群的数据持久性，可以将 min.insync.replicas 设置为 2，确保至少有两个副本跟生产者保持“同步”。生产者需要配合将 ack 设置为 all，这样就可以确保至少有两个副本（首领和另一个副本）确认写入成功，从而防止在以下情况下丢失数据：首领确认写入，然后发生停机，所有权

被转移到一个副本，但这个副本没有写入成功。如果没有这些配置，则生产者会认为已经写入成功，但实际上消息丢失了。不过，这样做是有副作用的，因为需要额外的开销，所以效率会有所降低。因此，对于能够容忍偶尔消息丢失的高吞吐量集群，不建议修改这个参数的默认值。第 7 章会讨论这方面的更多内容。

08. `message.max.bytes`

broker 通过设置 `message.max.bytes` 参数来限制单条消息的大小，默认值是 1 000 000，也就是 1 MB。如果生产者尝试发送超过这个大小的消息，那么不仅消息不会被 broker 接收，还会收到 broker 返回的错误信息。与其他 broker 配置参数一样，这个参数指的是压缩后的消息大小，也就是说，消息的实际大小可以远大于 `message.max.bytes`，只要压缩后小于这个值即可。

这个参数对性能有显著的影响。值越大，负责处理网络连接和请求的线程用在处理请求上的时间就越长。它还会增加磁盘写入块的大小，从而影响 I/O 吞吐量。其他的存储解决方案，比如大对象存储或分层存储，可能也是解决大磁盘写入问题的一种方案，但本章不对它们做更多的介绍。

协调服务器端和客户端之间的消息大小配置

消费者客户端设置的 `fetch.max.bytes` 需要与服务器端设置的消息大小保持一致。如果这个参数的值比 `message.max.bytes` 小，那么消费者就无法读取比较大的消息，进而造成阻塞，无法继续处理消息。在配置 broker 的 `replica.fetch.max.bytes` 参数时，也遵循同样的原则。

2.4 选择硬件

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中。不过，如果比较注重性能，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、网络以及 CPU。当 Kafka 扩展到非常大的规模时，由于需要更新大量的元数据，单个 broker 能够处理的分区数量就存在一定的限制。一旦确定了性能关注点，就可以在预算范围内选择最优的硬件配置。

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，直到至少有一台服务器确认消息已成功提交为止。也就是说，磁盘写入的速度越快，生成消息的延迟就越低。

如果考虑硬盘类型对磁盘吞吐量的影响，那么是选择传统的机械硬盘（HDD）还是固态硬盘（SSD），可以很容易做出决定。固态硬盘的查找和访问速度都很快，提供了最好的性能。机械硬盘则更便宜，单块硬盘容量也更大。我们可以在同一台服务器上使用多个机械硬盘，设置多个数据目录，或者把它们设置成磁盘阵列，以提升机械硬盘的性能。其他方面的因素，比如磁盘使用的特定技术（串行连接存储技术或 SATA）或磁盘控制器的质量，都会影响吞吐量。经验表明，机械硬盘更适用于高存储量但不经常被访问的集群，而如果有非常多的客户端连接，则固态硬盘是更好的选择。

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多少消息。如果服务器每天收到 1 TB 消息，需要保留 7 天，那么就需要 7 TB 的存储空间，另外还要为其他文件提供至少 10% 的额外空间。除此之外，还需要提供额外的缓冲区，用于应对消息流量的增长和波动。

在决定扩展 Kafka 集群规模时，存储容量是另一个需要考虑的因素。我们可以为主题设置多个分区，让集群的总流量均衡分布到整个集群。如果单个 broker 容量不足，则可以让其他 broker 提供可用的容量。存储容量的选择也受集群复制策略的影响（第 7 章将讨论更多的细节）。

2.4.3 内存

一般来说，Kafka 消费者从分区的末尾（也就是在生产者成功写入数据之后）读取数据，即使出现了滞后，也不会滞后太多。消费者正在读取的消息会被放到系统的页面缓存中，这比从磁盘上重新读取的速度更快。因此，为系统提供更多的内存用于页面缓存可以提高消费者客户端的性能。

Kafka 本身不需要太多内存。一个每秒处理 150 000 条消息和每秒 200 MB 数据速率的 broker，只需要 5 GB 堆内存，剩下的系统内存用于页面缓存。因为缓存了正在使用的日志片段，所以 Kafka 的性能得到了提升。这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，因为它们需要共享页面缓存，从而会降低 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 伸缩规模的主要因素。Kafka 支持多消费者，导致流入和流出的网络流量不均衡，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写入 1 MB 数据，但可能同时存在多个消费者，这就需要数倍的流出网络流量。其他的操作，比如集群复制（参见第 7 章）和镜像（参见第 10 章），也会占用网络流量。如果网络接口达到饱和状态，那么集群的复制出现延时就在所难免，从而让集群变得不堪一击。为了避免网络成为主要的制约因素，建议至少使用 10 GB 的网卡。配备 1 GB 网卡的老机器很容易达到饱和状态，不推荐使用。

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，不过在一定程度上还是会影响整体的性能。为了优化网络和节省磁盘空间，客户端会对消息进行压缩。服务器需要对消息进行解压缩，检查每条消息的校验和并分配偏移量，然后再将其压缩保存到磁盘上。这个过程是 Kafka 需要用到计算处理能力的地

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，除非集群变得非常大，大到单个集群有数百个节点和数百万个分区。如果真的到了那个时候，则可以选择更高性能的 CPU，避免集群规模太过膨胀。

2.5 云端的 Kafka

近年来，Kafka 更经常被部署在云计算环境中，比如微软 Azure、亚马逊云科技（AWS）或谷歌云平台。云端 Kafka 有很多选择，Confluent 和 Azure HDInsight 都提供了托管 Kafka，但如果你打算手动管理自己的 Kafka 集群，那么下面提供了一些简单的建议。大多数云环境提供了各种选项，你可以选择不同的计算实例，每个计算实例有不同的 CPU、内存、IOPS 和磁盘配置。要选择合适的实例配置，需要对 Kafka 的各种性能特征做一下优先级排列。

2.5.1 微软 Azure

在 Azure 中，虚拟机和磁盘是分开管理的，所以存储需求与虚拟机类型的选择就没有什么关系了。我们可以从需要保留的数据量开始考虑，然后再考虑生产者的性能。如果要求非常低的延迟，那么就需要使用配备了高级 SSD 存储的实例。否则的话，托管存储（比如 Azure 托管磁盘或 Azure Blob 存储）可能就足够了。

在实际当中，Azure 的使用经验表明，对小规模集群来说，`Standard_D16s_v3` 实例是一个很好的选择，其性能已经能够满足大多数应用场景。对大规模集群来说，`Standard_D64s_v4` 实例提供了更好的性能。建议用户在 Azure 中构建集群，将分区均衡地分布在 Azure 的计算故障区域之间，以此来确保可用性。在选择了虚拟机类型之后，接下来需要考虑存储类型。强烈建议用户选择 Azure 托管磁盘，而不是临时磁盘。这是因为如果选择临时磁盘，那么一旦虚拟机被移动，就存在 broker 会丢失所有数据的风险。托管机械磁盘相对便宜，但微软没有在可用性方面为其定义明确的 SLA。高级固态硬盘或超级固态硬盘配置相对较贵，但速度更快，而且微软为其提供了 99.99% 的 SLA 支持。另外，如果对延迟不敏感，则可以使用微软 Blob 存储。

2.5.2 AWS

在 AWS 中，如果要求非常低的延迟，那么可以选择配备了本地固态硬盘的实例。否则的话，临时存储方案（如亚马逊弹性块存储）可能就足够了。

AWS 中常见的选择是 `m4` 或 `r3` 实例类型。`m4` 实例可以更长时间地保留数据，但磁盘吞吐量相对较低，因为它是基于弹性块存储的。`r3` 实例配备了本地固态硬盘，提供了更高的吞吐量，但可以保留的数据量有限。如果在数据量和吞吐量两个方面都有很高的要求，那么可能需要考虑 `i2` 或 `d2` 实例类型，但它们要贵很多。

2.6 配置 Kafka 集群

在进行本地开发或概念验证时，单台 Kafka 服务器就足够了。不过，使用集群可以获得更多的好处，如图 2-2 所示。使用集群的最大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。在进行 Kafka 或底层系统维护时，集群可以为客户端提供高可用性。本节只介绍如何配置 Kafka 集群，第 7 章将介绍更多有关数据复制和数据持久性的内容。

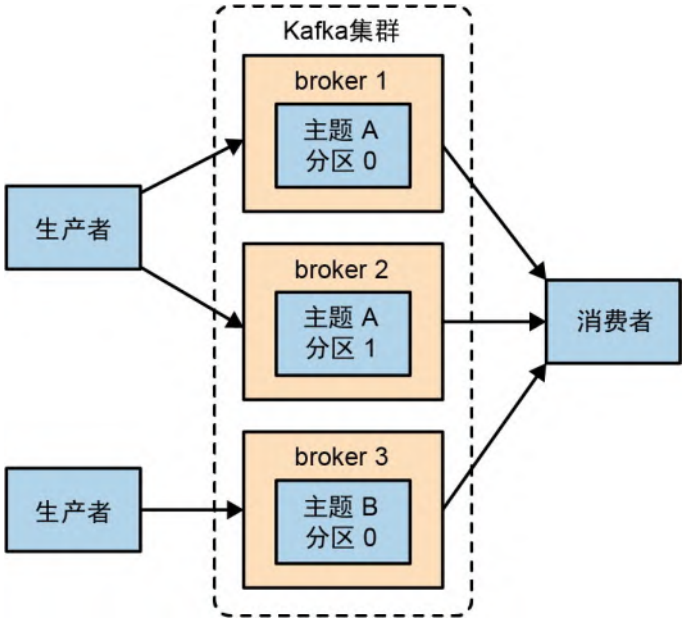

图 2-2：一个简单的 Kafka 集群

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。

- 磁盘容量
- 单个 broker 的复制容量
- CPU
- 网络

第一个要考虑的因素是需要多少磁盘空间来保留数据，以及单个 broker 有多少可用空间。如果整个集群需要保留 10 TB 数据，每个 broker 可以存储 2 TB，那么至少需要 5 个 broker。另外，如果增加了复制系数，那么至少还需要多一倍的空间，具体取决于配置的复制系数是多少（第 7 章将介绍复制系数）。复制系数是指一个 broker 的数据需要被复制到多少个其他 broker 上。如果这个集群配置的复制系数是 2，那么将需要至少 10 个 broker。

另一个要考虑的因素是集群处理请求的能力，这可以通过之前提到的 3 个瓶颈表现出来。

假设你有一个包含 10 个 broker 的集群，集群中有 100 多万个副本（也就是说，有 500 000 个分区，复制系数为 2），如果分区分布得均衡，那么每个 broker 大约有 100 000 个副本。这可能会导致生产者、消费者和控制器队列出现瓶颈。在过去，官方的建议是每个 broker 的分区副本不超过 4000 个，每个集群的分区副本不超过 200 000 个。随着集群效率的提升，Kafka 可以被扩展到更大的规模。目前，建议每个 broker 的分区副本不超过 14 000 个，每个集群的分区副本不超过 100 万个。

之前提到过，在大多数情况下，CPU 通常不是主要瓶颈，但如果有很多客户端连接和请求，则 CPU 可能会成为瓶颈。在观察大型集群的 CPU 整体使用情况时，需要关注有多少个客户端和消费者群组，并进行

相应的伸缩，以满足它们的需求，这样有助于确保更好的整体性能。在网络容量方面，应主要关注网络接口容量，以及它们是否能够在有多个消费者或流量与数据保留策略不一致（比如高峰期的流量激增）的情况下处理客户端流量。如果一个 broker 的网络接口在高峰期使用了 80% 的容量，并且有两个消费者，那么除非有两个 broker，否则消费者将无法及时读取高峰流量。如果集群启用了复制功能，那么就又多了一个消费者。为了解决因磁盘吞吐量不足或可用系统内存较少而引起的性能问题，你可能需要通过增加更多的 broker 来扩展集群。

2.6.2 broker 配置

要让一个 broker 加入集群，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`，这个参数指定了用于保存元数据的 ZooKeeper 的群组和路径。其次，每个 broker 都必须为 `broker.id` 指定唯一的值。如果两个 broker 使用相同的 `broker.id`，那么第二个 broker 将无法启动。还可以为集群配置其他的一些参数，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，通常配置在 /etc/sysctl.conf 文件中。在修改这些内核参数时，最好参考一下操作系统发行版的文档。

01. 虚拟内存

一般来说，Linux 的虚拟内存会根据系统的工作负载进行自动调整。我们可以对交换分区的处理方式以及内存脏页做一些调整，让 Kafka 更好地处理工作负载。

对大多数应用程序（特别是那些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数据交换对 Kafka 各方面的性能都有重大影响。Kafka 大量使用了系统页面缓存，如果虚拟内存被交换到磁盘，则说明已经没有多余内存可以分配给页面缓存。

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。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些，比如设置为 1。这个参数是指虚拟机子系统使用交换分区的权重百分比。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` 设置为 0？

在以前，人们建议把 `vm.swappiness` 设置为 0，意思是“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，这个值的含义才发生了变化。这个变化被移植到了其他发行版中，包括 Red Hat 企业版内核 2.6.32-303。于是，0 的意思就变成“在任何情况下都不要发生交换”。这也就是为什么现在建议把这个值设置为 1。

我们可以调整内核处理脏页的方式，并从中获得一些好处。Kafka 借助较高的 I/O 性能为生产者提供快速响应，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（如固态硬盘），还是具有 NVRAM 缓存的磁盘子系统（冗余磁盘阵列）。这样一来，在后台刷新进程将脏页写入磁盘之前脏页数量就减少了。我们可以将 `vm.dirty_background_ratio` 设置为小于默认值 10 的数字。这个参数是指脏页占系统内存的百分比，大多数情况下设置为 5 就可以了。它不应该被设置为 0，因为那样会促使内核频繁地刷新页面，从而降低内核为磁盘写入提供缓冲的能力。

如果要增加被内核进程刷新到磁盘之前的脏页数量，那么可以将 `vm.dirty_ratio` 设置为大于 20 的值（这个数值也是指脏页占系统内存的百分比）。这个值可设置的范围很广，60~80 是一个比较合理的区间。不过修改这个参数会带来一些风险，比如未刷新的磁盘操作的数量会增加，以及同步刷新导致的 I/O 等待时间会更长。如果将 `vm.dirty_ratio` 设置了较高的值，那么建议启用 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，建议在 Kafka 集群运行期间监控脏页的数量，不管是在生产环境中还是在模拟环境中。可以通过 /proc/vmstat 文件查看当前的脏页数量。

```
# cat /proc/vmstat | egrep "dirty|writeback"
nr_dirty 21845
nr_writeback 0
nr_writeback_temp 0
nr_dirty_threshold 32715981
nr_dirty_background_threshold 2726331
#
```

Kafka 的日志片段文件和已打开的连接都使用了文件描述符。除了已打开的连接，一个 broker 如果有很多分区，那么它至少还需要 (分区数量)×(分区大小 / 日志片段大小) 个文件描述符来跟踪所有的日志片段。因此，建议将 vm.max_map_count 设置为一个非常大的数字（基于前面的公式）。根据环境的不同，将这个值设置为 400 000 或 600 000 是没有问题的。另外，建议将 vm.overcommit_memory 设置为 0。0 表示内核会根据应用程序来确定空闲内存的数量。如果这个参数被设置为 0 以外的值，则可能会导致操作系统夺走过多内存，从而减少 Kafka 的运行内存，这种情况在高摄取率的应用程序中很常见。

02. 磁盘

除了磁盘硬件设备和冗余磁盘阵列，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，对本地文件系统来说，Ext4（第四代扩展文件系统）和 XFS（扩展文件系统）最为常见。XFS 是很多 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载，表现要优于 Ext4。Ext4 也可以表现得很好，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时间间隔（默认是 5），以便降低刷新频率。Ext4 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 要安全些。XFS 为 Kafka 提供了更好的性能，除了文件系统提供的自动调优，不需要再做额外的调优。另外，XFS 的批量磁盘写入的效率更高，所有这些组合在一起，提高了整体的 I/O 吞吐量。

不管使用哪一种文件系统来存储日志片段，都要合理设置挂载点的 noatime 属性。文件元数据包含 3 个时间戳：创建时间（ctime）、最后修改时间（mtime）和最后访问时间（atime）。在默认情况下，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。atime 属性的用处并不大，除非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（对于这种情况可以使用 relatime）。Kafka 没有用到 atime 这个属性，所以完全可以将其禁用。设置挂载点的 noatime 属性可以避免更新 atime，但又不影响 ctime 和 mtime。当有大量磁盘写入时，通过设置 largeio 也有助于提升 Kafka 的效率。

03. 网络

在默认情况下，系统内核并没有针对高速大流量网络传输做过优化，所以对需要处理大网络流量的应用程序来说，一般需要对 Linux 系统的网络栈进行调优。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，可以调整分配给 socket 读写缓冲区的默认内存和最大内存，这样可以显著提升大流量网络的传输性能。socket 读写缓冲区内存对应的参数分别是 net.core.wmem_default 和 net.core.rmem_default，合理的值是 131 072（也就是 128 KiB）。读写缓冲区最大内存对应的参数分别是 net.core.wmem_max 和 net.core.rmem_max，合理的值是 2 097 152（也就是 2 MiB）。需要注意的是，最大值并不意味着每个 socket 一定要分配这么大的缓冲空间，只是在必要的情况下才会达到这个上限。

除了配置 socket 参数，还需要配置 TCP socket 的读写缓冲区，对应的参数分别是 net.ipv4.tcp_wmem 和 net.ipv4.tcp_rmem。这些参数的值由 3 个整数组成，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 4 KiB、默认值是 64

KiB、最大值是 2 MiB。根据 Kafka 服务器接收流量的实际情况，可能需要设置更大的最大值，以便为网络连接提供更大的缓冲空间。

除此之外，还有其他一些有用的网络参数。例如，可以把 `net.ipv4.tcp_window_scaling` 设置为 1，启用 TCP 时间窗口扩展，以此来提升客户端传输数据的效率，并允许在服务器端缓冲传输的数据。也可以把 `net.ipv4.tcp_max_syn_backlog` 设置为比默认值 1024 更大的值，这样就可以处理更多的并发连接。还可以把 `net.core.netdev_max_backlog` 设置为比默认值 1000 更大的值，通过允许更多的数据包排队等待内核处理，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。

2.7 生产环境的注意事项

当你准备好把 Kafka 从测试环境部署到生产环境时，一些注意事项可以帮你构建出更可靠的消息服务。

2.7.1 垃圾回收器选项

调整 Java 垃圾回收器选项就像是一门艺术，不仅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 为我们带来了垃圾回收器 G1GC，让这种状况有所改观。最开始 G1GC 还不太稳定，但其在 JDK 8 和 JDK 11 中有了显著改进。建议将 G1GC 作为 Kafka 的默认垃圾回收器。在应用程序的整个生命周期中，G1GC 会根据工作负载进行自我调节，并且停顿时间是恒定的。它可以轻松收集大块堆内存，把堆内存分为若干小块区域，并不是每次都回收整个堆空间。

正常情况下，G1GC 只需要很少的配置就能完成这些工作。以下是 G1GC 的两个主要的性能调优参数。

`MaxGCPauseMillis`

该参数指定了每次垃圾回收的默认停顿时间。这个值不是固定的，G1GC 可以根据具体情况进行调整，默认是 200 毫秒。G1GC 可以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 200 毫秒。

`InitiatingHeapOccupancyPercent`

该参数指定了在 G1GC 启动新一轮垃圾回收之前可以使用的堆内存百分比，默认是 45。也就是说，在堆内存使用率达到 45% 之前，G1GC 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 使用堆内存以及处理垃圾对象的效率是比较高的，所以可以把这些参数设置得小一些。如果一台服务器有 64 GB 内存，并使用 5 GB 堆内存来运行 Kafka，那么可以将 `MaxGCPauseMillis` 设置为 20 毫秒，将 `InitiatingHeapOccupancyPercent` 设置为 35，让垃圾回收比默认的早一些启动。

Kafka 是在 G1GC 之前发布的，因此，它默认使用的是 CMS（并发标记和清除）垃圾回收器，以确保与所有的 JVM 兼容。现在，强烈建议使用 G1GC，只要 Java 版本大于等于 1.8 即可。我们可以通过修改环境变量来实现，比如，将之前的启动命令修改为如下形式。

```
# export KAFKA_JVM_PERFORMANCE_OPTS="-server -Xmx6g -Xms6g
-XX:MetaspaceSize=96m -XX:+UseG1GC
-XX:MaxGCPauseMillis=20 -XX:InitiatingHeapOccupancyPercent=35
-XX:G1HeapRegionSize=16M -XX:MinMetaspaceFreeRatio=50
-XX:MaxMetaspaceFreeRatio=80 -XX:+ExplicitGCInvokesConcurrent"
# /usr/local/kafka/bin/kafka-server-start.sh -daemon
/usr/local/kafka/config/server.properties
#
```

2.7.2 数据中心布局

在开发和测试环境中，人们并不太关心 Kafka 服务器在数据中心的物理位置，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，在生产环境中，服务不可用就意味着金钱的损失，具体表现为无法为用户提供服务或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（参见第 7 章），而服务器在数据中心的物理位置随之也变得重要起来。数据中心最好要有故障区域。如果在部署 Kafka 之前没有考虑好这个问题，那么以后就需要耗费更高的成本来移动服务器。

Kafka 可以将新创建的分区分配给部署在不同机架上的 broker（机架感知），确保单个分区的副本不会都位于同一个机架。要做到这一点，必须正确配置每个 broker 的 `broker.rack` 参数。云环境中的故障区域也可以进行同样的配置。不过，这只适用于新创建的分区。Kafka 集群不会监控哪些已有的分区不再有机架感知（例如，由于分区重分配导致的），也不会进行自动纠正。所以，建议使用集群均衡工具来保持分区的机架感知，比如 Cruise Control（参见附录 B）。恰当配置这个属性有助于确保持续的机架感知。

总的来说，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）。除了使用两个电源连接，最好把 broker 安装在不同的机架上，因为随着时间的推移，机架也需要维护，这个时候机器需要离线（比如移动机器或重新连接电源）。

2.7.3 共享 ZooKeeper

Kafka 使用 ZooKeeper 保存 broker、主题和分区的元数据。只有当消费者群组成员或 Kafka 集群本身发生变化时才会向 ZooKeeper 写入数据。这些流量通常很小，所以没有必要为单个 Kafka 集群使用专门的 ZooKeeper 群组。实际上，有很多 Kafka 部署环境使用单个 ZooKeeper 群组来保存多个 Kafka 集群的元数据（正如本章之前所描述的那样，每个 Kafka 集群使用一个单独的 chroot 路径）。

Kafka 消费者、工具、ZooKeeper 和你

随着时间的推移，Kafka 对 ZooKeeper 的依赖在减少。在 2.8.0 版本中，Kafka 做了一个完全无 ZooKeeper 的早期尝试，但还没有做好生产就绪的准备。一路走来，可以看到各个版本已经在逐步减少对 ZooKeeper 的依赖。例如，在旧版本 Kafka 中，除了 broker，消费者也利用 ZooKeeper 来保存消费者群组的信息和已消费的主题的信息，并定期提交分区的偏移量（为了实现消费者群组内的故障转移）。在 0.9.0.0 版本中，消费者接口发生了变化，我们可以直接在 Kafka 中完成上述的这些任务。在每一个 2.*x* 版本中，我们都看到 Kafka 的一些路径移除了对 ZooKeeper 的依赖。现在，Kafka 管理工具不再需要连到 ZooKeeper，它们可以直接连到 Kafka 集群完成主题创建、动态配置变更等操作。同样，很多以前使用 `--zookeeper` 选项的命令行工具现在改用了 `--bootstrap-server`。`--zookeeper` 选项仍然可以使用，但已经处于被弃用的状态。当将来 Kafka 不再需要通过连接 ZooKeeper 进行主题的创建、管理或消费时，这个选项将被移除。

不过，还有一个与消费者和 ZooKeeper 有关的问题需要注意。虽然不建议使用 ZooKeeper 来保存元数据，但消费者仍然可以选择是使用 ZooKeeper 还是 Kafka 来保存偏移量，还可以选择提交的时间间隔。如果消费者使用 ZooKeeper 来保存偏移量，那么每一个消费者会在每一个提交时间间隔内执行一次 ZooKeeper 写入操作。合理的偏移量提交时间间隔是 1 分钟，因为如果有消费者发生故障，那么消费者群组将在这段时间内读取到重复消息。这些提交偏移量的操作可能会给 ZooKeeper 带来很大的流量，特别是在有很多消费者的集群中。这个问题需要引起注意。如果 ZooKeeper 群组不能轻松地处理这些流量，则可能需要使用更长的提交时间间隔。不管怎样，还是建议使用新版的 Kafka 消费者，并将偏移量提交到 Kafka，消除对 ZooKeeper 的依赖。

如果可以的话，除了让多个 Kafka 集群共享一个 ZooKeeper 群组，不建议再把 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中断都可能导致 Kafka 出现不可预测的行为。如果有多个 broker 与 ZooKeeper 断开连接，那么它们就会离线，进而导致分区离线。这也会给集群控制器造成压力，因为在发生中断一段时间后，当控制器尝试关闭 broker 时，会表现出细微的异常。其他应用程序也会给 ZooKeeper 群组造成压力，它们可能会大量使用 ZooKeeper 或者执行一些不恰当的操作。所以，最好让其他应用程序使用自己的 ZooKeeper 群组。

2.8 小结

本章介绍了如何运行 Kafka，讨论了如何为 broker 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，我们将介绍基本的客户端应用程序。接下来的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 读取消息（第 4 章）。

第 3 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，总是需要一个可以往 Kafka 写入数据的生产者、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，在信用卡事务处理系统中，有一个客户端应用程序，它可能是在线商店，每当有支付行为发生时，就把事务消息发送到 Kafka 上。另一个应用程序会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会被写回 Kafka，然后再发送给发起事务的在线商店。第三个应用程序会从 Kafka 上读取事务消息和审核状态，把它们保存到数据库中，随后分析师开始审查这些结果，或许还能借此改进规则引擎。

可以用 Kafka 内置的客户端 API 来开发 Kafka 应用程序。

本章将从 Kafka 生产者的设计和组件讲起，学习如何使用 Kafka 生产者。我们将首先演示如何创建 `KafkaProducer` 对象和 `ProducerRecords` 对象、如何将记录发送给 Kafka，以及如何处理 Kafka 返回的错误响应。然后介绍用于控制生产者行为的重要配置参数。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。

第 4 章将介绍 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，Kafka 还提供了二进制连接协议。也就是说，可以通过直接向 Kafka 网络端口发送字节序列的方式从 Kafka 读取消息或向 Kafka 写入消息。有很多用 C++、Python、Go 等语言实现的 Kafka 客户端，它们都实现了 Kafka 的连接协议，使得 Kafka 的使用不仅限于 Java。这些客户端不属于 Apache Kafka 项目，但 Kafka 项目 wiki 页面上提供了一个清单，列出了所有可用的客户端。不过，与连接协议和第三方客户端相关的内容不在本章的讨论范围内。

3.1 生产者概览

一个应用程序会在很多情况下向 Kafka 写入消息：记录用户的活动（用于审计和分析）、记录指标、记录日志、记录从智能家电收集到的信息、与其他应用程序进行异步通信、缓冲即将写入数据库的数据，等等。

多样的应用场景意味着多样的需求：是否每条消息都很重要？是否允许丢失一小部分消息？是否可以接受偶尔出现重复消息？是否有严格的延迟和吞吐量需求？

之前提到的信用卡事务处理系统不允许消息丢失或重复，可接受的最高延迟为 500 毫秒，对吞吐量要求较高——我们希望每秒可以处理 100 万条消息。

另一种应用场景是保存网站的点击信息。在这个场景中，允许丢失少量消息或出现少量重复消息，延迟可以高一些，只要不影响用户体验就行。换句话说，只要保证用户点击链接后可以马上加载页面，我们并不介意消息需要在几秒之后才能到达 Kafka 服务器。吞吐量则取决于用户在网站上的活跃度。

不同的应用场景直接影响如何使用和配置生产者 API。

尽管生产者 API 使用起来很简单，但消息的发送过程还是有点儿复杂。图 3-1 展示了向 Kafka 发送消息的主要步骤。

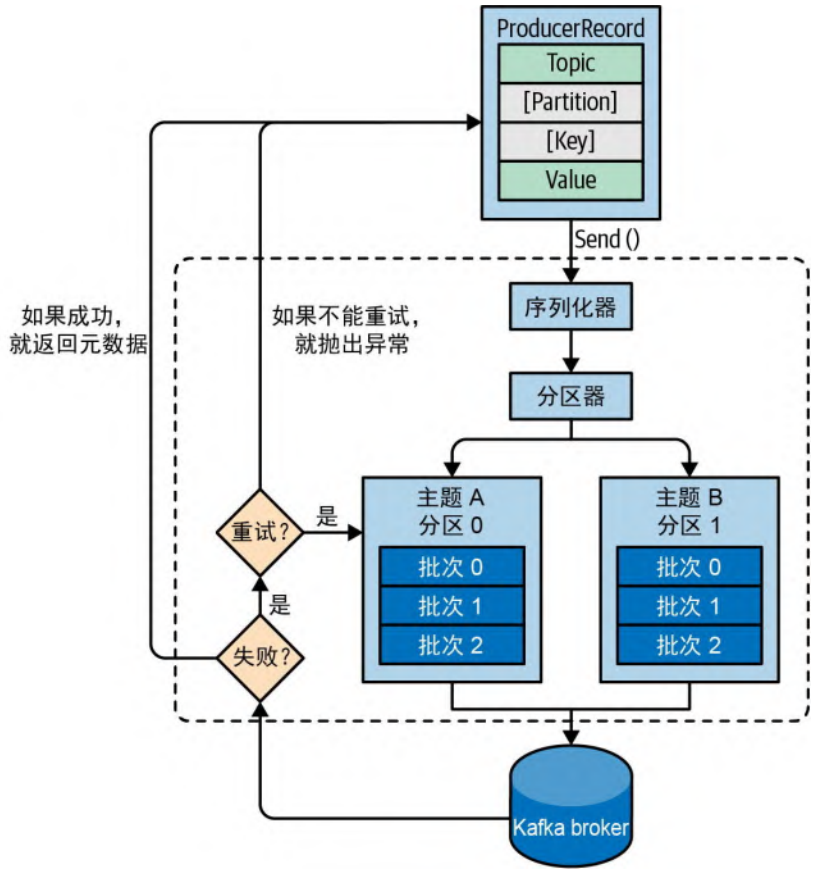

图 3-1：Kafka 生产者组件图

先从创建一个 `ProducerRecord` 对象开始，其中需要包含目标主题和要发送的内容。另外，还可以指定键、分区、时间戳或标头。在发送 `ProducerRecord` 对象时，生产者需要先把键和值对象序列化成字节数组，这样才能在网络上传输。

接下来，如果没有显式地指定分区，那么数据将被传给分区器。分区器通常会基于 `ProducerRecord` 对象的键选择一个分区。选好分区以后，生产者就知道该往哪个主题和分区发送这条消息了。紧接着，该消息会被添加到一个消息批次里，这个批次里的所有消息都将被发送给同一个主题和分区。有一个独立的线程负责把这些消息批次发送给目标 broker。

broker 在收到这些消息时会返回一个响应。如果消息写入成功，就返回一个 `RecordMetaData` 对象，其中包含了主题和分区信息，以及消息在分区中的偏移量。如果消息写入失败，则会返回一个错误。生产者在收到错误之后会尝试重新发送消息，重试几次之后如果还是失败，则会放弃重试，并返回错误信息。

3.2 创建 Kafka 生产者

要向 Kafka 写入消息，首先需要创建一个生产者对象，并设置一些属性。Kafka 生产者有 3 个必须设置的属性。

bootstrap.servers

broker 的地址。可以由多个 host:port 组成，生产者用它们来建立初始的 Kafka 集群连接。它不需要包含所有的 broker 地址，因为生产者在建立初始连接之后可以从给定的 broker 那里找到其他 broker 的信息。不过还是建议至少提供两个 broker 地址，因为一旦其中一个停机，则生产者仍然可以连接到集群。

key.serializer

一个类名，用来序列化消息的键。broker 希望接收到的消息的键和值都是字节数组。生产者可以把任意 Java 对象作为键和值发送给 broker，但它需要知道如何把这些 Java 对象转换成字节数组。key.serializer 必须被设置为一个实现了 org.apache.kafka.common.serialization.Serializer 接口的类，生产者会用这个类把键序列化成字节数组。Kafka 客户端默认提供了 ByteArraySerializer、StringSerializer 和 IntegerSerializer 等，如果你只使用常见的几种 Java 对象类型，就没有必要实现自己的序列化器。需要注意的是，必须设置 key.serializer 这个属性，尽管你可能只需要将值发送给 Kafka。如果只需要发送值，则可以将 Void 作为键的类型，然后将这个属性设置为 VoidSerializer。

value.serializer

一个类名，用来序列化消息的值。与设置 key.serializer 属性一样，需要将 value.serializer 设置成可以序列化消息值对象的类。

下面的代码片段演示了如何创建一个生产者。这里只指定了必需的属性，其他属性使用默认值。

```
Properties kafkaProps = new Properties(); ❶
kafkaProps.put("bootstrap.servers", "broker1:9092,broker2:9092");
kafkaProps.put("key.serializer",
    "org.apache.kafka.common.serialization.StringSerializer"); ❷
kafkaProps.put("value.serializer",
    "org.apache.kafka.common.serialization.StringSerializer");

producer = new KafkaProducer<String, String>(kafkaProps); ❸
```

❶ 新建一个 Properties 对象。

❷ 因为我们打算把键和值定义成字符串类型，所以使用内置的 StringSerializer。

❸ 创建一个新的生产者对象，为键和值设置类型，然后把 Properties 对象传给它。

这个接口很简单，通过配置生产者不同的属性可以控制它的行为。Kafka 文档列出了所有的配置参数，本章后面部分会介绍其中几个比较重要的参数。

实例化好生产者对象后，接下来就可以开始发送消息了。发送消息主要有以下 3 种方式。

发送并忘记

把消息发送给服务器，但并不关心它是否成功送达。大多数情况下，消息可以成功送达，因为 Kafka 是高可用的，而且生产者有自动尝试重发的机制。但是，如果发生了不可重试的错误或超时，那么消息将会丢失，应用程序将不会收到任何信息或异常。

同步发送

一般来说，生产者是异步的——我们调用 send() 方法发送消息，它会返回一个 Future 对象。可以调用 get() 方法等待 Future 完成，这样就可以在发送下一条消息之前知道当前消息是否发送成功。

异步发送

调用 send() 方法，并指定一个回调函数，当服务器返回响应时，这个函数会被触发。

在接下来的几个例子中，我们将介绍如何使用上述 3 种方式来发送消息，以及如何处理可能出现的异常情况。

本章所有的例子都使用了单线程，其实生产者对象是可以在多线程环境中发送消息的。

3.3 发送消息到 Kafka

最简单的消息发送方式如下所示。

```
ProducerRecord<String, String> record =
    new ProducerRecord<>("CustomerCountry", "Precision Products",
        "France"); ❶
try {
    producer.send(record); ❷
} catch (Exception e) {
    e.printStackTrace(); ❸
}
```

❶ 生产者会接受一个 ProducerRecord 对象作为参数，所以要先创建这样的对象。ProducerRecord 有多个构造函数，稍后会详细介绍。这里使用的构造函数需要指定目标主题名（一个字符串）和要发送的消息的键和值（在这里也是字符串）。键和值的类型必须与键序列化器和值序列化器相对应。

❷ 调用生产者的 send() 方法来发送 ProducerRecord 对象。从图 3-1 的生产者组件图中可以看到，消息会先被放进缓冲区，然后通过单独的线程发送给服务器端。send() 方法会返回一个包含 RecordMetadata 的 Future 对象。因为我们选择忽略返回值，所以不知道消息是否发送成功。如果可以接受潜在的数据丢失，那么就可以使用这种发送方式，但在生产环境中通常不会这么做。

❸ 可以忽略在发送消息时发生的错误或服务器端返回的错误，但在发送消息之前，生产者仍有可能抛出其他的异常。这些异常可能是 SerializationException（序列化消息失败）、BufferExhaustedException 或 TimeoutException（缓冲区已满），或者 InterruptException（发送线程被中断）。

3.3.1 同步发送消息

同步发送消息很简单，当 Kafka 返回错误或重试次数达到上限时，生产者可以捕获到异常。这里需要考虑性能问题。根据 Kafka 集群繁忙程度的不同，broker 可能需要 2 毫秒或更长的时间来响应请求。如果采用同步发送方式，那么发送线程在这段时间内就只能等待，什么也不做，甚至都不发送其他消息，这将导致糟糕的性能。因此，同步发送方式通常不会被用在生产环境中（但会经常被用在示例代码中）。

最简单的同步发送方式如下所示。

```
ProducerRecord<String, String> record =
    new ProducerRecord<>("CustomerCountry", "Precision Products", "France");
try {
    producer.send(record).get(); ❶
} catch (Exception e) {
    e.printStackTrace(); ❷
}
```

❶ 调用 Future.get() 方法等待 Kafka 响应。如果消息没有发送成功，那么这个方法将抛出一个异常。如果没有发生错误，那么我们将得到一个 RecordMetadata 对象，并能从中获取消息的偏移量和其他元数据。

❷ 如果在消息发送之前或发送过程中发生了错误，那么我们将捕捉到一个异常。这里只是简单地把异常信息打印了出来。

KafkaProducer 一般会出现两种错误。一种是**可重试**错误，这种错误可以通过重发消息来解决。例如，对于连接错误，只要再次建立连接就可以解决。对于“not leader for partition”（非分区首领）错误，只要重新为分区选举首领就可以解决，此时元数据也会被刷新。可以通过配置启用 KafkaProducer 的自动重试机制。如果在多次重试后仍无法解决问题，则应用程序会收到重试异常。另一种错误则无法通过重试解

决，比如“Message size too large”（消息太大）。对于这种错误，KafkaProducer 不会进行任何重试，而会立即抛出异常。

3.3.2 异步发送消息

假设一条消息在应用程序和 Kafka 集群之间往返需要 10 毫秒。如果在发送完每条消息后都需要等待响应，那么发送 100 条消息将需要 1 秒。如果只发送消息但不需要等待响应，那么发送 100 条消息所需要的时间就会少很多。大多数时候，并不需要等待响应——尽管 Kafka 会把消息的目标主题、分区信息和偏移量返回给客户端，但对客户端应用程序来说可能不是必需的。不过，当消息发送失败，需要抛出异常、记录错误日志或者把消息写入“错误消息”文件以便日后分析诊断时，就需要用到这些信息了。

为了能够在异步发送消息时处理异常情况，生产者提供了回调机制。下面是回调机制的示例。

```
private class DemoProducerCallback implements Callback { ❶
    @Override
    public void onCompletion(RecordMetadata recordMetadata, Exception e) {
        if (e != null) {
            e.printStackTrace(); ❷
        }
    }
}

ProducerRecord<String, String> record =
    new ProducerRecord<>("CustomerCountry", "Biomedical Materials", "USA"); ❸
producer.send(record, new DemoProducerCallback()); ❹
```

❶ 为了使用回调，需要一个实现了 org.apache.kafka.clients.producer.Callback 接口的类，这个接口只有一个 onCompletion 方法。

❷ 如果 Kafka 返回错误，那么 onCompletion 方法会收到一个非空（nonnull）异常。这里只是简单地把它打印了出来，但在生产环境中应该使用更好的处理方式。

❸ 发送的记录与之前一样。

❹ 在发送消息时将回调对象传进去。

回调的执行将在生产者主线程中进行，如果有两条消息被发送给同一个分区，则这可以保证它们的回调是按照发送的顺序执行的。这就要求回调的执行要快，避免生产者出现延迟或影响其他消息的发送。不建议在回调中执行阻塞操作，阻塞操作应该被放在其他线程中执行。

3.4 生产者配置

到目前为止，本书只介绍了生产者的几个配置参数，即必需的 `bootstrap.servers`URI 和序列化器。

生产者还有很多其他的可配置的参数，Kafka 文档中都有说明。它们大部分有合理的默认值，没有必要进行修改。不过有几个参数在内存使用、性能和可靠性方面对生产者影响比较大，接下来将详细介绍它们。

3.4.1 `client.id`

`client.id` 是客户端标识符，它的值可以是任意字符串，broker 用它来识别从客户端发送过来的消息。`client.id` 可以被用在日志、指标和配额中。选择一个好的客户端标识符可以让故障诊断变得更容易些，这就好比“我们看到很多来自 IP 地址 104.27.155.134 的身份验证失败了”要比“好像订单验证服务的身份验证失败了，你能不能让 Laura 看看”更容易诊断问题。

3.4.2 `acks`

`acks` 指定了生产者在多少个分区副本收到消息的情况下才会认为消息写入成功。在默认情况下，Kafka 会在首领副本收到消息后向客户端回应消息写入成功（Kafka 3.0 预计会改变这个默认行为）。这个参数对写入消息的持久性有重大影响，对于不同的场景，使用默认值可能不是最好的选择。第 7 章将深入讨论 Kafka 的可靠性保证，不过现在先来看一下这个参数可设置的 3 个值。

`acks=0`

如果 `acks=0`，则生产者不会等待任何来自 broker 的响应。也就是说，如果 broker 因为某些问题没有收到消息，那么生产者便无从得知，消息也就丢失了。不过，因为生产者不需要等待 broker 返回响应，所以它们能够以网络可支持的最大速度发送消息，从而达到很高的吞吐量。

`acks=1`

如果 `acks=1`，那么只要集群的首领副本收到消息，生产者就会收到消息成功写入的响应。如果消息无法到达首领副本（比如首领副本发生崩溃，新首领还未选举出来），那么生产者会收到一个错误响应。为了避免数据丢失，生产者会尝试重发消息。不过，在首领副本发生崩溃的情况下，如果消息还没有被复制到新的首领副本，则消息还是有可能丢失。

`acks=all`

如果 `acks=all`，那么只有当所有副本全部收到消息时，生产者才会收到消息成功写入的响应。这种模式是最安全的，它可以保证不止一个 broker 收到消息，就算有个别 broker 发生崩溃，整个集群仍然可以运行（第 6 章将讨论更多的细节）。不过，它的延迟比 `acks=1` 高，因为生产者需要等待不止一个 broker 确认收到消息。

你会发现，为 `acks` 设置的值越小，生产者发送消息的速度就越快。也就是说，我们通过牺牲可靠性来换取较低的生产者延迟。不过，端到端延迟是指从消息生成到可供消费者读取的时间，这对 3 种配置来说都是一样的。这是因为为了保持一致性，在消息被写入所有同步副本之前，Kafka 不允许消费者读取它们。因此，如果你关心的是端到端延迟，而不是生产者延迟，那么就不需要在可靠性和低延迟之间做权衡了：你可以选择最可靠的配置，但仍然可以获得相同的端到端延迟。

3.4.3 消息传递时间

有几个参数可用来控制开发人员最感兴趣的生产者行为：在调用 `send()` 方法后多长时间可以知道消息发送成功与否。这也是等待 Kafka 返回成功响应或放弃重试并承认发送失败的时间。

多年来，这些配置参数和相应的行为经历了多次变化。这里将介绍在 Kafka 2.1 中引入的最新实现。

从 Kafka 2.1 开始，我们将 `ProduceRecord` 的发送时间分成如下两个时间间隔，它们是被分开处理的。

- 异步调用 `send()` 所花费的时间。在此期间，调用 `send()` 的线程将被阻塞。
- 从异步调用 `send()` 返回到触发回调（不管是成功还是失败）的时间，也就是从 `ProduceRecord` 被放到批次中直到 Kafka 成功响应、出现不可恢复异常或发送超时的时间。

如果同步调用 `send()`，那么发送线程将持续阻塞，也就无法知道每个时间间隔是多长。这里讨论的是常见且我们建议的情况，即异步调用 `send()` 并使用回调。

图 3-2 展示了生产者的内部数据流以及不同的配置参数如何相互影响。[1]

[1]这张图片由 Sumant Tambe 为 Kafka 项目绘制，基于 ASLv2 许可。

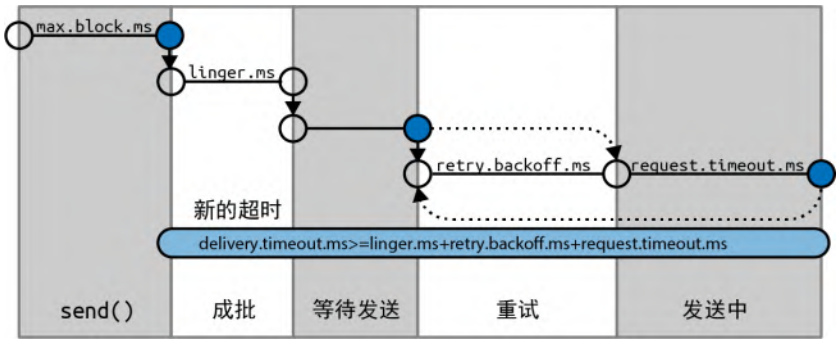

图 3-2：Kafka 生产者内部的发送时间分段序列图

下面将介绍用来控制这两个时间间隔的配置参数，以及它们之间的关系。

01. `max.block.ms`

    这个参数用于控制在调用 `send()` 或通过 `partitionsFor()` 显式地请求元数据时生产者可以发生阻塞的时间。当生产者的发送缓冲区被填满或元数据不可用时，这些方法就可能发生阻塞。当达到 `max.block.ms` 配置的时间时，就会抛出一个超时异常。

02. `delivery.timeout.ms`

    这个参数用于控制从消息准备好发送（`send()` 方法成功返回并将消息放入批次中）到 broker 响应或客户端放弃发送（包括重试）所花费的时间。如图 3-2 所示，这个时间应该大于 `linger.ms` 和 `request.timeout.ms`。如果配置的时间不满足这一点，则会抛出异常。通常，成功发送消息的速度要比 `delivery.timeout.ms` 快得多。

    如果生产者在重试时超出了 `delivery.timeout.ms`，那么将执行回调，并会将 broker 之前返回的错误传给它。如果消息批次还没有发送完毕就超出了 `delivery.timeout.ms`，那么也将执行回调，并会将超时异常传给它。

    

    可以将这个参数配置成你愿意等待的最长时间，通常是几分钟，并使用默认的重试次数（几乎无限制）。基于这样的配置，只要生产者还有时间（或者在发送成功之前），它都会持续重试。这是一种合理的重试方式。我们的重试策略通常是：“在 broker 发生崩溃的情况下，首领选举通常需要 30 秒才能完成，因此为了以防万一，我们会持续重试 120 秒。”为了避免烦琐地配置重试次数和重试时间间隔，只需将 `delivery.timeout.ms` 设置为 120。

03. request.timeout.ms

这个参数用于控制生产者在发送消息时等待服务器响应的时间。需要注意的是，这是指生产者在放弃之前等待每个请求的时间，不包括重试、发送之前所花费的时间等。如果设置的值已触及，但服务器没有响应，那么生产者将重试发送，或者执行回调，并传给它一个 TimeoutException。

04. retries 和retry.backoff.ms

当生产者收到来自服务器的错误消息时，这个错误有可能是暂时的（例如，一个分区没有首领）。在这种情况下，retries 参数可用于控制生产者在放弃发送并向客户端宣告失败之前可以重试多少次。在默认情况下，重试时间间隔是 100 毫秒，但可以通过 retry.backoff.ms 参数来控制重试时间间隔。

并不建议在当前版本的 Kafka 中使用这些参数。相反，你可以测试一下 broker 在发生崩溃之后需要多长时间恢复（也就是直到所有分区都有了首领副本），并设置合理的 delivery.timeout.ms，让重试时间大于 Kafka 集群从崩溃中恢复的时间，以免生产者过早放弃重试。

生产者并不会重试所有的错误。有些错误不是暂时的，生产者就不会进行重试（例如，“消息太大”错误）。通常，对于可重试的错误，生产者会自动进行重试，所以不需要在应用程序中处理重试逻辑。你要做的是集中精力处理不可重试的错误或者当重试次数达到上限时的情况。

如果想完全禁用重试，那么唯一可行的方法是将 retries 设置为 0。

3.4.4 linger.ms

这个参数指定了生产者在发送消息批次之前等待更多消息加入批次的时间。生产者会在批次被填满或等待时间达到 linger.ms 时把消息批次发送出去。在默认情况下，只要有可用的发送者线程，生产者都会直接把批次发送出去，就算批次中只有一条消息。把 linger.ms 设置成比 0 大的数，可以让生产者在将批次发送给服务器之前等待一会儿，以使更多的消息加入批次中。虽然这样会增加一点儿延迟，但也极大地提升了吞吐量。这是因为一次性发送的消息越多，每条消息的开销就越小，如果启用了压缩，则计算量也更少了。

3.4.5 buffer.memory

这个参数用来设置生产者要发送给服务器的消息的内存缓冲区大小。如果应用程序调用 send() 方法的速度超过生产者将消息发送给服务器的速度，那么生产者的缓冲空间可能会被耗尽，后续的 send() 方法调用会等待内存空间被释放，如果在 max.block.ms 之后还没有可用空间，就抛出异常。需要注意的是，这个异常与其他异常不一样，它是 send() 方法而不是 Future 对象抛出来的。

3.4.6 compression.type

在默认情况下，生产者发送的消息是未经压缩的。这个参数可以被设置为 snappy、gzip、lz4 或 zstd，这指定了消息被发送给 broker 之前使用哪一种压缩算法。snappy 压缩算法由谷歌发明，虽然占用较少的 CPU 时间，但能提供较好的性能和相当可观的压缩比。如果同时有性能和网络带宽方面的考虑，那么可以使用这种算法。gzip 压缩算法通常会占用较多的 CPU 时间，但提供了更高的压缩比。如果网络带宽比较有限，则可以使用这种算法。使用压缩可以降低网络传输和存储开销，而这些往往是向 Kafka 发送消息的瓶颈所在。

3.4.7 batch.size

当有多条消息被发送给同一个分区时，生产者会把它们放在同一个批次里。这个参数指定了一个批次可以使用的内存大小。需要注意的是，该参数是按照字节数而不是消息条数来计算的。当批次被填满时，批次里所有的消息都将被发送出去。但是生产者并不一定都会等到批次被填满时才将其发送出去。那些未填满的批次，甚至只包含一条消息的批次也有可能被发送出去。所以，就算把批次大小设置得很大，也不会导致延迟，只是会占用更多的内存而已。但如果把批次大小设置得太小，则会增加一些额外的开销，因为生产者需要更频繁地发送消息。

3.4.8 max.in.flight.requests.per.connection

这个参数指定了生产者在收到服务器响应之前可以发送多少个消息批次。它的值越大，占用的内存就越多，不过吞吐量也会得到提升。Apache wiki 页面上的实验数据表明，在单数据中心环境中，该参数被设置为 2 时可以获得最佳的吞吐量，但使用默认值 5 也可以获得差不多的性能。

顺序保证

Kafka 可以保证同一个分区中的消息是有序的。也就是说，如果生产者按照一定的顺序发送消息，那么 broker 会按照这个顺序把它们写入分区，消费者也会按照同样的顺序读取它们。在某些情况下，顺序是非常重要的。例如，向一个账户中存入 100 元再取出来与先从账户中取钱再存回去是截然不同的！不过，有些场景对顺序不是很敏感。

假设我们把 `retries` 设置为非零的整数，并把 `max.in.flight.requests.per.connection` 设置为比 1 大的数。如果第一个批次写入失败，第二个批次写入成功，那么 broker 会重试写入第一个批次，等到第一个批次也写入成功，两个批次的顺序就反过来了。

我们希望至少有 2 个正在处理中的请求（出于性能方面的考虑），并且可以进行多次重试（出于可靠性方面的考虑），这个时候，最好的解决方案是将 `enable.idempotence` 设置为 `true`。这样就可以在最多有 5 个正在处理中的请求的情况下保证顺序，并且可以保证重试不会引入重复消息。第 8 章将深入探讨幂等生产者。

3.4.9 max.request.size

这个参数用于控制生产者发送的请求的大小。它限制了可发送的单条最大消息的大小和单个请求的消息总量的大小。假设这个参数的值为 1 MB，那么可发送的单条最大消息就是 1 MB，或者生产者最多可以在单个请求里发送一条包含 1024 个大小为 1 KB 的消息。另外，broker 对可接收的最大消息也有限制（`message.max.bytes`），其两边的配置最好是匹配的，以免生产者发送的消息被 broker 拒绝。

3.4.10 receive.buffer.bytes 和 send.buffer.bytes

这两个参数分别指定了 TCP socket 接收和发送数据包的缓冲区大小。如果它们被设为–1，就使用操作系统默认值。如果生产者或消费者与 broker 位于不同的数据中心，则可以适当加大它们的值，因为跨数据中心网络的延迟一般都比较高，而带宽又比较低。

3.4.11 enable.idempotence

从 0.11 版本开始，Kafka 支持**精确一次性**（exactly once）语义。精确一次性语义是一个大课题，本书将专门用一章内容来讨论它。幂等生产者是它的一个简单且重要的组成部分。

假设为了最大限度地提升可靠性，你将生产者的 `acks` 设置为 `all`，并将 `delivery.timeout.ms` 设置为一个比较大的数，允许进行尽可能多的重试。这些配置可以确保每条消息被写入 Kafka 至少一次。但在某些情况下，消息有可能被写入 Kafka 不止一次。假设一个 broker 收到了生产者发送的消息，然后消息被写入本地磁盘并成功复制给了其他 broker。此时，这个 broker 还没有向生产者发送响应就发生了崩溃。而生产者将一直等待，直到达到 `request.timeout.ms`，然后进行重试。重试发送的消息将被发送给新的首领，而这个首领已经有这条消息的副本，因为之前写入的消息已经被成功复制给它了。现在，你就有了一条重复的消息。

为了避免这种情况，可以将 enable.idempotence 设置为 true。当幂等生产者被启用时，生产者将给发送的每一条消息都加上一个序列号。如果 broker 收到具有相同序列号的消息，那么它就会拒绝第二个副本，而生产者则会收到 DuplicateSequenceException，这个异常对生产者来说是无害的。

如果要启用幂等性，那么 max.in.flight.requests.per.connection 应小于或等于 5、retries 应大于 0，并且 acks 被设置为 all。如果设置了不恰当的值，则会抛出 ConfigException 异常。

3.5 序列化器

之前的例子中介绍过，创建一个生产者对象必须指定序列化器。我们已经知道如何使用默认的字符串序列化器，除此之外，Kafka 还提供了整型和字节数组等序列化器，但它们并不能覆盖大多数应用场景。毕竟，我们还需要序列化更多通用的记录类型。

接下来，我们将演示如何构建自己的序列化器，然后会将 Avro 序列化器作为推荐的备选方案。

3.5.1 自定义序列化器

如果要发送给 Kafka 的对象不是简单的字符串或整型，则既可以用通用的序列化框架（比如 Avro、Thrift 或 Protobuf）来创建消息，也可以使用自定义序列化器。强烈建议使用通用的序列化框架。不过，为了了解序列化器的工作原理，也为了说明为什么要使用序列化框架，本节将演示如何自定义一个序列化器。

假设你创建了一个简单的类来表示客户。

```
public class Customer {
    private int customerID;
    private String customerName;

    public Customer(int ID, String name) {
        this.customerID = ID;
        this.customerName = name;
    }

    public int getID() {
        return customerID;
    }

    public String getName() {
        return customerName;
    }
}
```

现在，我们要为这个类创建一个序列化器，它看起来如下所示。

```
import org.apache.kafka.common.errors.SerializationException;

import java.nio.ByteBuffer;
import java.util.Map;

public class CustomerSerializer implements Serializer<Customer> {

    @Override
    public void configure(Map configs, boolean isKey) {
        // 不需要配置什么
    }

    @Override
    /**
     Customer对象被序列化成：
     表示customerID的4字节整数
     表示customerName长度的4字节整数（如果customerName为空，则长度为0）
     表示customerName的N字节
     **/
    public byte[] serialize(String topic, Customer data) {
        try {
            byte[] serializedName;
            int stringSize;
            if (data == null)
                return null;
            else {
                if (data.getName() != null) {
                    serializedName = data.getName().getBytes("UTF-8");
                    stringSize = serializedName.length;
                } else {
```

```
                    serializedName = new byte[0];
                    stringSize = 0;
                }
            }

            ByteBuffer buffer = ByteBuffer.allocate(4 + 4 + stringSize);
            buffer.putInt(data.getID());
            buffer.putInt(stringSize);
            buffer.put(serializedName);

            return buffer.array();
        } catch (Exception e) {
            throw new SerializationException(
                "Error when serializing Customer to byte[] " + e);
        }
    }

    @Override
    public void close() {
        // 不需要关闭任何东西
    }
}
```

可以用 `CustomerSerializer` 把消息定义成 `ProducerRecord<String, Customer>`，并发送 `Customer` 数据，把 `Customer` 对象传给生产者。这个例子很简单，但代码看起来太“脆弱”了。如果有多种类型的消费者（比如把 `customerID` 字段变成长整型，或者给 `Customer` 添加 `startDate` 字段），则会出现新旧消息的兼容性问题。在不同版本的序列化器和反序列化器之间调试兼容性问题着实是个挑战——需要比较原始字节数组。更糟糕的是，如果同一家公司的不同团队都需要向 Kafka 写入 `Customer` 数据，那么他们需要使用相同的序列化器，并且需要同时修改代码。

基于以上几点原因，建议使用已有的序列化器和反序列化器，比如 JSON、Avro、Thrift 或 Protobuf。接下来将介绍 Avro，然后演示如何序列化 Avro 记录并将其发送给 Kafka。

3.5.2 使用 Avro 序列化数据

Apache Avro（以下简称 Avro）是一种与编程语言无关的序列化格式。Doug Cutting 创建了这个项目，目的是为大多数人提供一种共享数据文件的方式。

我们使用与语言无关的模式来定义 Avro 数据，并使用 JSON 来描述模式。数据一般会被序列化成二进制文件，不过也可以被序列化成 JSON。Avro 在读写文件时需要用到模式，模式通常被内嵌在数据文件中。

Avro 有一个很有意思的特性，当发送消息的应用程序使用了新模式时，读取消息的应用程序可以继续处理消息而无须做任何修改，这个特性使它特别适合用在像 Kafka 这样的消息系统中。

假设最初的模式是下面这样的。

```
{"namespace": "customerManagement.avro",
 "type": "record",
 "name": "Customer",
 "fields": [
     {"name": "id", "type": "int"},
     {"name": "name",  "type": "string"},
     {"name": "faxNumber", "type": ["null", "string"], "default": "null"} ❶
 ]
}
```

❶ `id` 字段和 `name` 字段是必需的，`faxNumber` 是可选的且默认值为 `null`。

假设我们已经使用这个模式几个月时间，并用它生成了几太字节的数据，现在决定对其做一些修改。因为在 21 世纪我们不再需要 `faxNumber` 这个字段，所以要用 `email` 字段来替代它。

新的模式如下所示。

```
{"namespace": "customerManagement.avro",
 "type": "record",
 "name": "Customer",
 "fields": [
     {"name": "id", "type": "int"},
     {"name": "name",  "type": "string"},
     {"name": "email", "type": ["null", "string"], "default": "null"}
 ]
}
```

更新到新模式后，旧记录仍然包含 `faxNumber` 字段，而新记录则包含 `email` 字段。在很多公司里，应用程序升级是一个缓慢的过程，通常需要几个月时间。所以，需要考虑如何让还在使用 `faxNumber` 字段的旧应用程序和已经使用 `email` 字段的新应用程序正常处理 Kafka 中的所有消息。

读取消息的应用程序会调用类似 `getName()`、`getId()` 和 `getFaxNumber()` 这样的方法。如果读取到使用新模式构建的消息，那么 `getName()` 方法和 `getId()` 方法仍然能够正常返回，但 `getFaxNumber()` 方法将返回 `null`，因为消息里不包含传真号码。

现在，假设升级了应用程序，用 `getEmail()` 方法替代了 `getFaxNumber()` 方法。如果读取到一个使用旧模式构建的消息，那么 `getEmail()` 方法将返回 `null`，因为旧消息不包含邮件地址。

现在可以看出 Avro 的好处了：即使在未更新所有读取消息的应用程序的情况下修改了模式，也不会遇到异常或阻断性错误，也不需要修改已有的数据。

不过有两个地方需要注意。

- 用于写入数据和读取数据的模式必须是相互兼容的。Avro 的文档提到了一些兼容性原则。
- 反序列化器需要获取写入数据时使用的模式，即使它可能与读取数据时使用的模式不一样。Avro 数据文件中就包含了用于写入数据的模式，不过在 Kafka 中有一种更好的处理方式，下一节会介绍。

3.5.3 在 Kafka 中使用 Avro 记录

Avro 的数据文件中包含了整个模式，这样的开销是可以接受的，但如果在每条记录中都嵌入模式，则会成倍增加记录的大小。不管怎样，在读取 Avro 记录时仍然需要用到整个模式，所以在读取记录之前需要先找到模式。为此，我们遵循常见的架构模式，使用了**模式注册表**。模式注册表并不是 Kafka 项目的一部分，现在已经有一些开源的模式注册表实现。我们的示例中使用的是 Confluent Schema Registry。这个注册表的代码可以在 GitHub 网站上找到，你也可以把它作为 Confluent 平台的一部分进行安装。如果决定使用这个注册表，那么建议你参考一下它的文档。

我们把所有在写入数据时需要用到的模式都保存在注册表中，然后在记录里引用模式的标识符。读取消息的应用程序使用标识符从注册表中拉取模式来反序列化记录。模式的保存和拉取分别由序列化器和反序列化器来完成。Avro 序列化器的使用方法与其他序列化器一样，图 3-3 演示了这个过程。

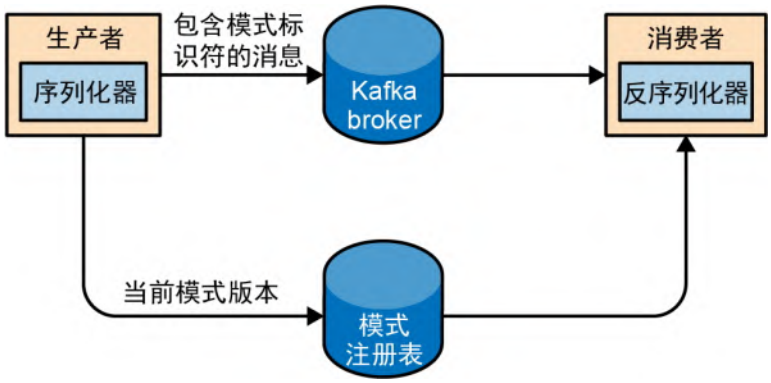

图 3-3：Avro 记录的序列化和反序列化流程图

下面的例子演示了如何把生成的 Avro 对象发送给 Kafka（关于如何从 Avro 模式生成对象请参考 Avro 文档）。

```
Properties props = new Properties();

props.put("bootstrap.servers", "localhost:9092");
props.put("key.serializer",
    "io.confluent.kafka.serializers.KafkaAvroSerializer");
props.put("value.serializer",
    "io.confluent.kafka.serializers.KafkaAvroSerializer"); ❶
props.put("schema.registry.url", schemaUrl); ❷

String topic = "customerContacts";

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

// 不断生成新事件，直到有人按下Ctrl-C组合键
while (true) {
    Customer customer = CustomerGenerator.getNext(); ❹
    System.out.println("Generated customer " +
        customer.toString());
    ProducerRecord<String, Customer> record =
        new ProducerRecord<>(topic, customer.getName(), customer); ❺
    producer.send(record); ❻
}
```

❶ 使用 KafkaAvroSerializer 来序列化对象。需要注意的是，KafkaAvroSerializer 也可以处理原始类型，这也是为什么稍后可以用 String 作为记录的键并用 Customer 对象作为记录的值。

❷ schema.registry.url 是生产者要传给序列化器的参数，其指向模式的存储位置。

❸ Customer 是生成的对象，也就是记录的值。

❹ Customer 类不是一个普通的 Java 类（POJO），而是基于模式生成的 Avro 对象。Avro 序列化器只能序列化 Avro 对象，不能序列化 POJO。可以用 avro-tools.jar 或 Avro 的 Maven 插件来生成 Avro 类，它们都是 Apache Avro 项目的一部分。关于如何生成 Avro 类，请参考 Apache Avro 入门指南。

❺ 实例化一个 ProducerRecord 对象，指定值的类型为 Customer，并传给它一个 Customer 对象。

❻ 把 Customer 对象作为记录发送出去，KafkaAvroSerializer 会处理剩下的事情。

也可以使用通用的 Avro 对象，就像使用 map 那样，这与基于模式生成的带有 getter 方法和 setter 方法的 Avro 对象不同。

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("key.serializer",
    "io.confluent.kafka.serializers.KafkaAvroSerializer"); ❶
props.put("value.serializer",
    "io.confluent.kafka.serializers.KafkaAvroSerializer");
props.put("schema.registry.url", url); ❷

String schemaString =
    "{\"namespace\": \"customerManagement.avro\",
     "\"type\": \"record\", " + ❸
     "\"name\": \"Customer\"," +
     "\"fields\": [" +
      "{\"name\": \"id\", \"type\": \"int\"}," +
      "{\"name\": \"name\", \"type\": \"string\"}," +
      "{\"name\": \"email\", \"type\": " + "[\"null\",\"string\"], " +
       "\"default\":\"null\" }" +
    "]}";
Producer<String, GenericRecord> producer =
    new KafkaProducer<String, GenericRecord>(props); ❹
```

```
Schema.Parser parser = new Schema.Parser();
Schema schema = parser.parse(schemaString);

for (int nCustomers = 0; nCustomers < customers; nCustomers++) {
    String name = "exampleCustomer" + nCustomers;
    String email = "example " + nCustomers + "@example.com"

    GenericRecord customer = new GenericData.Record(schema); ❺
    customer.put("id", nCustomers);
    customer.put("name", name);
    customer.put("email", email);

    ProducerRecord<String, GenericRecord> data =
        new ProducerRecord<>("customerContacts", name, customer);
    producer.send(data);
}
```

❶ 仍然使用 `KafkaAvroSerializer`。

❷ 提供同样的模式注册表 URI。

❸ 也需要提供 Avro 模式，因为没有使用 Avro 生成的对象也就没有模式。

❹ 对象类型是 `GenericRecord`，我们用模式和要写入的数据来初始化它。

❺ `ProducerRecord` 的值就是包含了模式和要发送的数据的一个 `GenericRecord` 对象。序列化器知道如何从记录里获取模式、把它保存到注册表中，以及用它序列化对象。

3.6 分区

在上面的例子中，ProducerRecord 对象包含了主题名称、记录的键和值。Kafka 消息就是一个个的键–值对，ProducerRecord 对象可以只包含主题名称和值，键默认情况下是 null。不过，大多数应用程序还是会用键来发送消息。键有两种用途：一是作为消息的附加信息与消息保存在一起，二是用来确定消息应该被写入主题的哪个分区（键在压缩主题中也扮演了重要角色，第 6 章将讨论这方面的内容）。具有相同键的消息将被写入同一个分区。如果一个进程只从主题的某些分区读取数据（第 4 章将介绍更多的细节），那么具有相同键的所有记录都会被这个进程读取。要创建一个包含键和值的记录，只需像下面这样创建一个 ProducerRecord 即可。

```
ProducerRecord<String, String> record =
    new ProducerRecord<>("CustomerCountry", "Laboratory Equipment", "USA");
```

如果要创建键为 null 的消息，那么不指定键就可以了。

```
ProducerRecord<String, String> record =
    new ProducerRecord<>("CustomerCountry", "USA"); ❶
```

❶ 这里的键将被设置为 null。

如果键为 null，并且使用了默认的分区器，那么记录将被随机发送给主题的分区。分区器使用轮询调度（round-robin）算法将消息均衡地分布到各个分区中。从 Kafka 2.4 开始，在处理键为 null 的记录时，默认分区器使用的轮询调度算法具备了黏性。也就是说，在切换到下一个分区之前，它会将同一个批次的消息全部写入当前分区。这样就可以使用更少的请求发送相同数量的消息，既降低了延迟，又减少了 broker 占用 CPU 的时间。

如果键不为空且使用了默认的分区器，那么 Kafka 会对键进行哈希（使用 Kafka 自己的哈希算法，即使升级 Java 版本，哈希值也不会发生变化），然后根据哈希值把消息映射到特定的分区。这里的关键在于同一个键总是被映射到同一个分区，所以在进行映射时，会用到主题所有的分区，而不只是可用的分区。这也意味着，如果在写入数据时目标分区不可用，那么就会出错。不过这种情况很少发生，第 7 章在讨论 Kafka 数据复制和可用性时将介绍这方面的更多内容。

除了默认的分区器，Kafka 客户端还提供了 RoundRobinPartitioner 和 UniformStickyPartitioner。在消息包含键的情况下，可以用它们来实现随机分区分配和黏性随机分区分配。对某些应用程序（例如，ETL 应用程序会将数据从 Kafka 加载到关系数据库中，并使用 Kafka 记录的键作为数据库的主键）来说，键很重要，但如果负载出现了倾斜，那么其中某些键就会对应较大的负载。这个时候，可以用 UniformStickyPartitioner 将负载均衡地分布到所有分区。

如果使用了默认的分区器，那么只有在不改变主题分区数量的情况下键与分区之间的映射才能保持一致。例如，只要分区数量保持不变，就可以保证用户 045189 的记录总是被写到分区 34。这样就可以在从分区读取数据时做各种优化。但是，一旦主题增加了新分区，这个就无法保证了——旧数据仍然留在分区 34，但新记录可能被写到了其他分区。如果要使用键来映射分区，那么最好在创建主题时就把分区规划好（Confluent 博客提供了一些对于如何选择分区数量的建议），而且永远不要增加新分区。

自定义分区策略

我们已经讨论了默认分区器的特点，它也是最为常用的分区器。除了哈希分区，有时也需要使用不一样的分区策略。假设你是 B2B 供应商，你有一个大客户，它是手持设备 Banana 的制造商。你的日常交易中有 10% 以上的交易与这个客户有关。如果使用默认的哈希分区算法，那么与 Banana 相关的记录就会和其他客户的记录一起被分配给相同的分区，导致这个分区比其他分区大很多。服务器可能会出现存储空间不足、请求处理缓慢等问题。因此，需要给 Banana 分配单独的分区，然后使用哈希分区算法将其他记录分配给其他分区。

下面是一个自定义分区器的例子。

```
import org.apache.kafka.clients.producer.Partitioner;
import org.apache.kafka.common.Cluster;
import org.apache.kafka.common.PartitionInfo;
import org.apache.kafka.common.record.InvalidRecordException;
import org.apache.kafka.common.utils.Utils;

public class BananaPartitioner implements Partitioner {

    public void configure(Map<String, ?> configs) {} ❶

    public int partition(String topic, Object key, byte[] keyBytes,
                         Object value, byte[] valueBytes,
                         Cluster cluster) {
        List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);
        int numPartitions = partitions.size();

        if ((keyBytes == null) || (!(key instanceOf String))) ❷
            throw new InvalidRecordException("We expect all messages "+
                "to have customer name as key");

        if (((String) key).equals("Banana"))
            return numPartitions - 1; // Banana的记录总是被分配到最后一个分区

        // 其他记录被哈希到其他分区
        return Math.abs(Utils.murmur2(keyBytes)) % (numPartitions - 1);
    }

    public void close() {}
}
```

❶ Partitioner 接口包含了 configure、partition 和 close 这 3 个方法。这里只实现 partition 方法。不能在 partition 方法里硬编码客户的名字，而应该通过 configure 方法传进来。

❷ 只接受字符串作为键，如果不是字符串，就抛出异常。

3.7 标头

除了键和值，记录还可以包含标头。可以在不改变记录键–值对的情况下向标头中添加一些有关记录的元数据。标头指明了记录数据的来源，可以在不解析消息体的情况下根据标头信息来路由或跟踪消息（消息有可能被加密，而路由器没有访问加密数据的权限）。

标头由一系列有序的键–值对组成。键是字符串，值可以是任意被序列化的对象，就像消息里的值一样。

下面这个简单的示例演示了如何给 `ProduceRecord` 添加标头。

```
ProducerRecord<String, String> record =
    new ProducerRecord<>("CustomerCountry", "Precision Products", "France");

record.headers().add("privacy-level","YOLO".getBytes(StandardCharsets.UTF_8));
```

3.8 拦截器

有时候，你希望在不修改代码的情况下改变 Kafka 客户端的行为。这或许是因为你想给公司所有的应用程序都加上同样的行为，或许是因为无法访问应用程序的原始代码。

Kafka 的 `ProducerInterceptor` 拦截器包含两个关键方法。

`ProducerRecord<K, V> onSend(ProducerRecord<K, V> record)`

这个方法会在记录被发送给 Kafka 之前，甚至是在记录被序列化之前调用。如果覆盖了这个方法，那么你就可以捕获到有关记录的信息，甚至可以修改它。只需确保这个方法返回一个有效的 `ProducerRecord` 对象。这个方法返回的记录将被序列化并发送给 Kafka。

`void onAcknowledgement(RecordMetadata metadata, Exception exception)`

这个方法会在收到 Kafka 的确认响应时调用。如果覆盖了这个方法，则不可以修改 Kafka 返回的响应，但可以捕获到有关响应的信息。

常见的生产者拦截器应用场景包括：捕获监控和跟踪信息、为消息添加标头，以及敏感信息脱敏。

下面是一个非常简单的生产者拦截器示例，它只是简单地统计在特定时间窗口内发送和接收的消息数量。

```
public class CountingProducerInterceptor implements ProducerInterceptor {

  ScheduledExecutorService executorService =
          Executors.newSingleThreadScheduledExecutor();
  static AtomicLong numSent = new AtomicLong(0);
  static AtomicLong numAcked = new AtomicLong(0);

  public void configure(Map<String, ?> map) {
      Long windowSize = Long.valueOf(
              (String) map.get("counting.interceptor.window.size.ms")); ❶
      executorService.scheduleAtFixedRate(CountingProducerInterceptor::run,
              windowSize, windowSize, TimeUnit.MILLISECONDS);
  }

  public ProducerRecord onSend(ProducerRecord producerRecord) {
      numSent.incrementAndGet();
      return producerRecord; ❷
  }

  public void onAcknowledgement(RecordMetadata recordMetadata, Exception e) {
      numAcked.incrementAndGet(); ❸
  }

  public void close() {
      executorService.shutdownNow(); ❹
  }

  public static void run() {
      System.out.println(numSent.getAndSet(0));
      System.out.println(numAcked.getAndSet(0));
  }
}
```

❶ `ProducerInterceptor` 继承了 `Configurable` 接口。你可以覆盖 `configure` 方法，并在调用其他方法之前设置好需要的东西。这个方法的参数包含了生产者所有的配置属性，你可以随意访问它们。在这个示例中，我们添加了一个自己的配置属性。

❷ 每当发送了一条记录，就增加记录的计数，并原封不动地将记录返回。

❸ 每当收到 Kafka 的确认响应，就增加确认的计数，不返回任何东西。

❹ 这个方法会在生产者被关闭时调用，我们可以借助这个机会清理拦截器的状态。在这个示例中，我们关闭了之前创建的线程。如果你打开了文件句柄、与远程数据库建立了连接，或者做了其他类似的操作，那么可以在这里关闭所有的资源，以免发生资源泄漏。

如前所述，可以在不修改客户端代码的情况下使用生产者拦截器。如果要在 `kafka-console-producer`（Kafka 附带的一个示例应用程序）中使用上述的拦截器，那么可以遵循以下 3 个步骤。

01. 将 jar 包加入类路径中。

```
export CLASSPATH=$CLASSPATH:~./target/CountProducerInterceptor-1.0-
SNAPSHOT.jar
```

02. 创建一个包含这些信息的配置文件。

```
interceptor.classes=com.shapira.examples.interceptors.CountProducer
Interceptor counting.interceptor.window.size.ms=10000
```

03. 正常运行应用程序，但要指定在上一步创建的配置文件。

```
bin/kafka-console-producer.sh --broker-list localhost:9092 --topic
interceptor-test --producer.config producer.config
```

3.9 配额和节流

Kafka 可以限制生产消息和消费消息的速率，这是通过配额机制来实现的。Kafka 提供了 3 种配额类型：生产、消费和请求。生产配额和消费配额限制了客户端发送和接收数据的速率（以字节 / 秒为单位）。请求配额限制了 broker 用于处理客户端请求的时间百分比。

可以为所有客户端（使用默认配额）、特定客户端、特定用户，或特定客户端及特定用户设置配额。特定用户的配额只在集群配置了安全特性并对客户端进行了身份验证后才有效。

默认的生产配额和消费配额是 broker 配置文件的一部分。如果要限制每个生产者平均发送的消息不超过 2 MBps，那么可以在 broker 配置文件中加入 `quota.producer.default=2M`。

也可以覆盖 broker 配置文件中的默认配额来为某些客户端配置特定的配额，尽管不建议这么做。如果允许 clientA 的配额达到 4 MBps、clientB 的配额达到 10 MBps，则可以这样配置：`quota.producer.override="clientA:4M,clientB:10M"`。

在配置文件中指定的配额都是静态的，如果要修改它们，则需要重启所有的 broker。因为随时都可能有新客户端加入，所以这种配置方式不是很方便。因此，特定客户端的配额通常采用动态配置。可以用 `kafka-config.sh` 或 AdminClient API 来动态设置配额。

下面来看一些例子。

```
bin/kafka-configs --bootstrap-server localhost:9092 --alter --add-config 'producer_
byte_rate=1024' --entity-name clientC --entity-type clients ❶

bin/kafka-configs --bootstrap-server localhost:9092 --alter --add-config 'producer_
byte_rate=1024,consumer_byte_rate=2048' --entity-name user1 --entity-type users ❷

bin/kafka-configs --bootstrap-server localhost:9092 --alter --add-config 'consumer_
byte_rate=2048' --entity-type users ❸
```

❶ 限制 clientC（通过客户端 ID 来识别）每秒平均发送不超过 1024 字节。

❷ 限制 user1（通过已认证的账号来识别）每秒平均发送不超过 1024 字节以及每秒平均消费不超过 2048 字节。

❸ 限制所有用户每秒平均消费不超过 2048 字节，有覆盖配置的特定用户除外。这也是动态修改默认配置的一种方式。

当客户端触及配额时，broker 会开始限制客户端请求，以防止超出配额。这意味着 broker 将延迟对客户端请求做出响应。对大多数客户端来说，这样会自动降低请求速率（因为执行中的请求数量也是有限制的），并将客户端流量降到配额允许的范围内。但是，被节流的客户端还是有可能向服务器端发送额外的请求，为了不受影响，broker 将在一段时间内暂停与客户端之间的通信通道，以满足配额要求。

节流行为通过 `produce-throttle-time-avg`、`produce-throttle-time-max`、`fetch-throttle-time-avg` 和 `fetch-throttle-time-max` 暴露给客户端，这几个参数是生产请求和消费请求因节流而被延迟的平均时间和最长时间。需要注意的是，这些时间对应的是生产消息和消费消息的吞吐量配额、请求时间配额，或两者兼而有之。其他类型的客户端请求只会因触及请求时间配额而被节流，这些节流行为也会通过其他类似的指标暴露出来。

如果你异步调用 `Producer.send()`，并且发送速率超过了 broker 能够接受的速率（无论是由于配额的限制还是由于处理能力不足），那么消息将会被放入客户端的内存队列。如果发送速率一直快于接收速率，那么客户端最终将耗尽内存缓冲区，并阻塞后续的 `Producer.send()` 调用。如果超时延迟不足以让 broker 赶上生产者，使其清理掉一些缓冲区空间，那么 `Producer.send()`

最终将抛出 TimeoutException 异常。或者，批次里的记录因为等待时间超过了 delivery.timeout.ms 而过期，导致执行 send() 的回调，并抛出 TimeoutException 异常。因此，要做好计划和监控，确保 broker 的处理能力总是与生产者发送数据的速率相匹配。

3.10 小结

我们以一个生产者示例（使用 10 行代码将消息发送给 Kafka）开始了本章的内容。然后在代码中加入错误处理逻辑，并介绍了同步和异步两种发送方式。接下来探究了生产者的一些重要配置参数以及它们对生产者行为的影响。还讨论了用于控制消息格式的序列化器，并深入探讨了 Avro，这是一种在 Kafka 中得到广泛应用的序列化方式。最后描述了 Kafka 的分区机制，并给出了一个自定义分区的例子。

现在，我们已经知道如何向 Kafka 写入消息，第 4 章将介绍如何从 Kafka 读取消息。

第 4 章　Kafka 消费者——从 Kafka 读取数据

应用程序使用 `KafkaConsumer` 向 Kafka 订阅主题，并从订阅的主题中接收消息。不同于从其他消息系统读取数据，从 Kafka 读取数据涉及一些独特的概念和想法。如果不先理解这些概念，则难以理解如何使用消费者 API。所以，本章将先解释这些重要的概念，然后再举几个例子，演示如何使用消费者 API 实现不同的应用程序。

4.1 Kafka 消费者相关概念

要想知道如何从 Kafka 读取消息，需要先了解消费者和消费者群组的概念。接下来将解释这些概念。

4.1.1 消费者和消费者群组

假设我们有一个应用程序，它从一个 Kafka 主题读取消息，在对这些消息做一些验证后再把它们保存起来。应用程序需要创建一个消费者对象，订阅主题并开始接收消息、验证消息和保存结果。但过了一阵子，生产者向主题写入消息的速度超过了应用程序验证数据的速度，这时候该怎么办呢？如果只使用单个消费者来处理消息，那么应用程序会远远跟不上消息生成的速度。显然，此时很有必要对消费者进行横向伸缩。就像多个生产者可以向相同的主题写入消息一样，也可以让多个消费者从同一个主题读取消息。

Kafka 消费者从属于**消费者群组**。一个群组里的消费者订阅的是同一个主题，每个消费者负责读取这个主题的部分消息。

假设主题 T1 有 4 个分区，我们创建了消费者 C1，它是群组 G1 中唯一的消费者，用于订阅主题 T1。消费者 C1 将收到主题 T1 全部 4 个分区的消息，如图 4-1 所示。

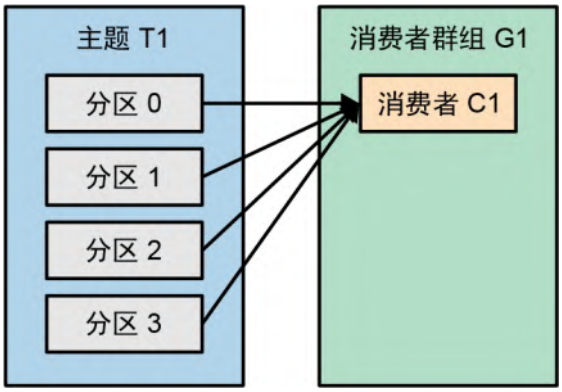

图 4-1：只包含一个消费者的群组接收 4 个分区的消息

如果在群组 G1 里新增一个消费者 C2，那么每个消费者将接收到两个分区的消息。假设消费者 C1 接收分区 0 和分区 2 的消息，消费者 C2 接收分区 1 和分区 3 的消息，如图 4-2 所示。

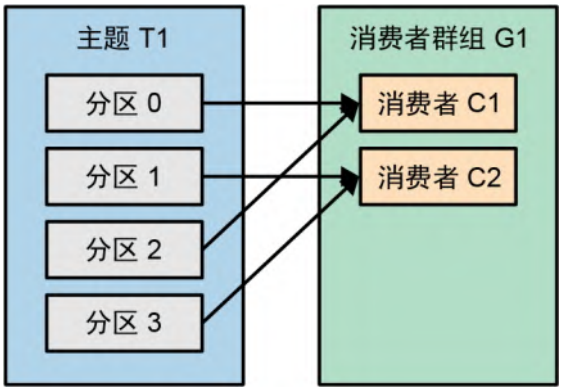

图 4-2：包含两个消费者的群组接收 4 个分区的消息

如果群组 G1 有 4 个消费者，那么每个消费者将可以分配到一个分区，如图 4-3 所示。

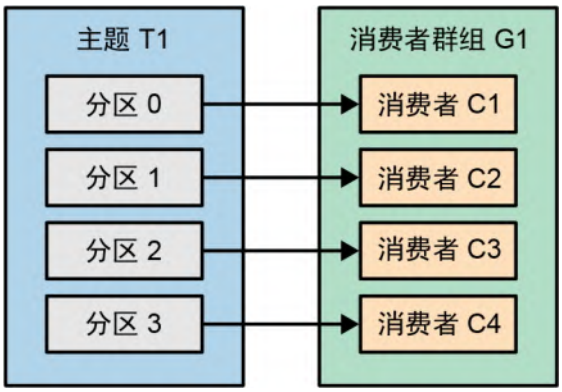

图 4-3：包含 4 个消费者的群组，每个消费者分配到一个分区

如果向群组里添加更多的消费者，以致超过了主题的分区数量，那么就会有一部分消费者处于空闲状态，不会接收到任何消息，如图 4-4 所示。

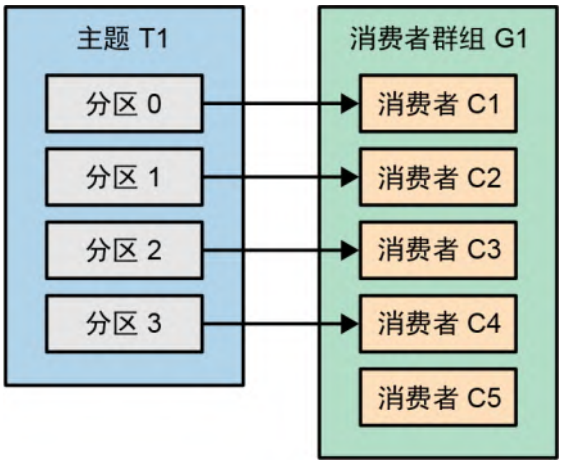

图 4-4：消费者数量超过分区数量，有消费者空闲

向群组里添加消费者是横向扩展数据处理能力的主要方式。Kafka 消费者经常需要执行一些高延迟的操作，比如把数据写到数据库或用数据做一些比较耗时的计算。在这些情况下，单个消费者无法跟上数据生成的速度，因此可以增加更多的消费者来分担负载，让每个消费者只处理部分分区的消息，这是横向扩展消费者的主要方式。于是，我们可以为主题创建大量的分区，当负载急剧增长时，可以加入更多的消费者。不过需要注意的是，不要让消费者的数量超过主题分区的数量，因为多余的消费者只会被闲置。第 2 章提供过一些如何为主题选择合适分区数的建议。

除了通过增加消费者数量来横向伸缩单个应用程序，我们还经常遇到多个应用程序从同一个主题读取数据的情况。实际上，Kafka 的一个主要设计目标是让 Kafka 主题里的数据能够满足企业各种应用场景的需求。在这些应用场景中，我们希望每一个应用程序都能获取到所有的消息，而不只是其中的一部分。只要保证每个应用程序都有自己的消费者群组就可以让它们获取到所有的消息。不同于传统的消息系统，横向伸缩消费者和消费者群组并不会导致 Kafka 性能下降。

在之前的例子中，如果新增一个只包含一个消费者的群组 G2，那么这个消费者将接收到主题 T1 的所有消息，与群组 G1 之间互不影响。群组 G2 可以增加更多的消费者，每个消费者会读取若干个分区，就像群组 G1 里的消费者那样。作为整体来说，群组 G2 还是会收到所有消息，不管有没有其他群组存在，如图 4-5 所示。

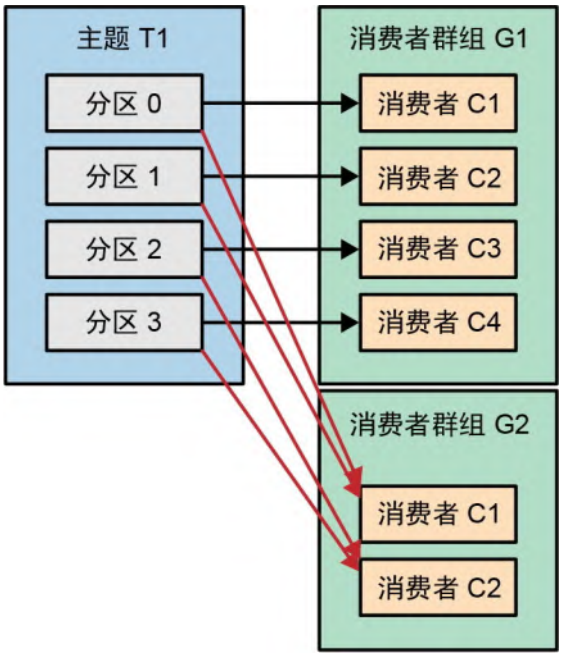

图 4-5：新增一个消费者群组，每个群组都能收到所有消息

总的来说，就是为每一个需要获取主题全部消息的应用程序创建一个消费者群组，然后向群组里添加更多的消费者来扩展读取能力和处理能力，让群组里的每个消费者只处理一部分消息。

4.1.2 消费者群组和分区再均衡

如上一节所述，消费者群组里的消费者共享主题分区的所有权。当一个新消费者加入群组时，它将开始读取一部分原本由其他消费者读取的消息。当一个消费者被关闭或发生崩溃时，它将离开群组，原本由它读取的分区将由群组里的其他消费者读取。主题发生变化（比如管理员添加了新分区）会导致分区重分配。

分区的所有权从一个消费者转移到另一个消费者的行为称为**再均衡**。再均衡非常重要，它为消费者群组带来了高可用性和伸缩性（你可以放心地添加或移除消费者）。不过，在正常情况下，我们并不希望发生再均衡。

根据消费者群组所使用的分区分配策略的不同，再均衡可以分为两种类型。[1]

[1]其中所列图片，即图 4-6 和图 4-7，来自 Sophie Blee-Goldman 在 2020 年 5 月撰写的博文“From Eager to Smarter in Apache Kafka Consumer Rebalances”。

主动再均衡

在进行主动再均衡期间，所有消费者都会停止读取消息，放弃分区所有权，重新加入消费者群组，并获得重新分配到的分区。这样会导致整个消费者群组在一个很短的时间窗口内不可用。这个时间窗口的长短取决于消费者群组的大小和几个配置参数。从图 4-6 可以看到，主动再均衡包含两个不同的阶段：第一个阶段，所有消费者都放弃分区所有权；第二个阶段，消费者重新加入群组，获得重新分配到的分区，并继续读取消息。

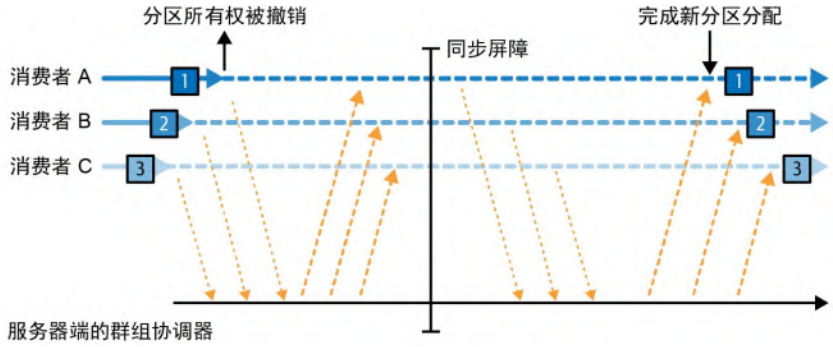

图 4-6：主动再均衡会撤销分区所有权，暂停消费消息，并重新分配分区

协作再均衡

协作再均衡（也称为**增量再均衡**）通常是指将一个消费者的部分分区重新分配给另一个消费者，其他消费者则继续读取没有被重新分配的分区。这种再均衡包含两个或多个阶段。

在第一个阶段，消费者群组首领会通知所有消费者，它们将失去部分分区的所有权，然后消费者会停止读取这些分区，并放弃对它们的所有权。在第二个阶段，消费者群组首领会将这些没有所有权的分区分配给其他消费者。虽然这种增量再均衡可能需要进行几次迭代，直到达到稳定状态，但它避免了主动再均衡中出现的“停止世界”停顿。这对大型消费者群组来说尤为重要，因为它们的再均衡可能需要很长时间。图 4-7 演示了协作再均衡的增量式执行过程，涉及部分消费者和分区。

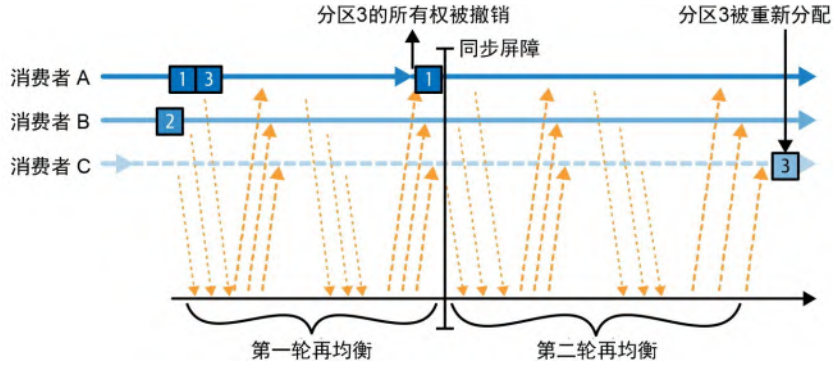

图 4-7：协作再均衡只暂停读取被重新分配的分区

消费者会向被指定为**群组协调器**的 broker（不同消费者群组的协调器可能不同）发送**心跳**，以此来保持群组成员关系和对分区的所有权关系。心跳是由消费者的一个后台线程发送的，只要消费者能够以正常的时间间隔发送心跳，它就会被认为还“活着”。

如果消费者在足够长的一段时间内没有发送心跳，那么它的会话就将超时，群组协调器会认为它已经“死亡”，进而触发再均衡。如果一个消费者发生崩溃并停止读取消息，那么群组协调器就会在几秒内收不到心跳，它会认为消费者已经“死亡”，进而触发再均衡。在这几秒时间里，“死掉”的消费者不会读取分区里的消息。在关闭消费者后，协调器会立即触发一次再均衡，尽量降低处理延迟。本章的后续部分将介绍一些用于控制心跳发送频率、会话过期时间和调节消费者行为的配置参数。

分配分区是怎样的一个过程？

当一个消费者想要加入消费者群组时，它会向群组协调器发送 `JoinGroup` 请求。第一个加入群组的消费者将成为群组首领。首领从群组协调器那里获取群组的成员列表（列表中包含了所有最近发送过心跳的消费者，它们被认为还“活着”），并负责为每一个消费者分配分区。它使用实现了 `PartitionAssignor` 接口的类来决定哪些分区应该被分配给哪个消费者。

Kafka 内置了一些分区分配策略，后文将深入介绍它们。分区分配完毕之后，首领会把分区分配信息发送给群组协调器，群组协调器再把这些信息发送给所有的消费者。每个消费者只能看到自己的分配信息，只有首领会持有所有消费者及其分区所有权的信息。每次再均衡都会经历这个过程。

4.1.3 群组固定成员

在默认情况下，消费者的群组成员身份标识是临时的。当一个消费者离开群组时，分配给它的分区所有权将被撤销；当该消费者重新加入时，将通过再均衡协议为其分配一个新的成员 ID 和新分区。

可以给消费者分配一个唯一的 `group.instance.id`，让它成为群组的**固定**成员。通常，当消费者第一次以固定成员身份加入群组时，群组协调器会按照分区分配策略给它分配一部分分区。当这个消费者被关闭时，它不会自动离开群组——它仍然是群组的成员，直到会话超时。当这个消费者重新加入群组时，它会继续持有之前的身份，并分配到之前所持有的分区。群组协调器缓存了每个成员的分区分配信息，只需要将缓存中的信息发送给重新加入的固定成员，不需要进行再均衡。

如果两个消费者使用相同的 `group.instance.id` 加入同一个群组，则第二个消费者会收到错误，告诉它具有相同 ID 的消费者已存在。

如果应用程序需要维护与消费者分区所有权相关的本地状态或缓存，那么群组固定成员关系就非常有用。如果重建本地缓存非常耗时，那么你肯定不希望在每次重启消费者时都经历这个过程。更重要的是，在消费者重启时，消费者所拥有的分区不会被重新分配。在重启过程中，消费者不会读取这些分区，所以当消费者重启完毕时，读取进度会稍稍落后，但你要相信它们一定会赶上。

需要注意的是，群组的固定成员在关闭时不会主动离开群组，它们何时“真正消失”取决于 `session.timeout.ms` 参数。你可以将这个参数设置得足够大，避免在进行简单的应用程序重启时触发再均衡，但又要设置得足够小，以便在出现严重停机时自动重新分配分区，避免这些分区的读取进度出现较大的滞后。

4.2 创建 Kafka 消费者

在读取消息之前，需要先创建一个 KafkaConsumer 对象。创建 KafkaConsumer 对象与创建 KafkaProducer 对象非常相似——把想要传给消费者的属性放在 Properties 对象里。本章后续部分将深入介绍所有的配置属性。为简单起见，这里只提供 3 个必要的属性：bootstrap.servers、key.deserializer 和 value.deserializer。

第一个属性 bootstrap.servers 指定了连接 Kafka 集群的字符串。它的作用与 KafkaProducer 中的 bootstrap.servers 一样，有关这个属性的详细定义可以参见第 3 章。另外两个属性 key.deserializer 和 value.deserializer 与生产者的 key.serializer 和 value.serializer 类似，只不过它们不是使用指定类把 Java 对象转成字节数组，而是把字节数组转成 Java 对象。

严格来说，第 4 个属性 group.id 不是必需的，但会经常被用到。它指定了一个消费者属于哪一个消费者群组。也可以创建不属于任何一个群组的消费者，只是这种做法不太常见，所以本书的大部分章节会假设消费者属于某个群组。

下面的代码片段演示了如何创建一个 KafkaConsumer 对象。

```
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("group.id", "CountryCounter");
props.put("key.deserializer",
    "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer",
    "org.apache.kafka.common.serialization.StringDeserializer");

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

如果你已经阅读过第 3 章中关于如何创建生产者的内容，那么就应该很熟悉上面的代码。我们假设消息的键和值都是字符串类型，唯一不同的是新增了 group.id 属性，它指定了消费者所属的群组的名字。

4.3 订阅主题

在创建好消费者之后，下一步就可以开始订阅主题了。subscribe() 方法会接收一个主题列表作为参数，看起来很简单。

```
consumer.subscribe(Collections.singletonList("customerCountries")); ❶
```

❶ 为简单起见，这里创建了只包含一个元素的主题列表，主题的名字叫作 customerCountries。

也可以在调用 subscribe() 方法时传入一个正则表达式。正则表达式可以匹配多个主题，如果有人创建了新主题，并且主题的名字与正则表达式匹配，那么就会立即触发一次再均衡，然后消费者就可以读取新主题里的消息。如果应用程序需要读取多个主题，并且可以处理不同类型的数据，那么这种订阅方式就很有用。在 Kafka 和其他系统之间复制数据的应用程序或流式处理应用程序经常使用正则表达式来订阅多个主题。

如果要订阅所有与测试相关的主题，那么可以这样做。

```
consumer.subscribe(Pattern.compile("test.*"));
```

如果你的 Kafka 集群包含了大量分区（比如 30 000 个或更多），则需注意，主题的过滤是在客户端完成的。当你使用正则表达式而不是指定列表订阅主题时，消费者将定期向 broker 请求所有已订阅的主题及分区。然后，客户端会用这个列表来检查是否有新增的主题，如果有，就订阅它们。如果主题很多，消费者也很多，那么通过正则表达式订阅主题就会给 broker、客户端和网络带来很大的开销。在某些情况下，主题元数据使用的带宽会超过用于发送数据的带宽。另外，为了能够使用正则表达式订阅主题，需要授予客户端获取集群全部主题元数据的权限，即全面描述整个集群的权限。

4.4 轮询

消费者 API 最核心的东西是通过一个简单的轮询向服务器请求数据。消费者代码的主要部分如下所示。

```
Duration timeout = Duration.ofMillis(100);

while (true) { ❶
    ConsumerRecords<String, String> records = consumer.poll(timeout); ❷

    for (ConsumerRecord<String, String> record : records) { ❸
        System.out.printf("topic = %s, partition = %d, offset = %d, " +
                          "customer = %s, country = %s\n",
        record.topic(), record.partition(), record.offset(),
                record.key(), record.value());
        int updatedCount = 1;
        if (custCountryMap.containsKey(record.value())) {
            updatedCount = custCountryMap.get(record.value()) + 1;
        }
        custCountryMap.put(record.value(), updatedCount);

        JSONObject json = new JSONObject(custCountryMap);
        System.out.println(json.toString()); ❹
    }
}
```

❶ 这是一个无限循环。消费者实际上是一个长时间运行的应用程序，它通过持续轮询来向 Kafka 请求数据。稍后我们将介绍如何退出循环，并关闭消费者。

❷ 这一行代码非常重要。像鲨鱼停止移动就会死掉一样，消费者必须持续对 Kafka 进行轮询，否则会被认为已经“死亡”，它所消费的分区将被移交给群组里其他的消费者。传给 `poll()` 的参数是一个超时时间间隔，用于控制 `poll()` 的阻塞时间（当消费者缓冲区里没有可用数据时会发生阻塞）。如果这个参数被设置为 0 或者有可用的数据，那么 `poll()` 就会立即返回，否则它会等待指定的毫秒数。

❸ `poll()` 方法会返回一个记录列表。列表中的每一条记录都包含了主题和分区的信息、记录在分区里的偏移量，以及记录的键–值对。我们一般会遍历这个列表，逐条处理记录。

❹ 把处理结果保存起来或者更新已有的记录后，处理过程就结束了。这里是为了统计各个国家的客户数量，所以使用了哈希表来保存结果，并把结果以 JSON 的格式打印了出来。在真实的场景中，结果一般会被保存到数据存储系统中。

轮询不只是获取数据那么简单。在第一次调用消费者的 `poll()` 方法时，它需要找到 `GroupCoordinator`，加入群组，并接收分配给它的分区。如果触发了再均衡，则整个再均衡过程也会在轮询里进行，包括执行相关的回调。所以，消费者或回调里可能出现的错误最后都会转化成 `poll()` 方法抛出的异常。

需要注意的是，如果超过 `max.poll.interval.ms` 没有调用 `poll()`，则消费者将被认为已经“死亡”，并被逐出消费者群组。因此，要避免在轮询循环中做任何可能导致不可预知的阻塞的操作。

线程安全

我们既不能在同一个线程中运行多个同属一个群组的消费者，也不能保证多个线程能够安全地共享一个消费者。按照规则，一个消费者使用一个线程。如果要在应用程序的同一个消费者群组里运行多个消费者，则需要让每个消费者运行在自己的线程中。最好是把消费者的逻辑封装在自己的对象里，然后用 Java 的 `ExecutorService` 启动多个线程，让每个消费者运行在自己的线程中。Confluent 博客上的教程展示了具体该怎么做。

在旧版本 Kafka 中，轮询方法的完整签名是 `poll(long)`。现在，这个签名被弃用了，新 API 的签名是 `poll(Duration)`。除了参数类型发生变化，方法体里的阻塞语义也发生了细微的改变。原来的方法会一直阻塞，直到从 Kafka 获取所需的元数据，即使阻塞时间比指定的超时时间还长。新方法将遵守超时限制，不会一直等待元数据返回。如果你已经有一个消费者使用 `poll(0)` 来获取 Kafka 元数据（不消费任何记录，这是一种相当常见的做法），那么就不要指望把它改成 `poll(Duration.ofMillis(0))` 后还能获得同样的效果。你需要想新的办法来达到目的。通常的解决办法是将逻辑放在 `rebalanceListener.onPartitionAssignment()` 方法里，这个方法一定会在获取分区元数据之后以及记录开始到达之前被调用。Jesse Anderson 在他的博文 “Kafka's Got a Brand-New Poll” 中分享了另一种解决方案。

另外，还可以让消费者维护一个事件队列，并让多个工作线程执行队列里的任务。Igor Buzatović在他的博文“Multi-Threaded Message Consumption with the Apache Kafka Consumer”中提供了这种模式的示例。

4.5 配置消费者

到目前为止，本章介绍了如何使用消费者 API，但只涉及了几个配置属性——bootstrap.servers、group.id、key.deserializer 和 value.deserializer。Kafka 的文档中列出了所有与消费者相关的配置属性。大部分属性有合理的默认值，一般不需要修改它们。不过，有一些属性与消费者的性能和可用性有很大关系，接下来将介绍这些比较重要的属性。

4.5.1 fetch.min.bytes

这个属性指定了消费者从服务器获取记录的最小字节数，默认是 1 字节。broker 在收到消费者的获取数据请求时，如果可用数据量小于 fetch.min.bytes 指定的大小，那么它就会等到有足够可用数据时才将数据返回。这样可以降低消费者和 broker 的负载，因为它们在主题流量不是很大的时候（或者一天里的低流量时段）不需要来来回回地传输消息。如果消费者在没有太多可用数据时 CPU 使用率很高，或者在有很多消费者时为了降低 broker 的负载，那么可以把这个属性的值设置得比默认值大。但需要注意的是，在低吞吐量的情况下，加大这个值会增加延迟。

4.5.2 fetch.max.wait.ms

通过设置 fetch.min.bytes，可以让 Kafka 等到有足够多的数据时才将它们返回给消费者，feth.max.wait.ms 则用于指定 broker 等待的时间，默认是 500 毫秒。如果没有足够多的数据流入 Kafka，那么消费者获取数据的请求就得不到满足，最多会导致 500 毫秒的延迟。如果要降低潜在的延迟（为了满足 SLA），那么可以把这个属性的值设置得小一些。如果 fetch.max.wait.ms 被设置为 100 毫秒，fetch.min.bytes 被设置为 1 MB，那么 Kafka 在收到消费者的请求后，如果有 1 MB 数据，就将其返回，如果没有，就在 100 毫秒后返回，就看哪个条件先得到满足。

4.5.3 fetch.max.bytes

这个属性指定了 Kafka 返回的数据的最大字节数（默认为 50 MB）。消费者会将服务器返回的数据放在内存中，所以这个属性被用于限制消费者用来存放数据的内存大小。需要注意的是，记录是分批发送给客户端的，如果 broker 要发送的批次超过了这个属性指定的大小，那么这个限制将被忽略。这样可以保证消费者能够继续处理消息。值得注意的是，broker 端也有一个与之对应的配置属性，Kafka 管理员可以用它来限制最大获取数量。broker 端的这个配置属性可能很有用，因为请求的数据量越大，需要从磁盘读取的数据量就越大，通过网络发送数据的时间就越长，这可能会导致资源争用并增加 broker 的负载。

4.5.4 max.poll.records

这个属性用于控制单次调用 poll() 方法返回的记录条数。可以用它来控制应用程序在进行每一次轮询循环时需要处理的记录条数（不是记录的大小）。

4.5.5 max.partition.fetch.bytes

这个属性指定了服务器从每个分区里返回给消费者的最大字节数（默认值是 1 MB）。当 KafkaConsumer.poll() 方法返回 ConsumerRecords 时，从每个分区里返回的记录最多不超过 max.partition.fetch.bytes 指定的字节。需要注意的是，使用这个属性来控制消费者的内存使用量会让事情变得复杂，因为你无法控制 broker 返回的响应里包含多少个分区的数据。因此，对于这种情况，建议用 fetch.max.bytes 替代，除非有特殊的需求，比如要求从每个分区读取差不多的数据量。

4.5.6 session.timeout.ms 和 heartbeat.interval.ms

session.timeout.ms 指定了消费者可以在多长时间内不与服务器发生交互而仍然被认为还“活着”，默认是 10 秒。如果消费者没有在 session.timeout.ms 指定的时间内发送心跳给群组协调器，则会被认为已“死亡”，协调器就会触发再均衡，把分区分配给群组里的其他消费者。session.timeout.ms 与 heartbeat.interval.ms 紧密相关。heartbeat.interval.ms 指定了消费者向协调器发送心跳的

频率，session.timeout.ms 指定了消费者可以多久不发送心跳。因此，我们一般会同时设置这两个属性，heartbeat.interval.ms 必须比 session.timeout.ms 小，通常前者是后者的 1/3。如果 session.timeout.ms 是 3 秒，那么 heartbeat.interval.ms 就应该是 1 秒。把 session.timeout.ms 设置得比默认值小，可以更快地检测到崩溃，并从崩溃中恢复，但也会导致不必要的再均衡。把 session.timeout.ms 设置得比默认值大，可以减少意外的再均衡，但需要更长的时间才能检测到崩溃。

4.5.7 max.poll.interval.ms

这个属性指定了消费者在被认为已经“死亡”之前可以在多长时间内不发起轮询。前面提到过，心跳和会话超时是 Kafka 检测已“死亡”的消费者并撤销其分区的主要机制。我们也提到了心跳是通过后台线程发送的，而后台线程有可能在消费者主线程发生死锁的情况下继续发送心跳，但这个消费者并没有在读取分区里的数据。要想知道消费者是否还在处理消息，最简单的方法是检查它是否还在请求数据。但是，请求之间的时间间隔是很难预测的，它不仅取决于可用的数据量、消费者处理数据的方式，有时还取决于其他服务的延迟。在需要耗费时间来处理每个记录的应用程序中，可以通过 max.poll.records 来限制返回的数据量，从而限制应用程序在再次调用 poll() 之前的等待时长。但是，即使设置了 max.poll.records，调用 poll() 的时间间隔仍然很难预测。于是，设置 max.poll.interval.ms 就成了一种保险措施。它必须被设置得足够大，让正常的消费者尽量不触及这个阈值，但又要足够小，避免有问题的消费者给应用程序造成严重影响。这个属性的默认值为 5 分钟。当这个阈值被触及时，后台线程将向 broker 发送一个“离开群组”的请求，让 broker 知道这个消费者已经“死亡”，必须进行群组再均衡，然后停止发送心跳。

4.5.8 default.api.timeout.ms

如果在调用消费者 API 时没有显式地指定超时时间，那么消费者就会在调用其他 API 时使用这个属性指定的值。默认值是 1 分钟，因为它比请求超时时间的默认值大，所以可以将重试时间包含在内。poll() 方法是一个例外，因为它需要显式地指定超时时间。

4.5.9 request.timeout.ms

这个属性指定了消费者在收到 broker 响应之前可以等待的最长时间。如果 broker 在指定时间内没有做出响应，那么客户端就会关闭连接并尝试重连。它的默认值是 30 秒。不建议把它设置得比默认值小。在放弃请求之前要给 broker 留有足够长的时间来处理其他请求，因为向已经过载的 broker 发送请求几乎没有什么好处，况且断开并重连只会造成更大的开销。

4.5.10 auto.offset.reset

这个属性指定了消费者在读取一个没有偏移量或偏移量无效（因消费者长时间不在线，偏移量对应的记录已经过期并被删除）的分区时该做何处理。它的默认值是 latest，意思是说，如果没有有效的偏移量，那么消费者将从最新的记录（在消费者启动之后写入 Kafka 的记录）开始读取。另一个值是 earliest，意思是说，如果没有有效的偏移量，那么消费者将从起始位置开始读取记录。如果将 auto.offset.reset 设置为 none，并试图用一个无效的偏移量来读取记录，则消费者将抛出异常。

4.5.11 enable.auto.commit

这个属性指定了消费者是否自动提交偏移量，默认值是 true。你可以把它设置为 false，选择自己控制何时提交偏移量，以尽量避免出现数据重复和丢失。如果它被设置为 true，那么还有另外一个属性 auto.commit.interval.ms 可以用来控制偏移量的提交频率。本章后续部分将深入介绍与提交偏移量相关的其他内容。

4.5.12 partition.assignment.strategy

我们知道，分区会被分配给群组里的消费者。PartitionAssignor 根据给定的消费者和它们订阅的主题来决定哪些分区应该被分配给哪个消费者。Kafka 提供了几种默认的分配策略。

区间（**range**）

这个策略会把每一个主题的若干个连续分区分配给消费者。假设消费者 C1 和消费者 C2 同时订阅了主题 T1 和主题 T2，并且每个主题有 3 个分区。那么消费者 C1 有可能会被分配到这两个主题的分区 0 和分区 1，消费者 C2 则会被分配到这两个主题的分区 2。因为每个主题拥有奇数个分区，并且都遵循一样的分配策略，所以第一个消费者会分配到比第二个消费者更多的分区。只要使用了这个策略，并且分区数量无法被消费者数量整除，就会出现这种情况。

轮询（**roundRobin**）

这个策略会把所有被订阅的主题的所有分区按顺序逐个分配给消费者。如果使用轮询策略为消费者 C1 和消费者 C2 分配分区，那么消费者 C1 将分配到主题 T1 的分区 0 和分区 2 以及主题 T2 的分区 1，消费者 C2 将分配到主题 T1 的分区 1 以及主题 T2 的分区 0 和分区 2。一般来说，如果所有消费者都订阅了相同的主题（这种情况很常见），那么轮询策略会给所有消费者都分配相同数量（或最多就差一个）的分区。

黏性（**sticky**）

设计黏性分区分配器的目的有两个：一是尽可能均衡地分配分区，二是在进行再均衡时尽可能多地保留原先的分区所有权关系，减少将分区从一个消费者转移给另一个消费者所带来的开销。如果所有消费者都订阅了相同的主题，那么黏性分配器初始的分配比例将与轮询分配器一样均衡。后续的重新分配将同样保持均衡，但减少了需要移动的分区的数量。如果同一个群组里的消费者订阅了不同的主题，那么黏性分配器的分配比例将比轮询分配器更加均衡。

协作黏性（**cooperative sticky**）

这个分配策略与黏性分配器一样，只是它支持协作（增量式）再均衡，在进行再均衡时消费者可以继续从没有被重新分配的分区读取消息。可以参考 4.1.2 节了解更多有关协作再均衡的内容。需要注意的是，如果你从 Kafka 2.3 之前的版本开始升级，并希望使用协作黏性分配策略，则需要遵循特定的升级路径，具体请参看相关升级指南。

可以通过 `partition.assignment.strategy` 来配置分区策略，默认值是 `org.apache.kafka.clients.consumer.RangeAssignor`，它实现了区间策略。你也可以把它改成 `org.apache.kafka.clients.consumer.RoundRobinAssignor`、`org.apache.kafka.clients.consumer.StickyAssignor` 或 `org.apache.kafka.clients.consumer.CooperativeStickyAssignor`。还可以使用自定义分配策略，如果是这样，则需要把 `partition.assignment.strategy` 设置成自定义类的名字。

4.5.13 `client.id`

这个属性可以是任意字符串，broker 用它来标识从客户端发送过来的请求，比如获取请求。它通常被用在日志、指标和配额中。

4.5.14 `client.rack`

在默认情况下，消费者会从每个分区的首领副本那里获取消息。但是，如果集群跨越了多个数据中心或多个云区域，那么让消费者从位于同一区域的副本那里获取消息就会具有性能和成本方面的优势。要从最近的副本获取消息，需要设置 `client.rack` 这个参数，用于标识客户端所在的区域。然后，可以将 broker 的 `replica.selector.class` 参数值改为 `org.apache.kafka.common.replica.RackAwareReplicaSelector`。

你也可以实现自己的 `replica.selector.class`，根据客户端元数据和分区元数据选择想要读取的副本。

4.5.15 `group.instance.id`

这个属性可以是任意具有唯一性的字符串，被用于消费者群组的固定名称。

4.5.16 `receive.buffer.bytes` 和 `send.buffer.bytes`

这两个属性分别指定了 socket 在读写数据时用到的 TCP 缓冲区大小。如果它们被设置为–1，就使用操作系统的默认值。如果生产者或消费者与 broker 位于不同的数据中心，则可以适当加大它们的值，因为跨数据中心网络的延迟一般都比较高，而带宽又比较低。

4.5.17 `offsets.retention.minutes`

这是 broker 端的一个配置属性，需要注意的是，它也会影响消费者的行为。只要消费者群组里有活跃的成员（也就是说，有成员通过发送心跳来保持其身份），群组提交的每一个分区的最后一个偏移量就会被 Kafka 保留下来，在进行重分配或重启之后就可以获取到这些偏移量。但是，如果一个消费者群组失去了所有成员，则 Kafka 只会按照这个属性指定的时间（默认为 7 天）保留偏移量。一旦偏移量被删除，即使消费者群组又“活”了过来，它也会像一个全新的群组一样，没有了过去的消费记忆。需要注意的是，这个行为在不同的版本中经历了几次变化，如果你使用的 Kafka 版本小于 2.1.0，那么请仔细查阅相关文档以了解其对应的行为。

4.6 提交和偏移量

每次调用 `poll()` 方法，它总是会返回还没有被消费者读取过的记录，这意味着我们有办法来追踪哪些记录是被群组里的哪个消费者读取过的。之前提到过，Kafka 不像其他 JMS 队列系统那样需要收到来自消费者的确认，这是 Kafka 的一个独特之处。相反，消费者可以用 Kafka 来追踪已读取的消息在分区中的位置（偏移量）。

我们把更新分区当前读取位置的操作叫作**偏移量提交**。与传统的消息队列不同，Kafka 不会提交每一条记录。相反，消费者会将已成功处理的最后一条消息提交给 Kafka，并假定该消息之前的每一条消息都已成功处理。

那么消费者是如何提交偏移量的呢？消费者会向一个叫作 `__consumer_offset` 的主题发送消息，消息里包含每个分区的偏移量。如果消费者一直处于运行状态，那么偏移量就没有什么实际作用。但是，如果消费者发生崩溃或有新的消费者加入群组，则会**触发再均衡**。再均衡完成之后，每个消费者可能会被分配新的分区，而不是之前读取的那个。为了能够继续之前的工作，消费者需要读取每个分区最后一次提交的偏移量，然后从偏移量指定的位置继续读取消息。

如果最后一次提交的偏移量小于客户端处理的最后一条消息的偏移量，那么处于两个偏移量之间的消息就会被重复处理，如图 4-8 所示。

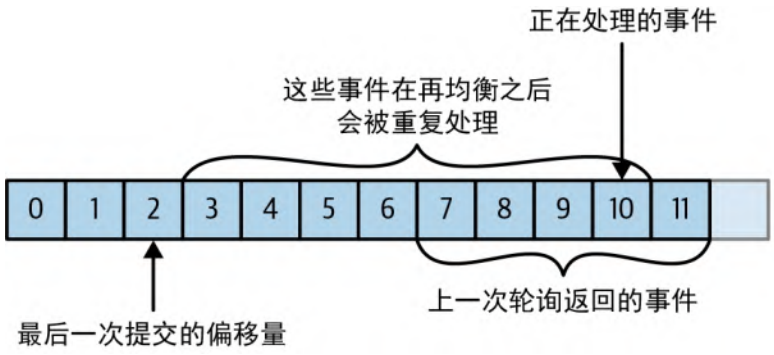

图 4-8：重复处理消息

如果最后一次提交的偏移量大于客户端处理的最后一条消息的偏移量，那么处于两个偏移量之间的消息就会丢失，如图 4-9 所示。

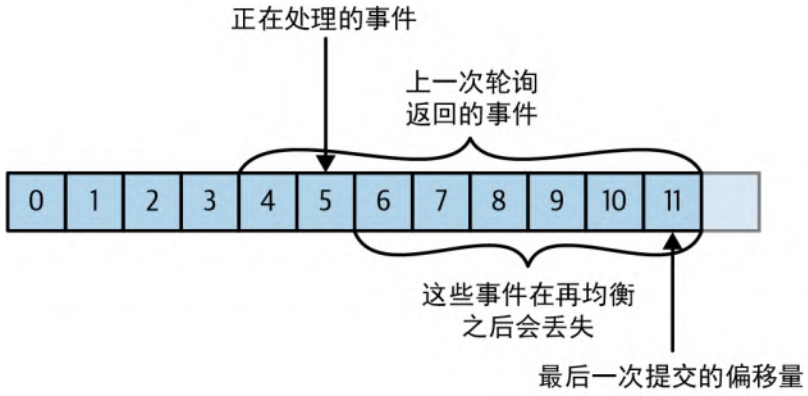

图 4-9：两个偏移量之间的消息丢失了

所以，如何管理偏移量对客户端应用程序有很大的影响。`KafkaConsumer`API 提供了多种提交偏移量的方式。

提交哪一个偏移量？

如果使用自动提交或不指定提交的偏移量，那么将默认提交 poll() 返回的最后一个位置之后的偏移量。在进行手动提交或需要提交特定的偏移量时，一定要记住这一点。另外，本书一直在强调“提交比客户端从 poll() 返回的最后一个位置大 1 的偏移量”是件很烦琐的事情，因此，每当提到默认的偏移量提交行为时，就直接写成“提交最后一个偏移量”。如果需要手动提交偏移量，那么请记住这一点。

4.6.1 自动提交

最简单的提交方式是让消费者自动提交偏移量。如果 enable.auto.commit 被设置为 true，那么每过 5 秒，消费者就会自动提交 poll() 返回的最大偏移量。提交时间间隔通过

auto.commit.interval.ms 来设定，默认是 5 秒。与消费者中的其他处理过程一样，自动提交也是在轮询循环中进行的。消费者会在每次轮询时检查是否该提交偏移量了，如果是，就会提交最后一次轮询返回的偏移量。

不过，在使用这种提交方式之前，需要知道它会带来怎样的后果。

假设我们使用默认的 5 秒提交时间间隔，并且消费者在最后一次提交偏移量之后 3 秒会发生崩溃。再均衡完成之后，接管分区的消费者将从最后一次提交的偏移量的位置开始读取消息。这个偏移量实际上落后了 3 秒，所以在这 3 秒内到达的消息会被重复处理。可以通过修改提交时间间隔来更频繁地提交偏移量，缩小可能导致重复消息的时间窗口，但无法完全避免。

在使用自动提交时，到了该提交偏移量的时候，轮询方法将提交上一次轮询返回的偏移量，但它并不知道具体哪些消息已经被处理过了，所以，在再次调用 poll() 之前，要确保上一次 poll() 返回的所有消息都已经处理完毕（调用 close() 方法也会自动提交偏移量）。通常情况下这不会有什么问题，但在处理异常或提前退出轮询循环时需要特别小心。

虽然自动提交很方便，但是没有为避免开发者重复处理消息留有余地。

4.6.2 提交当前偏移量

大部分开发者通过控制提交时间来降低丢失消息的可能性和减少可能在再均衡期间发生的消息重复。消费者 API 提供了另一种提交偏移量的方式，开发者可以在必要的时候手动提交当前偏移量，而不是基于时间间隔提交。

把 enable.auto.commit 设置为 false，让应用程序自己决定何时提交偏移量。使用 commitSync() 提交偏移量是最简单可靠的方式。这个 API 会提交 poll() 返回的最新偏移量，提交成功后马上返回，如果由于某些原因提交失败就抛出异常。

需要注意的是，commitSync() 将会提交 poll() 返回的最新偏移量，所以，如果你在处理完所有记录之前就调用了 commitSync()，那么一旦应用程序发生崩溃，就会有丢失消息的风险（消息已被提交但未被处理）。如果应用程序在处理记录时发生崩溃，但 commitSync() 还没有被调用，那么从最近批次的开始位置到发生再均衡时的所有消息都将被再次处理——这或许比丢失消息更好，或许更坏。

下面是在处理完最近一批消息后使用 commitSync() 提交偏移量的例子。

```
Duration timeout = Duration.ofMillis(100);

while (true) {
    ConsumerRecords<String, String> records = consumer.poll(timeout);
    for (ConsumerRecord<String, String> record : records) {
        System.out.printf("topic = %s, partition = %d, offset =
            %d, customer = %s, country = %s\n",
            record.topic(), record.partition(),
            record.offset(), record.key(), record.value()); ❶
```

```
    }
    try {
        consumer.commitSync(); ❷
    } catch (CommitFailedException e) {
        log.error("commit failed", e) ❸
    }
}
```

❶ 假设把消息内容打印出来就算处理完毕。你的应用程序可能会对记录做更多的处理——修改、填充、聚合、展示在仪表盘上或者向用户发送通知。你需要根据具体的使用场景来决定怎么处理。

❷ 处理完当前批次后，在轮询更多的消息之前，调用 commitSync() 方法提交当前批次最新的偏移量。

❸ 只要没有发生不可恢复的错误，commitSync() 方法就会一直尝试直至提交成功。如果提交失败，就把异常记录到错误日志里。

4.6.3 异步提交

手动提交有一个缺点，在 broker 对请求做出回应之前，应用程序会一直阻塞，这样会限制应用程序的吞吐量。可以通过降低提交频率来提升吞吐量，但如果发生了再均衡，则会增加潜在的消息重复。

这个时候可以使用异步提交 API。只管发送请求，无须等待 broker 做出响应。

```
Duration timeout = Duration.ofMillis(100);

while (true) {
    ConsumerRecords<String, String> records = consumer.poll(timeout);
    for (ConsumerRecord<String, String> record : records) {
        System.out.printf("topic = %s, partition = %s,
            offset = %d, customer = %s, country = %s\n",
            record.topic(), record.partition(), record.offset(),
            record.key(), record.value());
    }
    consumer.commitAsync(); ❶
}
```

❶ 提交最后一个偏移量，然后继续处理。

在提交成功或碰到无法恢复的错误之前，commitSync() 会一直重试，但 commitAsync() 不会，这是 commitAsync() 的一个缺点。之所以不进行重试，是因为 commitAsync() 在收到服务器端的响应时，可能已经有一个更大的偏移量提交成功。假设我们发出一个提交偏移量 2000 的请求，这个时候出现了短暂的通信问题，服务器收不到请求，自然也不会做出响应。与此同时，我们处理了另外一批消息，并成功提交了偏移量 3000。如果此时 commitAsync() 重新尝试提交偏移量 2000，则有可能在偏移量 3000 之后提交成功。这个时候如果发生再均衡，就会导致消息重复。

之所以提到这个问题并强调提交顺序的重要性，是因为 commitAsync() 也支持回调，回调会在 broker 返回响应时执行。回调经常被用于记录偏移量提交错误或生成指标，如果要用它来重试提交偏移量，那么一定要注意提交顺序。

```
Duration timeout = Duration.ofMillis(100);

while (true) {
    ConsumerRecords<String, String> records = consumer.poll(timeout);
    for (ConsumerRecord<String, String> record : records) {
        System.out.printf("topic = %s, partition = %s,
        offset = %d, customer = %s, country = %s\n",
        record.topic(), record.partition(), record.offset(),
        record.key(), record.value());
    }
    consumer.commitAsync(new OffsetCommitCallback() {
        public void onComplete(Map<TopicPartition,
        OffsetAndMetadata> offsets, Exception e) {
            if (e != null)
```

```
                log.error("Commit failed for offsets {}", offsets, e);
            }
        }); ❶
}
```

❶ 发送请求，然后继续处理。如果提交失败，则错误信息和偏移量会被记录下来。

异步提交中的重试

可以用一个单调递增的消费者序列号变量来维护异步提交的顺序。每次调用 commitAsync() 后增加序列号，并在回调中更新序列号变量。在准备好进行重试时，先检查回调的序列号与序列号变量是否相等。如果相等，就说明没有新的提交，可以安全地进行重试。如果序列号变量比较大，则说明已经有新的提交了，此时应该停止重试。

4.6.4 同步和异步组合提交

一般情况下，偶尔提交失败但不进行重试不会有太大问题，因为如果提交失败是由于临时问题导致的，后续的提交总会成功。但如果这是发生在消费者被关闭或再均衡前的最后一次提交，则要确保提交是成功的。

如果是消费者被关闭，那么一般会使用 commitAsync() 和 commitSync() 的组合。这种模式的原理如下。（后面讲到再均衡监听器时将讨论如何在发生再均衡前提交偏移量）。

```
Duration timeout = Duration.ofMillis(100);

try {
    while (!closing) {
        ConsumerRecords<String, String> records = consumer.poll(timeout);
        for (ConsumerRecord<String, String> record : records) {
            System.out.printf("topic = %s, partition = %s, offset = %d,
                customer = %s, country = %s\n",
                record.topic(), record.partition(),
                record.offset(), record.key(), record.value());
        }
        consumer.commitAsync();❶
    }
    consumer.commitSync();❷
} catch (Exception e) {
    log.error("Unexpected error", e);
} finally {
        consumer.close();
}
```

❶ 如果一切正常，就用 commitAsync() 提交偏移量。这样速度更快，而且即使这次提交失败，下一次提交也会成功。

❷ 如果直接关闭消费者，那么就没有所谓的“下一次提交”了。commitSync() 会一直重试，直到提交成功或发生无法恢复的错误。

4.6.5 提交特定的偏移量

提交最后一个偏移量的频率应该与处理消息批次的频率一样。但如果想要更频繁地提交偏移量该怎么办？如果 poll() 返回了一大批数据，那么为了避免可能因再均衡引起的消息重复，想要在批次处理过程中提交偏移量该怎么办？这个时候不能只是调用 commitSync() 或 commitAsync()，因为它们只会提交消息批次里的最后一个偏移量。

幸运的是，消费者 API 允许在调用 commitSync() 和 commitAsync() 时传给它们想要提交的分区和偏移量。假设你正在处理一个消息批次，刚处理好来自主题“customers”的分区 3 的消息，它的偏移量是 5000，那么就可以调用 commitSync() 来提交这个分区的偏移量 5001。需要注意的是，因为一个消费者

可能不止读取一个分区，你需要跟踪所有分区的偏移量，所以通过这种方式提交偏移量会让代码变得复杂。

下面是提交特定偏移量的例子。

```
private Map<TopicPartition, OffsetAndMetadata> currentOffsets =
    new HashMap<>(); ❶
int count = 0;

....
Duration timeout = Duration.ofMillis(100);

while (true) {
    ConsumerRecords<String, String> records = consumer.poll(timeout);
    for (ConsumerRecord<String, String> record : records) {
        System.out.printf("topic = %s, partition = %s, offset = %d,
            customer = %s, country = %s\n",
            record.topic(), record.partition(), record.offset(),
            record.key(), record.value()); ❷
        currentOffsets.put(
            new TopicPartition(record.topic(), record.partition()),
            new OffsetAndMetadata(record.offset()+1, "no metadata")); ❸
        if (count % 1000 == 0) ❹
            consumer.commitAsync(currentOffsets, null); ❺
        count++;
    }
}
```

❶ 用于跟踪偏移量的 map。

❷ `printf` 表示处理消息的过程。

❸ 在读取每一条记录之后，将下一条要处理的消息的偏移量更新到 map 中。提交的偏移量应该是应用程序要处理的下一条消息的偏移量，下一次就从这个位置开始读取。

❹ 我们决定每处理 1000 条记录就提交一次偏移量。在实际当中，可以基于时间或记录的内容来提交偏移量。

❺ 这里调用的是 `commitAsync()`（没有回调，所以第二个参数是 `null`），不过调用 `commitSync()` 也是可以的。当然，在提交特定偏移量时仍然要处理可能出现的错误，就像之前那样。

4.7 再均衡监听器

之前在讨论提交偏移量时提到过，消费者会在退出和进行分区再均衡之前做一些清理工作。

如果知道消费者即将失去对一个分区的所有权，那么你就会马上提交最后一个已处理的记录的偏移量。可能还需要关闭文件句柄、数据库连接等。

消费者 API 提供了一些方法，让你可以在消费者分配到新分区或旧分区被移除时执行一些代码逻辑。你所要做的就是在调用 `subscribe()` 方法时传进去一个 `ConsumerRebalanceListener` 对象。`ConsumerRebalanceListener` 有 3 个需要实现的方法。

`public void onPartitionsAssigned(Collection<TopicPartition> partitions)`

这个方法会在重新分配分区之后以及消费者开始读取消息之前被调用。你可以在这个方法中准备或加载与分区相关的状态信息、找到正确的偏移量，等等。这里所有的事情都应该保证在 `max.poll.timeout.ms` 内完成，以便消费者可以成功地加入群组。

`public void onPartitionsRevoked(Collection<TopicPartition> partitions)`

这个方法会在消费者放弃对分区的所有权时调用——可能是因为发生了再均衡或者消费者正在被关闭。通常情况下，如果使用了主动再平衡算法，那么这个方法会在再平衡开始之前以及消费者停止读取消息之后调用。如果使用了协作再平衡算法，那么这个方法会在再均衡结束时调用，而且只涉及消费者放弃所有权的那些分区。如果你要提交偏移量，那么可以在这里提交，无论是哪个消费者接管这个分区，它都知道应该从哪里开始读取消息。

`public void onPartitionsLost(Collection<TopicPartition> partitions)`

这个方法只会在使用了协作再均衡算法并且之前不是通过再均衡获得的分区被重新分配给其他消费者时调用（之前通过再均衡获得的分区被重新分配时会调用 `onPartitions Revoked()`）。你可以在这里清除与这些分区相关的状态或资源。需要注意的是，在清理状态时要非常小心，因为分区的新所有者可能也保存了分区状态，需要避免发生冲突。如果你没有实现这个方法，则 `onPartitionsRevoked()` 将被调用。

如果使用了协作再均衡算法，那么需要注意以下几点。

- `onPartitionsAssigned()` 在每次进行再均衡时都会被调用，以此来告诉消费者发生了再均衡。如果没有新的分区分配给消费者，那么它的参数就是一个空集合。
- `onPartitionsRevoked()` 会在进行正常的再均衡并且有消费者放弃分区所有权时被调用。如果它被调用，那么参数就不会是空集合。
- `onPartitionsLost()` 会在进行意外的再均衡并且参数集合中的分区已经有新的所有者的情况下被调用。

如果这 3 个方法你都实现了，那么就可以保证在一个正常的再均衡过程中，分区新所有者的 `onPartitionsAssigned()` 会在之前的所有者的 `onPartitionsRevoked()` 被调用完毕并放弃了所有权之后被调用。

下面的例子演示了如何在失去分区所有权之前通过 `onPartitionsRevoked()` 方法来提交偏移量。

```
private Map<TopicPartition, OffsetAndMetadata> currentOffsets =
    new HashMap<>();
Duration timeout = Duration.ofMillis(100);

private class HandleRebalance implements ConsumerRebalanceListener { ❶
    public void onPartitionsAssigned(Collection<TopicPartition>
        partitions) { ❷
```

```
    }

    public void onPartitionsRevoked(Collection<TopicPartition> partitions) {
        System.out.println("Lost partitions in rebalance. " +
            "Committing current offsets:" + currentOffsets);
        consumer.commitSync(currentOffsets); ❸
    }
}

try {
    consumer.subscribe(topics, new HandleRebalance()); ❹

    while (true) {
        ConsumerRecords<String, String> records = consumer.poll(timeout);
        for (ConsumerRecord<String, String> record : records) {
            System.out.printf("topic = %s, partition = %s, offset = %d,
                customer = %s, country = %s\n",
                record.topic(), record.partition(), record.offset(),
                record.key(), record.value());
            currentOffsets.put(
                new TopicPartition(record.topic(), record.partition()),
                new OffsetAndMetadata(record.offset()+1, null));
        }
        consumer.commitAsync(currentOffsets, null);
    }
} catch (WakeupException e) {
    // 忽略异常
} catch (Exception e) {
    log.error("Unexpected error", e);
} finally {
    try {
        consumer.commitSync(currentOffsets);
    } finally {
        consumer.close();
        System.out.println("Closed consumer and we are done");
    }
}
```

❶ 首先，需要实现 `ConsumerRebalanceListener` 接口。

❷ 在这个例子中，在分配到新分区时我们不做任何事情，直接开始读取消息。

❸ 如果发生了再均衡，则要在即将失去分区所有权时提交偏移量。我们提交的是所有分区而不只是那些即将失去所有权的分区的偏移量——因为我们提交的是已处理过的消息的偏移量，所以不会有什么问题。况且，我们会使用 `commitSync()` 方法确保在再均衡发生之前提交偏移量。

❹ 把 `ConsumerRebalanceListener` 对象传给 `subscribe()` 方法，这样消费者才能调用它，这是非常重要的一步。

4.8 从特定偏移量位置读取记录

到目前为止，我们已经知道如何使用 `poll()` 从各个分区的最新偏移量位置读取消息，但有时候也需要从不同的偏移量位置读取消息。Kafka 提供了一些方法，可以让 `poll()` 从不同的位置读取消息。

如果你想从分区的起始位置读取所有的消息，或者直接跳到分区的末尾读取新消息，那么 Kafka API 分别提供了两个方法：`seekToBeginning(Collection<Topic Partition> tp)` 和 `seekToEnd(Collection<TopicPartition> tp)`。

Kafka 还提供了用于查找特定偏移量的 API。这个 API 有很多用途，比如，对时间敏感的应用程序在处理速度滞后的情况下可以向前跳过几条消息，或者如果消费者写入的文件丢失了，则它可以重置偏移量，回到某个位置进行数据恢复。

下面的例子演示了如何将分区的当前偏移量定位到在指定时间点生成的记录。

```
Long oneHourEarlier = Instant.now().atZone(ZoneId.systemDefault())
          .minusHours(1).toEpochSecond();
Map<TopicPartition, Long> partitionTimestampMap = consumer.assignment()
        .stream()
        .collect(Collectors.toMap(tp -> tp, tp -> oneHourEarlier)); ❶
Map<TopicPartition, OffsetAndTimestamp> offsetMap
        = consumer.offsetsForTimes(partitionTimestampMap); ❷

for(Map.Entry<TopicPartition,OffsetAndTimestamp> entry: offsetMap.entrySet()) {
    consumer.seek(entry.getKey(), entry.getValue().offset()); ❸
}
```

❶ 首先，创建一个 map，将所有分配给这个消费者的分区（通过调用 `consumer.assignment()`）映射到我们想要回退到的时间戳。

❷ 然后，通过时间戳获取对应的偏移量。这个方法会向 broker 发送请求，通过时间戳获取对应的偏移量。

❸ 最后，将每个分区的偏移量重置成上一步返回的偏移量。

4.9 如何退出

之前在讨论轮询循环时提到过，无须担心消费者在一个无限循环里轮询消息，因为我们可以让其优雅地退出。那么接下来就来看看如何优雅地退出轮询循环。

如果你确定马上要关闭消费者（即使消费者还在等待一个 poll() 返回），那么可以在另一个线程中调用 consumer.wakeup()。如果轮询循环运行在主线程中，那么可以在 ShutdownHook 里调用这个方法。需要注意的是，consumer.wakeup() 是消费者唯一一个可以在其他线程中安全调用的方法。调用 consumer.wakeup() 会导致 poll() 抛出 WakeupException，如果调用 consumer.wakeup() 时线程没有在轮询，那么异常将在下一次调用 poll() 时抛出。不一定要处理 WakeupException，但在退出线程之前必须调用 consumer.close()。消费者在被关闭时会提交还没有提交的偏移量，并向消费者协调器发送消息，告知自己正在离开群组。协调器会立即触发再均衡，被关闭的消费者所拥有的分区将被重新分配给群组里其他的消费者，不需要等待会话超时。

下面是让运行在主线程中的消费者退出循环的代码。这段代码是经过简化的，你可以在 GitHub 网站上查看完整代码。

```
Runtime.getRuntime().addShutdownHook(new Thread() {
    public void run() {
        System.out.println("Starting exit...");
        consumer.wakeup(); ❶
        try {
            mainThread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
});

...
Duration timeout = Duration.ofMillis(10000); ❷

try {
    // 一直循环，直到按下Ctrl-C组合键，关闭钩子会在退出时做清理工作
    while (true) {
        ConsumerRecords<String, String> records =
            movingAvg.consumer.poll(timeout);
        System.out.println(System.currentTimeMillis() +
            "-- waiting for data...");
        for (ConsumerRecord<String, String> record : records) {
            System.out.printf("offset = %d, key = %s, value = %s\n",
                record.offset(), record.key(), record.value());
        }
        for (TopicPartition tp: consumer.assignment())
            System.out.println("Committing offset at position:" +
                consumer.position(tp));
        movingAvg.consumer.commitSync();
    }
} catch (WakeupException e) {
    // 忽略异常 ❸
} finally {
    consumer.close(); ❹
    System.out.println("Closed consumer and we are done");
}
```

❶ ShutdownHook 运行在单独的线程中，所以退出轮询循环最安全的方式只能是调用 wakeup()。

❷ 一个比较长的轮询超时时间。如果轮询的时间足够短，并且你不介意在退出之前等一小会儿，那么就没有必要调用 wakeup()，只需在每次轮询时检查一下原子布尔变量即可。较长的轮询超时时间在处理低吞吐量主题时比较有用，因为当 broker 没有新数据返回时，客户端在轮询时占用的 CPU 时间会更少。

❸ 因为在另一个线程中调用了 wakeup()，所以 poll() 会抛出 WakeupException。你可能想捕获异常，确保应用程序不会意外退出，但实际上在这里无须对它做任何处理。

❹ 在退出之前，确保彻底关闭了消费者。

4.10 反序列化器

第 3 章提到过，生产者需要用**序列化器**把对象转换成字节数组后再发送给 Kafka。类似地，消费者需要用**反序列化器**把从 Kafka 接收到的字节数组转换成 Java 对象。在前面的例子中，我们假设每条消息的键–值对都是字符串，所以使用了默认的 `StringDeserializer`。

第 3 章在讨论 Kafka 生产者时已经介绍过如何序列化自定义对象类型，以及如何使用 Avro 和 `AvroSerializer` 根据定义好的模式生成 Avro 对象，并将其序列化后发送给 Kafka。现在来看看如何为对象自定义反序列化器，以及如何使用 Avro 和 Avro 反序列化器。

很显然，生成消息所使用的序列化器与读取消息所使用的反序列化器应该是相对应的。如果用 `IntSerializer` 序列化对象，但用 `StringDeserializer` 反序列化它，那么就不会出现什么好结果。对开发者来说，他们必须知道写入主题的消息使用的是哪一种序列化器，并确保每个主题里只包含能够被反序列化器解析的数据。使用 Avro 和模式注册表进行序列化和反序列化的优势在于：`AvroSerializer` 可以保证写入主题的数据与主题的模式是兼容的，也就是说，可以使用相应的反序列化器和模式来反序列化数据。另外，不管是在生产者端还是消费者端出现的任何一个与兼容性有关的错误都会被捕捉到，而且这些错误都带有描述性信息，这也就意味着，当出现序列化错误时，无须再费劲地调试字节数组了。

尽管自定义反序列化器不太常用，我们还是简单地演示一下如何进行操作，然后再举例演示如何使用 Avro 来反序列化消息的键和值。

4.10.1 自定义反序列化器

下面以在第 3 章中使用过的自定义对象为例，为它开发一个反序列化器。

```
public class Customer {
    private int customerID;
    private String customerName;

    public Customer(int ID, String name) {
        this.customerID = ID;
        this.customerName = name;
    }

    public int getID() {
        return customerID;
    }

    public String getName() {
        return customerName;
    }
}
```

自定义反序列化器如下所示。

```
import org.apache.kafka.common.errors.SerializationException;

import java.nio.ByteBuffer;
import java.util.Map;

public class CustomerDeserializer implements Deserializer<Customer> { ❶

    @Override
    public void configure(Map configs, boolean isKey) {
        // 不需要做任何配置
    }

    @Override
    public Customer deserialize(String topic, byte[] data) {
        int id;
        int nameSize;
        String name;
```

```
        try {
            if (data == null)
                return null;
            if (data.length < 8)
                throw new SerializationException("Size of data received " +
                    "by deserializer is shorter than expected");

            ByteBuffer buffer = ByteBuffer.wrap(data);
            id = buffer.getInt();
            nameSize = buffer.getInt();

            byte[] nameBytes = new byte[nameSize];
            buffer.get(nameBytes);

            name = new String(nameBytes, "UTF-8");

            return new Customer(id, name); ❷

        } catch (Exception e) {
            throw new SerializationException("Error when deserializing " +
                 "byte[] to Customer " + e);
        }
    }

    @Override
    public void close() {
        // 不需要关闭任何东西
    }
}
```

❶ 消费者也需要用到 `Customer` 类，这个类和序列化器在生产者端和消费者端必须是对应的。在一个大型的企业里，会有很多消费者和生产者共享数据，这给企业带来了一定的挑战。

❷ 把序列化器的逻辑反过来，从字节数组中获取客户 ID 和名字，再用它们构建我们需要的对象。

使用反序列化器的消费者代码如下所示。

```
Duration timeout = Duration.ofMillis(100);
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("group.id", "CountryCounter");
props.put("key.deserializer",
    "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer",
    CustomerDeserializer.class.getName());


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

consumer.subscribe(Collections.singletonList("customerCountries"))

while (true) {
    ConsumerRecords<String, Customer> records = consumer.poll(timeout);
    for (ConsumerRecord<String, Customer> record : records) {
        System.out.println("current customer Id: " +
            record.value().getID() + " and
            current customer name: " +  record.value().getName());
    }
    consumer.commitSync();
}
```

再强调一次，并不建议使用自定义序列化器和自定义反序列化器，因为它们把生产者和消费者紧紧地耦合在一起，容易出错。建议使用标准的消息格式，比如 JSON、Thrift、Protobuf 或 Avro。接下来将介绍如何使用 Avro 反序列化器。有关 Avro 的项目背景、模式和模式兼容性问题可以参见第 3 章。

4.10.2 在消费者里使用 Avro 反序列器

继续使用第 3 章中出现过的 Customer 类。为了读取这些对象，需要实现一个类似下面这样的消费者应用程序。

```
Duration timeout = Duration.ofMillis(100);
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("group.id", "CountryCounter");
props.put("key.deserializer",
    "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer",
    "io.confluent.kafka.serializers.KafkaAvroDeserializer"); ❶
props.put("specific.avro.reader","true");
props.put("schema.registry.url", schemaUrl); ❷
String topic = "customerContacts"

KafkaConsumer<String, Customer> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Collections.singletonList(topic));

System.out.println("Reading topic:" + topic);

while (true) {
    ConsumerRecords<String, Customer> records = consumer.poll(timeout); ❸

    for (ConsumerRecord<String, Customer> record: records) {
        System.out.println("Current customer name is: " +
            record.value().getName()); ❹
    }
    consumer.commitSync();
}
```

❶ 用 KafkaAvroDeserializer 来反序列化 Avro 消息。

❷ schema.registry.url 是一个新的参数，它指向模式的存放位置。消费者可以使用生产者注册的模式来反序列化消息。

❸ 将生成的类 Customer 作为值类型。

❹ record.value() 返回的是一个 Customer 实例，我们把它的名字打印出来。

4.11 独立的消费者：为什么以及怎样使用不属于任何群组的消费者

到目前为止，我们已经讨论了消费者群组，在消费者群组中，分区会被自动分配给消费者，当群组新增消费者或移除消费者时会自动触发再均衡。通常情况下，这些刚好可以满足你的需求，但有时候你可能需要一些更简单的东西。比如，你可能只需要用一个消费者读取一个主题所有的分区或某个分区。这个时候就不需要使用消费者群组和再均衡了，只需要把主题或分区分配给这个消费者，然后开始读取消息，并时不时地提交偏移量。（尽管为了提交偏移量仍然需要配置 `group.id`，但只要不调用 `subscribe()`，消费者就不会加入任何群组。）

如果知道需要读取哪些分区，就不需要**订阅**主题了，可以直接将目标分区**分配**给消费者。消费者既可以订阅主题（并加入消费者群组），也可以为自己分配分区，但不能同时做这两件事情。

下面的例子演示了消费者是如何为自己分配分区并从分区读取消息的。

```
Duration timeout = Duration.ofMillis(100);
List<PartitionInfo> partitionInfos = null;
partitionInfos = consumer.partitionsFor("topic"); ❶

if (partitionInfos != null) {
    for (PartitionInfo partition : partitionInfos)
        partitions.add(new TopicPartition(partition.topic(),
            partition.partition()));
    consumer.assign(partitions); ❷

    while (true) {
        ConsumerRecords<String, String> records = consumer.poll(timeout);

        for (ConsumerRecord<String, String> record: records) {
            System.out.printf("topic = %s, partition = %s, offset = %d,
                customer = %s, country = %s\n",
                record.topic(), record.partition(), record.offset(),
                record.key(), record.value());
        }
        consumer.commitSync();
    }
}
```

❶ 向集群请求主题的可用分区。如果只打算读取特定分区，则可以跳过这一步。

❷ 在知道了需要读取哪些分区之后，调用 `assign()` 方法。

除了不会发生再均衡和不需要手动查找分区，其他的跟平常一样。不过需要注意的是，如果主题增加了新分区，那么消费者并不会收到通知。所以，要么定时调用 `consumer.partitionsFor()` 方法来检查是否有新分区加入，要么在每次添加新分区后重启应用程序。

4.12 小结

本章首先解释了 Kafka 消费者群组的概念，以及消费者群组如何支持多个消费者从主题读取消息。在介绍完这些概念之后，给出了一个消费者订阅主题并读取消息的例子，然后介绍了一些重要的消费者配置参数以及它们对消费者行为的影响。本章用了很大一部分内容解释偏移量以及消费者是如何管理偏移量的。了解消费者提交偏移量的方式有助于更好地使用消费者客户端，所以本章介绍了几种不同的偏移量提交方式。之后又介绍了消费者 API 的其他内容，比如如何处理再均衡以及如何关闭消费者。

本章在最后介绍了反序列化器。消费者客户端可以使用反序列化器将保存在 Kafka 中的字节转换成可以被应用程序处理的 Java 对象。我们主要介绍了 Avro 反序列化器，虽然还有很多其他的可用的反序列化器，但在 Kafka 中，Avro 是最为常用的一个。

第 5 章　编程式管理 Kafka

虽然现在已经有很多基于命令行或图形界面的 Kafka 管理工具（第 9 章将介绍它们），但有时候你可能想直接在客户端应用程序中执行一些管理命令。基于用户的输入或数据创建新主题就是一个非常常见的场景：物联网应用程序从用户设备接收事件，并根据设备类型将事件写入不同的主题。如果制造商生产了一种新设备，那么你要么为其手动创建一个新主题，要么让应用程序在接收到包含未识别类型设备的事件时动态创建一个新主题。虽然后一种方式有缺点，但在一些应用场景中，不依赖于额外的处理过程来生成主题是一种很有吸引力的解决方案。

Kafka 在 0.11 版本中加入了 AdminClient，为之前只能通过命令行完成的管理功能提供了编程 API：查看、创建和删除主题，描述集群，管理 ACL 和修改配置。

假设你的应用程序要为某个主题生成事件，在生成第一个事件之前，这个主题必须存在。在 Kafka 加入 AdminClient 之前，我们只有很少的选择，并且没有一个是特别友好的：可以捕获 `producer.send()` 方法的 `UNKNOWN_TOPIC_OR_PARTITION` 异常，让用户知道需要先创建主题，或者寄希望于 Kafka 集群启用了自动创建主题功能，或者依赖 Kafka 的内部 API 并接受没有兼容性保证的后果。现在，Kafka 提供了 AdminClient，我们就有了一种更好的解决方案：用 AdminClient 检查主题是否存在，如果不存在，就当场创建。

本章将先对 AdminClient 进行概览，然后再深入探讨如何在应用程序中使用它。我们将重点关注最常用的功能：主题管理、消费者群组管理和配置管理。

5.1 AdminClient 概览

在开始使用 AdminClient 之前，最好先了解一下它的核心设计原则。在了解了 AdminClient 的设计原则和使用方法之后，你就会对它有更直观的感受。

5.1.1 异步和最终一致性 API

AdminClient 是异步的，它的每一个方法在向集群控制器发出请求后会立即返回一个或多个 `Future` 对象。`Future` 对象是异步操作的结果，提供了用于检查异步操作状态、取消操作、等待操作完成以及在操作完成后执行其他函数的一些方法。AdminClient 会将 `Future` 对象封装到 `Result` 对象中，`Result` 对象提供了一些方法以用于等待操作完成和执行常见的后续操作。例如，`KafkaAdminClient.createTopics` 会返回 `CreateTopicsResult` 对象，你可以用它来等待所有主题创建完成、检查每个主题的状态，并在主题创建成功之后获取主题的配置信息。

在 Kafka 中，从控制器到 broker 的元数据传播是异步的，所以当控制器状态被完全更新时，AdminClient API 返回的 `Future` 将被视为已完成。这个时候，并不是每个 broker 都知道发生了状态变更，所以 `listTopics` 请求可能会由不包含最新创建主题的 broker 负责处理。这叫作**最终一致性**：最终每个 broker 都会知道每一个主题的存在，但我们不能保证是在什么时候。

5.1.2 配置参数

AdminClient 的每一个方法都会接受一个特定于该方法的 `Options` 对象作为参数。例如，`listTopics` 方法的参数是 `ListTopicsOptions` 对象，`describeCluster` 的参数是 `DescribeClusterOptions` 对象。这些对象包含了 broker 将如何处理请求的配置参数。所有 AdminClient 方法都会有的一个配置参数是 `timeoutMs`：它指定了客户端在抛出 `TimeoutException` 之前等待集群返回响应的时间。这个参数用于限制应用程序被 AdminClient 操作阻塞的时间。其他参数还包括 `listTopics` 是否应该返回集群的内部主题，以及 `describeCluster` 是否应该返回客户端被授权了哪些集群操作。

5.1.3 扁平的结构

Kafka 协议支持的所有管理操作都可以直接用 `KafkaAdminClient` 来实现。AdminClient 的接口没有对象层次结构或命名空间，这有点儿争议，因为这个接口有很多方法，但这么做最主要的好处是，如果你想知道如何通过编程的方式执行 Kafka 管理操作，那么只需要搜索一个 JavaDoc，无须担心忘记搜索哪里了，而且 IDE 可以自动完成代码，非常方便。如果你要找的东西不在 AdminClient 中，则说明它还没有被实现（但欢迎贡献者参与贡献）。

如果你有兴趣为 Kafka 做贡献，那么可以查看一下我们的贡献指南。在对 Kafka 架构或协议做出重要的修改之前，先从较小的、没有争议的缺陷修复和改进开始。也鼓励大家参与非代码贡献，比如缺陷报告、文档改进、回答问题和撰写博文。

5.1.4 额外的话

所有修改集群状态的操作（创建、删除和更改）都是由控制器来处理的。读取集群状态的操作（列出清单和描述）可以由任意一个 broker 来处理，而且一般会被重定向到负载最低的 broker（基于客户端所知道的信息）。这个不影响我们使用 API，但当你遇到一些意想不到的行为，或者看到一些操作成功而另一些操作失败，或者想弄清楚为什么某个操作花费了很长时间时，知道这个很有好处。

在撰写本章内容的时候（Kafka 2.5 即将发布），大多数操作可以通过 AdminClient 或者直接修改 ZooKeeper 中的集群元数据来完成。但不建议直接修改 ZooKeeper，如果不得不这么做，那么请把这个当成 bug（缺陷）报告给 Kafka 项目组。这是因为在不久的将来，Kafka 社区将会移除对 ZooKeeper 的依

赖，每一个通过修改 ZooKeeper 实现管理操作的应用程序也都会做出修改，但 AdminClient API 将保持不变，只是在 Kafka 集群内部采用了不同的实现。

5.2 AdminClient 生命周期：创建、配置和关闭

要使用 Kafka 的 AdminClient，首先需要构造一个 AdminClient 对象。

```
Properties props = new Properties();
props.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
AdminClient admin = AdminClient.create(props);
// TODO: 用AdminClient做一些有用的事情
admin.close(Duration.ofSeconds(30));
```

`create` 静态方法会接受一个包含配置参数的 `Properties` 对象作为参数。唯一必须提供的配置参数是集群的 URI：用逗号分隔的要连接的 broker 地址清单。通常，在生产环境中至少需要指定 3 个 broker 地址，以防其中一个不可用。第 11 章将介绍如何配置带有身份验证的安全连接。

如果你启动了一个 AdminClient，那么到最后总是要关闭它。需要注意的是，在调用 `close` 方法时，可能还会有一些正在执行中的 AdminClient 操作。为此，`close` 方法提供了一个超时参数。一旦调用了 `close` 方法，就不能再调用其他方法或发送其他请求，客户端会一直等待响应，直到超时。超时之后，客户端会异常中止所有正在执行中的操作并释放资源。如果没有指定超时时间，则客户端会一直等待所有正在执行中的操作完成。

你可能还记得第 3 章和第 4 章分别介绍过 `KafkaProducer` 和 `KafkaConsumer` 的很多配置参数。AdminClient 要比它们简单得多，没有太多需要配置的参数。你可以在 Kafka 文档中看到所有的配置参数。接下来的内容中列出了几个重要的配置参数。

5.2.1 `client.dns.lookup`

这个参数是在 Kafka 2.1.0 中引入的。

在默认情况下，Kafka 根据连接串中提供的主机名来验证、解析和创建连接（然后是根据 `advertised.listeners` 指定的监听器监听的主机名）。这种方式在大多数情况下没有问题，但不适用于两个重要的场景：使用 DNS 别名以及单个 DNS 映射了多个 IP 地址。二者看起来差不多，但还是略有不同。下面让我们更详细地了解一下这两个互斥场景。

01. 使用 **DNS** 别名

    假设你有多个 broker，它们的命名规则是这样的：`broker1.hostname.com`、`broker2.hostname.com`，以此类推。与其在连接串中指定所有这些地址（维护起来比较困难），不如创建一个 DNS 别名，将其与所有的主机地址映射起来。可以用 `all-brokers.hostname.com` 作为连接串，因为你并不关心哪个 broker 将接受来自客户端的初始连接。这样做非常方便，除非你启用了 SASL 身份验证。如果使用了 SASL，那么客户端会尝试对 `all-brokers.hostname.com` 进行身份验证，而实际的服务器主体可能是 `broker2.hostname.com`。如果名字不匹配，则 SASL 将拒绝身份验证（broker 认证有可能是中间人攻击），进而连接失败。

    对于这个场景，需要配置 `client.dns.lookup=resolve_canonical_bootstrap_servers_only`，客户端将会“展开”DNS 别名，其效果与在连接串中指定所有的 broker 主机地址一样。

02. 单个 **DNS** 映射了多个 **IP** 地址

    在现代网络架构中，通常会将所有的 broker 放在代理或负载均衡器后面。这在 Kubernetes 集群中尤为常见，因为 Kubernetes 集群需要用负载均衡器来转发外部连接。在这些场景中，我们不希望负载均衡器成为单点故障点。因此，常见的做法是将 `broker1.hostname.com` 指向一组 IP 地址，也就是负载均衡器的 IP 地址，它们总是能够将流量路由到相同的 broker 上。这些 IP 地址可能会发生变化。

在默认情况下，Kafka 客户端会尝试连接第一个 IP 地址。如果这个 IP 地址不可用，那么客户端就无法连接到 broker，即使 broker 是可用的。因此，建议配置 `client.dns.lookup=use_all_dns_ips`，确保客户端可以充分利用高可用负载均衡器所带来的好处。

5.2.2 `request.timeout.ms`

这个参数指定了应用程序等待 AdminClient 返回响应的时间，包括客户端收到可重试错误时进行重试的时间。

这个参数的默认值是 120 秒，时间有点儿长，但确实有一些 AdminClient 操作，特别是消费者群组操作，需要较长的时间。正如 5.1 节中提到的，每个 AdminClient 方法都会接受一个 `Options` 对象作为参数，这个对象可以包含超时时间。如果 AdminClient 操作位于应用程序的关键路径上，你可能就会想要设置短一点儿的超时时间，并通过其他方式处理 Kafka 未能及时做出响应的问题。一个常见的例子是，应用程序会在第一次启动时尝试验证主题是否存在，如果 Kafka 在 30 秒内未能做出响应，你可能就会选择继续启动应用程序，稍后再来验证主题是否存在（或者完全跳过验证）。

5.3 基本的主题管理操作

我们已经创建并配置了一个 AdminClient 对象，接下来看看可以用它做些什么。AdminClient 最常见的应用是主题管理，包括列出主题、描述主题、创建主题和删除主题。

先来列出集群的所有主题。

```
ListTopicsResult topics = admin.listTopics();
topics.names().get().forEach(System.out::println);
```

需要注意的是，admin.listTopics() 返回的 ListTopicsResult 是对一组 Future 对象的包装器。topics.name() 返回的是一组主题名字的 Future 对象。当调用 Future 的 get() 方法时，执行线程将会等待，直到服务器返回一组主题名字或抛出超时异常。在获得主题名字列表后，我们会将它们逐个打印出来。

现在，尝试做一些更有用的事情：检查主题是否存在，如果不存在，就创建一个。要检查主题是否存在，一种方法是获取所有主题的名字，然后检查你需要的主题是否在列表中。在大型集群中，这么做可能效率低下，况且有时候你可能不只想检查主题是否存在，还想知道主题分区和副本的数量是否正确。例如，Connect 和 Confluent 模式注册表就使用一个主题来保存配置信息，在启动时，它们会检查这个主题是否存在、主题是否只有一个分区（确保配置变更严格有序）、是否有 3 个副本（确保可用性），以及是否被压实（及时更新旧的配置信息）。

```
DescribeTopicsResult demoTopic = admin.describeTopics(TOPIC_LIST); ❶

try {
    topicDescription = demoTopic.values().get(TOPIC_NAME).get(); ❷
    System.out.println("Description of demo topic:" + topicDescription);

    if (topicDescription.partitions().size() != NUM_PARTITIONS) { ❸
      System.out.println("Topic has wrong number of partitions. Exiting.");
      System.exit(-1);
    }
} catch (ExecutionException e) { ❹
    // 对于大部分异常，提前退出
    if (! (e.getCause() instanceof UnknownTopicOrPartitionException)) {
        e.printStackTrace();
        throw e;
    }

    // 如果执行到这里，则说明主题不存在
    System.out.println("Topic " + TOPIC_NAME +
        " does not exist. Going to create it now");
    // 需要注意的是，分区和副本数是可选的
    // 如果没有指定，那么将使用broker的默认配置
    CreateTopicsResult newTopic = admin.createTopics(Collections.singletonList(
            new NewTopic(TOPIC_NAME, NUM_PARTITIONS, REP_FACTOR))); ❺

    // 检查主题是否已创建成功：
    if (newTopic.numPartitions(TOPIC_NAME).get() != NUM_PARTITIONS) { ❻
        System.out.println("Topic has wrong number of partitions.");
        System.exit(-1);
    }
}
```

❶ 为了检查主题是否配置正确，可以调用 describeTopics() 方法，并传入一组想要验证的主题名字作为参数。它会返回 DescribeTopicResult 对象，这个对象对 map（主题名字到 Future 的映射）进行了包装。

❷ 如果一直等待 Future 完成，那么可以调用 get() 得到想要的结果——在这里是一个 TopicDescription 对象。但服务器也可能无法正确处理请求——如果主题不存在，那么服务器就不会返回我们想要的结果。在这种情况下，服务器将返回一个错误，Future 将抛出

ExecutionException，这个异常是服务器返回的错误导致的。因为我们想要处理主题不存在的情况，所以需要处理这些异常。

❸ 如果主题存在，那么 Future 将返回一个 TopicDescription 对象，其中包含了主题的所有分区、分区首领所在的 broker、副本清单和同步副本清单。需要注意的是，这个对象并不包含主题的配置信息。本章后面会讨论配置管理。

❹ 如果 Kafka 返回错误，那么所有的 AdminClient 结果对象都会抛出 ExecutionException 异常，因为 AdminClient 结果对 Future 对象进行了包装，而 Future 又对异常进行了包装，所以需要检查 ExecutionException 的嵌套异常才能获取到 Kafka 返回的错误信息。

❺ 如果主题不存在，就创建一个新主题。在创建主题时，可以只指定主题名字，其他参数使用默认值。当然，也可以指定分区数量、副本数量和其他配置参数。

❻ 最后，等待主题创建完成并验证结果。这里只检查分区数量。因为在创建主题时指定了分区数量，所以要确保它是对的。如果你在创建主题时使用了 broker 默认配置，则更加需要验证结果。再次调用 get() 获取 CreateTopic 的结果，这个方法也有可能抛出异常，最常见的是 TopicExistsException，我们需要处理这个异常。

现在我们有了一个主题，接下来把它删除。

```
admin.deleteTopics(TOPIC_LIST).all().get();

// 检查主题是否已删除
// 需要注意的是，由于删除是异步操作，这个时候主题可能还存在
try {
    topicDescription = demoTopic.values().get(TOPIC_NAME).get();
    System.out.println("Topic " + TOPIC_NAME + " is still around");
} catch (ExecutionException e) {
    System.out.println("Topic " + TOPIC_NAME + " is gone");
}
```

上面的代码你应该非常熟悉了。我们调用 deleteTopics，并传给它要删除的主题的名字，然后调用 get() 等待操作完成。

虽然代码很简单，但需要注意的是，在 Kafka 中，删除主题是最终的操作，因为没有回收站或垃圾桶可用于拯救被删除的主题，也没有特定的检查来验证主题是否为空以及你是否真的想要删除它们。误删其他主题也就意味着造成了无法恢复的数据丢失，因此在删除主题时要格外小心。

到目前为止，所有的示例都调用了 Future 对象的 get() 方法。大多数时候，这些已经足够了，毕竟管理操作不会太多，而且等待操作完成或超时都是可接受的。但有一种情况例外：你正在向一个需要处理大量管理请求的服务器写入数据。在这种情况下，你并不希望在等待 Kafka 返回响应的同时阻塞服务器线程，而是希望服务器继续接受来自客户端的请求，然后将请求发送给 Kafka，等 Kafka 返回响应后再将响应发送给客户端。这个时候，KafkaFuture 会非常有用。下面是一个简单的例子。

```
vertx.createHttpServer().requestHandler(request -> { ❶
    String topic = request.getParam("topic"); ❷
    String timeout = request.getParam("timeout");
    int timeoutMs = NumberUtils.toInt(timeout, 1000);

    DescribeTopicsResult demoTopic = admin.describeTopics( ❸
            Collections.singletonList(topic),
            new DescribeTopicsOptions().timeoutMs(timeoutMs));

    demoTopic.values().get(topic).whenComplete( ❹
            new KafkaFuture.BiConsumer<TopicDescription, Throwable>() {
               @Override
               public void accept(final TopicDescription topicDescription,
                                  final Throwable throwable) {
```

```
                    if (throwable != null) {
                       request.response().end("Error trying to describe topic "
                               + topic + " due to " + throwable.getMessage()); ❺
                    } else {
                        request.response().end(topicDescription.toString()); ❻
                    }
                }
            });
}).listen(8080);
```

❶ 用 Vert.x 创建一个简单的 HTTP 服务器。服务器在收到请求时会调用我们定义的 `requestHandler`。

❷ 请求当中包含了一个主题名字，我们将用这个主题的描述信息作为响应。

❸ 像往常一样调用 `AdminClient.describeTopics`，并得到一个包装好的 `Future` 对象。

❹ 这里没有调用 `get()` 方法，而是构造了一个函数，`Future` 在完成时会调用这个函数。

❺ 如果 `Future` 抛出异常，就将错误返回给 HTTP 客户端。

❻ 如果 `Future` 顺利完成，就将主题描述信息返回给客户端。

这里的关键在于我们不会等待 Kafka 返回响应。当 Kafka 返回响应时，`DescribeTopicResult` 会将响应返回给 HTTP 客户端。与此同时，HTTP 服务器可以继续处理其他请求。你可以尝试用 `SIGSTOP` 暂停 Kafka（不要在生产环境中这么做），并向 Vert.x 发送两个 HTTP 请求：一个超时时间较长，一个超时时间较短。虽然你是在第一个请求之后发送的第二个请求，但是因为超时时间较短，第二个请求会更早收到响应，而且不会被阻塞在第一个请求之后。

5.4 配置管理

配置管理是通过描述和更新一系列配资资源（ConfigResource）来实现的。配置资源可以是 broker、broker 日志记录器和主题。我们通常会用 kafka-config.sh 或其他 Kafka 管理工具来检查和修改 broker 及 broker 日志配置，但主题配置管理是在应用程序中完成的。

例如，很多应用程序使用了压实的主题，它们会定期（为安全起见，要比默认的保留期限更加频繁一些）检查主题是否被压实，如果没有，就采取相应的行动来纠正主题配置。

请看下面的例子。

```
ConfigResource configResource =
        new ConfigResource(ConfigResource.Type.TOPIC, TOPIC_NAME); ❶
DescribeConfigsResult configsResult =
        admin.describeConfigs(Collections.singleton(configResource));
Config configs = configsResult.all().get().get(configResource);
// 打印非默认配置
configs.entries().stream().filter(
        entry -> !entry.isDefault()).forEach(System.out::println); ❷

// 检查主题是否被压实
ConfigEntry compaction = new ConfigEntry(TopicConfig.CLEANUP_POLICY_CONFIG,
        TopicConfig.CLEANUP_POLICY_COMPACT);
if (!configs.entries().contains(compaction)) {
    // 如果主题没有被压实，就将其压实
    Collection<AlterConfigOp> configOp = new ArrayList<AlterConfigOp>();
    configOp.add(new AlterConfigOp(compaction, AlterConfigOp.OpType.SET)); ❸
    Map<ConfigResource, Collection<AlterConfigOp>> alterConf = new HashMap<>();
    alterConf.put(configResource, configOp);
    admin.incrementalAlterConfigs(alterConf).all().get();
} else {
    System.out.println("Topic " + TOPIC_NAME + " is compacted topic");
}
```

❶ 如上所述，ConfigResource 有几种类型，这里检查的是主题配置。也可以在同一个请求中指定多个不同类型的资源。

❷ describeConfigs 的结果是一个 map（从 ConfigResource 到配置的映射）。每个配置项都有一个 isDefault() 方法，可以让我们知道哪些配置被修改了。如果用户为主题配置了非默认值，或者修改了 broker 级别的配置，而创建的主题继承了 broker 的非默认配置，那么我们便能知道这个配置不是默认的。

❸ 为了修改配置，这里指定了需要修改的 ConfigResource 和一组操作。每个修改操作都由一个配置条目（配置的名字和值，此处名字是 cleanup.policy，值是 compacted）和操作类型组成。Kafka 的 4 种操作类型分别是：SET（用于设置值）、DELETE（用于删除值并重置为默认值）、APPEND 和 SUBSTRACT。后两种只适用于 List 类型的配置，用于向列表中添加值或从列表中移除值，这样就不用每次都把整个列表发送给 Kafka 了。

在紧急情况下，我们可以非常方便地获取到配置信息。记得有一次，我们在升级期间不小心用损坏的 broker 配置文件替换了正确的文件。直到重启第一个 broker 失败，我们才发现配置文件是错的。我们无法恢复原始文件，只能进行大量的试错，希望能够找出正确的配置。最后，一位站点可靠性工程师（SRE）通过连接到还没有被重启的 broker 并使用 AdminClient 转储了原先的配置解决了这个问题。

5.5 消费者群组管理

之前提到过，与其他大多数消息队列不同，Kafka 允许按照之前读取和处理数据的顺序重新处理数据。第 4 章介绍过消费者群组，解释了如何用消费者 API 回到之前的位置重新读取主题中的旧数据。但是，使用 API 意味着你需要事先在应用程序中写好重新处理数据的逻辑，也就是说，应用程序本身必须内置“重新处理”的功能。

但是，在某些情况下，即使应用程序没有预先内置这个功能，你也希望它们能够重新处理消息。一种情况是在发生事故期间对应用程序进行诊断，另一种情况是在故障转移过程中准备在新集群中运行应用程序（第 9 章在讨论灾难恢复技术时将更详细地探讨这方面的内容）。

本节将介绍如何使用 AdminClient 以编程的方式查看和修改消费者群组以及消费者群组提交的偏移量。第 10 章将介绍可用于实现同样操作的外部工具。

5.5.1 查看消费者群组

如果想查看和修改消费群组，那么第一步是将它们列出来。

```
admin.listConsumerGroups().valid().get().forEach(System.out::println);
```

需要注意的是，通过调用 `valid()` 方法，可以让 `get()` 返回的消费者群组只包含由集群正常返回的消费者群组。错误都将被忽略，不作为异常抛出。可以用 `errors()` 方法获取所有的异常。如果调用了 `all()` 方法，则只有集群返回的第一个错误会作为异常抛出。这类错误有可能是因没有查看群组的权限或某些群组协调器不可用导致的。

如果想了解更多有关某些群组的信息，那么可以用如下信息描述它们。

```
ConsumerGroupDescription groupDescription = admin
        .describeConsumerGroups(CONSUMER_GRP_LIST)
        .describedGroups().get(CONSUMER_GROUP).get();
        System.out.println("Description of group " + CONSUMER_GROUP
                + ":" + groupDescription);
```

描述信息中包含了大量有关群组的信息，包括群组成员、它们的标识符和主机地址、分配给它们的分区、分配分区的算法以及群组协调器的主机地址。在对消费者群组进行故障诊断时，这些描述信息非常有用。但这里缺了一个比较重要的信息——我们想要知道消费者群组最后消费的每个分区的偏移量以及相比最新消息滞后了多少。

在过去，获取这些信息唯一的方法是解析消费者群组写到内部主题的偏移量消息。虽然这种方法可以实现我们的目的，但 Kafka 不保证内部消息格式是兼容的，因此不建议使用。下面来看看 AdminClient 是如何获取这些信息的。

```
Map<TopicPartition, OffsetAndMetadata> offsets =
        admin.listConsumerGroupOffsets(CONSUMER_GROUP)
                .partitionsToOffsetAndMetadata().get(); ❶

Map<TopicPartition, OffsetSpec> requestLatestOffsets = new HashMap<>();

for(TopicPartition tp: offsets.keySet()) {
    requestLatestOffsets.put(tp, OffsetSpec.latest()); ❷
}

Map<TopicPartition, ListOffsetsResult.ListOffsetsResultInfo> latestOffsets =
        admin.listOffsets(requestLatestOffsets).all().get();

for (Map.Entry<TopicPartition, OffsetAndMetadata> e: offsets.entrySet()) { ❸
    String topic = e.getKey().topic();
    int partition =  e.getKey().partition();
    long committedOffset = e.getValue().offset();
    long latestOffset = latestOffsets.get(e.getKey()).offset();
```

```
    System.out.println("Consumer group " + CONSUMER_GROUP
            + " has committed offset " + committedOffset
            + " to topic " + topic + " partition " + partition
            + ". The latest offset in the partition is "
            + latestOffset + " so consumer group is "
            + (latestOffset - committedOffset) + " records behind");
}
```

❶ 获取消费者群组读取的所有主题和分区，以及每个分区最新的提交偏移量。与 `describeConsumerGroups` 不同，`listConsumerGroupOffsets` 只接受一个消费者群组而不是一个集合作为参数。

❷ 我们希望获取到结果集中每一个分区最后一条消息的偏移量。`OffsetSpec` 提供了 3 个非常方便的实现：`earliest()`、`latest()` 和 `forTimestamp()`，分别用于获取分区中最早和最近的偏移量，以及在指定时间或紧接在指定时间之后写入的消息的偏移量。

❸ 最后，遍历所有分区，将最近提交的偏移量、分区中最近的偏移量以及它们之间的差值打印出来。

5.5.2 修改消费者群组

到目前为止，我们都只是查看消费者群组的信息。AdminClient 也提供了修改消费者群组的方法：删除群组、移除成员、删除提交的偏移量和修改偏移量。SRE 通常用它们来构建临时工具，以在紧急情况下进行故障恢复。

在所有这些方法当中，修改偏移量是最有用的。删除偏移量似乎就是让消费者“从头开始”读取数据，但实际上也取决于消费者的配置——如果消费者在启动时没有可用的偏移量，那么是从头开始读取还是直接跳到最新的位置开始读取呢？除非配置了 `auto.offset.reset`，否则就无从得知。显式地将提交的偏移量修改为最早的偏移量，可以强制消费者从主题开头位置开始读取，实际上就是“重置”消费者。

需要注意的是，当偏移量主题中的偏移量发生变化时，消费者群组并不会收到通知。它们只在分配了新分区或启动时读取偏移量主题。为了防止在消费者无法知晓的情况下修改偏移量（会导致偏移量被覆盖），Kafka 不允许在消费者群组处于活动状态时修改偏移量。

另外，如果消费者应用程序会维护状态（大多数流式处理应用程序会维护状态），那么重置偏移量并让消费者群组从主题的开始位置读取数据可能会对保存的状态造成奇怪的影响。假设你的一个应用程序一直在统计商店已售出的鞋子的数量，你在早上 8 点时发现输入数据中有一个错误，并希望从凌晨 3 点开始重新计算。如果你将偏移量重置到凌晨 3 点，却没有对保存的状态做出相应的修改，那么今天售出的每只鞋子都会被计算两次（假设处理凌晨 3 点到 8 点的数据是纠正错误所必需的)。你需要相应地更新保存的状态。在开发环境中，我们通常会在重置偏移量之前将保存的状态完全删除。

下面来看一个例子。

```
Map<TopicPartition, ListOffsetsResult.ListOffsetsResultInfo> earliestOffsets =
    admin.listOffsets(requestEarliestOffsets).all().get(); ❶

Map<TopicPartition, OffsetAndMetadata> resetOffsets = new HashMap<>();
for (Map.Entry<TopicPartition, ListOffsetsResult.ListOffsetsResultInfo> e:
        earliestOffsets.entrySet()) {
  resetOffsets.put(e.getKey(), new OffsetAndMetadata(e.getValue().offset())); ❷
}

try {
  admin.alterConsumerGroupOffsets(CONSUMER_GROUP, resetOffsets).all().get(); ❸
} catch (ExecutionException e) {
  System.out.println("Failed to update the offsets committed by group "
            + CONSUMER_GROUP + " with error " + e.getMessage());
  if (e.getCause() instanceof UnknownMemberIdException)
     System.out.println("Check if consumer group is still active."); ❹
}
```

❶ 要重置消费者群组，并让它从最早的偏移量位置开始消费，需要先获取最早的偏移量。获取最早的偏移量与获取最新的偏移量差不多，如上一个示例所示。

❷ 在这个循环中，将 `listOffsets` 返回的 `ListOffsetsResultInfo` 转成 `alterConsumerGroupOffsets` 需要的 `OffsetAndMetadata`。

❸ 调用 `alterConsumerGroupOffsets` 之后等待 `Future` 完成，这样便可知道是否执行成功。

❹ 导致 `alterConsumerGroupOffsets` 执行失败最常见的一个原因是没有停止消费者群组（只能直接关闭消费者应用程序，因为没有可用于关闭消费者群组的命令）。如果消费者群组仍然处于活跃状态，那么一旦修改了偏移量，群组协调器就会认为有非群组成员正在提交偏移量，并抛出 `UnknownMemberIdException` 异常。

5.6 集群元数据

大多数时候，应用程序并不需要知道它所连接的集群的信息，你甚至可以在不知道有多少个 broker 以及哪个 broker 是控制器的情况下生产和消费消息。Kafka 客户端抽离了这些信息——应用程序只需要关心主题和分区。

但如果你感到好奇，那么下面这一小段代码或许可以满足你的好奇心。

```
DescribeClusterResult cluster = admin.describeCluster();

System.out.println("Connected to cluster " + cluster.clusterId().get()); ❶
System.out.println("The brokers in the cluster are:");
cluster.nodes().get().forEach(node -> System.out.println("    * " + node));
System.out.println("The controller is: " + cluster.controller().get());
```

❶ 集群标识符是一个 GUID，人类不可读，但仍然可用于检查客户端是否连接到了正确的集群。

5.7 高级的管理操作

本节将讨论一些很少被用到的方法，使用这些方法可能存在风险，但在必要的时候它们非常有用。这些方法对 SRE 来说是最重要的，但不要等到发生事故了才开始了解它们。趁还来得及，可以多了解、多实践。需要注意的是，虽然这些方法属于同一类操作，但彼此之间没有什么联系。

5.7.1 为主题添加分区

通常，分区数量是在创建主题时就设置好的。而且，由于每一个分区都可以有非常高的吞吐量，因此我们很少会遇到主题容量瓶颈。此外，如果主题中的消息包含了键，则消费者可以假设包含了相同键的消息总是会被写入相同的分区，这些消息可以由同一个消费者按照它们被写入的顺序来读取。

因此，我们很少需要为主题添加分区，而且添加分区可能存在风险。你需要确保这个操作不会影响正在消费该主题的应用程序。但是，有时候确实会遇到已有分区的吞吐量达到上限的情况，这个时候则别无选择，只能添加分区。

可以用 `createPartitions` 方法为主题添加分区。需要注意的是，如果一次性为多个主题添加分区，则可能会出现一些主题添加成功一些主题添加失败的情况。

```
Map<String, NewPartitions> newPartitions = new HashMap<>();
newPartitions.put(TOPIC_NAME, NewPartitions.increaseTo(NUM_PARTITIONS+2)); ❶
admin.createPartitions(newPartitions).all().get();
```

❶ 在添加分区时，需要指定添加分区后主题将拥有的分区总数，而不是要添加的新分区的数量。

因为传给 `createPartitions` 方法的参数是添加分区以后主题所拥有的分区总数，所以在添加分区之前可能需要先查看一下已有的分区数。

5.7.2 从主题中删除消息

隐私保护条例规定了特定数据的保留策略。可惜的是，虽然 Kafka 提供了主题数据保留策略，但并不支持按照合规性来保留数据。一个保留策略为 30 天的主题，如果所有分区中的数据都可以放在一个日志片段中，那么这个主题就可以保留超过 30 天的数据。

`deleteRecords` 方法会将所有偏移量早于指定偏移量的消息标记为已删除，使消费者无法读取这些数据。这个方法将返回被删除的消息的最大偏移量，这样我们就可以检查删除操作是否按预期执行了。从磁盘上彻底删除数据是异步进行的。需要注意的是，可以用 `listOffsets` 方法获取在特定时间点或之后写入的消息的偏移量。也可以组合使用这些方法来删除早于任意特定时间点的消息。

```
Map<TopicPartition, ListOffsetsResult.ListOffsetsResultInfo> olderOffsets =
        admin.listOffsets(requestOlderOffsets).all().get();
Map<TopicPartition, RecordsToDelete> recordsToDelete = new HashMap<>();
for (Map.Entry<TopicPartition, ListOffsetsResult.ListOffsetsResultInfo>  e:
        olderOffsets.entrySet())
    recordsToDelete.put(e.getKey(),
            RecordsToDelete.beforeOffset(e.getValue().offset()));
admin.deleteRecords(recordsToDelete).all().get();
```

5.7.3 首领选举

首领选举有以下两种类型。

首选首领选举

每一个分区都有一个可以被指定为首选首领的副本。如果所有分区的首领都是它们的首选首领副本，那么每个 broker 上的首领数量应该是均衡的。在默认情况下，Kafka 每 5 分钟会检查一次首领是否就是首选首领副本，如果不是，但它有资格成为首领，就会选择首选首领副本作为首领。如果 `auto.leader.rebalance.enable` 被设置为 `false`，或者你想快一点儿执行选举，则可以调用 `electLeader()` 方法。

不彻底的首领选举

如果一个分区的首领副本变得不可用，而其他副本没有资格成为首领（通常是因为缺少数据），那么这个分区将没有首领，也就不可用了。解决这个问题的一种方法是触发不彻底的首领选举，也就是选举一个本来没有资格成为首领的副本作为首领。这可能导致数据丢失——所有写入旧首领但未被复制到新首领的消息都将丢失。`electLeader()` 方法也可以用来触发不彻底的首领选举。

这个方法是异步的，也就是说，即使它成功返回，也需要一段时间才能让所有 broker 都知道发生了首领选举，在这期间调用 `describeTopics()` 可能会返回不一致的结果。如果你触发了多个分区的首领选举，则可能会导致一些分区执行成功，一些分区执行失败。

```
Set<TopicPartition> electableTopics = new HashSet<>();
electableTopics.add(new TopicPartition(TOPIC_NAME, 0));
try {
    admin.electLeaders(ElectionType.PREFERRED, electableTopics).all().get(); ❶
} catch (ExecutionException e) {
    if (e.getCause() instanceof ElectionNotNeededException) {
        System.out.println("All leaders are preferred already"); ❷
    }
}
```

❶ 这里选举的是某个特定主题的某个分区的首选首领。我们可以指定任意数量的分区和主题。如果在调用这个方法时传入 `null` 而不是分区列表，则它将触发所有分区的首领选举。

❷ 如果集群的状态是健康的，那么这个方法将不执行任何操作。首选首领选举和不彻底的首领选举只在当前首领不是首选首领副本时才有效。

5.7.4 重新分配副本

有时候，你可能不喜欢某些副本所在的位置。例如，一个 broker 超载了，你可能想移走一些副本；或者你可能想增加更多的副本；或者你可能想将一个 broker 所有的副本都移走，以便关闭整台机器；又或者可能有一些主题太过繁杂，你想将它们与其他主题隔离开来。在这些场景中，可以用 `alterPartitionReassignments` 更细粒度地控制分区的每个副本要放在什么位置。需要注意的是，将副本从一个 broker 重新分配给另一个 broker 可能需要复制大量的数据。所以要注意可用的网络带宽，并在必要时使用配额限制数据复制。配额是 broker 级别的配置，可以用 `AdminClient` 来查看和修改它们。

在下面的例子中，假设有一个 ID 为 0 的 broker。我们的主题有几个分区，每个分区都有一个副本位于这个 broker 上。在添加了一个新 broker 后，我们想用它来保存主题的一些副本，因此将通过一种略微不同的方式来分配主题分区。

```
Map<TopicPartition, Optional<NewPartitionReassignment>> reassignment = new Hash-
Map<>();
reassignment.put(new TopicPartition(TOPIC_NAME, 0),
        Optional.of(new NewPartitionReassignment(Arrays.asList(0,1)))); ❶
reassignment.put(new TopicPartition(TOPIC_NAME, 1),
        Optional.of(new NewPartitionReassignment(Arrays.asList(1)))); ❷
reassignment.put(new TopicPartition(TOPIC_NAME, 2),
        Optional.of(new NewPartitionReassignment(Arrays.asList(1,0)))); ❸
reassignment.put(new TopicPartition(TOPIC_NAME, 3), Optional.empty()); ❹

admin.alterPartitionReassignments(reassignment).all().get();

System.out.println("currently reassigning: " +
```

```
        admin.listPartitionReassignments().reassignments().get()); ❺
demoTopic = admin.describeTopics(TOPIC_LIST);
topicDescription = demoTopic.values().get(TOPIC_NAME).get();
System.out.println("Description of demo topic:" + topicDescription); ❻
```

❶ 为分区 0 添加一个副本，并将新副本放在 ID 为 1 的新 broker 上，但保持首领不变。

❷ 没有为分区 1 添加副本，只是将现有的副本移动到新 broker 上。因为只有一个副本，所以它也是首领。

❸ 为分区 2 添加一个副本，并将它作为首选首领。下一次首选首领选举将选举新 broker 上的这个新副本作为新首领。现有的副本将成为跟随者。

❹ 分区 3 没有进行中的重分配，如果有，则会被取消，并且状态会被恢复到重分配操作开始之前的样子。

❺ 可以列出正在进行的重分配。

❻ 还可以打印出新的分区状态，但需要注意的是，可能需要等一会儿才能获取到最终的状态。

5.8 测试

Kafka 提供了一个测试类 MockAdminClient，你可以用它来初始化任意数量的 broker，并对它执行管理操作，以便在没有真实 Kafka 集群的情况下测试应用程序的行为是否正确。MockAdminClient 不是 Kafka API 的一部分，所以其发生变更时并不会告知用户，但它模拟的方法是公开的，方法签名将保持兼容。需要注意的是，这个类可能会修改和破坏你的测试，所以你需要在便利性和测试风险之间做出一些权衡。

这个测试类吸引人的地方是对一些常用方法进行了非常全面的模拟：你可以用 MockAdminClient 创建一个主题，然后调用 listTopics()，它将返回你“创建”的主题。

但 MockAdminClient 并没有模拟所有的方法。如果使用 2.5 或更早版本的 AdminClient，那么调用 MockAdminClient 的 incrementalAlterConfigs() 方法将抛出 UnsupportedOperationException。你可以注入自己的实现类来解决这个问题。

为了演示如何使用 MockAdminClient，下面来实现一个类，并用 AdminClient 创建主题。

```
public TopicCreator(AdminClient admin) {
    this.admin = admin;
}

// 如果指定的主题名字以“test”开头，就创建这个主题
public void maybeCreateTopic(String topicName)
        throws ExecutionException, InterruptedException {
    Collection<NewTopic> topics = new ArrayList<>();
    topics.add(new NewTopic(topicName, 1, (short) 1));
    if (topicName.toLowerCase().startsWith("test")) {
        admin.createTopics(topics);

        // 为了演示而修改配置
        ConfigResource configResource =
                  new ConfigResource(ConfigResource.Type.TOPIC, topicName);
        ConfigEntry compaction =
                  new ConfigEntry(TopicConfig.CLEANUP_POLICY_CONFIG,
                        TopicConfig.CLEANUP_POLICY_COMPACT);
        Collection<AlterConfigOp> configOp = new ArrayList<AlterConfigOp>();
        configOp.add(new AlterConfigOp(compaction, AlterConfigOp.OpType.SET));
        Map<ConfigResource, Collection<AlterConfigOp>> alterConf =
            new HashMap<>();
        alterConf.put(configResource, configOp);
        admin.incrementalAlterConfigs(alterConf).all().get();
    }
}
```

这里的逻辑并不复杂：如果主题名字以“test”开头，那么 maybeCreateTopic 方法就会创建这个主题。我们还修改了主题配置，以便演示如何处理模拟客户端没有实现某些方法的情况。

我们使用 Mockito 测试框架来验证 MockAdminClient 方法是否按预期被调用，并填充未实现的方法。Mockito 是一个相当简单的模拟测试框架，提供了非常好的 API，非常适用于小型的单元测试。

下面从实例化模拟客户端开始我们的测试。

```
@Before
public void setUp() {
    Node broker = new Node(0,"localhost",9092);
    this.admin = spy(new MockAdminClient(Collections.singletonList(broker),
        broker)); ❶

    // 如果没有这个，那么该测试将抛出
    // java.lang.UnsupportedOperationException: Not implemented yet异常
```

```
    AlterConfigsResult emptyResult = mock(AlterConfigsResult.class);
    doReturn(KafkaFuture.completedFuture(null)).when(emptyResult).all();
    doReturn(emptyResult).when(admin).incrementalAlterConfigs(any()); ❷
}
```

❶ 我们用一个 broker 列表（这里只用了一个 broker）初始化了 `MockAdminClient`，并指定了控制器。broker 的 ID、主机名和端口都是假的。在执行这些测试时并不会真的运行 broker。我们使用了 Mockito 的“间谍”注入，以便稍后可以检查 `TopicCreator` 是否被正确执行。

❷ 这里调用 Mockito 的 `doReturn` 方法来确保模拟客户端不会抛出异常。我们正在测试的方法期望 `AlterConfigResult` 对象返回一个 `KafkaFuture`，所以要确保 `incrementalAlterConfigs` 返回的就是这个。

现在，我们有了一个想要的假 AdminClient，接下来就可以用它来测试 `maybeCreateTopic()` 方法了。

```
@Test
public void testCreateTestTopic()
        throws ExecutionException, InterruptedException {
    TopicCreator tc = new TopicCreator(admin);
    tc.maybeCreateTopic("test.is.a.test.topic");
    verify(admin, times(1)).createTopics(any()); ❶
}

@Test
public void testNotTopic() throws ExecutionException, InterruptedException {
    TopicCreator tc = new TopicCreator(admin);
    tc.maybeCreateTopic("not.a.test");
    verify(admin, never()).createTopics(any()); ❷
}
```

❶ 主题名字以“test”开头，所以 `maybeCreateTopic()` 应该要创建这个主题。然后，验证 `createTopics()` 被调用过一次。

❷ 如果主题名字不是以“test”开头，就验证 `createTopics()` 没有被调用。

最后需要注意的是：`MockAdminClient` 位于 Kafka 的一个测试 JAR 包中，所以 pom.xml 中要包含这个依赖项。

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

5.9 小结

AdminClient 是 Kafka 开发工具包中一个非常有用的工具。应用程序开发人员可以用它动态创建主题，并验证主题的配置是否符合要求。SRE 可以用它创建基于 Kafka 的工具、实现自动化或进行故障恢复。AdminClient 提供了很多有用的方法，SRE 可以把它看作管理 Kafka 的“瑞士军刀”。

本章介绍了 AdminClient 的基础知识：主题管理、配置管理、消费者群组管理，以及其他一些有用的方法。可以事先了解这些方法，因为你永远不知道什么时候会用到它们。

第 6 章　深入 Kafka

如果只是在生产环境中运行 Kafka 或使用 Kafka 开发应用程序，则并非一定要了解 Kafka 的内部工作原理。不过，了解 Kafka 的内部工作原理有助于理解 Kafka 的行为和诊断问题。本章并不会涵盖 Kafka 的每一个设计和实现细节，而是集中讨论以下几个与 Kafka 使用者密切相关的话题。

- Kafka 控制器
- Kafka 的复制原理
- Kafka 如何处理来自生产者和消费者的请求
- Kafka 的存储细节，比如文件格式和索引

在对 Kafka 进行调优时，深入理解这些问题是很有必要的。了解了内部机制，就可以更精确地进行深层调优，而不是随意而为。

6.1 集群的成员关系

Kafka 使用 ZooKeeper 维护集群的成员信息。每个 broker 都有一个唯一的标识符，这个标识符既可以在配置文件中指定，也可以自动生成。broker 在启动时通过创建 ZooKeeper **临时节点**把自己的 ID 注册到 ZooKeeper 中。broker、控制器和其他的一些生态系统工具会订阅 ZooKeeper 的 /brokers/ids 路径（也就是 broker 在 ZooKeeper 上的注册路径），当有 broker 加入或退出集群时，它们可以收到通知。

如果你试图启动另一个具有相同 ID 的 broker，则会收到一个错误——新 broker 会尝试进行注册，但不会成功，因为 ZooKeeper 中已经有一个相同的节点。

当 broker 与 ZooKeeper 断开连接（通常会在关闭 broker 时发生，但在发生网络分区或长时间垃圾回收停顿时也会发生）时，它在启动时创建的临时节点会自动从 ZooKeeper 上移除。监听 broker 节点路径的 Kafka 组件会被告知这个 broker 已被移除。

broker 对应的 ZooKeeper 节点会在 broker 被关闭之后消失，但它的 ID 会继续存在于其他数据结构中。例如，每个主题的副本集（参见 6.3 节）中就可能包含这个 ID。在完全关闭一个 broker 后，如果使用相同的 ID 启动另一个全新的 broker，则它会立即加入集群，并获得与之前相同的分区和主题。

6.2 控制器

控制器其实也是一个 broker，只不过除了提供一般的 broker 功能之外，它还负责选举分区首领。集群中第一个启动的 broker 会通过在 ZooKeeper 中创建一个名为 `/controller` 的临时节点让自己成为控制器。其他 broker 在启动时也会尝试创建这个节点，但它们会收到“节点已存在”异常，并“意识”到控制器节点已存在，也就是说集群中已经有一个控制器了。其他 broker 会在控制器节点上创建 ZooKeeper watch，这样就可以收到这个节点的变更通知了。我们通过这种方式来确保集群中只有一个控制器。

如果控制器被关闭或者与 ZooKeeper 断开连接，那么这个临时节点就会消失。控制器使用的 ZooKeeper 客户端没有在 `zookeeper.session.timeout.ms` 指定的时间内向 ZooKeeper 发送心跳是导致连接断开的原因之一。当临时节点消失时，集群中的其他 broker 将收到控制器节点已消失的通知，并尝试让自己成为新的控制器。第一个在 ZooKeeper 中成功创建控制器节点的 broker 会成为新的控制器，其他节点则会收到“节点已存在”异常，并会在新的控制器节点上再次创建 ZooKeeper watch。每个新选出的控制器都会通过 ZooKeeper 条件递增操作获得一个数值更大的 epoch。其他 broker 也会知道当前控制器的 epoch，如果收到由控制器发出的包含较小 epoch 的消息，就会忽略它们。这一点很重要，因为控制器会因长时间垃圾回收停顿与 ZooKeeper 断开连接——在停顿期间，新控制器将被选举出来。当旧控制器在停顿之后恢复时，它并不知道已经选出了新的控制器，并会继续发送消息——在这种情况下，旧控制器会被认为是一个“僵尸控制器”。消息里的 epoch 可以用来忽略来自旧控制器的消息，这是防御“僵尸”的一种方式。

控制器必须先从 ZooKeeper 加载最新的副本集状态，然后才能开始管理集群元数据和执行首领选举。这个加载过程使用了异步 API，为了减少延迟，请求会以管道的形式发送给 ZooKeeper。但即便如此，在有大量分区的集群中，加载过程可能仍然需要几秒——Kafka 1.1.0 的主页为此提供了几个测试基准和对比结果。

当控制器发现有一个 broker 离开了集群（通过观察相关的 ZooKeeper 路径或收到了一个来自 broker 的 `ControlledShutdownRequest`）时，它知道，原先首领位于这个 broker 上的所有分区需要一个新首领。它将遍历所有需要新首领的分区，并决定应该将哪个分区作为新首领（简单一点儿，它可能就是副本集中的下一个副本）。然后，它会将更新后的状态持久化到 ZooKeeper 中（同样，为了减少延迟，以管道的方式异步发送请求），再向所有包含这些分区副本的 broker 发送一个 `LeaderAndISR` 请求，请求中包含了新首领和跟随者的信息。出于效率方面的考虑，这些请求会被分成批次，所以每个请求都包含了多个分区最新的成员关系信息（这些分区在相同的 broker 上都有一个副本）。每一个新首领都知道自己要开始处理来自生产者和消费者的请求，而跟随者也知道它们要开始从新首领那里复制消息。集群中的每一个 broker 都有一个 `MetadataCache`，其中包含了一个保存所有 broker 和副本信息的 map。控制器通过 `UpdateMetadata` 请求向所有 broker 发送有关首领变更的信息，broker 会在收到请求后更新缓存。在启动 broker 副本时也会有类似的过程——主要的区别是 broker 所有的分区副本都是跟随者，并且需要在自己有资格被选为首领之前与首领保持同步。

总的来说，Kafka 会使用 ZooKeeper 的临时节点来选举控制器，并会在 broker 加入或退出集群时通知控制器。控制器负责在 broker 加入或退出集群时进行首领选举。控制器会使用 epoch 来避免“脑裂”。所谓的“脑裂”，就是指两个 broker 同时认为自己是集群当前的控制器。

新控制器 KRaft

2019 年，Kafka 社区启动了一个雄心勃勃的项目：使用基于 Raft 的控制器替换基于 ZooKeeper 的控制器。新控制器叫作 KRaft，其预览版包含在 Kafka 2.8 中。于 2021 年 9 月发布的 Kafka 3.0 包含了它的第一个生产版本，Kafka 集群既可以使用基于 ZooKeeper 的传统控制器，也可以使用 KRaft。

为什么 Kafka 社区决定替换控制器？传统控制器经过了好几次重写，尽管在使用 ZooKeeper 保存主题、分区和副本信息方面有所改进，但已有的模型无法支持我们希望 Kafka 支持的分区数量。下面的这些顾虑是导致这一变更的动因。

- 虽然元数据更新是同步写入 ZooKeeper 的，但是异步发送给 broker 的。此外，从 ZooKeeper 接收更新也是异步的。所有这些都有可能导致 broker、控制器和 ZooKeeper 之间的元数据出现不一致。一旦出现这些情况，将很难检测到。

- 控制器在重新启动时需要从 ZooKeeper 读取所有的 broker 和分区的元数据，然后再将它们发送给所有的 broker。尽管经过了多年的努力，这个过程仍然是一个主要瓶颈——随着分区和 broker 数量的增加，重启控制器的速度将会更慢。
- 元数据所有权关系的内部架构不够好——有些操作是通过控制器完成的，有些操作是通过 broker 完成的，还有一些操作是直接通过修改 ZooKeeper 完成的。
- 与 Kafka 一样，ZooKeeper 本身就是一个分布式系统，要求开发者具备一定的专业知识。因此，想要用好 Kafka 的开发者需要了解两个分布式系统，而不是一个。

鉴于以上原因，Kafka 社区决定替换掉基于 ZooKeeper 的控制器。

在现有架构中，ZooKeeper 起到了两个重要作用：一是用于选举控制器，二是用于保存集群元数据（broker、配置、主题、分区和副本）。此外，控制器本身也需要管理元数据——用于选举首领、创建和删除主题，以及重新分配副本。所有这些功能在新控制器中都将被替换掉。

新控制器背后的核心设计思想是：Kafka 本身有一个基于日志的架构，其中用户会将状态的变化表示成一个事件流。开发社区对这种表示非常熟悉——多个消费者可以通过重放事件快速赶上最新的状态。日志保留了事件之间的顺序，并能确保消费者始终沿着单个时间轴移动。新控制器架构为 Kafka 的元数据管理带来了同样的好处。

在新架构中，控制器节点形成了一个 Raft 仲裁，管理着元数据事件日志。这个日志中包含了集群元数据的每一个变更。原先保存在 ZooKeeper 中的所有东西（比如主题、分区、ISR、配置等）都将被保存在这个日志中。

因为使用了 Raft 算法，所以控制器节点可以在不依赖外部系统的情况下选举首领。首领节点被称为**主控制器**，负责处理所有来自 broker 的 RPC 调用。跟随者控制器会从主控制器那里复制数据，并会作为主控制器的热备。因为控制器会跟踪最新的状态，所以当发生控制器故障转移时（在此期间，所有的状态都将被转移给新控制器），很快就可以完成状态的重新加载。

其他 broker 将通过新的 `MetadataFetch`API 从主控制器获取更新，而不是让主控制器将更新发送给它们。与获取请求类似，broker 会跟踪它们已经获取到的最新的元数据偏移量，并只向主控制器请求较新的元数据。broker 会将元数据持久化到磁盘上，这可以实现快速启动，即使有数百万个分区。

broker 会将自己注册到控制器仲裁上，在管理员将其注销之前一直保持注册状态，即使被关闭并离线，仍然是注册状态。那些在线但没有保持最新元数据的 broker 将被隔离，不能处理来自客户端的请求。这样可以防止客户端向已经过时很久但还不知道自己已经不是首领的非首领节点发送消息。

作为迁移到控制器仲裁的一部分，之前所有涉及直接与 ZooKeeper 通信的客户端和 broker 操作都将通过控制器来路由。这样就可以通过替换控制器来进行无缝的迁移，无须对 broker 做出任何修改。

新架构的整体设计请参见 KIP-500。Raft 协议的细节请参见 KIP-595。新控制器仲裁的详细设计，包括控制器配置和用于与集群元数据交互的新命令行工具，请参见 KIP-631。

6.3 复制

复制是 Kafka 架构核心的一部分。Kafka 经常被描述成“一个分布式、分区、可复制的提交日志服务”。复制之所以这么重要，是因为它可以在个别节点失效时仍能保证 Kafka 的可用性和持久性。

前面介绍过，Kafka 的数据保存在主题中，每个主题被分成若干个分区，每个分区可以有多个副本。副本保存在 broker 上，每个 broker 可以保存成百上千个主题和分区的副本。

副本有以下两种类型。

首领副本

每个分区都有一个首领副本。为了保证一致性，所有生产者请求和消费者请求都会经过这个副本。客户端可以从首领副本或跟随者副本读取数据。

跟随者副本

首领以外的副本都是跟随者副本。如果没有特别指定，则跟随者副本将不处理来自客户端的请求，它们的主要任务是从首领那里复制消息，保持与首领一致的状态。如果首领发生崩溃，那么其中的一个跟随者就会被提拔为新首领。

从跟随者副本读取数据

KIP-392 中加入了从跟随者副本读取数据的特性。这个特性的主要目的是允许客户端从最近的同步副本而不是首领副本读取数据，以此来降低网络流量成本。要使用这个特性，消费者端需要配置可以标识客户端位置的 `client.rack`。broker 端需要配置 `replica.selector.class`，默认为 `LeaderSelector`（表示总是从首领副本读取数据），可以把它设置为 `RackAwareReplicaSelector`，它将选择一个最近的副本，这个副本所在的 broker 的 `rack.id` 与客户端配置的 `client.rack` 相匹配。也可以实现 `ReplicaSelector` 接口，使用自定义副本选择逻辑。

复制协议经过了扩展，保证从跟随者副本读取的消息都是已提交的。也就是说，即使是从跟随者副本读取数据，仍然可以获得与之前一样的可靠性保证。为了提供这种保证，所有副本都需要知道首领提交了哪些消息。为此，首领在发送给跟随者的数据中加入了当前高水位标记（最近提交的偏移量）。传输高水位标记会导致一些延迟，也就是说，从跟随者副本读取到可用数据将比从首领副本读取晚一些。如果要减少消费者延迟，则需要从首领副本读取数据。

首领的另一项任务是搞清楚哪些跟随者副本的状态与自己是一致的。为了保持与首领同步，跟随者会尝试从首领那里复制消息，但它们可能会因为各种原因无法与首领保持同步。例如，网络拥塞导致复制变慢，或者 broker 发生崩溃并重启，都将导致所有副本的复制滞后。

为了与首领保持同步，跟随者需要向首领发送 `Fetch` 请求，这与消费者为了读取消息而发送的请求是一样的。作为响应，首领会将消息返回给跟随者。`Fetch` 请求消息里包含了跟随者想要获取的消息的偏移量，这些偏移量总是有序的。这样，首领就可以知道一个副本是否已经获取了最近一条消息之前的所有消息。通过检查每个副本请求的最后一个偏移量，首领就可以知道每个副本的滞后程度。如果副本没有在 30 秒内发送请求，或者即使发送了请求但与最新消息的间隔超过了 30 秒，那么它将被认为是**不同步**的。如果一个副本未能跟上首领，那么一旦首领发生故障，它将不能再成为新首领——毕竟，它并未拥有所有的消息。

与此相反，持续发出获取最新消息请求的副本被称为**同步副本**。当首领发生故障时，只有同步副本才有资格被选为新首领。

允许跟随者可以多久不活跃或允许跟随者在多久之后成为不同步副本是通过 `replica.lag.time.max.ms` 参数来配置的。这个时间直接影响首领选举期间的客户端行为和数据保留机制。第 7 章在介绍可靠性保证时将深入探讨这个话题。

除了当前的首领，每个分区都有一个**首选首领**，即创建主题时选定的首领。之所以是首选的，是因为在创建分区时，分区首领在 broker 间的分布已经是均衡的。因此，我们希望当首选首领成为当前首领时，broker 之间的负载是均衡的。在默认情况下，Kafka 的 `auto.leader.rebalance.enable` 会被设置为 `true`，它会检查首选首领是不是当前首领以及是不是同步的。如果是同步的，但不是当前首领，就会触发首领选举，让首选首领成为当前首领。

找到首选首领

可以很容易地从分区的副本列表中找到首选首领（可以用 `kafka.topics.sh` 查看分区和副本的详细信息。第 13 章将介绍这个工具以及其他管理工具）。列表中的第一个副本一般就是首选首领，不管当前首领是哪一个副本，或者使用副本分配工具将副本重新分配给了其他 broker，这一点都不会改变。需要注意的是，如果你手动重新分配了副本，那么第一个指定的副本就是首选首领。因此，要确保首选首领被分配给不同的 broker，避免出现少部分包含了首领的 broker 负载过重，其他 broker 却无法为它们分担负载的情况。

6.4 处理请求

broker 的大部分工作是处理客户端、分区副本和控制器发送给分区首领的请求。Kafka 提供了一种二进制协议（基于 TCP），指定了请求消息的格式以及 broker 如何对请求做出响应——既包括成功处理请求，也包括在处理请求过程中出现错误。

Kafka 项目提供了由 Kafka 项目贡献者实现和维护的 Java 客户端，除此之外，还有其他使用不同的编程语言（比如 C 语言、Python、Go 语言等）开发的客户端。你可以在 Kafka 官方网站上看到完整的清单。这些客户端都使用这个协议与 broker 通信。

客户端总是发起连接并发送请求，而 broker 负责处理这些请求并做出响应。broker 会按照请求到达的顺序来处理它们，这种顺序既能保证让 Kafka 具备消息队列的特性，又能保证保存的消息是有序的。

所有的请求消息都有标头，其中包含了如下信息。

- 请求类型。
- 请求版本（broker 可以处理来自不同版本客户端的请求，并根据不同的版本做出不同的响应）。
- 关联 ID——一个用于唯一标识请求消息的数字，同时也会出现在响应消息和错误日志里（可用于诊断问题）。
- 客户端 ID——用于标识发送请求的客户端应用程序。

本书不打算在这里对这个协议进行过多介绍，因为 Kafka 文档已经对它有很详细的说明了。不过，了解 broker 如何处理请求还是有用处的。后面在讨论如何监控 Kafka 和各种监控配置选项时，会介绍那些与队列和线程相关的指标和配置参数。

broker 会在它监听的每一个端口上运行一个**接收器线程**，这个线程会创建一个连接，并把它交给**处理器线程**处理。处理器线程（也叫**网络线程**）的数量是可配置的。网络线程负责从客户端获取请求，把它们放进**请求队列**，然后从**响应队列**取出响应，把它们发送给客户端。有时候，服务器端需要延迟对客户端做出响应，例如，消费者要求只在有可用数据时接收响应，或者发出 `DeleteTopic` 请求的客户端要求在开始删除主题之后才接收响应。延迟的响应会被放在**炼狱**（临时内存）中，直到它们可以被发送给客户端。图 6-1 为 Kafka 内部的请求处理流程。

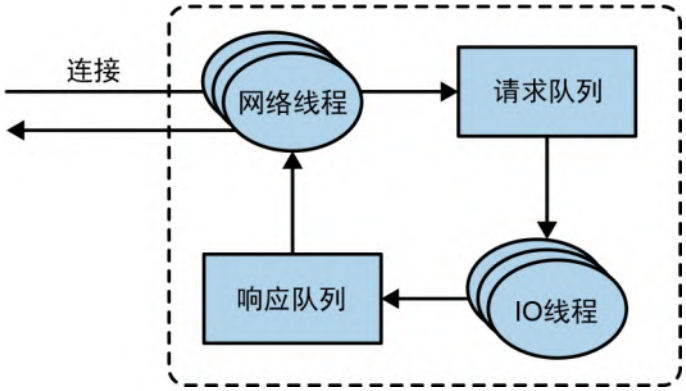

图 6-1：Kafka 内部的请求处理流程

请求消息被放入请求队列后，IO 线程（也叫**请求处理线程**）会负责处理它们。下面是几种最常见的请求类型。

生产请求

生产者发送的请求，其中包含了客户端要写入 broker 的消息。

获取请求

消费者和跟随者副本发送的请求，用于从 broker 读取消息。

管理请求

管理客户端发送的请求，用于执行元数据操作，比如创建和删除主题。

生产请求和获取请求都必须发送给分区的首领。如果 broker 收到一个针对某个分区的写入请求，而这个分区的首领在另一个 broker 上，那么发送请求的客户端将收到“非分区首领”错误响应。如果针对某个分区的读取请求被发送到一个不包含这个分区首领的 broker 上，那么也会收到同样的错误。Kafka 客户端负责把生产请求和获取请求发送到包含分区首领的 broker 上。

那么客户端怎么知道该向哪里发送请求呢？客户端使用了另一种请求类型，也就是**元数据请求**，请求中包含了客户端感兴趣的主题清单。这种请求的响应消息里指明了这些主题所包含的分区、每个分区都有哪些副本，以及哪个副本是首领。元数据请求可以被发送给任意一个 broker，因为所有 broker 都缓存了这些元数据信息。

一般情况下，客户端会把这些信息缓存起来，并直接向目标 broker 发送生产请求和获取请求。它们需要时不时地通过发送元数据请求来刷新缓存（刷新的时间间隔可以通过 `metadata.max.age.ms` 参数来配置），以便知道元数据是否发生了变化，比如，在新 broker 加入集群时，部分副本会被移动到新 broker 上（参见图 6-2）。另外，如果客户端收到“非分区首领”错误，那么它会在重新发送请求之前刷新元数据，因为这个错误说明客户端正在使用过期的元数据。

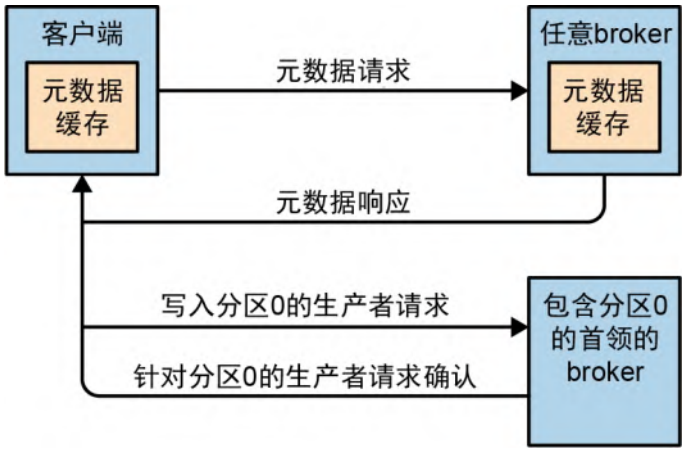

图 6-2：客户端请求路由

6.4.1 生产请求

第 3 章在介绍如何配置生产者时提到过 `acks` 这个配置参数，该参数指定了需要有多少个 broker 确认才能认为一个消息写入是成功的。生产者“写入成功”的判定条件根据配置的不同而不同。如果 `acks=1`，那么只要首领确认收到消息就算写入成功；如果 `acks=all`，则需要所有同步副本确认收到消息才算写入成功；如果 `acks=0`，则生产者只管发送消息，完全不需要等待 broker 返回响应。

首先，包含某个分区首领的 broker 在收到生产请求时会对请求做一些验证。

- 发送数据的用户是否有主题写入权限？
- 请求中指定的 `acks` 是否有效（只允许出现 `0`、`1` 或 `all`）？
- 如果 `acks=all`，那么是否有足够多的同步副本保证消息已经被安全写入？（可以将 broker 配置成如果同步副本的数量达不到指定的值就拒绝处理新消息。第 7 章在介绍 Kafka 的持久性和可靠性保证时将讨论这方面的更多细节。）

然后，消息将被写入本地磁盘。在 Linux 系统中，消息会被写入文件系统缓存，但不能保证何时会被冲刷到磁盘上。Kafka 不会一直等待数据被持久化到磁盘上，它主要通过复制功能来保证消息的持久性。

一旦消息被写入分区的首领，broker 就会检查 `acks` 配置参数——如果 `acks` 是 `0` 或 `1`，那么 broker 就会立即返回响应；如果 `acks` 是 `all`，则请求将被保存在一个叫作**炼狱**的缓冲区中，直到首领确认跟随者副本复制了消息，才将响应返回给客户端。

6.4.2 获取请求

broker 处理获取请求的方式与处理生产请求的方式很相似。客户端发送请求，希望 broker 返回指定主题、分区和特定偏移量位置的消息，就好像在说：“请把主题 Test 的分区 0 中偏移量从 53 开始的消息以及主题 Test 的分区 3 中偏移量从 64 开始的消息发给我。”客户端还可以指定一个分区最多可以返回多少数据。这个限制非常重要，因为客户端需要为 broker 返回的数据分配足够的内存。如果没有这个限制，并且 broker 返回了大量的数据，则可能会耗尽客户端的内存。

之前说过，请求需要发送给指定的分区首领，所以客户端需要通过查询元数据来确保请求被路由到正确的节点上。首领在收到请求时会先检查请求是否有效，比如，指定的偏移量在分区中是否存在？如果客户端请求的数据已被删除，或者请求的偏移量不存在，则 broker 会返回错误。

如果请求的偏移量存在，那么 broker 将按照客户端指定的数量上限从分区中读取消息，再把消息返回给客户端。Kafka 使用**零复制**技术向客户端发送消息，也就是说，Kafka 会直接把消息从文件（或者更确切地说是 Linux 文件系统缓存）里发送到网络通道，不需要经过任何中间缓冲区。这是 Kafka 与其他大部分数据库系统不一样的地方，其他数据库在将数据发送给客户端之前会先把它们保存在本地缓存中。这项技术避免了字节复制，也不需要管理内存缓冲区，从而能够获得更好的性能。

除了可以设置 broker 返回数据的上限，客户端也可以设置 broker 返回数据的下限。如果把下限设置为 10 KB，就好像是在告诉 broker“等到有 10 KB 数据时再把它们返回给我”。在主题消息流量不是很大的情况下，这样可以减少 CPU 和网络开销。具体操作如下：客户端发送一个请求，broker 在等到有足够数据时才把它们返回给客户端，然后客户端再发起另一个请求，而不是让客户端每隔几毫秒就发送一次请求，可能每次都只能拿到很少的数据，甚至没有数据（参见图 6-3）。对比这两种情况，虽然它们最终读取的数据总量是一样的，但前者的来回传送次数更少，开销也更小。

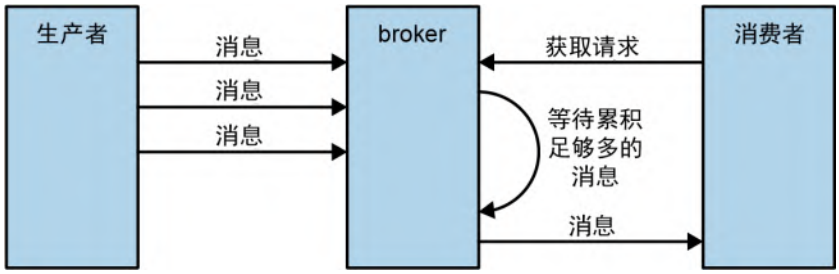

图 6-3：broker 延迟做出响应以便累积足够多的数据

当然，我们不会一直让客户端等待 broker 累积数据。客户端可以在等待了一段时间之后就开始处理可用的数据，而不是一直等待下去。所以，客户端可以定义一个超时时间，告诉 broker“如果你无法在 *x* 毫秒内累积足够多的数据，就把当前这些数据返回给我”。

有意思的是，并不是所有保存在分区首领上的数据都可以被客户端读取。大部分客户端只能读取已经被写入所有同步副本 [跟随者副本除外（尽管它们也是消费者），否则复制功能将无法正常工作] 的消息。分区首领知道哪些消息已经被复制到哪些副本上，所以消息在还没有被写入所有同步副本之前是不会被发送给消费者的——尝试获取这些消息的请求会得到空响应，而不是错误。

之所以这样，是因为还没有被足够多分区副本复制的消息被认为是“不安全”的——如果首领发生崩溃，另一个副本成为新首领，那么这些消息就丢失了。如果允许客户端读取只存在于首领中的消息，则可能会出现不一致的行为。试想，某个消费者先读取了一条消息，然后首领发生了崩溃，因为其他 broker 不包含这条消息，所以该消息就丢失了，其他消费者也就不可能读取到该消息，这样它们的行为与读取到该消息的消费者就会不一致。所以，我们会等到所有同步副本都复制了消息，才允许消费者读取它们（参见图 6-4）。这也意味着，如果 broker 间的消息复制因为某些原因变慢，那么消息到达消费者的时间也会变长

（因为会先等待消息复制完毕）。最大延迟时间可以通过参数 `replica.lag.time.max.ms` 来配置，它指定了分区副本在复制消息时最多出现多长的延迟仍然被认为是同步的。

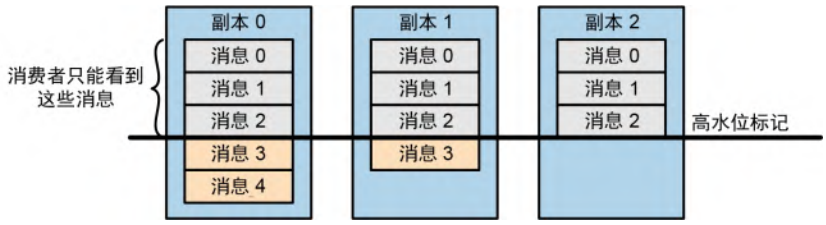

图 6-4：消费者只能看到已经被复制给同步副本的消息

在某些情况下，消费者需要读取大量的分区。在每个发送给 broker 的请求中都包含整个分区清单并让 broker 返回所有的元数据的做法是非常低效的，因为分区及其元数据其实很少会发生变化。所以，为了最小化这种开销，Kafka 提供了**请求会话缓存**。消费者可以尝试创建一个会话，会话中保存了它们正在读取的分区及其元数据信息。在创建了会话之后，消费者将不需要在每个请求中都指定分区，而是使用增量式请求。broker 只会在分区及其元数据发生变化时才将它们包含在响应中。不过，会话缓存的空间是有限的，Kafka 会优先保存需要处理大量分区的跟随者副本和消费者的会话，所以，在某些情况下，broker 可能不会创建会话，甚至会将会话清理掉。对于这两种情况，broker 将向客户端返回错误，客户端可以重新发起包含所有分区元数据的请求。

6.4.3 其他请求

我们已经介绍了 Kafka 客户端最常见的几种请求类型：元数据请求、生产请求和获取请求。Kafka 协议目前可以处理 61 种请求类型，后续将增加更多的请求类型。消费者可以使用 15 种请求类型来创建群组、协调消费，并允许开发人员管理消费者群组。除此之外，还有很多与元数据管理和安全相关的请求类型。

broker 之间也使用了同样的通信协议。这些请求发生在 Kafka 内部，客户端不应该使用它们。例如，当一个新首领被选举出来，控制器会将 `LeaderAndIsr` 请求发送给新首领（这样它就可以开始接收来自客户端的请求）和跟随者（这样它们就知道要开始跟随新首领）。

Kafka 的协议在持续演化——随着 Kafka 社区不断给客户端增加新功能，协议也要随之不断演进。例如，在过去，Kafka 消费者使用 ZooKeeper 来跟踪偏移量，消费者会在启动时检查保存在 ZooKeeper 上的偏移量，然后从这个位置开始处理消息。因为各种原因，社区决定不再使用 ZooKeeper 来保存偏移量，而是把偏移量保存在特定的 Kafka 主题上。为此，贡献者们不得不在协议里新增了几种请求类型：`OffsetCommitRequest`、`OffsetFetchRequest` 和 `ListOffsetsRequest`。现在，应用程序在提交偏移量时不会再把偏移量写入 ZooKeeper，而是会向 Kafka 发送 `OffsetCommitRequest` 请求。

过去，主题的创建需要通过命令行工具来完成，命令行工具会直接更新 ZooKeeper 中的主题列表。后来，Kafka 社区新增了 `CreateTopicRequest` 和其他用于管理元数据的请求类型。Java 应用程序通过调用 Kafka 的 `AdminClient`（第 5 章对其进行过深入介绍）来执行元数据操作。因为这些操作现在已经成为 Kafka 协议的一部分，所以那些使用不支持 ZooKeeper 的编程语言开发的 Kafka 客户端可以通过直接向 Kafka 发送请求来创建主题。

除了在协议中增加新的请求类型，我们也对已有的请求类型进行了修改，为它们添加了新的能力。例如，从 Kafka 0.9.0 到 0.10.0，我们希望能够让客户端知道谁是当前的控制器，于是把控制器信息添加到了元数据响应消息里。为此，我们在元数据请求和响应消息里增加了一个新的 version 字段。现在，0.9.0 版本客户端发送的元数据请求的 version 是 0（因为 version 1 在 0.9.0 版本客户端不存在）。不管是 0.9.0 版本的 broker，还是 0.10.0 版本的 broker，它们都知道应该返回 version 为 0 的响应，也就是不包含控制器信息。0.9.0 版本客户端不期待 broker 返回控制器信息，而且也不知道该如何解析它。0.10.0 版本客户端会发送 version 为 1 的元数据请求，0.10.0 版本的 broker 将返回 version 为 1 的响应，其中包含了控制器的信息。如果 0.10.0 版本客户端发送 version 为 1 的请求给 0.9.0 版本的 broker，broker 不知道该如何处理这个请求，就会返回错误。这就是为什么建议在升级客户端之前先升级 broker，因为新版 broker 知道如何处理旧请求，反过来则不然。

Kafka 社区在 Kafka 0.10.0 中加入了 `ApiVersionRequest`，其允许客户端询问 broker 都支持哪些版本，并使用相应的版本。有了这个功能，客户端就可以使用正确的请求类型版本与旧 broker 进行通信。增加新的 API 能让客户端知道 broker 都支持哪些特性，并让 broker 限定某个版本能够支持的特性，这项工作还在进行当中。有关这项改进的提议请参见 KIP-584。

6.5 物理存储

Kafka 的基本存储单元是分区。分区既无法在多个 broker 间再细分，也无法在同一个 broker 的多个磁盘间再细分。所以，分区的大小受单个挂载点可用空间的限制。（一个挂载点可以是单个磁盘或多个磁盘。如果配置了 JBOD，就是单个磁盘；如果配置了 RAID，就是多个磁盘。这方面的更多内容请参见第 2 章。）

在配置 Kafka 时，管理员会指定一个用于保存分区数据的目录列表，也就是 `log.dirs` 参数（不要把它与存放错误日志的目录混淆了，日志目录是配置在 log4j.properties 文件中的）。这个参数一般会包含 Kafka 将要使用的每一个挂载点的目录。

下面来看看 Kafka 是如何使用这些目录存储数据的。我们首先会介绍数据是如何被分配给 broker 以及 broker 目录的。然后会介绍 broker 是如何管理文件的，特别是如何根据策略保留数据。接下来会深入介绍文件和索引格式。最后会介绍日志压实及其工作原理。日志压实是 Kafka 的一个高级特性，有了这个特性，我们可以将 Kafka 作为长期的数据存储系统。

6.5.1 分层存储

从 2018 年年底开始，Kafka 社区启动了一个雄心勃勃的项目，为 Kafka 增加分层存储能力，并计划在 3.0 中发布。

这个项目的动机很简单：之所以使用 Kafka 存储海量数据，要么是因为吞吐量高，要么是因为需要长时间保留数据，但以下几点不容忽视。

- 一个分区可以存储的数据量是有限的。因此，分区数量不仅由产品需求驱动，也受物理磁盘大小的限制。
- 磁盘和集群大小的选择取决于存储需求。但是，如果将延迟和吞吐量作为主要考虑因素，那么集群的规模通常比实际需要的要大，从而增加了成本。
- 在 broker 间移动分区（当扩展或缩小集群时）所需要的时间是由分区大小决定的。大分区会降低集群的弹性。如今，我们可以充分利用灵活的云部署，所以在进行架构设计时会偏向于追求更大的弹性。

在分层存储架构中，Kafka 集群配置了两个存储层：本地存储层和远程存储层。本地存储层和当前的 Kafka 存储层一样，使用 broker 的本地磁盘存储日志片段，远程存储层则使用 HDFS、S3 等专用存储系统存储日志片段。

Kafka 用户可以单独为每一层配置保留策略。由于本地存储的成本通常远高于远程存储，因此本地存储的数据保留时间通常是几小时，甚至更短，而远程存储的保留时间则比较长，可以是几天，甚至几个月。

本地存储的延迟明显低于远程存储。对延迟敏感的应用程序通常从本地存储的分区尾部读取数据，因此可以受益于现有的 Kafka 存储机制，比如可以有效地利用页面缓存。在进行数据回填或故障恢复时，应用程序需要用到旧数据，所以需要从远程存储读取。

分层存储架构让 Kafka 集群的存储扩展可以独立于内存和 CPU，因此可以将 Kafka 作为一种长期的存储解决方案。这既减少了存储在 broker 上的数据量，也减少了在进行故障恢复和再均衡时需要复制的数据量。远程存储中的日志片段不需要恢复到 broker 上，当然，如果有必要也可以进行按需恢复。因为不是所有的数据都存储在 broker 上，所以要延长集群数据保留时间就不再需要扩展集群存储或添加新节点。与此同时，延长系统总体数据保留时间也无须像其他系统那样使用单独的数据管道将数据从 Kafka 复制到外部存储。

KIP-405 详细描述了分层存储的设计，包括新引入的组件 `RemoteLogManager` 以及如何与现有功能（比如首领选举、副本如何与首领保持同步）交互。

KIP-405 还记录了一个有趣的测试结果，即分层存储对性能的影响。负责开发分层存储特性的团队针对几种不同的场景进行了性能测试。在第一个场景中使用 Kafka 通常的高吞吐量工作负载。在这个场景中，延迟增加了一点点（从 21 毫秒增加到了 25 毫秒），因为 broker 需要将日志片段发送给远程存储。第二个场景是让一些消费者读取旧数据。在没有使用分层存储的情况下，消费者读取旧数据对延迟有很大的影响

（21 毫秒对 60 毫秒），但在启用了分层存储后，影响显著降低（25 毫秒对 42 毫秒）。这是因为分层存储读取是通过网络从 HDFS 或 S3 读取的，不会与基于本地磁盘 I/O 或页面缓存的本地读取产生竞争，让页面缓存只保留新鲜的数据。

所以，除了无限存储、更低的成本和更高的弹性外，分层存储还提供了历史数据读取和实时数据读取之间的隔离。

6.5.2 分区的分配

在创建主题时，Kafka 首先要决定如何在 broker 间分配分区。假设你有 6 个 broker，打算创建一个包含 10 个分区的主题，并且复制系数为 3，那么总共会有 30 个分区副本，它们将被分配给 6 个 broker。在进行分区分配时，要达到以下这些目标。

- 在 broker 间平均分布分区副本。对我们的例子来说，就是要保证每个 broker 可以分到 5 个副本。
- 确保每个分区的副本分布在不同的 broker 上。假设分区 0 的首领在 broker 2 上，那么可以把跟随者副本分别放在 broker 3 和 broker 4 上，但不能放在 broker 2 上，也不能将两个都放在 broker 3 上。
- 如果为 broker 指定了机架信息（Kafka 0.10.0 及之后的版本才支持），那么尽可能把每个分区的副本分配给不同机架上的 broker。这样做是为了保证一个机架的不可用不会导致整体分区不可用。

为了实现这些目标，先随机选择一个 broker（假设是 4），然后使用轮询的方式给每个 broker 分配分区首领。于是，分区 0 的首领在 broker 4 上，分区 1 的首领在 broker 5 上，分区 2 的首领在 broker 0 上（因为只有 6 个 broker），以此类推。接下来，从分区首领开始，依次分配跟随者副本。如果分区 0 的首领在 broker 4 上，那么它的第一个跟随者副本就在 broker 5 上，第二个跟随者副本就在 broker 0 上。如果分区 1 的首领在 broker 5 上，那么它的第一个跟随者副本就在 broker 0 上，第二个跟随者副本在 broker 1 上。

如果配置了机架信息，那么就不是按照数字顺序而是按照机架交替的方式来选择 broker 了。假设 broker 0 和 broker 1 被放置在一个机架上，broker 2 和 broker 3 被放置在另一个机架上。我们不是按照从 0 到 3 的顺序来选择 broker，而是按照 0、2、1、3 的顺序来选择，以保证相邻的 broker 总是位于不同的机架上（参见图 6-5）。于是，如果分区 0 的首领在 broker 2 上，那么第一个跟随者副本就在 broker 1 上，以保证它们位于不同的机架上。因为如果第一个机架离线，则还有其他幸存的副本，所以分区仍然可用。这对所有副本来说都是一样的，因此在机架离线时仍然能够保证可用性。

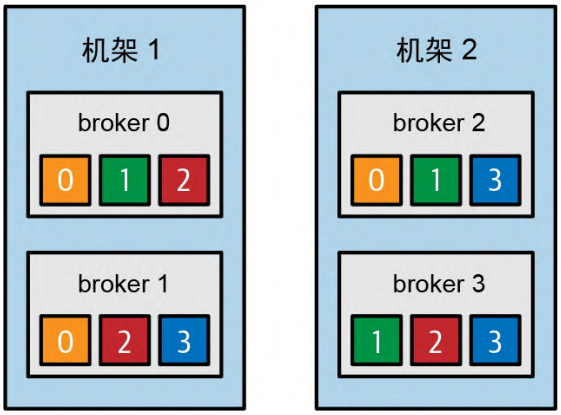

图 6-5：为位于不同机架上的 broker 分配分区和副本

为分区和副本选好合适的 broker 之后，接下来要决定新分区应该被放在哪个目录。我们会为每个分区分配目录，规则很简单：计算每个目录里的分区数量，新分区总是会被放在分区数量最少的那个目录。也就是说，如果添加了一个新磁盘，那么所有新分区都会被放到这个磁盘。这是因为在达到均衡分配的状态之前，新磁盘的分区数量总是最少的。

注意磁盘空间

需要注意的是，在将分区分配给 broker 时，我们不会考虑可用空间和工作负载，而在将分区分配给磁盘时会考虑分区数量，但不考虑分区大小。如果有些 broker 的磁盘空间比其他 broker 大（有可能是因为集群同时使用了旧服务器和新服务器）、有些分区特别大，或者同一个 broker 上有大小不同的磁盘，那么在分配分区时就要小心一些。

6.5.3 文件管理

数据保留是 Kafka 的一个重要概念。Kafka 不会一直保留数据，也不会一直等到消息被所有消费者读取了之后才将其删除。相反，Kafka 管理员会为每个主题配置数据保留期限，主题的数据要么在达到指定的时间之后被清除，要么在达到指定的数量之后被清除。

在一个大文件中查找和删除消息既费时又很容易出错，所以我们会把分区分成若干个**片段**。在默认情况下，每个片段包含 1 GB 或一周的数据，以较小的那个为准。在 broker 向分区写入数据时，如果触及任意一个上限，就关闭当前文件，并打开一个新文件。

当前正在写入数据的片段叫作**活动片段**。活动片段永远不会被删除，所以，如果你配置的保留时间是 1 天，但片段里包含了 5 天的数据，那么这些数据就会被保留 5 天，因为在片段被关闭之前，这些数据是不会被删除的。如果你要保留数据一周，并且每天使用一个新片段，那么每天就会有一个新片段被创建，同时最旧的一个片段会被删除，因此这个分区在大部分时间里会有 7 个片段。

第 2 章讲过，broker 会为分区的每一个打开的日志片段分配一个文件句柄，哪怕是非活动片段。这样就会打开很多文件句柄，因此必须根据实际情况对操作系统做一些调优。

6.5.4 文件格式

每个日志片段被保存在一个单独的数据文件中，文件中包含了消息和偏移量。保存在磁盘上的数据格式与生产者发送给服务器的消息格式以及服务器发送给消费者的消息格式是一样的。因为磁盘存储和网络传输采用了相同的格式，所以 Kafka 可以使用零复制技术向消费者发送消息，并避免对生产者压缩过的消息进行解压和再压缩。如果修改了消息格式，那么网络传输和磁盘保存的消息格式也需要修改，并且 broker 需要知道如何处理因升级导致文件中包含了两种格式的消息。

Kafka 消息由有效负载和系统标头组成。有效负载包括一个可选的键、值和一些可选的用户标头，其中每个标头也是一个键–值对。

从 Kafka 0.11 和 v2 版消息格式开始，生产者都是以批次的方式发送消息。如果每次只发送一条消息，那么使用批次反而会增加开销。但如果每次发送两条或更多的消息，那么使用批次就可以节约空间，减少网络带宽和磁盘的使用。这也是为什么配置了 `linger.ms=10`，Kafka 会表现得更好的一个原因——要求的延迟越小，消息被放在同一个批次里发送的可能性就越高。因为 Kafka 会为每个分区创建一个单独的批次，所以写入的分区越少，生产者的效率就越高。需要注意的是，生产者可以在同一个生产请求中包含多个批次。如果生产者端使用了压缩（推荐这么做），那么更大的批次将意味着不管是通过网络传输还是磁盘保存都能获得更好的压缩比。

消息批次的标头包括以下信息。

- 一个魔数，表示消息格式的版本。
- 批次第一条消息的偏移量以及与最后一条消息的偏移量的差值——即使之后在压实批次时可能会删除一些消息，这些也会一直保留下来。在生产者创建并发送批次时，第一条消息的偏移量会被设置为 0，第一个持久化这个批次的 broker（分区的首领）会用实际偏移量替换它。
- 批次第一条消息的时间戳和最大时间戳。如果设置的时间戳类型是追加时间而不是创建时间，那么可以由 broker 来设置时间戳。
- 批次大小，以字节为单位。
- 收到批次的 broker 首领的 epoch（用于在首领选举后截短消息，KIP-101 和 KIP-279 详细说明了用法）。
- 用于验证批次完整性的校验和。
- 表示不同属性的 16 位（bit）：压缩类型、时间戳类型（时间戳可以在客户端或 broker 端设置），以及批次是作为事务的一部分还是作为控制批次。
- 生产者 ID、生产者 epoch 和批次里的第一个序列——这些都用于实现精确一次性保证。

- 当然，还有组成批次的消息集合。

正如你所看到的，批次标头包含了很多信息。记录本身也有系统标头（不要与用户设置的标头相混淆）。每条记录包含以下信息。

- 记录大小，以字节为单位。
- 属性——目前没有记录级别的属性，所以这个标头没有被用到。
- 这条记录的偏移量与批次第一条消息的偏移量之间的差值。
- 这条记录的时间戳与批次第一条消息的时间戳之间的差值，以毫秒为单位。
- 有效载荷：键、值和用户设置的标头。

需要注意的是，每条记录的开销其实很小，而且大多数系统信息是批次级别的。标头中只保存批次第一条消息的偏移量和时间戳，每条记录中只保存差值，这极大地减少了每条记录的开销，从而让传输更大的批次变得更加高效。

除了包含用户数据的消息批次，Kafka 还有另一种批次类型，即控制批次，比如用于说明事务提交情况的批次。这些由消费者负责处理，不会传给用户应用程序。目前这种批次里包含了一个版本信息和一个类型指示器：0 表示中止的事务，1 表示提交的事务。

如果你想看到这些信息，那么可以使用 Kafka 的 `DumpLogSegment` 工具，该工具允许查看分区日志片段的内容。可以通过以下命令来运行这个工具。

```
bin/kafka-run-class.sh kafka.tools.DumpLogSegments
```

如果你指定了 `--deep-iteration` 参数，那么它将显示有关被压实的消息的信息。

向下转换消息格式

前面介绍的消息格式是在 Kafka 0.11 中引入的。因为 Kafka 支持在升级客户端之前升级 broker，所以它必须支持 broker、生产者和消费者之间的任意版本组合。大多数组合是没有问题的——新版本 broker 知道如何处理旧版本生产者发送的旧消息格式，新版本生产者在向旧版本 broker 发送消息时会使用旧消息格式。但是，当新版本生产者向新版本 broker 发送 v2 版本的消息时，消息将以 v2 格式保存，如果不支持 v2 消息格式的旧版本消费者试图读取这些消息，就会给 broker 带来一个难题。在这种情况下，broker 需要将消息从 v2 格式转换成 v1 格式，让旧版本消费者能够解析它们。这种转换将比平常消耗更多的 CPU 和内存，所以应该尽量避免。KIP-188 为此引入了几个重要的指标，包括 `FetchMessageConversionsPerSec` 和 `MessageConversionsTimeMs`。如果你还在使用旧版客户端，那么建议检查一下这些指标，并尽快升级客户端。

6.5.5 索引

消费者可以从 Kafka 任意可用的偏移量位置开始读取消息。假设消费者希望从偏移量 100 开始读取 1 MB 消息，那么 broker 就必须立即定位到偏移量 100（可能是在分区的任意一个片段里），然后从这个位置开始读取消息。为了帮助 broker 更快定位到指定的偏移量，Kafka 为每个分区维护了一个索引。该索引将偏移量与片段文件以及偏移量在文件中的位置做了映射。

类似地，Kafka 还有第二个索引，该索引将时间戳与消息偏移量做了映射。在按时间戳搜索消息时会用到这个索引。这种搜索方式在 Kafka Streams 中使用广泛，在一些故障转移场景中也很有用。

索引也会被分成片段，所以，在删除消息时也可以删除相应的索引。Kafka 没有为索引维护校验和。如果索引损坏，那么 Kafka 将通过重新读取消息并记录偏移量和位置来再次生成索引。如果有必要，管理员也可以删除索引，这样做绝对安全（尽管可能需要较长的恢复时间），因为 Kafka 会自动重新生成索引。

6.5.6 压实

一般情况下，Kafka 会根据设置的时间来保留数据，把超过时效的旧数据删除。但是，请试想一种场景，假设你用 Kafka 来保存客户的收货地址，那么保存客户的最新地址比保存客户上周甚至去年的地址更有意义，这样你就不用保留客户的旧地址了。另外一种场景是应用程序使用 Kafka 来保存它的当前状态，每次状态发生变化，就将新状态写入 Kafka。当应用程序从故障中恢复时，它会从 Kafka 读取之前保存的消息，以便恢复到最近的状态。应用程序只关心发生崩溃前的那个状态，并不关心在运行过程中发生的所有状态变化。

Kafka 通过改变主题的保留策略来满足这些应用场景。如果保留策略是 `delete`，那么早于保留时间的旧事件将被删除；如果保留策略是 `compact`（压实），那么只为每个键保留最新的值。很显然，只有当应用程序生成的事件里包含了键–值对时，设置 `compact` 才有意义。如果主题中包含了 `null` 键，那么这个策略就会失效。

主题的数据保留策略也可以被设置成 `delete.and.compact`，也就是以上两种策略的组合。超过保留时间的消息将被删除，即使它们的键对应的值是最新的。组合策略可以防止压实主题变得太大，同时也可以满足业务需要在一段时间后删除数据的要求。

6.5.7 压实的工作原理

每个日志片段可以分为以下两个部分（参见图 6-6）。

干净的部分

这些消息之前被压实过，每个键只有一个对应的值，这个值是上一次压实时保留下来的。

浑浊的部分

这些消息是在上一次压实之后写入的。

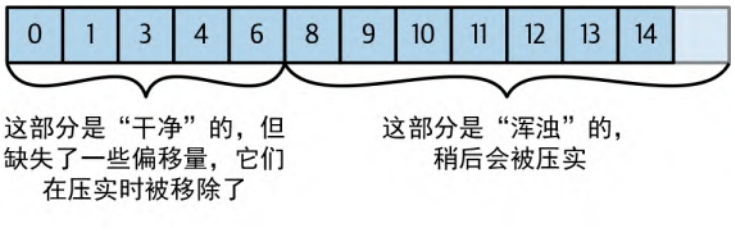

图 6-6：包含干净和浑浊部分的分区

如果启用了压实功能（通过配置 `log.cleaner.enabled` 参数来开启），那么 broker 在启动时会创建一个压实管理器线程和一些压实工作线程来执行压实任务。这些线程会选择浑浊率（浑浊的消息占分区总体消息的比例）最高的分区来压实。

为了压实分区，压实线程会读取分区的浑浊部分，并在内存中创建一个 map。map 的每个元素都包含消息键的哈希值（16 字节）和上一条具有相同键的消息的偏移量（8 字节）。也就是说，每个 map 的元素只占用 24 字节的内存。假设要压实一个 1 GB 的日志片段，每条消息大小为 1 KB，总共有 100 万条消息，那么只需要用 24 MB 的 map 就可以压实这个片段。（如果有重复的键，则可以重用哈希项，从而使用更少的内存。）这个效率是非常高的。

Kafka 管理员可以配置压实线程在执行压实时可以为 map 分配多少内存。每个线程都会创建自己的 map，但这个参数指的是所有线程可使用的内存总大小。如果你为 map 分配了 1 GB 内存，并使用了 5 个压实线程，那么每个线程将可以使用 200 MB 内存。Kafka 不要求这个 map 可以放下整个分区的浑浊部分，但至少要能够放下一个片段的浑浊部分，否则 Kafka 会报错。管理员要么为 map 分配更多的内存，要么减少压实线程数量。如果有几个片段都可以被放进 map，那么 Kafka 将从最旧的片段开始压实，其他片段则继续保持浑浊，等待下一轮压实。

在创建好 map 后，压实线程会开始从干净的片段读取消息，它会先读取最旧的消息，把它们的内容与 map 中的内容进行比对。对于每一条消息，它会检查消息的键是否存在于 map 中，如果不存在，则说明

这条消息的值是最新的，就把它复制到替换片段上。如果键已存在，就忽略这条消息，因为后面会有一条更新的包含相同键的消息。在复制完所有消息后，将替换片段与原始片段进行交换，然后开始压实下一个片段。完成整个压实过程后，每一个键对应一条消息，这些消息的值都是最新的，如图 6-7 所示。

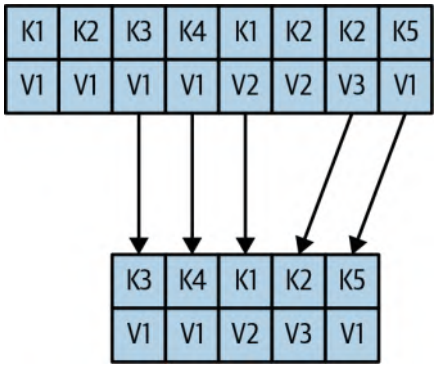

图 6-7：压实前后的分区片段

6.5.8 被删除的事件

如果想删除某个键对应的所有消息，应该怎么做？例如，在用户使用完服务之后，我们在法律方面有义务把用户跟踪信息从系统中删除。

要彻底把一个键从系统中删除，应用程序必须发送包含这个键且值为 `null` 的消息。压实线程在发现这条消息时，会先进行常规的压实操作，只保留值为 `null` 的消息。这条消息（被称为**墓碑消息**）会根据配置的参数保留一段时间。在此期间，消费者可以读取到这条消息，并且发现它的值已经被置空。消费者在将 Kafka 数据复制到关系数据库时，如果它看到这条墓碑消息，就知道应该要把相关的用户信息从数据库中删除。在超过保留期限之后，清理线程会移除墓碑消息，它们的键也将从 Kafka 分区中消失。这里的关键是要让消费者有足够的时间看到墓碑消息，因为如果消费者离线几小时，那么可能就错过了墓碑消息，也就不会去删除数据库中的相关数据了。

值得一提的是，Kafka 的管理客户端提供了一个 `deleteRecords` 方法。这个方法可用于删除指定偏移量之前的所有记录，但它使用的是一种完全不同的机制。当这个方法被调用时，Kafka 会将低水位标记（分区的第一个偏移量）移动到指定的偏移量。这样可以防止消费者读取低水位标记之前的记录，保证这些记录在被清理线程删除之前都是不可访问的。这个方法可用于删除设置了保留策略的主题和压实主题。

6.5.9 何时会压实主题

就像 `delete` 策略不会删除当前的活动片段一样，`compact` 策略也不会压实当前的活动片段，只有旧片段里的消息才会被压实。

在默认情况下，Kafka 会在主题中有 50% 的数据包含脏记录的情况下进行压实。这样做的目的是避免压实太过频繁（因为压实会影响主题的读写性能），同时也能避免存在太多脏记录（因为它们会占用磁盘空间）。让主题的 50% 的磁盘空间包含脏记录，然后压实一次，这看起来是个合理的折中，当然，管理员也可以调整这个百分比。

此外，管理员可以通过两个配置参数来控制压实时间。

- `min.compaction.lag.ms` 用于确保消息被写入之后最短需要经过多长时间才可以被压实。
- `max.compaction.lag.ms` 用于确保消息从写入到可以被压实最长可以是多长时间。当业务要求在一定时间内进行压实时可以使用这个参数，例如，GDPR 要求在收到删除请求之后 30 天内删除某些信息。

6.6 小结

我们无法在这一章中涵盖所有的内容，只是希望大家能够对 Kafka 社区在这个项目上所做的设计和优化有所了解，也希望向大家解释清楚在使用 Kafka 时可能碰到的一些晦涩难懂的现象和参数配置问题。

如果大家真的对 Kafka 内部原理感兴趣，那么唯一的途径是阅读它的源代码。Kafka 开发者邮件组（dev@kafka.apache.org）是一个非常友好的社区，在这里总会有人热心回答有关 Kafka 工作原理的问题。或许你在阅读源代码时还能修复一些 bug——开源社区的大门总是向贡献者敞开。

第 7 章　可靠的数据传递

可靠性是系统而不是某个独立组件的一个属性，所以，在讨论 Kafka 的可靠性保证时，需要从系统的整体出发。说到可靠性，那些与 Kafka 集成的系统与 Kafka 本身一样重要。正因为可靠性是系统层面的概念，所以它不只是某个个体的事情。Kafka 管理员、Linux 系统管理员、网络和存储管理员，以及应用程序开发者，所有人必须协同作战才能构建出一个可靠的系统。

Kafka 在数据传递可靠性方面具备很大的灵活性，它可以被应用在很多场景中——从跟踪用户点击动作到处理信用卡支付操作。有些场景对可靠性要求很高，有些则更看重速度和简便性。Kafka 是高度可配置的，它的客户端 API 也提供了高度灵活性，可以满足不同程度的可靠性权衡。

不过，灵活性也很容易让人掉入陷阱。有时候，你的系统看起来是可靠的，但实际上可能不是。本章将首先讨论各种各样的可靠性以及它们在 Kafka 中的含义；然后介绍 Kafka 的复制功能以及它是如何提高系统可靠性的；接下来探讨如何配置 Kafka 的 broker 和主题来满足不同应用场景的需求；随后讨论如何在各种可靠性场景中使用生产者和消费者客户端；最后介绍如何验证系统的可靠性。因为系统可靠性涉及方方面面，所以对其做出的任何假设都必须得到充分验证。

7.1 可靠性保证

在讨论可靠性时，通常会使用**保证**这个词，它是指系统在各种不同的环境中会保持行为的一致性。

ACID 大概是大家最熟悉的一个例子，它是关系数据库普遍支持的标准可靠性保证。ACID 指的是**原子性**、**一致性**、**隔离性**和**持久性**。如果一个数据库厂商说他们的数据库遵循 ACID 规范，那么就是在说他们的数据库可以保证事务行为具有事务性。

有了这些保证，我们才能相信在应用程序中使用关系数据库是安全的——我们知道它们给出了哪些承诺，也知道它们在不同环境条件下会表现出怎样的行为。我们了解这些保证机制，并会基于这些保证机制开发安全的应用程序。

所以，了解系统的保证机制对构建可靠的应用程序来说至关重要，这也是在不同的环境条件下解释系统行为的前提。那么 Kafka 可以在哪些方面做出保证呢？

- Kafka 可以保证分区中的消息是有序的。如果使用同一个生产者向同一个分区中写入消息，并且消息 B 在消息 A 之后写入，那么 Kafka 可以保证消息 B 的偏移量比消息 A 的偏移量大，而且消费者会先读取消息 A 再读取消息 B。
- 一条消息只有在被写入分区所有的同步副本时才被认为是“已提交”的（但不一定要冲刷到磁盘上）。生产者可以选择接收不同类型的确认，比如确认消息被完全提交，或者确认消息被写入首领副本，或者确认消息被发送到网络上。
- 只要还有一个副本是活动的，已提交的消息就不会丢失。
- 消费者只能读取已提交的消息。

在构建系统时，借助这些基本的保证机制可以提升系统可靠性，但它们本身并不能让系统完全可靠。构建一个可靠的系统需要做出一些权衡，Kafka 为管理员和开发者提供了一套配置参数，他们可以控制这些参数以获得想要的可靠性。这是消息存储可靠性和一致性的重要程度与可用性、高吞吐量、低延迟和硬件成本的重要程度之间的一种权衡。

下面将介绍 Kafka 的复制机制，并探讨 Kafka 是如何实现可靠性的，最后将介绍一些重要的配置参数。

7.2 复制

Kafka 的复制机制和分区多副本架构是 Kafka 可靠性保证的核心。把消息写入多个副本可以保证 Kafka 在发生崩溃时仍然能够提供消息的持久性。

第 6 章已经深入解释过 Kafka 的复制机制，现在重新回顾一下主要内容。

Kafka 的主题会被分为多个**分区**，分区是最基本的数据构建块。分区存储在单个磁盘上，Kafka 可以保证分区中的事件是有序的。一个分区可以在线（可用），也可以离线（不可用）。每个分区可以有多个副本，其中有一个副本是首领。所有的事件都会被发送给首领副本，通常情况下消费者也直接从首领副本读取事件。其他副本只需要与首领保持同步，及时从首领那里复制最新的事件。当首领副本不可用时，其中一个同步副本将成为新首领（这里有一个例外，第 6 章中已经讨论过了）。

分区的首领肯定是同步副本，而对跟随者副本来说，则需要满足以下条件才能被认为是同步副本。

- 与 ZooKeeper 之间有一个活跃的会话，也就是说，它在过去的 6 秒（可配置）内向 ZooKeeper 发送过心跳。
- 在过去的 10 秒（可配置）内从首领那里复制过消息。
- 在过去的 10 秒内从首领那里复制过最新的消息。仅从首领那里复制消息是不够的，它还必须在每 10 秒（可配置）内复制一次最新的消息。

如果跟随者副本不能满足以上任何一点（比如与 ZooKeeper 断开连接，或者不再复制新消息，或者复制消息滞后了 10 秒以上），那么它就会被认为是不同步的。一个不同步的副本可以通过与 ZooKeeper 重新建立连接并从首领那里复制最新的消息重新变成同步副本。如果网络出现了临时问题，并快速得到了修复，那么重新变成同步副本就很快。但如果副本所在的 broker 发生了崩溃，那么这个过程就需要较长时间。

不同步副本

在旧版本的 Kafka 中，经常会有一个或多个副本在同步和不同步状态之间快速切换。这说明集群出问题了。一个相对常见的原因是设置了较大的请求最大值和 JVM 堆内存。我们需要调整这些参数，以防止长时间的垃圾回收停顿，因为停顿会导致 broker 暂时与 ZooKeeper 断开连接。不过，现在这个问题已经很少见了，特别是在 Kafka 2.5.0 及以上版本中。使用 JVM 8 及以上版本（支持 G1 垃圾回收器）有助于缓解这个问题，尽管仍然需要针对大消息做一些调优。Kafka 复制协议在本书第 1 版（英文版）出版后的几年里变得更加可靠了。有关 Kafka 复制协议演进的细节，可以参见 Jason Gustafson 的精彩演讲“Hardening Apache Kafka Replication”和格温 • 沙皮拉对 Kafka 整体改进的概述“Please Upgrade Apache Kafka Now”。

一个稍有滞后的同步副本会导致生产者和消费者变慢，因为在消息被认为已**提交**之前，客户端会等待所有同步副本确认消息。如果一个副本变成不同步的，那么我们就不再关心它是否已经收到消息。这个时候，虽然不同步副本同样是滞后的，但它不影响性能。然而，更少的同步副本意味着更小的有效复制系数，因此在停机时丢失数据的风险就更大了。

下一节将讲解在实战中这意味着什么。

7.3 broker 配置

broker 中有 3 个配置参数会影响 Kafka 的消息存储可靠性。与其他配置参数一样，它们既可以配置在 broker 级别，用于控制所有主题的行为，也可以配置在主题级别，用于控制个别主题的行为。

主题级别的可靠性配置让 Kafka 集群中可以同时存在可靠的主题和不可靠的主题。例如，在银行系统中，管理员可能会把整个集群设置为可靠的，但一些用于保存客户投诉信息的主题可以例外，因为这些主题的消息如果发生丢失，问题也不大。

下面来逐个介绍一下这些配置参数，看看它们如何影响消息存储的可靠性，以及 Kafka 在哪些方面做出了权衡。

7.3.1 复制系数

主题级别的配置参数是 `replication.factor`。在 broker 级别，可以通过 `default.replication.factor` 来设置自动创建的主题的复制系数。

到目前为止，本书都是假设主题的复制系数是 3，也就是说每个分区总共会被 3 个不同的 broker 复制 3 次。这样的假设是合理的，因为这就是 Kafka 默认的复制系数，不过用户可以修改它。即使是在主题被创建之后，仍然可以使用 Kafka 的副本分配工具新增或移除副本，以此来改变复制系数。

如果复制系数是 *N*，那么在 *N*–1 个 broker 失效的情况下，客户端仍然能够从主题读取数据或向主题写入数据。所以，更高的复制系数会带来更高的可用性、可靠性和更少的灾难性事故。另外，复制系数 *N* 需要至少 *N* 个 broker，也就是说我们会有 *N* 个数据副本，并且它们会占用 *N* 倍的磁盘空间。基本上，我们是在用硬件换取可用性。

那么该如何确定一个主题需要几个副本呢？这个时候需要考虑以下因素。

可用性

如果一个分区只有一个副本，那么它在 broker 例行重启期间将不可用。副本越多，可用性就越高。

持久性

每个副本都包含了一个分区的所有数据。如果一个分区只有一个副本，那么一旦磁盘损坏，这个分区的所有数据就丢失了。如果有更多的副本，并且这些副本位于不同的存储设备中，那么丢失所有副本的概率就降低了。

吞吐量

每增加一个副本都会增加 broker 内的复制流量。如果以 10 MBps 的速率向一个分区发送数据，并且只有 1 个副本，那么不会增加任何的复制流量。如果有 2 个副本，则会增加 10 MBps 的复制流量，3 个副本会增加 20 MBps 的复制流量，5 个副本会增加 40 MBps 的复制流量。在规划集群大小和容量时，需要把这个考虑在内。

端到端延迟

每一条记录必须被复制到所有同步副本之后才能被消费者读取。从理论上讲，副本越多，出现滞后的可能性就越大，因此会降低消费者的读取速度。在实际当中，如果一个 broker 由于各种原因变慢，那么它就会影响所有的客户端，而不管复制系数是多少。

成本

一般来说，出于成本方面的考虑，非关键数据的复制系数应该小于 3。数据副本越多，存储和网络成本就越高。因为很多存储系统已经将每个数据块复制了 3 次，所以有时候可以将 Kafka 的复制系数设置为

2，以此来降低成本。需要注意的是，与复制系数 3 相比，这样做仍然会降低可用性，但可以由存储设备来提供持久性保证。

副本的位置分布也很重要。Kafka 可以确保分区的每个副本被放在不同的 broker 上。但是，在某些情况下，这样仍然不够安全。如果一个分区的所有副本所在的 broker 位于同一个机架上，那么一旦机架的交换机发生故障，不管设置了多大的复制系数，这个分区都不可用。为了避免机架级别的故障，建议把 broker 分布在多个不同的机架上，并通过 `broker.rack` 参数配置每个 broker 所在的机架的名字。如果配置了机架名字，那么 Kafka 就会保证分区的副本被分布在多个机架上，从而获得更高的可用性。如果是在云端运行 Kafka，则可以将可用区域视为机架。第 6 章已经介绍过如何将副本分配给不同的 broker 和机架。

7.3.2 不彻底的首领选举

`unclean.leader.election.enable` 只能在 broker 级别（实际上是在集群范围内）配置，它的默认值是 `false`。

前面讲过，当分区的首领不可用时，一个同步副本将被选举为新首领。如果在选举过程中未丢失数据，也就是说所有同步副本都包含了已提交的数据，那么这个选举就是“彻底”的。

但如果在首领不可用时其他副本都是不同步的，该怎么办呢？

这种情况会在以下两种场景中出现。

- 分区有 3 个副本，其中的两个跟随者副本不可用（比如有两个 broker 发生崩溃）。这个时候，随着生产者继续向首领写入数据，所有消息都会得到确认并被提交（因为此时首领是唯一的同步副本）。现在，假设首领也不可用了（又一个 broker 发生崩溃），这个时候，如果之前的一个跟随者重新启动，那么它就会成为分区的唯一不同步副本。
- 分区有 3 个副本，由于网络问题导致两个跟随者副本复制消息滞后，因此即使它们还在复制，但已经不同步了。作为唯一的同步副本，首领会继续接收消息。这个时候，如果首领变为不可用，则只剩下两个不同步的副本可以成为新首领。

对于这两种场景，我们要做出一个两难的选择。

- 如果不允许不同步的副本被提升为新首领，那么分区在旧首领（最后一个同步副本）恢复之前是不可用的。有时候这种状态会持续数小时（比如更换内存芯片）。
- 如果允许不同步的副本被提升为新首领，那么在这个副本变为不同步之后写入旧首领的数据将全部丢失，消费者读取的数据将会出现不一致。为什么会这样？假设在副本 0 和副本 1 不可用的情况下，向副本 2（也就是首领）写入偏移量为 100~200 的消息。现在，副本 2 变为不可用，副本 0 变为可用，但副本 0 只包含偏移量为 0~100 的消息。如果允许副本 0 成为新首领，那么生产者就可以继续写入数据，消费者则可以继续读取数据。于是，新首领就有了偏移量为 100~200 的新消息。这样，部分消费者会读取到偏移量为 100~200 的旧消息，部分消费者会读取到偏移量为 100~200 的新消息，还有部分消费者会读取到二者的混合消息。这样会导致非常不好的结果，比如生成不准确的报表。另外，副本 2 可能会重新变为可用，并成为新首领的跟随者。这个时候，它会把在当前首领中不存在的消息全部删除，以致所有消费者都将无法读取到这些消息。

总的来说，如果允许不同步副本成为首领，那么就要承担丢失数据和消费者读取到不一致的数据的风险。如果不允许它们成为首领，那么就要接受较低的可用性，因为必须等待原先的首领恢复到可用状态。

在默认情况下，`unclean.leader.election.enable` 的值是 `false`，也就是不允许不同步副本成为首领。这是最安全的选项，因为它可以保证数据不丢失。这也意味着在之前描述的极端不可用场景中，一些分区将一直不可用，直到手动恢复。当遇到这种情况时，管理员可以决定是否允许数据丢失，以便让分区可用，如果可以，就在启动集群之前将其设置为 `true`，在集群恢复之后不要忘了再将其改回 `false`。

7.3.3 最少同步副本

`min.insync.replicas` 参数可以配置在主题级别和 broker 级别。

如前所述，尽管为一个主题配置了 3 个副本，还是会出现只剩下一个同步副本的情况。如果这个同步副本变为不可用，则必须在可用性和一致性之间做出选择，而这是一个两难的选择。根据 Kafka 对可靠性保证的定义，一条消息只有在被写入所有同步副本之后才被认为是已提交的，但如果这里的“所有”只包含一个同步副本，那么当这个副本变为不可用时，数据就有可能丢失。

如果想确保已提交的数据被写入不止一个副本，就要把最少同步副本设置得大一些。对于一个包含 3 个副本的主题，如果 `min.insync.replicas` 被设置为 2，那么至少需要有两个同步副本才能向分区写入数据。

如果 3 个副本都是同步的，那么一切正常进行。即使其中一个副本变为不可用，也不会有什么问题。但是，如果有两个副本变为不可用，那么 broker 就会停止接受生产者的请求。尝试发送数据的生产者会收到 `NotEnoughReplicasException` 异常，不过消费者仍然可以继续读取已有的数据。实际上，如果使用这样的配置，那么当只剩下一个同步副本时，它就变成只读的了。这样做是为了避免在发生不彻底的选举时数据的写入和读取出现非预期的行为。要脱离这种只读状态，必须让两个不可用分区中的一个重新变为可用（比如重启 broker），并等待它变为同步的。

7.3.4 保持副本同步

前面提到过，不同步副本会降低总体可靠性，所以要尽量避免出现这种情况。一个副本可能在两种情况下变得不同步：要么它与 ZooKeeper 断开连接，要么它从首领复制消息滞后。对于这两种情况，Kafka 提供了两个 broker 端的配置参数。

`zookeeper.session.timeout.ms` 是允许 broker 不向 ZooKeeper 发送心跳的时间间隔。如果超过这个时间不发送心跳，则 ZooKeeper 会认为 broker 已经“死亡”，并将其从集群中移除。在 Kafka 2.5.0 中，这个参数的默认值从 6 秒增加到了 18 秒，以提高 Kafka 集群在云端的稳定性，因为云环境的网络延迟更加多变。一般来说，我们希望将这个值设置得足够大，以避免因垃圾回收停顿或网络条件造成的随机抖动，但又要设置得足够小，以确保及时检测到确实已经发生故障的 broker。

如果一个副本未能在 `replica.lag.time.max.ms` 指定的时间内从首领复制数据或赶上首领，那么它将变成不同步副本。在 Kafka 2.5.0 中，这个参数的默认值从 10 秒增加到了 30 秒，以提高集群的弹性，并避免不必要的抖动。需要注意的是，这个值也会影响消费者的最大延迟——值越大，等待一条消息被写入所有副本并可被消费者读取的时间就越长，最长可达 30 秒。

7.3.5 持久化到磁盘

之前提到过，即使消息还没有被持久化到磁盘上，Kafka 也可以向生产者发出确认，这取决于已接收到消息的副本的数量。Kafka 会在重启之前和关闭日志片段（默认 1 GB 大小时关闭）时将消息冲刷到磁盘上，或者等到 Linux 系统页面缓存被填满时冲刷。其背后的想法是，拥有 3 台放置在不同机架或可用区域的机器，并在每台机器上放置一份数据副本比只将消息写入首领的磁盘更加安全，因为两个不同的机架或可用区域同时发生故障的可能性非常小。不过，也可以让 broker 更频繁地将消息持久化到磁盘上。配置参数 `flush.messages` 用于控制未同步到磁盘的最大消息数量，`flash.ms` 用于控制同步频率。在配置这些参数之前，最好先了解一下 `fsync` 是如何影响 Kafka 的吞吐量的以及如何尽量避开它的缺点。

7.4 在可靠的系统中使用生产者

即使我们会尽可能地把 broker 配置得很可靠，但如果没有对生产者进行可靠性方面的配置，则整个系统仍然存在丢失数据的风险。

请看下面的两个例子。

- 我们为 broker 配置了 3 个副本，并禁用了不彻底的首领选举，这样应该可以保证已提交的消息不会丢失。不过，我们把生产者发送消息的 `acks` 设置成了 `1`。生产者向首领发送了一条消息，虽然其被首领成功写入，但其他同步副本还没有收到这条消息。首领向生产者发送了一个响应，告诉它“消息写入成功”，然后发生了崩溃，而此时其他副本还没有复制这条消息。另外两个副本此时仍然被认为是同步的（我们需要一小段时间才能判断一个副本是否变成了不同步的），并且其中的一个副本会成为新首领。因为消息还没有被写入这两个副本，所以就丢失了，但发送消息的客户端认为消息已经成功写入。从消费者的角度来看，系统仍然是一致的，因为它们看不到丢失的消息（副本没有收到这条消息，不算已提交），但从生产者的角度来看，这条消息丢失了。
- 我们为 broker 配置了 3 个副本，并禁用了不彻底的首领选举。我们接受了之前的教训，把生产者的 `acks` 设置成了 `all`。假设现在生产者向 Kafka 发送了一条消息，此时分区首领刚好发生崩溃，新首领正在选举当中，Kafka 会向生产者返回“首领不可用”的响应。在这个时候，如果生产者未能正确处理这个异常，也没有重试发送消息，那么消息也有可能丢失。这不算是 broker 的可靠性问题，因为 broker 并没有收到这条消息；这也不是一致性问题，因为消费者也不会读取到这条消息。问题在于，如果生产者未能正确处理异常，就有可能丢失数据。

从上面的两个例子可以看出，开发人员需要注意两件事情。

- 根据可靠性需求配置恰当的 `acks`。
- 正确配置参数，并在代码里正确处理异常。

第 3 章深入介绍过如何配置生产者，现在再来回顾几个要点。

7.4.1 发送确认

生产者可以选择以下 3 种确认模式。

`acks=0`

如果生产者能够通过网络把消息发送出去，那么就认为消息已成功写入 Kafka。不过，在这种情况下仍然有可能出现错误，比如发送的消息对象无法被序列化或者网卡发生故障。如果此时分区离线、正在进行首领选举或整个集群长时间不可用，则并不会收到任何错误。在 `acks=0` 模式下，生产延迟是很低的（这就是为什么很多基准测试是基于这种模式的），但它对端到端延迟并不会带来任何改进（在消息被所有可用副本复制之前，消费者是看不到它们的）。

`acks=1`

首领在收到消息并把它写入分区数据文件（不一定要冲刷到磁盘上）时会返回确认或错误响应。在这种模式下，如果首领被关闭或发生崩溃，那么那些已经成功写入并确认但还没有被跟随者复制的消息就丢失了。另外，消息写入首领的速度可能比副本从首领那里复制消息的速度更快，这样会导致分区复制不及时，因为首领在消息被副本复制之前就向生产者发送了确认响应。

`acks=all`

首领在返回确认或错误响应之前，会等待所有同步副本都收到消息。这个配置可以和 `min.insync.replicas` 参数结合起来，用于控制在返回确认响应前至少要有多少个副本收到消息。这是最安全的选项，因为生产者会一直重试，直到消息提交成功。不过，这种模式下的生产者延迟也最大，因为生产者在发送下一批次消息之前需要等待所有副本都收到当前批次的消息。

7.4.2 配置生产者的重试参数

生产者需要处理的错误包括两个部分：一部分是由生产者自动处理的错误，另一部分是需要开发者手动处理的错误。

生产者可以自动处理**可重试**的错误。当生产者向 broker 发送消息时，broker 可以返回一个成功响应或者错误响应。错误响应可以分为两种，一种是在重试之后可以解决的，另一种是无法通过重试解决的。如果 broker 返回 `LEADER_NOT_AVAILABLE` 错误，那么生产者可以尝试重新发送消息——或许新首领被选举出来了，那么第二次尝试发送就会成功。也就是说，`LEADER_NOT_AVAILABLE` 是一个**可重试**错误。如果 broker 返回 `INVALID_CONFIG` 错误，那么即使重试发送消息也无法解决这个问题，所以这样的重试是没有意义的，这是**不可重试**错误。

一般来说，如果你的目标是不丢失消息，那么就让生产者在遇到可重试错误时保持重试。正如第 3 章所建议的那样，最好的重试方式是使用默认的重试次数（整型最大值或无限），并把 `delivery.timeout.ms` 配置成我们愿意等待的时长，生产者会在这个时间间隔内一直尝试发送消息。

重试发送消息存在一定的风险，因为如果两条消息都成功写入，则会导致消息重复。通过重试和小心翼翼地处理异常，可以保证每一条消息都会被保存**至少一次**，但不能保证**只保存一次**。如果把 `enable.idempotence` 参数设置为 `true`，那么生产者就会在消息里加入一些额外的信息，broker 可以使用这些信息来跳过因重试导致的重复消息。第 8 章将讨论这方面的细节。

7.4.3 额外的错误处理

使用生产者内置的重试机制可以在不造成消息丢失的情况下轻松地处理大部分错误，但开发人员仍然需要处理以下这些其他类型的错误。

- 不可重试的 broker 错误，比如消息大小错误、身份验证错误等。
- 在将消息发送给 broker 之前发生的错误，比如序列化错误。
- 在生产者达到重试次数上限或重试消息占用的内存达到上限时发生的错误。
- 超时。

第 3 章介绍过如何在同步发送和异步发送消息时处理错误。这些错误的处理逻辑与具体的应用程序及其目标有关——丢弃“不合法的消息”？把错误记录下来？停止从源系统读取消息？对源系统应用回压策略以便暂停发送消息？把消息保存到本地磁盘的某个目录里？具体使用哪一种逻辑要根据实际的架构和产品需求来决定。只需记住，如果错误处理只是为了重试发送消息，那么最好还是使用生产者内置的重试机制。

7.5 在可靠的系统中使用消费者

我们已经学习了如何在保证 Kafka 可靠性的前提下生产数据，现在来看看如何在同样的前提下读取数据。

本章在开头部分提到过，只有已经被提交到 Kafka 的数据，也就是已经被写入所有同步副本的数据，对消费者是可用的。这保证了消费者读取到的数据是一致的。消费者唯一要做的是跟踪哪些消息是已经读取过的，哪些消息是还未读取的，这是消费者在读取消息时不丢失消息的关键。

在从分区读取数据时，消费者会先获取一批消息，检查批次的最后一个偏移量，然后从这个偏移量开始读取下一批消息。这样可以保证消费者总能以正确的顺序获取新数据，不会错过任何消息。

如果一个消费者退出，那么另一个消费者需要知道从什么地方开始继续处理，以及前一个消费者在退出处理前的最后一个偏移量是多少。所谓的“另一个”消费者，也可能是原来的消费者重启之后重新上线。不过这个不重要，因为总归会有一个消费者继续从这个分区读取数据，重要的是它需要知道该从哪里开始读取。这也就是为什么消费者要“提交”偏移量。消费者会把读取的每一个分区的偏移量都保存起来，这样在重启或其他消费者接手之后就知道从哪里开始读取了。造成消费者丢失消息最主要的一种情况是它们提交了已读取消息的偏移量却未能全部处理完。在这种情况下，如果其他消费者接手了工作，那么那些没有被处理的消息就会被忽略，永远不会得到处理。这就是为什么我们非常重视何时以及如何提交偏移量。

已提交的消息与已提交的偏移量

需要注意的是，与之前讨论的“已提交的消息”不同，这里已提交的消息是指已经被写入所有同步副本并且对消费者可见的消息，而**已提交的偏移量**是指消费者发送给 Kafka 的偏移量，用于确认它已接收到的最后一条消息在分区中的位置。

第 4 章详细介绍过消费者 API 和多种提交偏移量的方式。接下来我们将介绍一些关键的注意事项。如果要了解消费者 API 的使用细节，请参见第 4 章。

7.5.1 消费者的可靠性配置

为了保证消费者行为的可靠性，需要注意以下 4 个非常重要的配置参数。

第一个是 `group.id`，这个参数在第 4 章中已经详细介绍过了。如果两个消费者具有相同的群组 ID，并订阅了同一个主题，那么每个消费者将分到主题分区的一个子集，也就是说它们只能读取到所有消息的一个子集（但整个群组可以读取到主题所有的消息）。如果你希望一个消费者可以读取主题所有的消息，那么就需要为它设置唯一的 `group.id`。

第二个是 `auto.offset.reset`，这个参数指定了当没有偏移量（比如在消费者首次启动时）或请求的偏移量在 broker 上不存在时（第 4 章已经介绍过这种场景）消费者该作何处理。这个参数有两个值，一个是 `earliest`，如果配置了这个值，那么消费者将从分区的开始位置读取数据，即使它没有有效的偏移量。这会导致消费者读取大量的重复数据，但可以保证最少的数据丢失。另一个值是 `latest`，如果配置了这个值，那么消费者将从分区的末尾位置读取数据。这样可以减少重复处理消息，但很有可能会错过一些消息。

第三个是 `enable.auto.commit`，你可以决定让消费者自动提交偏移量，也可以在代码里手动提交偏移量。自动提交的一个最大好处是可以少操心一些事情。如果是在消费者的消息轮询里处理数据，那么自动提交可以确保不会意外提交未处理的偏移量。自动提交的主要缺点是我们无法控制应用程序可能重复处理的消息的数量，比如消费者在还没有触发自动提交之前处理了一些消息，然后被关闭。如果应用程序的处理逻辑比较复杂（比如把消息交给另外一个后台线程去处理），那么就只能使用手动提交了，因为自动提交机制有可能会在还没有处理完消息时就提交偏移量。

第四个配置参数 `auto.commit.interval.ms` 与第三个参数有直接的联系。如果选择使用自动提交，那么可以通过这个参数来控制提交的频率，默认每 5 秒提交一次。一般来说，频繁提交会增加额外的开销，但也会降低重复处理消息的概率。

如果一个消费者经常因为发生再均衡而暂停处理消息，则很难说它是可靠的，尽管这与数据处理的可靠性没有直接关系。第 4 章提供过一些关于如何减少不必要的再均衡以及在再均衡期间减少消费者停顿的建议。

7.5.2 手动提交偏移量

如果想要更大的灵活性，选择了手动提交，那么就需要考虑正确性和性能方面的问题。

这里不再重复介绍手动提交机制以及如何使用相关的 API，因为第 4 章已经很详细地介绍过了。相反，我们会着重说明几个在开发可靠的消费者应用程序时需要注意的事项。下面先从简单的开始，再逐个深入。

01. 总是在处理完消息后提交偏移量

    如果所有的处理逻辑都是在轮询里进行的，并且不需要维护轮询之间的状态（比如为了聚合数据），那么就很简单。我们可以使用自动提交，在轮询结束时提交偏移量，也可以在轮询里提交偏移量，并选择一个合适的提交频率，在额外的开销和重复消息量之间取得平衡。如果涉及额外线程或有状态处理，那么情况就复杂一些。第 4 章介绍过如何实现这些逻辑并提供了一些参考示例。

02. 提交频率是性能和重复消息数量之间的权衡

    即使是在最简单的场景中（比如所有的处理逻辑都在轮询里进行，并且不需要维护轮询之间的状态），仍然可以选择在一个轮询里提交多次或多个轮询提交一次。提交偏移量需要额外的开销，这有点儿类似生产者配置了 `acks=all`，但同一个消费者群组提交的偏移量会被发送给同一个 broker，这可能会导致 broker 超载。提交频率需要在性能需求和重复消息量之间取得平衡。处理一条消息就提交一次偏移量的方式只适用于吞吐量非常低的主题。

03. 在正确的时间点提交正确的偏移量

    在轮询过程中提交偏移量有一个缺点，就是有可能会意外提交已读取但未处理的消息的偏移量。一定要在处理完消息后再提交偏移量，这点很关键——提交已读取但未处理的消息的偏移量会导致消费者错过消息。具体示例可参见第 4 章。

04. 再均衡

    在设计应用程序时，需要考虑到消费者会发生再均衡并需要处理好它们。第 4 章展示过几个示例，主要是关于如何在分区被撤销之前提交偏移量，或者在应用程序被分配到新分区并清理状态时提交偏移量。

05. 消费者可能需要重试

    有时候，在调用了轮询方法之后，有些消息需要稍后再处理。假设我们要把 Kafka 的数据写到数据库，但此时数据库不可用，那么就需要稍后再重试。需要注意的是，消费者提交偏移量并不是对单条消息的“确认”，这与传统的发布和订阅消息系统不一样。也就是说，如果记录 #30 处理失败，但记录 #31 处理成功，那么就不应该提交记录 #31 的偏移量——如果提交了，就表示 #31 以内的记录都已处理完毕，包括记录 #30 在内，但这可能不是我们想要的结果。不过，可以采用以下两种模式来解决这个问题。

    第一种模式，在遇到可重试错误时，提交最后一条处理成功的消息的偏移量，然后把还未处理好的消息保存到缓冲区（这样下一个轮询就不会把它们覆盖掉），并调用消费者的 `pause()` 方法，确保其他的轮询不会返回数据，之后继续处理缓冲区里的消息。

第二种模式，在遇到可重试错误时，把消息写到另一个重试主题，并继续处理其他消息。另一个消费者群组负责处理重试主题中的消息，或者让一个消费者同时订阅主主题和重试主题。这种模式有点儿像其他消息系统中的死信队列。

06. 消费者可能需要维护状态

在一些应用程序中，需要维护多个轮询之间的状态。如果想计算移动平均数，就需要在每次轮询之后更新结果。如果应用程序重启，则不仅需要从上一个偏移量位置开始处理消息，还需要恢复之前保存的移动平均数。一种办法是在提交偏移量的同时把算好的移动平均数写到一个“结果”主题中。当一个线程重新启动时，它就可以获取到之前算好的移动平均数，并从上一次提交的偏移量位置开始读取数据。第 8 章将介绍如何在一个事务中将结果写入主题并提交偏移量。一般来说，这是一个比较复杂的问题，建议尝试使用其他框架，比如 Kafka Streams 或 Flink，它们为聚合、连接、时间窗和其他复杂的分析操作提供了高级的 DSL API。

7.6 验证系统可靠性

经过了所有这些流程（确认可靠性需求、配置 broker、配置客户端和正确使用 API），现在可以把所有东西都部署到生产环境中，然后高枕无忧，自信不会丢失任何消息了，对吗？

建议还是先对系统可靠性做 3 个层面的验证：验证配置、验证应用程序以及在生产环境中监控可靠性。下面来看看每一步都需要验证什么以及如何验证。

7.6.1 验证配置

可以抛开应用程序逻辑，单独验证 broker 和客户端的配置。之所以建议这么做，主要是因为以下两个原因。

- 有助于验证配置是否能够满足需求。
- 有助于了解系统的预期行为。

Kafka 提供了两个重要的配置验证工具：`org.apache.kafka.tools` 包下面的 `VerifiableProducer` 类和 `VerifiableConsumer` 类。可以在命令行中运行这两个工具，或把它们嵌入自动化测试框架中。

`VerifiableProducer` 会生成一系列消息，消息里包含了一个从 1 到指定数字的数字。可以使用与真实的生产者相同的配置参数来配置 `VerifiableProducer`，比如配置相同的 `acks`、`retries`、`delivery.timeout.ms` 和消息生成速率。在运行过程中，`VerifiableProducer` 会根据收到的确认响应将每条消息发送成功或失败的结果打印出来。反过来，`VerifiableConsumer` 会读取由 `VerifiableProducer` 生成的消息，并按照读取顺序把它们打印出来。它也会把与偏移量和再均衡相关的信息打印出来。

要仔细考虑需要测试哪些场景，比如以下场景。

- 首领选举：如果停掉首领会发生什么事情？生产者和消费者需要多长时间来恢复状态？
- 控制器选举：重启控制器后系统需要多少时间来恢复状态？
- 滚动重启：可以滚动重启 broker 而不丢失消息吗？
- 不彻底的首领选举：如果依次停止一个分区的所有副本（确保每个副本都变为不同步的），然后启动一个不同步的 broker 会发生什么？要怎样才能恢复正常？这样做是可接受的吗？

从中选择一个场景，比如停掉正在写入消息的分区的首领，然后启动 `VerifiableProducer` 和 `VerifiableConsumer` 开始进行测试。如果期望在一个短暂的停顿之后恢复正常，并且没有丢失任何消息，那么只需验证一下生产者生成的消息条数与消费者读取的消息条数相匹配即可。

Kafka 项目代码库提供了大量的测试用例。大部分测试遵循相同的原则，并使用 `VerifiableProducer` 和 `VerifiableConsumer` 来确保不同的迭代版本能够正常运行。

7.6.2 验证应用程序

在确定 broker 和客户端的配置可以满足需求之后，接下来要验证应用程序是否能够提供我们想要的保证。应用程序的验证包括检查错误处理逻辑、偏移量提交的方式、再均衡监听器以及其他使用了 Kafka 客户端的地方。

应用程序的逻辑千变万化，关于如何测试它们，我们也只能提供这些指导。建议将应用程序集成测试作为开发流程的一部分，并针对以下这些故障场景做一些测试。

- 客户端与服务器断开连接
- 客户端与服务器之间存在高延迟
- 磁盘被填满
- 磁盘被挂起（也就是所谓的“掉电”）

- 首领选举
- 滚动重启 broker
- 滚动重启消费者
- 滚动重启生产者

现今有很多优秀的网络故障和磁盘故障注入工具，这里就不一一推荐了。Kafka 项目本身也提供了一个故障注入框架 Trogdor。我们对每一种测试场景都有**期望的行为**（也就是在开发应用程序时所期望看到的行为），然后运行测试看看实际会发生什么。例如，在测试“滚动重启消费者”这一场景时，我们期望在进行再均衡时出现短暂的停顿，然后继续读取消息，并且重复消息的条数不超过 1000 条。测试结果会告诉我们应用程序提交偏移量和处理再均衡的方式是否与预期一样。

7.6.3 在生产环境中监控可靠性

对应用程序进行测试固然重要，但它无法取代在生产环境中对应用程序进行持续的监控，以确保数据按照期望的方式流动。第 12 章将详细介绍如何监控 Kafka 集群，不过除了监控集群的健康状况，也要对客户端和数据流进行监控。

Kafka 的 Java 客户端提供了一些 JMX 指标，可用于监控客户端的状态和事件。对生产者来说，最重要的两个可靠性指标是消息的错误率和重试率（聚合过的）。如果这两个指标上升，则说明系统出问题了。除此之外，还要监控生产者日志，注意那些被设为 `WARN` 级别的错误日志，以及包含“Got error produce response with correlation id 5689 on topic-partition [topic-1,3], retrying (two attempts left). Error: ...”的日志。如果看到消息剩余的重试次数为 0，则说明生产者已经没有多余的重试机会。第 3 章介绍过如何通过配置 `delivery.timeout.ms` 和 `retries` 来改进生产者的重试机制，避免过早放弃重试。当然，如果能把导致错误的问题解决掉则最好。生产者端 `ERROR` 级别的日志可以告诉我们更多信息，比如不可重试错误导致彻底发送失败、可重试错误的重试次数达到上限，或者消息发送超时。如果有必要，那么也可以把 broker 端返回的错误记录下来。

在消费者端，最重要的指标是消费者滞后指标。这个指标告诉我们消费者正在处理的消息与最新提交到分区的消息偏移量之间有多少差距。理想情况下，这个指标的值是 0，也就是说消费者读取的是最新的消息。但是，在实际当中，因为 `poll()` 方法会返回很多消息，消费者在读取下一批数据之前需要花一些时间来处理它们，所以这个指标会有些波动。关键在于要确保消费者最终会赶上去，而不是越落越远。因为会出现正常波动，所以为这个指标配置告警有一定难度。Burrow 是 LinkedIn 公司开发的一款消费者滞后检测工具，其可以让这个指标的配置变得容易一些。

监控数据流是为了确保所有生成的数据会被及时地读取（这里的“及时”视具体的业务需求而定）。为了确保数据能够被及时读取，需要知道数据是什么时候生成的。从 0.10.0 版本开始，Kafka 在所有消息里加入了生成消息的时间戳。（但需要注意的是，发送消息的应用程序或配置了相应参数的 broker 都可以覆盖这个时间戳。）

为了确保所有消息在合理的时间内被读取，应用程序需要记录生成消息的数量（一般用每秒多少条消息来表示）。消费者需要记录单位时间内已读取消息的数量以及消息生成时间与读取时间之间的时间差（根据消息的时间戳来判断）。然后，我们需要一个系统将生产者和消费者记录的消息数量收集起来（确保没有丢失消息），确保二者之间的时间差在合理的范围内。要实现这种端到端的监控系统，不仅耗时，还有一定难度，到目前为止还没有开源的实现。不过，作为 Confluent Control Center 的一部分，Confluent 提供了一个商业版实现。

除了监控客户端和端到端数据流，broker 还提供了一些指标，用于说明 broker 发送给客户端的错误响应率。建议收集 `kafka.server:type=BrokerTopicMetrics,name=FailedProduceRequestsPerSec` 和 `kafka.server:type=BrokerTopicMetrics,name=FailedFetchRequestsPerSec` 这两个指标。一些错误响应会时不时地出现，如果我们关闭了一个 broker，另一个 broker 成了新首领，那么生产者可能会收到 `NOT_LEADER_FOR_PARTITION` 错误，并在继续发送消息之前请求更新的元数据。如果失败的请求数量激增，则有必要对它们进行诊断。为了帮助诊断这类问题，Kafka 会将与失败请求相关的指标与 broker 发送的错误响应关联在一起。

7.7 小结

正如本章开头所述，可靠性并不只是 Kafka 单方面的事情。应该从整个系统层面来考虑可靠性问题，包括应用程序架构、生产者和消费者 API 的使用方式、生产者和消费者的配置、主题的配置以及 broker 的配置。为了提升系统的可靠性，需要在许多方面（比如复杂性、性能、可用性和磁盘空间）做出权衡。了解了 Kafka 的各种配置和常用模式，并对应用场景的需求做到心中有数，就可以在应用程序和 Kafka 的可靠性以及各种权衡之间做出更好的决策。

第 8 章　精确一次性语义

第 7 章介绍了用于控制 Kafka 可靠性保证的配置参数和最佳实践。我们主要关注至少一次性传递——保证 Kafka 不会丢失已确认写入的消息。但仍然可能出现重复消息。

在一些简单的系统中，生产者生成的消息会被各种应用程序读取，即使出现了重复消息也很容易处理。大多数真实的应用程序有唯一标识符，可用于对消息进行去重处理。

但是，在处理聚合事件的流式处理应用程序中，事情就没那么简单了。在检查读取消息、计算平均值并生成结果的应用程序时，我们通常无法检测出平均值是不正确的，因为有可能在计算平均值时把一个事件处理了两次。对于这种情况，需要提供更强的保证，这种保证就是精确一次性处理语义。

本章将介绍 Kafka 的精确一次性语义及其应用场景和局限性。与之前介绍至少一次性保证一样，我们将更深入地探究精确一次性语义，并了解它的实现原理。在第一次阅读本章时，也可以跳过这些内容，但在使用这个特性之前如果能够先深入理解它们，则会对你有很大帮助——本章解释了不同配置参数和 API 的含义，以及如何最有效地使用它们。

Kafka 的精确一次性语义由两个关键特性组成：幂等生产者（避免因重试导致的消息重复）和事务语义（保证流式处理应用程序中的精确一次性处理）。下面先来介绍既简单又常用的幂等生产者。

8.1 幂等生产者

如果一个操作被执行多次的结果与被执行一次相同，那么这个操作就是幂等的。对数据库来说，可以通过 `UPDATE t SET x=x+1 where y=5` 和 `UPDATE t SET x=18 where y=5` 之间的区别来说明什么是幂等性。第一个语句不是幂等的，因为执行它 3 次得到的结果与执行一次是不一样的。第二个语句是幂等的，因为不管执行它多少次，`x` 都等于 18。

这与 Kafka 生产者有什么关系呢？如果为生产者配置了至少一次性语义而不是幂等性语义，则意味着在发生了不确定性事件的情况下，生产者将重试发送消息，这样可以保证消息至少有一次是送达的，但可能会因为重试而出现重复。

一个最典型的场景是分区首领收到生产者发送的一条消息，这条消息被跟随者成功复制，然后，首领所在的 broker 在向生产者发送响应之前崩溃了。生产者没有收到回应，在一段时间之后将重新发送消息。消息被发送给了新首领，而新首领已经有了上一次写入的消息副本，结果导致消息重复。

对一些应用程序来说，消息重复并不是什么问题，但对另外一些应用程序来说，消息重复可能导致出现错误的库存、糟糕的财务报表，或者给只订购一把雨伞的人邮寄了两把雨伞。

Kafka 的幂等生产者可以自动检测并解决消息重复问题。

8.1.1 幂等生产者的工作原理

如果启用了幂等生产者，那么每条消息都将包含生产者 ID（PID）和序列号。我们将它们与目标主题和分区组合在一起，用于唯一标识一条消息。broker 会用这些唯一标识符跟踪写入每个分区的最后 5 条消息。为了减少每个分区需要跟踪的序列号数量，生产者需要将 `max.inflight.requests` 设置成 5 或更小的值（默认值是 5）。

如果 broker 收到之前已经收到过的消息，那么它将拒绝这条消息，并返回错误。生产者会记录这个错误，并反映在指标当中，但不抛出异常，也不触发告警。在生产者客户端，错误将被添加到 `record-error-rate` 指标当中。在 broker 端，错误是 `ErrorsPerSec` 指标的一部分（`RequestMetrics` 类型）。

如果 broker 收到一个非常大的序列号该怎么办？如果 broker 期望消息 2 后面跟着消息 3，但收到了消息 27，那么这个时候该怎么办？在这种情况下，broker 将返回“乱序”错误。如果使用了不带事务的幂等生产者，则这个错误可能会被忽略。

虽然生产者在遇到“乱序”异常后将继续正常运行，但这个错误通常说明生产者和 broker 之间出现了消息丢失——如果 broker 在收到消息 2 之后直接收到消息 27，那么说明从消息 3 到消息 26 一定发生了什么。如果你在日志中看到这样的错误，那么最好重新检查一下生产者和主题的配置，确保为生产者配置了高可靠性参数，并检查是否发生了不彻底的首领选举。

就像在分布式系统中一样，研究幂等生产者在故障条件下的行为会非常有趣。可以考虑以下两种情况：生产者重启和 broker 故障。

01. 生产者重启

    当一个生产者发生故障时，我们通常会创建新生产者来代替它——可能是手动重启机器或使用像 Kubernetes 这样提供了自动故障恢复功能的复杂框架。关键的问题在于，如果启用了幂等生产者，那么生产者在重启时就会连接 broker 并生成生产者 ID。生产者在每次初始化时都会产生一个新 ID（假设没有启用事务）。这意味着如果一个生产者发生故障，取代它的生产者发送了一条旧生产者已经发送过的消息，那么 broker 将无法检测到重复，因为这两条消息有不同的生产者 ID 和序列号，将被视为两条不同的消息。需要注意的是，如果一个旧生产者被挂起，但在替代它的新生产者启动之后又“活”过来了，那么情况也一样——旧生产者不会被认为是“僵尸”，它们是两个拥有不同 ID 的生产者。

02. **broker 故障**

当一个 broker 发生故障时，控制器将为首领副本位于这个 broker 上的分区选举新首领。假设我们有一个生产者，它向主题 A 的分区 0 生成消息，分区 0 的首领副本在 broker 5 上，跟随者副本在 broker 3 上。如果 broker 5 发生故障，那么 broker 3 就会成为新首领。生产者通过元数据协议发现 broker 3 是新首领，并开始向它生成消息。

但 broker3 如何知道哪些序列号已经生成过了？

每次生成新消息时，首领都会用最后 5 个序列号更新内存中的生产者状态。每次从首领复制新消息时，跟随者副本都会更新自己的内存。当跟随者成为新首领时，它的内存中已经有了最新的序列号，并且可以继续验证新生成的消息，不会有任何问题或延迟。

但是，如果旧首领又“活”过来了，会发生什么呢？在重启之后，内存中没有旧首领的生产者状态。为了能够恢复状态，每次在关闭或创建日志片段时 broker 都会将生产者状态快照保存到文件中。broker 在启动时会从快照文件中读取最新状态，然后通过复制当前首领来更新生产者状态。当它准备好再次成为首领时，内存中已经有了最新的序列号。

如果 broker 发生崩溃，但没有更新最后一个快照，会发生什么呢？生产者 ID 和序列号也是 Kafka 消息格式的一部分。在进行故障恢复时，我们将通过读取旧快照和分区最新日志片段里的消息来恢复生产者状态。等故障恢复完成，一个新的快照就保存好了。

如果分区里没有消息，会发生什么呢？假设某个主题的数据保留时间是两小时，但在过去的两小时内没有新消息到达——如果 broker 发生崩溃，则没有消息可以用来恢复状态。幸运的是，没有消息也就意味着没有重复消息。我们可以立即开始接收新消息（同时将状态缺失的警告信息记录下来），并创建生产者状态。

8.1.2 幂等生产者的局限性

幂等生产者只能防止由生产者内部重试逻辑引起的消息重复。对于使用同一条消息调用两次 `producer.send()` 就会导致消息重复的情况，即使使用幂等生产者也无法避免。这是因为生产者无法知道这两条消息实际上是一样的。通常建议使用生产者内置的重试机制，而不是在应用程序中捕获异常并自行进行重试。使用幂等生产者是在进行重试时避免消息重复的最简单的方法。

应用程序有多个实例或一个实例有多个生产者的情况非常常见。如果两个生产者尝试发送同样的消息，则幂等生产者将无法检测到消息重复。这在一些应用程序中非常常见，例如，从数据源（一个文件目录）获取数据，并将其生成到 Kafka 中。如果碰巧应用程序有两个实例在读取相同的文件并将记录生成到 Kafka，那么我们将会收到多个同样的消息副本。

幂等生产者只能防止因生产者自身的重试机制而导致的消息重复，不管这种重试是由生产者、网络还是 broker 错误所导致。

8.1.3 如何使用幂等生产者

幂等生产者使用起来非常简单，只需在生产者配置中加入 `enable.idempotence=true`。如果生产者已经配置了 `acks=all`，那么在性能上就不会有任何差异。在启用了幂等生产者之后，会发生下面这些变化。

- 为了获取生产者 ID，生产者在启动时会调用一个额外的 API。
- 每个消息批次里的第一条消息都将包含生产者 ID 和序列号（批次里其他消息的序列号基于第一条消息的序列号递增）。这些新字段给每个消息批次增加了 96 位（生产者 ID 是长整型，序列号是整型），这对大多数工作负载来说几乎算不上是额外的开销。
- broker 将会验证来自每一个生产者实例的序列号，并保证没有重复消息。

- 每个分区的消息顺序都将得到保证，即使 `max.in.flight.requests.per.connection` 被设置为大于 1 的值（5 是默认值，这也是幂等生产者可以支持的最大值）。

在 Kafka 2.5 中，幂等生产者的逻辑和错误处理都有了显著改进（生产者端和 broker 端），这要得益于 KIP-360。在 Kafka 2.5 之前，生产者状态并非总是保留足够长的时间，从而导致在各种情况下出现致命的 `UNKNOWN_PRODUCER_ID` 错误。（分区重分配存在一种已知的边缘情况，即如果一个分区副本在某个生产者写入消息之前成为新首领，那么它就没有这个分区的状态。）另外，之前的版本会试图在发生某些错误的情况下重写序列号，这可能会导致消息重复。在新版本中，如果遇到一个致命错误，那么这个消息批次和请求中的所有批次都将被拒绝。应用程序开发人员可以捕捉到异常，并决定是跳过这些记录，还是冒着消息重复和乱序的风险进行重试。

8.2 事务

本章开头提到，为保证 Streams 应用程序的正确性，Kafka 中加入了事务机制。为了让流式处理应用程序生成正确的结果，要保证每个输入的消息都被精确处理一次，即使是在发生故障的情况下。Kafka 的事务机制可以保证流式处理应用程序生成准确的结果，这样开发人员就可以在对准确性要求较高的场景中使用流式处理了。

Kafka 的事务机制是专门为流式处理应用程序而添加的。因此，它非常适用于流式处理应用程序的基础模式，即“消费–处理–生产”。事务可以保证流式处理的精确一次性语义——在更新完应用程序内部状态并将结果成功写入输出主题之后，对每个输入消息的处理就算完成了。8.2.4 节将介绍几种不适合启用 Kafka 精确一次性保证的场景。

事务是底层机制的名字。精确一次性语义或精确一次性保证是流式处理应用程序的行为。Streams 使用事务来实现精确一次性保证。其他流式处理框架，比如 Spark Streaming 或 Flink，则使用不同的机制来为用户提供精确一次性保证。

8.2.1 事务的应用场景

一些流式处理应用程序对准确性要求较高，特别是如果处理过程包含了聚合或连接操作，那么事务对它们来说就会非常有用。如果流式处理应用程序只进行简单的转换和过滤，那么就不需要更新内部状态，即使出现了重复消息，也可以很容易地将它们过滤掉。但是，如果流式处理应用程序对几条消息进行了聚合，一些输入消息被统计了不止一次，那么就很难知道结果是不是错误的。如果不重新处理输入消息，则不可能修正结果。

金融行业的应用程序就是典型的复杂流式处理的例子，在这些应用程序中，精确一次性被用于保证精确的聚合结果。不过，因为可以非常容易地在 Streams 应用程序中启用精确一次性保证，所以已经有非常多的应用场景（如聊天机器人）启用了这个特性。

8.2.2 事务可以解决哪些问题

假设有一个简单的流式处理应用程序：它从源主题读取消息，然后可能会对消息做一些处理，再将结果写入另一个主题。我们想要确保处理的每一条消息的结果只被写入一次。那么，哪些地方有可能出错呢？

事实证明，很多地方有可能出错。下面来看看其中的两种情况。

01. 应用程序崩溃导致的重复处理

    在从源集群读取并处理了消息之后，应用程序必须做两件事：一是将结果写入输出主题，二是提交已处理的消息的偏移量。假设这两个动作就按照这个顺序发生。如果应用程序在发送结果之后发生崩溃，但偏移量还没有提交，该怎么办？

    第 4 章讨论过当消费者崩溃时会发生什么。几秒之后，因为没有心跳，所以将触发再均衡，消费者读取的分区将被重新分配给其他消费者。新消费者将从最后提交的偏移量的位置开始读取这些分区的消息。在最后一个提交的偏移量和应用程序发生崩溃那个位置之间的消息将被再次处理，结果也将被再次写入输出主题——这就出现了重复。

02. “僵尸”应用程序导致的重复处理

    如果应用程序从 Kafka 读取了一个消息批次，但还没有开始处理它们就被挂起或与 Kafka 断开了连接，那么这个时候会发生什么？

就像前面的场景一样，在停止发送心跳一段时间之后，应用程序将被认为已经“死亡”，它的分区将被重新分配给消费者群组里的其他消费者。新消费者将重新读取这个消息批次，对其进行处理，并将结果写入输出主题，然后继续。

这个时候，之前的应用程序实例（被挂起的那个）可能又恢复过来了：继续处理它最近读取的消息批次，并将结果写入输出主题。所有这些都可以在它向 Kafka 轮询更多消息或发送心跳，然后发现它被认为已经“死亡”，并且现在有另外一个实例拥有这些分区之前完成。

一个“死亡”但不知道自己已经“死亡”的消费者被称为“僵尸”。在这个场景中，如果没有额外的保证，则“僵尸”消费者可以向输出主题生成结果，进而导致重复。

8.2.3 事务是如何保证精确一次性的

继续以流式处理应用程序为例。它会从一个主题读取数据，对数据进行处理，再将结果写入另一个主题。精确一次处理意味着消费、处理和生产都是**原子**操作，要么提交偏移量和生成结果这两个操作都成功，要么都不成功。我们要确保不会出现只有部分操作执行成功的情况（提交了偏移量但没有生成结果，反之亦然）。

为了支持这种行为，Kafka 事务引入了**原子多分区写入**的概念。我们知道，提交偏移量和生成结果都涉及向分区写入数据，结果会被写入输出主题，偏移量会被写入 `consumer_offsets` 主题。如果可以打开一个事务，向这两个主题写入消息，如果两个写入操作都成功就提交事务，如果不成功就中止，并进行重试，那么就会实现我们所追求的精确一次性语义。

图 8-1 是一个简单的流式处理应用程序，它会在执行原子多分区写入的同时提交消息偏移量。

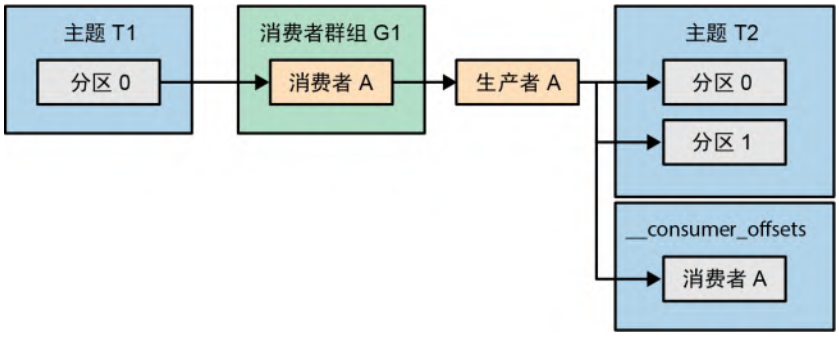

图 8-1：执行原子多分区写入的事务性生产者

为了启用事务和执行原子多分区写入，我们使用了**事务性生产者**。事务性生产者实际上就是一个配置了 `transactional.id` 并用 `initTransactions()` 方法初始化的 Kafka 生产者。与 `producer.id`（由 broker 自动生成）不同，`transactional.id` 是一个生产者配置参数，在生产者重启之后仍然存在。实际上，`transactional.id` 主要用于在重启之后识别同一个生产者。broker 维护了 `transactional.id` 和 `producer.id` 之间的映射关系，如果对一个已有的 `transactional.id` 再次调用 `initTransactions()` 方法，则生产者将分配到与之前一样的 `producer.id`，而不是一个新的随机数。

防止“僵尸”应用程序实例重复生成结果需要一种**“僵尸”隔离**机制，或者防止“僵尸”实例将结果写入输出流。通常可以使用 epoch 来隔离“僵尸”。在调用 `initTransaction()` 方法初始化事务性生产者时，Kafka 会增加与 `transactional.id` 相关的 epoch。带有相同 `transactional.id` 但 epoch 较小的发送请求、提交请求和中止请求将被拒绝，并返回 `FencedProducer` 错误。旧生产者将无法写入输出流，并被强制 `close()`，以防止“僵尸”引入重复记录。Kafka 2.5 及以上版本支持将消费者群组元数据添加到事务元数据中。这些元数据也被用于隔离“僵尸”，在对“僵尸”实例进行隔离的同时允许带有不同事务 ID 的生产者写入相同的分区。

在很大程度上，事务是一个生产者特性。创建事务性生产者、开始事务、将记录写入多个分区、生成偏移量并提交或中止事务，这些都是由生产者完成的。然而，这些还不够。以事务方式写入的记录，即使是最

终被中止的部分，也会像其他记录一样被写入分区。消费者也需要配置正确的隔离级别，否则将无法获得我们想要的精确一次性保证。

我们通过设置 `isolation.level` 参数来控制消费者如何读取以事务方式写入的消息。如果设置为 `read_committed`，那么调用 `consumer.poll()` 将返回属于已成功提交的事务或以非事务方式写入的消息，它不会返回属于已中止或执行中的事务的消息。默认的隔离级别是 `read_uncommitted`，它将返回所有记录，包括属于执行中或已中止的事务的记录。配置成 `read_committed` 并不能保证应用程序可以读取到特定事务的所有消息。也可以只订阅属于某个事务的部分主题，这样就可以只读取部分消息。此外，应用程序无法知道事务何时开始或结束，或者哪些消息是哪个事务的一部分。

图 8-2 对比了在 `read_committed` 隔离级别和默认的 `read_uncommitted` 隔离级别下，消费者可以看到哪些记录。

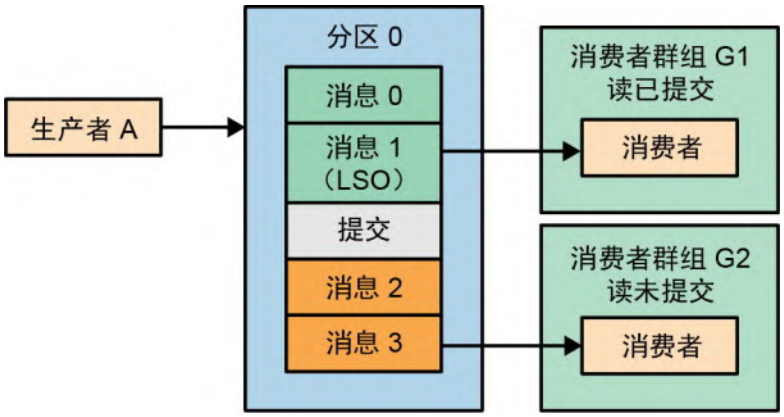

图 8-2：`read_committed` 隔离级别下的消费者比默认隔离级别下的消费者滞后

为了保证按顺序读取消息，`read_committed` 隔离级别将不返回在事务开始之后（这个位置也被叫作最后稳定偏移量，last stable offset，LSO）生成的消息。这些消息将被保留，直到事务被生产者提交或终止，或者事务超时（通过 `transaction.timeout.ms` 参数指定，默认为 15 分钟）并被 broker 终止。长时间使事务处于打开状态会导致消费者延迟，从而导致更高的端到端延迟。

我们的流式处理应用程序的输出结果具备了精确一次性保证，即使输入消息是以非事务方式写入的。原子多分区写入可以保证在将输出记录提交到输出主题的同时也提交了输入记录的偏移量，所以输入记录不会被重复处理。

8.2.4 事务不能解决哪些问题

之前讲过，在 Kafka 中加入事务是为了提供多分区原子写入（不是读取），并隔离流式处理应用程序中的“僵尸”生产者。在“消费–处理–生产”流式处理任务中，事务为我们提供了精确一次性保证。在其他场景中，事务要么不起作用，要么需要做额外的工作才能获得我们想要的结果。

人们对事务存在两个误解，他们假设精确一次性保证也适用于向 Kafka 写入消息之外的动作，并认为消费者总是能够读取整个事务和获取事务的边界信息。

下面是 Kafka 事务无法实现精确一次性保证的几种场景。

01. 在流式处理中执行外部操作

    假设流式处理应用程序在处理数据时需要向用户发送电子邮件。在应用程序中启用精确一次性语义并不能保证邮件只发送一次，这个保证只适用于将记录写入 Kafka。使用序列号去重，或者使用标记中止或取消事务在 Kafka 中是有效的，但它不会撤销已发送的电子邮件。在流式处理应用程序中执行的任何带有外部效果的操作都是如此：调用 REST API、写入文件，等等。

02. 从 **Kafka** 中读取数据并写入数据库

在这种情况下，应用程序会将数据写入外部数据库，而不是 Kafka。这里没有生产者参与，我们用数据库驱动器（如 JDBC）将记录写入数据库，消费者会将偏移量提交给 Kafka。没有任何一种机制允许将外部数据库写入操作与向 Kafka 提交偏移量的操作放在同一个事务中。不过，我们可以在数据库中维护偏移量（参见第 4 章），并在一个事务中将数据和偏移量一起提交到数据库——这将依赖于数据库的事务保证机制而不是 Kafka 的事务保证机制。

微服务通常需要在一个原子事务中更新数据库并向 Kafka 发布消息，要么两个操作都生效，要么都不生效。前面的两个例子中已经解释过，Kafka 事务做不到这样。

这个问题的一种常见解决方案是使用发件箱模式。微服务只会将消息发布到一个 Kafka 主题（也就是“发件箱”），然后另外一个独立的消息中继服务会从 Kafka 读取消息并更新数据库。因为 Kafka 不会保证数据库操作的精确一次性更新，所以需要确保数据库更新是幂等的。

这个模式可以保证消息最终到达 Kafka、主题消费者和数据库，或者都不到达。

这个模式的反向模式是将数据库作为发件箱，另外一个中继服务将确保将数据库更新也作为消息发送给 Kafka。如果可以使用关系数据库内置的约束（比如唯一索引和外键），那么这种模式就是首选。Debezium 项目主页上的一篇博文“Reliable Microservices Data Exchange With the Outbox Pattern”深入探讨了发件箱模式，并提供了详细的例子。

03. 从一个数据库读取数据写入 **Kafka**，再从 **Kafka** 将数据写入另一个数据库

我们倾向于认为可以构建出这样的应用程序：从一个数据库读取数据写入 Kafka，再从 Kafka 将数据写入另一个数据库，并仍然能够保持源数据库的事务。

不幸的是，Kafka 并不支持这种端到端的事务保证。除了很难保证在同一个事务中提交记录和偏移量，还有另外一个难点：Kafka 的 `read_committed` 隔离级别太弱了，根本无法保留数据库的事务。消费者不仅看不到未提交的消息，也不保证可以看到事务中已提交的所有消息，因为消息在某些主题上可能会滞后，而消费者没有事务的边界信息，所以它不知道事务何时开始和结束，也不知道看到的是部分消息还是全部消息。

04. 将数据从一个集群复制到另一个集群

这个场景有点儿微妙：将数据从一个 Kafka 集群复制到另一个集群是有可能支持精确一次性保证的。一个为 MirrorMaker 2.0 添加精确一次性语义的改进提议（“KIP-656: MirrorMaker2 Exactly-once Semantics”）对此进行了描述。在撰写本章时，这个提议仍处于草稿阶段，但算法已经描述得很清楚了，就是保证源集群中的每条记录都将被精确地复制到目标集群一次。

但这并不能保证事务是原子的。如果一个应用程序以事务的方式生成了几条记录和偏移量，然后 MirrorMaker 2.0 将它们复制到了另一个 Kafka 集群，那么在复制过程中事务属性就有可能会丢失。造成事务属性丢失的原因与将数据从 Kafka 复制到关系数据库一样：从 Kafka 读取数据的消费者并不知道或者能够保证已经读取了事务的所有记录。如果它只订阅了部分主题，那么就会只复制事务的一部分记录。

05. 发布和订阅模式

这里有一个更为微妙的例子。前面讨论的精确一次性保证是“消费–处理–生产”的模式，而发布和订阅也是一个非常常见的模式。事务为发布和订阅模式提供了一些保证：配置了 `read_committed` 隔离

级别的消费者将看不到已中止事务的消息，但这种保证并不是精确一次性的。消费者可以多次处理一条消息，具体取决于它们的偏移量提交逻辑。

对于发布和订阅模式，Kafka 提供的保证与 JMS 事务类似，消费者需要配置成 `read_committed` 隔离级别，以保证未提交的事务对消费者是不可见的。JMS broker 对所有消费者隐藏了未提交的事务。

我们要避免的一种模式是在发布消息之后等待另一个应用程序响应，然后才提交事务，而那个应用程序直到事务被提交之后才能收到消息，从而导致死锁。

8.2.5 如何使用事务

事务既是一个 broker 特性，也是 Kafka 协议的一部分，所以有多种客户端支持事务。

使用事务的最常见也最推荐的方式是在 Streams 中启用精确一次性保证。无须直接管理事务，Streams 会自动提供我们需要的保证。事务最初就是为这个场景而设计的，所以在 Streams 中启用事务是最简单也最有可能符合我们预期的方式。

要在 Streams 应用程序中启用精确一次性保证，只需要将 `processing.guarantee` 设置为 `exactly_once` 或 `exactly_once_beta`。

`exactly_once_beta` 在处理发生崩溃或因执行中的事务而被挂起的应用程序时会有一些不同。它是在 Kafka 2.5 中被引入 broker 的，并在 Kafka 2.6 中被引入 Streams。它的主要好处是可以用一个事务性生产者处理多个分区，因此可以创建出更加可伸缩的 Streams 应用程序。可以在 Kafka 改进提案“KIP-447: Producer scalability for exactly once semantics”中看到有关它的更多信息。

如果想在不使用 Streams 的情况下获得精确一次性保证，该怎么办？这个时候，可以直接使用事务 API。下面的代码片段演示了如何使用事务 API。Kafka 的 GitHub 代码库中有完整的例子，包括一个演示驱动器和一个简单的精确一次性处理器。

```
Properties producerProps = new Properties();
producerProps.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
producerProps.put(ProducerConfig.CLIENT_ID_CONFIG, "DemoProducer");
producerProps.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG, transactionalId); ❶

producer = new KafkaProducer<>(producerProps);

Properties consumerProps = new Properties();
consumerProps.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
consumerProps.put(ConsumerConfig.GROUP_ID_CONFIG, groupId);
props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "false"); ❷
consumerProps.put(ConsumerConfig.ISOLATION_LEVEL_CONFIG, "read_committed"); ❸

consumer = new KafkaConsumer<>(consumerProps);

producer.initTransactions(); ❹

consumer.subscribe(Collections.singleton(inputTopic)); ❺

while (true) {
  try {
    ConsumerRecords<Integer, String> records =
      consumer.poll(Duration.ofMillis(200));
    if (records.count() > 0) {
      producer.beginTransaction(); ❻
      for (ConsumerRecord<Integer, String> record : records) {
        ProducerRecord<Integer, String> customizedRecord = transform(record); ❼
        producer.send(customizedRecord);
      }
      Map<TopicPartition, OffsetAndMetadata> offsets = consumerOffsets();
      producer.sendOffsetsToTransaction(offsets, consumer.groupMetadata());❽
```

```
      producer.commitTransaction(); ❾
    }
  } catch (ProducerFencedException|InvalidProducerEpochException e) { ❿
    throw new KafkaException(String.format(
    "The transactional.id %s is used by another process", transactionalId));
  } catch (KafkaException e) {
    producer.abortTransaction(); ⓫
    resetToLastCommittedPositions(consumer);
  }}
```

❶ 为生产者配置 `transactional.id`，让它成为一个能够进行原子多分区写入的事务性生产者。事务 ID 必须是唯一且长期存在的，因为本质上就是用它定义了应用程序的一个实例。

❷ 消费者不提交自己的偏移量——生产者会将偏移量提交作为事务的一部分，所以需要禁用自动提交。

❸ 在这个例子中，消费者会从输入主题读取数据。假设输入主题中的消息是由事务性生产者写入的（只是为了好玩儿，实际上我们对输入没有硬性要求）。为了干净地读取事务（忽略执行中和已中止的事务），可以将消费者隔离级别设置为 `read_committed`。需要注意的是，除了读取已提交的事务，消费者也会读取非事务性的写入。

❹ 事务性生产者要做的第一件事是初始化，包括注册事务 ID 和增加 epoch 的值（确保其他具有相同 ID 的生产者将被视为“僵尸”，并中止具有相同事务 ID 的旧事务）。

❺ 这里使用了消费者订阅 API，分配给应用程序实例的分区可以在触发再均衡时发生变更。在 Kafka 2.5（KIP-447 的 API 变更是在这个版本中引入的）之前这么做有一定的困难。事务性生产者需要被固定分配到一组分区，因为事务隔离机制要求相同分区对应的事务 ID 必须相同。（如果事务 ID 发生变化，则“僵尸”隔离保护机制将失效。）KIP-447 中加入了新 API（已在这个例子中使用），将消费者群组信息附加到了事务中，用于实现“僵尸”隔离。在使用这个方法时，如果相关分区被撤销，则事务也需要被提交。

❻ 我们读取了记录，现在要处理它们并生成结果。这个方法可以保证从调用它开始，一直到事务被提交或被中止，生成的所有内容都是事务的一部分。

❼ 在这里处理消息——所有的业务逻辑都在这里。

❽ 之前已经提到过，需要将偏移量提交作为事务的一部分，这样可以保证如果生成结果失败，则未成功处理的消息的偏移量将不会被提交。这个方法会将偏移量提交作为事务的一部分。需要注意的是，不要通过其他方式提交偏移量（禁用偏移量自动提交），也不要调用其他提交偏移量的 API。通过其他方式提交偏移量将无法提供事务保证。

❾ 我们生成了需要的东西，并将偏移量提交作为事务的一部分，现在可以提交事务了。一旦这个方法成功返回，整个事务就完成了，就可以继续读取和处理下一批消息了。

❿ 如果遇到这个异常，则说明应用程序实例变成“僵尸”了。我们的应用程序实例可能由于某种原因被挂起或断开连接，而另一个具有相同事务 ID 的应用程序实例已经在运行当中。很有可能我们启动的事务已经被中止，其他应用程序正在处理这些记录。这个应用程序实例除了优雅地“死去”，别无他法。

⓫ 如果在提交事务时遇到错误，则可以中止事务，重置消费者偏移量位置，并进行重试。

8.2.6 事务 ID 和隔离

为生产者设置事务 ID 这一步非常重要，它不像表面上看起来那么简单。错误地分配事务 ID 有可能导致应用程序出现错误或无法提供精确一次性保证。非常关键的是，一个应用程序实例的事务 ID 在重启前后必须保持一致，而且应用程序的不同实例的事务 ID 不能一样，否则 broker 将无法隔离“僵尸”实例。

在 Kafka 2.5 之前，隔离“僵尸”的唯一方法是将事务 ID 与分区形成固定映射，这样可以保证每个分区总是对应相同的事务 ID。假设有一个事务 ID 为 A 的生产者从主题 T 读取消息并断开了连接，事务 ID 为 B 的

新生产者取代了它，后来，生产者 A 变成了“僵尸”，但它不会被隔离，因为它的事务 ID 与生产者 B 的事务 ID 不一样。我们希望生产者 A 总是被新生产者 A 取代，新生产者 A 有更高的 epoch，这样“僵尸”A 就会被隔离。在旧版本中，事务 ID 是随机分配给线程的，不保证始终使用相同的事务 ID 写入相同的分区。

Kafka 2.5 中引入了除事务 ID 之外的第二种基于消费者群组元数据的隔离方法（KIP-447）。我们会调用生产者的偏移量提交方法，并将消费者群组元数据（而不只是消费者群组 ID）作为参数传给它。

假设主题 T1 有两个分区，分别是 `t-0` 和 `t-1`。两个分区分别被同一消费者群组中的两个消费者消费，每个消费者都将消息传给对应的事务性生产者——一个事务 ID 为 A，另一个事务 ID 为 B，它们分别向主题 T2 的分区 0 和分区 1 写入结果，如图 8-3 所示。

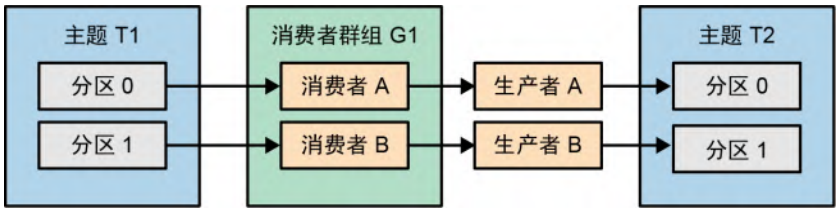

图 8-3：事务性消息处理器

如图 8-4 所示，如果消费者 A 和生产者 A 所在的应用程序实例变成“僵尸”，则消费者 B 将开始读取两个分区。如果想保证不会有“僵尸”写入分区 0，那么消费者 B 就不能读取分区 0 以及用事务 ID B 写入分区 0。应用程序需要实例化一个事务 ID 为 A 的新生产者，该生产者可以安全地写入分区 0，并隔离事务 ID 为 A 的旧生产者。但这样做有点儿浪费，我们可以在事务中包含消费者群组信息，生产者 B 的事务将显示它们来自新一代消费者群组，所以它们可以通过，而“僵尸”生产者 A 的事务将显示它们来自老一代消费者群组，所以它们将被隔离。

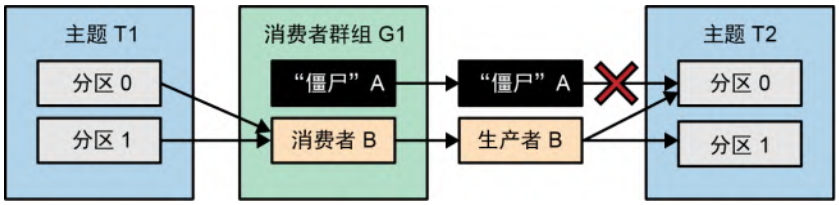

图 8-4：发生再均衡之后的事务性消息处理器

8.2.7 事务的工作原理

要使用事务，可以直接调用 API，不需要知道它的工作原理。但是，在诊断行为不符合预期的应用程序时，了解其内部原理有助于更好地诊断问题。

Kafka 事务的基本算法受到了 Chandy-Lamport 快照的启发，它会将一种被称为“标记”（marker）的消息发送到通信通道中，并根据标记的到达情况来确定一致性状态。Kafka 事务根据标记消息来判断跨多个分区的事务是否被提交或被中止——当生产者要提交一个事务时，它会发送“提交”消息给事务协调器，事务协调器会将提交标记写入所有涉及这个事务的分区。如果生产者在向部分分区写入提交消息后发生崩溃，该怎么办？Kafka 事务使用两阶段提交和事务日志来解决这个问题。总的来说，这个算法会执行如下步骤。

01. 记录正在执行中的事务，包括所涉及的分区。
02. 记录提交或中止事务的意图——一旦被记录下来，到最后要么被提交，要么被中止。
03. 将所有事务标记写入所有分区。
04. 记录事务的完成情况。

要实现这个算法，Kafka 需要一个事务日志。这里使用了一个叫作 `__transaction_state` 的内部主题。

我们将通过分析前面代码片段中所使用的事务 API 的内部原理来了解这个算法。

在开始第一个事务之前，生产者需要通过调用 `initTransaction()` 来注册自己。这个请求会被发送给一个 broker，它将成为这个事务性生产者的**事务协调器**。就像每一个 broker 都是部分消费者群组的消费者群组协调器一样，每一个 broker 都是部分生产者的事务协调器。每一个事务 ID 对应的事务协调器就是映射到这个事务 ID 的事务日志分区的首领。

`initTransaction()` API 注册了一个带有新事务 ID 的协调器或者增加现有事务 ID 的 epoch，用以隔离变成“僵尸”的旧生产者。当 epoch 增加时，挂起的事务将被中止。

下一步是调用 `beginTransaction()`。这个方法不是协议的一部分，它只是告诉生产者，现在有一个正在执行中的事务。broker 端的事务协调器仍然不知道事务已经开始。不过，一旦生产者开始发送消息，每次生产者检测到消息被发送给一个新分区时，都会向 broker 发送 `AddPartitionsToTxnRequest` 请求，告诉 broker 自己有一个执行中的事务，并且这些分区是事务的一部分。这些信息将被记录在事务日志中。

当生成结果并准备提交事务时，首先需要提交在这个事务中处理好的消息的偏移量。偏移量可以在任何时候提交，但一定要在事务提交之前。`sendOffsetsToTransaction()` 方法将向事务协调器发送一个请求，其中包含了偏移量和消费者群组 ID。事务协调器将用消费者群组 ID 查找群组协调器，并提交偏移量。

现在是提交或中止事务的时候了。`commitTransaction()` 方法或 `abortTransaction()` 方法将向事务协调器发送一个 `EndTransactionRequest`。事务协调器会把提交或中止事务的意图记录到事务日志中。如果这个步骤执行成功，那么事务协调器将负责完成提交（或中止）过程。它会向所有涉及事务的分区写入一个提交标记，然后将提交成功的信息写入事务日志。需要注意的是，如果事务协调器在记录提交意图之后以及在完成提交流程之前被关闭或发生崩溃，那么将会选举出一个新的事务协调器，它会从事务日志中获取提交意图，并完成提交流程。

如果一个事务未能在 `transaction.timeout.ms` 指定的时间内提交或中止，则事务协调器将自动中止它。

每个收到由事务性或幂等生产者发送的消息的 broker 都会在内存中保存生产者 ID 或事务性 ID，以及生产者发送的最后 5 个消息批次的相关状态：序列号、偏移量等。这些状态在生产者停止活动之后会继续保留 `transactional.id.expiration.ms` 指定的时间（默认为 7 天）。这样生产者就可以在不抛出 `UNKNOWN_PRODUCER_ID` 异常的情况下恢复活动。如果以非常高的速率创建新的幂等生产者或事务 ID，但从不重用它们，则可能会导致 broker 发生内存泄漏。如果在一周内连续每秒新增 3 个幂等生产者，那么将产生 180 万个状态条目，总共需要保存 900 万个消息批次元数据，占用大约 5 GB 内存。这可能会导致 broker 内存不足或出现严重的垃圾回收停顿。建议在应用程序启动时初始化几个长期使用的生产者，并在应用程序生命周期中重用它们。如果不能这么做（FaaS 会让这变得很困难），那么建议减小 `transactional.id.expiration.ms` 的值，这样事务 ID 就会更快过期，不会让旧状态占用 broker 太大的内存。

8.3 事务的性能

事务给生产者带来了一些额外的开销。事务 ID 注册在生产者生命周期中只会发生一次。分区事务注册最多会在每个分区加入每个事务时发生一次，然后每个事务会发送一个提交请求，并向每个分区写入一个额外的提交标记。事务初始化和事务提交请求都是同步的，在它们成功、失败或超时之前不会发送其他数据，这进一步增加了开销。

需要注意的是，生产者在事务方面的开销与事务包含的消息数量无关。因此，一个事务包含的消息越多，相对开销就越小，同步调用次数也就越少，从而提高了总体吞吐量。

在消费者方面，读取提交标记会增加一些开销。事务对消费者的性能影响主要是在 `read_committed` 隔离级别下的消费者无法读取未提交事务所包含的记录。提交事务的时间间隔越长，消费者在读取到消息之前需要等待的时间就越长，端到端延迟也就越高。

但是，消费者不需要缓冲未提交事务所包含的消息，因为 broker 不会将它们返回给消费者。由于消费者在读取事务时不需要做额外的工作，因此吞吐量不受影响。

8.4 小结

Kafka 的精确一次性语义与国际象棋正好相反：要理解它不容易，但用起来很简单。

本章介绍了 Kafka 实现精确一次性语义的两个关键机制：幂等生产者（避免由重试机制导致的重复处理）和事务（Streams 精确一次性语义的基础）。

通过一个配置就可以启用它们，这样就可以很方便地在要求更少重复和更高正确性的应用程序中使用 Kafka 了。

本章深入讨论了一些特定场景，并介绍了一些实现细节。在进行应用程序故障诊断或直接使用事务 API 时，了解这些细节至关重要。

了解 Kafka 精确一次性语义能够在哪些场景中提供什么样的保证，有助于设计出具有精确一次性保证的应用程序。我们不应该为应用程序的任何行为感到意外，而本章内容可以帮助我们避免这些意外。

第 9 章　构建数据管道

在使用 Kafka 构建数据管道时，通常有两种应用场景：第一种，把 Kafka 作为数据管道的两个端点之一，例如，把 Kafka 中的数据移动到 S3，或者把 MongoDB 中的数据移动到 Kafka；第二种，把 Kafka 作为数据管道两个端点的中间媒介，例如，为了把 Twitter 中的数据移动到 ElasticSearch，需要先把它们移动到 Kafka，然后再从 Kafka 中移动到 ElasticSearch。

LinkedIn 和其他一些大公司都将 Kafka 用于上述两种场景中，后来，Kafka 0.9 中加入了 Kafka Connect（以下简称 Connect）。每家企业在将 Kafka 集成到数据管道中时总会碰到一些特定的问题，于是我们决定加入一些 API 来帮助它们解决这些问题，而不是让它们自己从头开始想办法。

Kafka 为数据管道带来的主要价值在于，它可以作为数据管道各个数据阶段之间的大型缓冲区，有效解耦数据的生产者和消费者，让同一个数据源的数据可以被多个具有不同可用性需求的系统和应用程序使用。Kafka 因其出色的解耦能力以及在可靠性、安全和效率方面的良好表现，非常适合用来构建数据管道。

数据集成

有些组织会将 Kafka 看成数据管道的一个端点，它们想的是“怎样才能把数据从 Kafka 中移动到 ElasticSearch”。这么想情有可原，特别是当你需要的 ElasticSearch 数据还停留在 Kafka 中的时候，你也会这么想。不过，我们要讨论的是如何在更大的场景中使用 Kafka，这些场景至少包含了两个端点（可能会更多），并且这些端点都不是 Kafka。建议那些面临数据集成问题的人要从大局考虑问题，而不是只关注眼前的这些端点。过度聚焦短期问题会导致后期过高的复杂性和维护成本。

本章将讨论在构建数据管道时需要考虑的几个常见问题。这些问题并非 Kafka 独有，它们都是与数据集成相关的一般性问题。我们首先会解释为什么可以使用 Kafka 进行数据集成，以及它是如何解决这些问题的。还会介绍 Connect API 与客户端 API（生产者和消费者）的区别，以及它们分别适用于什么样的场景。然后会详细介绍 Connect。Connect 的完整手册不在本章的讨论范围之内，不过我们会举几个例子帮助你入门，而且会告诉你可以从哪里了解到更多有关 Connect 的信息。最后会介绍其他的数据集成系统，以及它们如何与 Kafka 集成。

9.1 构建数据管道时需要考虑的问题

本书不打算讲解所有与构建数据管道相关的细节，但会着重讨论在集成多个系统时需要考虑的几个最重要的问题。

9.1.1 及时性

有些系统希望每天一次性接收大量数据，有些系统则希望在数据生成几毫秒之后就能获取它们。大部分数据管道介于这两者之间。良好的数据集成系统可以支持各种数据管道的及时性需求，而且在业务需求发生变更时能够轻松地在不同的时间表之间迁移。作为一个基于流的数据平台，Kafka 提供了可靠且可伸缩的数据存储，可以支持从几近实时的数据管道到基于天的批处理。生产者既可以频繁地向 Kafka 写入数据，也可以根据实际需要写入。消费者可以在数据到达的第一时间就读取它们，也可以进行批处理：每小时运行一次，连接到 Kafka，并读取前一小时积压的数据。

Kafka 在这里扮演了一个大型缓冲区的角色，解开了生产者和消费者之间的时间敏感性耦合。生产者实时写入数据，消费者进行批处理，反之亦然。应用回压策略也因此变得更加容易。Kafka 本身就针对生产者使用了回压策略（必要时延后向生产者发送确认），因为读取速度完全是由消费者决定的。

9.1.2 可靠性

我们要避免单点故障，并能够自动从各种故障中快速恢复。数据通常经由数据管道到达业务系统，哪怕出现几秒的故障，也会造成灾难性的影响，对那些要求毫秒级及时性的系统来说尤为如此。数据传递保证是可靠性的另一个重要考量因素。有些系统允许数据丢失，但在大多数情况下，它们要求**至少一次传递**，也就是要求源系统的每一个事件都必须到达目的地，只是有时候重试操作可能造成重复传递。有些系统甚至要求**精确一次传递**——源系统的每一个事件都必须到达目的地，不允许丢失，也不允许重复。

第 7 章深入讨论过 Kafka 的可用性和可靠性保证。Kafka 本身支持至少一次传递，如果再结合支持事务模型或唯一键的外部存储系统，那么 Kafka 也能实现精确一次传递。因为大部分端点是数据存储系统，它们提供了精确一次传递的原语支持，所以基于 Kafka 的数据管道也能实现精确一次传递。值得一提的是，Connect API 为集成外部系统提供了处理偏移量的 API，我们可以很方便地构建出支持精确一次传递的端到端数据管道。实际上，很多开源的连接器支持精确一次传递。

9.1.3 高吞吐量和动态吞吐量

为了满足现代数据系统的需求，数据管道需要支持非常高的吞吐量。更重要的是，在某些情况下，数据管道还要能够应对突发的吞吐量增长。

如果将 Kafka 作为生产者和消费者之间的缓冲区，那么消费者吞吐量和生产者吞吐量就不会耦合在一起。没有必要实现复杂的回压机制，因为如果生产者吞吐量超过了消费者吞吐量，则可以把数据积压在 Kafka 中，等待消费者追赶上来。Kafka 支持单独增加额外的消费者或生产者，我们可以在数据管道的任何一端进行动态伸缩，以便满足不断变化的需求。

Kafka 是高吞吐量的分布式系统，一个一般规模的集群每秒可以处理数百兆数据，所以根本无须担心数据管道无法满足伸缩性需求。另外，Connect API 还擅长并行处理，既可以在单节点上进行，也可以扩展到多个节点，具体取决于系统的需求。稍后将介绍数据源和数据池（data sink）如何在多个线程间拆分任务，最大限度地利用 CPU 资源，即使运行在单个节点上。

Kafka 支持多种类型的压缩算法，在吞吐量增加时，Kafka 用户和管理员可以通过压缩来控制对网络和存储资源的使用。

9.1.4 数据格式

数据管道需要协调各种数据格式和数据类型，这是其在构建时需要考虑的一个非常重要的方面。数据类型取决于不同的数据库和存储系统。你可以通过 Avro 将 XML 或关系数据加载到 Kafka 中，然后将它们转成 JSON 写入 ElasticSearch，或者转成 Parquet 写入 HDFS，或者转成 CSV 写入 S3。

Kafka 本身和 Connect API 与数据格式无关。之前的章节介绍过，生产者和消费者可以使用各种序列化器来表示任意格式的数据。Connect API 有自己的包括数据类型和模式的内存对象模型，我们也可以使用一些可插拔的转换器将这些对象保存成任意格式。也就是说，不管数据是什么格式，都不会影响我们选择使用哪一种连接器。

很多数据源和数据池提供了模式，我们从数据源读取数据时会一并读取模式，并把它们保存起来，用于验证数据格式兼容性或更新数据池的模式。从 MySQL 到 Snowflake 的数据管道就是一个很好的例子。如果有人在 MySQL 中增加了一个字段，那么好的数据管道需要确保在加载新数据时，Snowflake 中也添加了同样的字段。

另外，在将 Kafka 中的数据写入外部系统时，数据池连接器需要负责处理数据格式。有些连接器的数据格式处理过程是可插拔的，比如 S3 的连接器就支持 Avro 和 Parquet。

通用的数据集成框架不仅要支持各种不同的数据类型，还要处理好不同数据源和数据池之间的行为差异。例如，Syslog 是一个可以推送数据的数据源，但如果要从关系数据库读取数据，则需要使用框架拉取。HDFS 只支持追加写入，我们只能向 HDFS 中写入数据，而其他大部分系统既可以追加数据，也可以更新数据。

9.1.5 转换

数据转换比其他方面的需求更具争议性。构建数据管道有两种方式，即 ETL 和 ELT。ETL 表示**提取–转换–加载**（extract-transform-load），当数据流经数据管道时，数据管道负责修改它们。这种方式为我们节省了时间和存储空间，因为不需要经历保存数据、修改数据、再保存数据这样的过程。不过，这种好处也要视情况而定。有时候，这种方式会给我们带来实实在在的好处，但有时候也可能给数据管道带来不适当的计算和存储负担。这种方式最主要的不足在于，在数据管道中进行数据转换会对数据管道下游的应用程序造成一些限制，特别是当下游应用程序希望对数据做进一步处理的时候。假设有人在 MongoDB 和 MySQL 之间建立了数据管道，并过滤掉了一些记录，或者移除了一些字段，那么下游应用程序只能从 MySQL 中访问到部分数据。如果它们想要访问被移除的字段，则只能重新构建管道，并重新处理历史数据（如果历史数据还可用的话）。

ELT 表示**提取–加载–转换**（extract-load-transform）。在这种模式下，数据管道只做少量的转换（主要是数据类型转换），确保到达数据池的数据尽可能与数据源保持相似。在这种数据管道架构中，目标系统会收集“原始数据”，并完成所有的处理任务。这种架构的好处在于，它为目标系统用户提供了最大的灵活性，因为它们可以访问到完整的数据。在这种架构中诊断问题也更加容易，因为数据处理被限定在一个系统中，而不是分散在数据管道和其他应用程序中。这种架构的不足在于，数据的转换占用了目标系统太多的 CPU 和存储资源。有时候，目标系统造价高昂，如果有可能，人们希望将计算任务移出这些系统。

Connect 提供了单一消息转换功能，在将消息从源系统复制到 Kafka 或从 Kafka 复制到目标系统时，可以对消息进行转换，包括将消息路由到不同的主题、过滤消息、修改数据类型、修改特定字段，等等。涉及连接和聚合的复杂转换可以通过 Kafka Streams 来完成，本书将在其他章节中详细探讨这些问题。

需要注意的是，在用 Kafka 构建 ETL 系统时，可以构建一对多的数据管道，也就是说，源数据被写入 Kafka 一次，然后被多个应用程序读取，并写入多个目标系统。我们可能需要做一些预处理和清理，比如标准化时间戳和数据类型、添加溯源信息以及移除个人信息，所有这些转换对数据使用者都是有利的。但不要过早地进行清理和优化，因为在其他地方可能需要更原始的数据。

9.1.6 安全性

安全性是人们一直关心的问题。对于数据管道的安全性，我们主要关心以下几个方面。

- 谁有权限访问写入 Kafka 的数据？
- 能否保证流经数据管道的数据是经过加密的？这是跨数据中心数据管道通常要考虑的一个关键问题。
- 谁有权限修改数据管道？
- 如果数据管道需要从一个不受信任的系统读取或写入数据，那么是否有适当的身份验证机制？
- PII（个人识别信息）的存储、访问和使用是否符合法律和监管的要求？

Kafka 支持加密传输数据，从数据源到 Kafka，再从 Kafka 到数据池。它还支持身份验证（通过 SASL 来实现）和授权，所以你可以确信，如果一个主题包含了敏感信息，那么在不经授权的情况下，数据不会流到不安全的系统中。Kafka 还提供了审计日志来跟踪未经授权的访问。通过编写额外的代码，还可能跟踪到每条消息来自哪里以及谁修改了消息，从而可以对每条消息进行溯源。

第 11 章将详细讨论 Kafka 的安全性。Connect 及其连接器需要在身份验证之后连接到外部数据系统，因此连接器的配置信息当中需要包含身份验证凭证。

不建议将凭证保存在配置文件中，因为如果这样就不得不格外小心地保护好这些文件。一种常见的解决方案是使用外部密钥管理系统，比如 HashiCorp Vault。Connect 支持外部密钥配置。Kafka 只提供了允许接入外部可插拔配置提供程序的框架和一个从文件中读取配置的提供程序示例。Kafka 社区开发了很多外部配置提供程序，可以与 Vault、AWS 和 Azure 集成。

9.1.7 故障处理

不能总是假设数据是完美的，而是要事先做好应对故障的准备。我们能否总是把缺损的数据挡在数据管道之外？我们能否恢复无法解析的记录？我们能否修复（或许由人工手动修复）并重新处理缺损的数据？如果在若干天之后才发现原先看起来正常的数据其实是缺损数据，该怎么办？

因为 Kafka 可以长时间保留数据，所以如果有必要，那么可以在适当的时候回过头来重新处理出错的数据。如果目标系统错过了一些消息，那么可以重放保存在 Kafka 中的事件。

9.1.8 耦合性和灵活性

数据管道最重要的作用之一是解耦数据源和数据池。数据管道会在很多情况下出现耦合。

临时数据管道

有些公司会为每一对相互连接的应用程序建立单独的数据管道。例如，它们使用 Logstash 将日志导入 ElasticSearch，使用 Flume 将日志导入 HDFS，使用 GoldenGate 将 Oracle 的数据导入 HDFS，使用 Informatica 将 MySQL 的数据或 XML 导入 Oracle，等等。它们将数据管道与特定的端点耦合起来，并创建了大量的集成点，这需要额外的部署、维护和监控。当有新的系统加入，它们需要构建额外的数据管道，从而增加了采用新技术的成本，也遏制了创新。

元数据丢失

如果数据管道没有保留模式元数据，并且不允许模式发生变更，那么最终会导致生产者和消费者之间发生紧密的耦合。没有了模式，生产者和消费者就需要额外的信息来解析数据。假设数据从 Oracle 流向 HDFS，如果 DBA 在 Oracle 中添加了一个字段，但没有保留模式信息，那么每一个从 HDFS 读取数据的应用程序都会出错，开发人员需要修改所有的应用程序才能解决问题。如果数据管道支持模式变更，那么各个应用程序就可以随意修改自己的代码，无须担心对整个系统造成破坏。

末端处理

之前在讨论数据转换时就已提到，数据管道难免要对数据做一些处理工作。毕竟，我们是在不同的系统之间移动数据，所以肯定会遇到不同的数据格式，需要支持不同的应用场景。不过，如果数据管道过多地处理数据，则会将下游系统与在构建数据管道时所做的决策联系在一起，比如如何保留字段、如何聚合数据，等等。如果下游系统发生变化，那么数据管道就要做出相应的修改，这种方式不仅不灵活，还低效且不安全。更为灵活的方式是尽量保留原始数据的完整性，让下游系统自己决定如何处理和聚合数据。

9.2 何时使用 Connect API 或客户端 API

在向 Kafka 写入数据或从 Kafka 读取数据时，要么使用传统的生产者和消费者客户端，就像第 3 章和第 4 章所描述的那样，要么使用即将在后续章节中介绍的 Connect API 和连接器。在具体介绍 Connect API 之前，你可能会问：“什么时候该用哪一个 API 呢？”

我们知道，Kafka 客户端是内嵌到应用程序中的，应用程序用它向 Kafka 写入数据或从 Kafka 读取数据。如果可以修改与 Kafka 相连接的应用程序的代码，或者想要将数据推送到 Kafka 或从 Kafka 读取数据，那么就可以使用 Kafka 客户端。

如果要将 Kafka 连接到数据存储系统，而这些系统不是你开发的，你无法或者不想修改它们的代码，那么可以使用 Connect。Connect 可用于从外部数据存储系统读取数据，或者将数据推送到外部存储系统。要使用 Connect，首先要有数据存储系统连接器，现在有很多可用的连接器。在实际使用时，Connect 用户只需提供配置文件即可。

如果要连接的数据存储系统没有相应的连接器，则可以考虑使用客户端 API 或 Connect API 开发一个。建议首选 Connect，因为它不仅提供了像配置管理、偏移量存储、并行处理、错误处理这样的开箱即用的特性，而且支持多种数据类型和标准的 REST 管理 API。开发一个连接 Kafka 和外部数据存储系统的小应用程序看起来很简单，但其实还有很多细节需要处理，比如数据类型和配置选项，这些无疑加大了开发的难度。另外，你还需要维护它并提供文档，因为你的队友需要学习如何使用它。Connect 是 Kafka 生态系统的一个标准组件，它为你处理了大部分细节，让你可以专注于处理与外部存储系统之间的数据传输。

9.3 KafkaConnect

Connect 是 Kafka 的一部分，它为 Kafka 和外部数据存储系统之间的数据移动提供了一种可靠且可伸缩的方式。它提供了一组 API 和一个运行时，我们可以用它们开发和运行**连接器插件**。Connect 会执行这些插件，并用它们移动数据。Connect 会以 **worker 集群**的方式运行。我们会在 worker 节点上安装连接器插件，然后用 REST API 来管理和配置**连接器**。连接器会启动额外的**任务**，充分利用 worker 节点上的资源来移动大量数据。数据源的连接器任务负责从源系统读取数据，并把数据对象提供给 worker。数据池的连接器任务负责从 worker 获取数据，并把它们写入目标系统。Connect 会使用**转换器**在 Kafka 中存储不同格式的数据。Kafka 本身支持 JSON 格式，Confluent 模式注册表提供了 Avro、Protobuf 和 JSON 格式的模式转换器。开发人员可以选择数据的存储格式，并且完全独立于他们所使用的连接器和他们处理模式（如果有的话）的方式。

本章无法完全涵盖 Connect 的所有细节和它的各种连接器，这些内容本身就可以单独写成一本书。不过，我们会对 Connect 进行概览，还会介绍如何使用它，并提供一些额外的参考资料。

9.3.1 运行 Connect

Connect 随 Kafka 一起发布，所以无须单独安装。如果你打算在生产环境中用 Connect 来移动大量数据或运行多个连接器，那么最好把 Connect 部署在单独的服务器上。你可以在所有的服务器上安装 Kafka，只启动部分服务器上的 broker，然后启动剩余服务器上的 Connect。

启动 worker 与启动 broker 差不多，只需在调用启动脚本时传入一个属性配置文件即可。

```
bin/connect-distributed.sh config/connect-distributed.properties
```

worker 有以下几个重要的配置参数。

`bootstrap.servers`

这个参数列出了将要与 Connect 协同工作的 broker 服务器，连接器会向这些 broker 写入数据或从它们那里读取数据。无须为这个参数提供所有 broker 的地址，不过建议至少指定 3 个。

`group.id`

具有相同组 ID 的 worker 属于同一个 Connect 集群。集群的连接器和它的任务可以运行在任意一个工作节点上。

`plugin.path`

Connect 采用了一种可插拔的架构，支持添加连接器、转换器、转换和密钥提供程序。为此，Connect 必须能够找到和加载这些插件。

可以配置一个或多个目录，Connect 可以在这些目录中找到连接器及其依赖项。例如，可以配置 `plugin.path=/opt/connectors,/home/gwenshap/connectors`。我们通常会在其中的一个目录中为每个连接器创建一个子目录。例如，对于前面的这个例子，我们将创建 /opt/connectors/jdbc 和 /opt/connectors/elastic 这两个子目录。在每个子目录中，我们会放置连接器的 JAR 包及其所有的依赖项。如果连接器是作为胖 JAR 包发行的，那么就可以直接把它放在 `plugin.path` 指定的目录中，不需要创建子目录。但需要注意的是，将依赖项放在顶级目录中是无效的。

还可以将连接器及其所有依赖项添加到 Connect 的类路径中。但不建议这么做，因为如果连接器的依赖项与 Kafka 的某个依赖项冲突，则可能会出现错误。推荐使用 `plugin.path` 配置参数。

`key.converter` 和 `value.converter`

Connect 可以处理多种数据格式。这两个参数分别指定了消息的键和值所使用的转换器。默认使用的是 Kafka 提供的 `JSONConverter`，当然也可以配置成 Confluent 模式注册表提供的 `AvroConverter`、

ProtobufConverter 或 JscoSchemaConverter。

有些转换器还提供了特定的配置参数。如果想使用这些配置参数，则需要给它们加上 key.converter. 或 value.converter. 这样的前缀。举个例子，JSON 格式的消息既可以包含模式也可以不包含模式，只需分别将 key.converter.schema.enable 设置成 true 或者 false 即可。消息的值类似，只需分别将 value.converter.schema.enable 设置成 true 或 false 即可。Avro 消息也包含了模式，不过需要使用 key.converter.schema.registry.url 和 value.converter.schema.registry.url 来指定模式注册表的位置。

rest.host.name 和 rest.port

一般通过 Connect 的 REST API 来配置和监控连接器。可以为 REST API 指定特定的端口。

在启动 worker 集群之后，可以通过 REST API 来验证它们是否运行正常。

```
$ curl http://localhost:8083/
{"version":"3.0.0-
SNAPSHOT","commit":"fae0784ce32a448a","kafka_cluster_id":"pfkYIGZQSXm8RylvACQHdg"}%
```

这个 REST URI 应该会返回当前 Connect 的版本号。我们运行的是 Kafka 3.0.0（预发布）快照版本。还可以检查已经安装好的连接器插件。

```
$ curl http://localhost:8083/connector-plugins

[
  {
    "class": "org.apache.kafka.connect.file.FileStreamSinkConnector",
    "type": "sink",
    "version": "3.0.0-SNAPSHOT"
  },
  {
    "class": "org.apache.kafka.connect.file.FileStreamSourceConnector",
    "type": "source",
    "version": "3.0.0-SNAPSHOT"
  },
  {
    "class": "org.apache.kafka.connect.mirror.MirrorCheckpointConnector",
    "type": "source",
    "version": "1"
  },
  {
    "class": "org.apache.kafka.connect.mirror.MirrorHeartbeatConnector",
    "type": "source",
    "version": "1"
  },
  {
    "class": "org.apache.kafka.connect.mirror.MirrorSourceConnector",
    "type": "source",
    "version": "1"
  }
]
```

我们运行的是最简单的 Kafka，所以只有文件数据源、文件数据池和 MirrorMaker 2.0 的连接器插件。

接下来看看如何配置和使用这些内置的示例连接器，然后我们会再提供一些需要连接外部数据存储系统的高级示例。

单机模式

需要注意的是，Connect 也支持单机模式。与分布式模式类似，单机模式只是在启动时用 bin/connect-standalone.sh 代替 bin/connect-distributed.sh。你也可以通过命令行传入连接器的配置文件，不需要使用 REST API。在单机模式下，所有的连接器和任务都运行在单个独

立的 worker 上。如果需要让连接器和任务运行在某台特定机器上（例如，Syslog 连接器会监听某个端口，你需要知道它运行在哪台机器上），就可以采用这种模式。

9.3.2 连接器示例：文件数据源和文件数据池

这个示例使用了文件连接器和 JSON 转换器，它们都是 Kafka 自带的。接下来要确保 ZooKeeper 和 Kafka 都处于运行状态。

首先启动一个分布式 worker。为了实现高可用性，真实的生产环境一般需要至少 2~3 个 worker。不过这个示例中只启动了一个。

```
bin/connect-distributed.sh config/connect-distributed.properties &
```

现在启动一个文件数据源。为方便起见，直接让它读取 Kafka 的配置文件，也就是说把 Kafka 配置文件中的内容发送到一个主题上。

```
echo '{"name":"load-kafka-config", "config":{"connector.class":
"FileStreamSource","file":"config/server.properties","topic":
"kafka-config-topic"}}' | curl -X POST -d @- http://localhost:8083/connectors
-H "Content-Type: application/json"

{
  "name": "load-kafka-config",
  "config": {
    "connector.class": "FileStreamSource",
    "file": "config/server.properties",
    "topic": "kafka-config-topic",
    "name": "load-kafka-config"
  },
  "tasks": [
    {
      "connector": "load-kafka-config",
      "task": 0
    }
  ],
  "type": "source"
}
```

我们写了一个 JSON 片段，里面包含了连接器的名字 `load-kafka-config` 和连接器的配置信息，包括连接器的类名、要加载的文件名和主题的名字。

接下来通过 Kafka 的控制台消费者来验证配置文件的内容是否已经被加载到主题上。

```
gwen$ bin/kafka-console-consumer.sh --bootstrap-server=localhost:9092
--topic kafka-config-topic --from-beginning
```

如果一切正常，那么可以看到如下输出。

```
{"schema":{"type":"string","optional":false},"payload":"# Licensed to the
Apache Software Foundation (ASF) under one or more"}

<省略的部分>
{"schema":{"type":"string","optional":false},"payload":"############################
Server Basics
############################"}
{"schema":{"type":"string","optional":false},"payload":""}
{"schema":{"type":"string","optional":false},"payload":"# The id of the broker.
This must be set to a unique integer for each broker."}
{"schema":{"type":"string","optional":false},"payload":"broker.id=0"}
{"schema":{"type":"string","optional":false},"payload":""}

<省略的部分>
```

以上输出的是 config/server.properties 文件中的内容，这些内容被一行一行地转成 JSON，并被连接器发送到了 `kafka-config-topic` 主题上。在默认情况下，JSON 转换器会在每条记录里附带上模式。这里的

模式非常简单，只有一个字符串类型的 payload 列。每条记录包含了文件中的一行内容。

现在，使用文件数据池转换器把主题中的内容导到文件中。导出的文件内容应该与原始 server.properties 文件的内容完全一样，JSON 转换器会把每条 JSON 记录转成一行文本。

```
echo '{"name":"dump-kafka-config", "config":
{"connector.class":"FileStreamSink","file":"copy-of-serverproperties","
topics":"kafka-config-topic"}}' | curl -X POST -d @- http://localhost:
8083/connectors --header "content-Type:application/json"

{"name":"dump-kafka-config","config":
{"connector.class":"FileStreamSink","file":"copy-of-serverproperties","
topics":"kafka-config-topic","name":"dump-kafka-config"},"tasks":
[]}
```

这次的配置发生了变化，我们使用的连接器类名是 FileStreamSink，而不是 FileStreamSource。文件指向了目标文件，而不是原先的 Kafka 配置文件。我们还指定了 topics，而不是 topic。需要注意的是，可以使用数据池将多个主题的内容写入一个文件，但一个数据源的内容只允许被写入一个主题。

如果一切正常，那么你会得到一个叫作 copy-of-server-properties 的文件，文件内容与 config/server.properties 完全一样。

如果要删除一个连接器，那么可以运行下面的命令。

```
curl -X DELETE http://localhost:8083/connectors/dump-kafka-config
```

这个例子使用了 FileStream 连接器，它们很简单，并且是 Kafka 自带的，所以只需安装了 Kafka 就可以用它创建你的第一个数据管道。但不要把它们用在生产环境中，因为它们有很多限制，也不具备可靠性保证。如果要从文件摄取数据，那么有几种替代方案可供选择：FilePulse Connector、FileSystem Connector 或 SpoolDir。

9.3.3 连接器示例：从 MySQL 到 ElasticSearch

下面来做一些更有用的事情。这次，将一张 MySQL 表的数据导入 Kafka 主题，然后再将它们加载到 ElasticSearch 中。

在 MacBook 中运行测试，并使用下面的命令安装 MySQL 和 ElasticSearch。

```
brew install mysql
brew install elasticsearch
```

下一步要确保有可用的连接器。可以通过以下几种方式获取。

01. 使用 Confluent Hub 客户端下载和安装。
02. 从 Confluent Hub 页面下载。（或者从其他提供了你需要的连接器的网站下载。）

03. 从源代码构建。要从源代码构建，需要做以下事情。

    a. 克隆连接器源代码。

    ```
    git clone https://github.com/confluentinc/kafka-connect-elasticsearch
    ```

    b. 运行 Maven 命令 mvn install -DskipTests 构建项目。

    c. 使用同样的步骤构建 JDBC 连接器。

接下来需要加载这些连接器。创建一个目录（如 /opt/connectors），并在 config/connect-distributed.properties 中加入 plugin.path=/opt/connectors。

在构建连接器项目时会在 target 目录下生成 JAR 包和依赖项，分别把它们复制到 plugin.path 指定的子目录中。

```
gwen$ mkdir /opt/connectors/jdbc
gwen$ mkdir /opt/connectors/elastic
gwen$ cp .../kafka-connect-jdbc/target/kafka-connect-jdbc-10.3.x-
SNAPSHOT.jar /opt/connectors/jdbc
gwen$ cp ../kafka-connect-elasticsearch/target/kafka-connectelasticsearch-
11.1.0-SNAPSHOT.jar /opt/connectors/elastic
gwen$ cp ../kafka-connect-elasticsearch/target/kafka-connectelasticsearch-
11.1.0-SNAPSHOT-package/share/java/kafka-connectelasticsearch/*
/opt/connectors/elastic
```

另外，因为要连接到 MySQL 数据库，所以需要下载并安装 MySQL 的 JDBC 驱动程序。受许可协议的限制，连接器本身不带有驱动程序。可以自行从 MySQL 官网下载，并放到 /opt/connectors/jdbc 目录下。

重启 worker 并检查新的连接器插件是否已经安装成功。

```
gwen$  bin/connect-distributed.sh config/connect-distributed.properties &

gwen$  curl http://localhost:8083/connector-plugins
[
  {
    "class": "io.confluent.connect.elasticsearch.ElasticSearchSinkConnector",
    "type": "sink",
    "version": "11.1.0-SNAPSHOT"
  },
  {
    "class": "io.confluent.connect.jdbc.JdbcSinkConnector",
    "type": "sink",
    "version": "10.3.x-SNAPSHOT"
  },
  {
    "class": "io.confluent.connect.jdbc.JdbcSourceConnector",
    "type": "source",
    "version": "10.3.x-SNAPSHOT"
  }
```

从上面的输出可以看到，新的连接器插件已经安装成功了。

下一步需要在 MySQL 中创建一张表，稍后我们将用 JDBC 连接器将表中的数据以流的方式发送给 Kafka。

```
gwen$ mysql.server restart
gwen$  mysql --user=root

mysql> create database test;
Query OK, 1 row affected (0.00 sec)

mysql> use test;
Database changed
mysql> create table login (username varchar(30), login_time datetime);
Query OK, 0 rows affected (0.02 sec)

mysql> insert into login values ('gwenshap', now());
Query OK, 1 row affected (0.01 sec)

mysql> insert into login values ('tpalino', now());
Query OK, 1 row affected (0.00 sec)
```

我们创建了一个数据库和一张表，并插入了一些测试数据。

接下来需要配置 JDBC 连接器。可以从文档中找到所有可用的配置参数，也可以通过 REST API 找到它们。

```
gwen$ curl -X PUT -d '{"connector.class":"JdbcSource"}' localhost:8083/
connector-plugins/JdbcSourceConnector/config/validate/ --header "content-
Type:application/json"
```

```
{
    "configs": [
        {
            "definition": {
                "default_value": "",
                "dependents": [],
                "display_name": "Timestamp Column Name",
                "documentation": "The name of the timestamp column to use
                to detect new or modified rows. This column may not be
                nullable.",
                "group": "Mode",
                "importance": "MEDIUM",
                "name": "timestamp.column.name",
                "order": 3,
                "required": false,
                "type": "STRING",
                "width": "MEDIUM"
            },
            <省略的部分>
```

向 REST API 发起验证连接器配置的请求，并传给它一个只包含连接器类名的配置（这是最简单的配置）。我们得到的 JSON 响应里包含了所有可用的配置项。

有了这些信息，就可以创建和配置 JDBC 连接器了。

```
echo '{"name":"mysql-login-connector", "config":{"connector.class":"JdbcSource-
Connector","connection.url":"jdbc:mysql://127.0.0.1:3306/test?
user=root","mode":"timestamp","table.whitelist":"login","validate.
non.null":false,"timestamp.column.name":"login_time","topic.prefix":"
mysql."}}' | curl -X POST -d @- http://localhost:8083/connectors --header
"content-Type:application/json"


{
  "name": "mysql-login-connector",
  "config": {
    "connector.class": "JdbcSourceConnector",
    "connection.url": "jdbc:mysql://127.0.0.1:3306/test?user=root",
    "mode": "timestamp",
    "table.whitelist": "login",
    "validate.non.null": "false",
    "timestamp.column.name": "login_time",
    "topic.prefix": "mysql.",
    "name": "mysql-login-connector"
  },
  "tasks": []
}
```

为了确保连接器运行正常，可以从 `mysql.login` 主题上读取数据。

```
gwen$ bin/kafka-console-consumer.sh --bootstrap-server=localhost:9092 --topic
mysql.login --from-beginning
```

如果返回错误，告知主题不存在，或者看不到任何数据，那么可以检查一下 Connect 的日志，看看有没有类似下面这样的错误信息。

```
[2016-10-16 19:39:40,482] ERROR Error while starting connector mysql-loginconnector
(org.apache.kafka.connect.runtime.WorkerConnector:108)
org.apache.kafka.connect.errors.ConnectException: java.sql.SQLException: Access
denied for user 'root;'@'localhost' (using password: NO)
        at io.confluent.connect.jdbc.JdbcSourceConnector.start(JdbcSourceConnector.
java:78)
```

如果还有其他问题，则可以检查一下类路径中是否包含了驱动程序，或者是否有权限读取数据库表。

连接器正常运行起来之后，如果向 `login` 表中插入数据，则它们会立即出现在 `mysql.login` 主题上。

变更数据捕获与 Debezium 项目

我们使用的 JDBC 连接器通过 JDBC 和 SQL 来扫描数据库表以查找新记录。它使用时间戳字段或递增的主键来检测新记录。这种方式相对低效，有时候也不准确。所有的关系数据库都将事务日志（也称为重做日志、binlog 或提前预写日志）作为其实现的一部分，它们中的大部分允许外部系统直接读取它们的事务日志。这种方式更准确、更高效，也就是所谓的变更数据捕获。大多数现代 ETL 系统会将变更数据捕获作为数据源。Debezium 项目为各种数据库提供了一系列高质量的开源变更捕获连接器。如果你正计划将关系数据库中的数据流转到 Kafka，那么强烈建议使用 Debezium 提供的变更捕获连接器（如果有你的数据库对应的连接器的话）。另外，Debezium 项目的文档是我们见过的最好的文档之一——除了连接器，它还涵盖与变更数据捕获相关的设计模式和应用场景，特别是它们在微服务架构当中的应用。

把数据从 MySQL 中移动到 Kafka 就算完成了。接下来我们要把数据从 Kafka 中写到 ElasticSearch，这个会更有意思。

首先启动 ElasticSearch，并通过访问本地端口来验证它是否已启动。

```
gwen$ elasticsearch &
gwen$ curl http://localhost:9200/
{
  "name" : "Chens-MBP",
  "cluster_name" : "elasticsearch_gwenshap",
  "cluster_uuid" : "X69zu3_sQNGb7zbMh7NDVw",
  "version" : {
    "number" : "7.5.2",
    "build_flavor" : "default",
    "build_type" : "tar",
    "build_hash" : "8bec50e1e0ad29dad5653712cf3bb580cd1afcdf",
    "build_date" : "2020-01-15T12:11:52.313576Z",
    "build_snapshot" : false,
    "lucene_version" : "8.3.0",
    "minimum_wire_compatibility_version" : "6.8.0",
    "minimum_index_compatibility_version" : "6.0.0-beta1"
  },
  "tagline" : "You Know, for Search"
}
```

现在创建并启动连接器。

```
echo '{"name":"elastic-login-connector", "config":{"connector.class":"ElasticsearchSinkConnector","
connection.url":"http://localhost:
9200","type.name":"mysql-data","topics":"mysql.login","key.ignore":true}}' |
curl -X POST -d @- http://localhost:8083/connectors --header "content-
Type:application/json"
{
  "name": "elastic-login-connector",
  "config": {
    "connector.class": "ElasticSearchSinkConnector",
    "connection.url": "http://localhost:9200",
    "topics": "mysql.login",
    "key.ignore": "true",
    "name": "elastic-login-connector"
  },
  "tasks": [
    {
      "connector": "elastic-login-connector",
      "task": 0
    }
  ]
}
```

这里有一些配置项需要解释一下。`connection.url` 是本地 ElasticSearch 服务器的地址。在默认情况下，每个 Kafka 主题对应 ElasticSearch 中的一个索引，主题的名字与索引的名字相同。我们只将 `mysql.login` 主题的数据写入 ElasticSearch。因为之前的 JDBC 连接器没有为消息指定键，所以 Kafka

消息的键都是空的。我们要让 ElasticSearch 连接器将主题名字、分区 ID 和偏移量作为消息的键。为此，需要把 `key.ignore` 设置为 `true`。

下面验证一下 `mysql.login` 主题对应的索引是否创建好了。

```
gwen$ curl 'localhost:9200/_cat/indices?v'
health status index       uuid                   pri rep docs.count
docs.deleted store.size pri.store.size
yellow open   mysql.login wkeyk9-bQea6NJmAFjv4hw   1   1          2
0      3.9kb          3.9kb
```

如果索引不存在，则可以检查一下 Connect 的日志。如果发现有错误，那么一般是因为缺少配置项或依赖包。如果一切正常，就可以从索引中搜索到我们的记录。

```
gwen$ curl -s -X "GET" "http://localhost:9200/mysql.login/_search?pretty=true"
{
  "took" : 40,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 2,
      "relation" : "eq"
    },
    "max_score" : 1.0,
    "hits" : [
      {
        "_index" : "mysql.login",
        "_type" : "_doc",
        "_id" : "mysql.login+0+0",
        "_score" : 1.0,
        "_source" : {
          "username" : "gwenshap",
          "login_time" : 1621699811000
        }
      },
      {
        "_index" : "mysql.login",
        "_type" : "_doc",
        "_id" : "mysql.login+0+1",
        "_score" : 1.0,
        "_source" : {
          "username" : "tpalino",
          "login_time" : 1621699816000
        }
      }
    ]
  }
}
```

如果在 MySQL 表中插入新数据，那么它们会自动出现在 Kafka 的 `mysql.login` 主题以及 ElasticSearch 的索引中。

现在，我们已经知道如何构建和安装 JDBC 连接器和 ElasticSearch 连接器了。我们可以根据具体的需要构建和安装任意的连接器。Confluent 提供了一些它们预先构建的连接器以及社区和其他一些供应商提供的连接器，可以在 Confluent Hub 上找到。你可以从中挑选你想要的连接器，把它们下载下来，并根据文档或通过 REST API 获取配置项，配置好以后就可以在自己的 Connect 集群中运行它们了。

构建自己的连接器

任何人都可以基于公开的 Connector API 创建自己的连接器。如果你要集成的数据存储系统没有相应的连接器，就自己开发一个。你也可以把自己开发的连接器放到 Confluent Hub 上，让更多的人知道并使用。有关构建连接器的更多细节已经超出了本章的讨论范围，不过有很多博文介绍了如何构建连接器，比如“4 Steps to Creating Apache Kafka Connectors with the Kafka Connect API”。纽约 2019 年 Kafka 峰会、伦敦 2018 年 Kafka 峰会以及 ApacheCon 的一些演讲也是很好的参考资料。建议将已有的连接器作为入门参考，或者从使用 Maven archtype 开始。另外，也可以通过 Kafka 社区邮件组（users@kafka.apache.org）寻求帮助或在邮件组中展示自己的连接器，或者把连接器提交到 Confluent Hub，这样它们就可以更容易被其他人找到了。

9.3.4 单一消息转换

将数据从 MySQL 复制到 Kafka，再从 Kafka 复制到 ElasticSearch，这个过程很有用，不过 ETL 管道通常还需要一个转换步骤。在 Kafka 生态系统中，我们将转换分为单一消息转换（single message transformation，SMT）和流式处理。前者是无状态的，后者是有状态的。SMT 可以在 Connect 中完成，也就是在复制消息时对消息进行转换，通常不需要编写额外的代码。对于更复杂的转换，比如涉及连接或聚合操作，则需要使用有状态的 Kafka Streams 框架。后面的章节将介绍 Kafka Streams。

Kafka 支持以下这些 SMT。

Cast

改变一个字段的数据类型。

MaskField

将一个字段的内容替换成 `null`。这在移除敏感信息或个人识别数据时非常有用。

Filter

丢弃或包含符合指定条件的记录。内置的条件包括匹配主题名称、匹配特定的标头、消息是否为墓碑消息（值为 `null`）。

Flatten

将嵌套的数据结构扁平化，也就是将所有字段的名字连接成路径的形式。

HeaderFrom

将消息里的字段移动或复制到标头里。

InsertHeader

在每一条消息的标头里加入一个固定的字符串。

InsertField

在消息里添加一个字段，字段的值既可以来自元数据（如偏移量），也可以是一个固定的值。

RegexRouter

使用正则表达式和替换字符串改变目标主题。

ReplaceField

移除或重命名消息里的字段。

TimestampConverter

修改一个字段的时间格式，比如将 Unix Epoch 转成字符串。

TimestampRouter

根据消息的时间戳来改变主题。这在数据池连接器中很有用，当我们希望基于消息的时间戳将它们复制到特定的分区表中时，主题将被用于查找目标系统的相关数据集。

此外，Kafka 项目之外的贡献者也提供了一些 SMT，可以在 Lenses.io、Aiven 和 Jeremy Custenborder 的 GitHub 仓库或 Confluent Hub 上找到它们。

要了解更多有关 Connect SMT 的信息，可以参考“Twelve Days of SMT”系列博文提供的详细示例。另外，也可以跟着教程开发自己的转换器并深入研究。

假设我们想要为 MySQL 连接器生成的每一条记录添加一个标头。这个标头将用于说明这条记录是由这个 MySQL 连接器创建的，这在审计这些记录的沿袭信息时非常有用。

为此，可以用下面的内容替换之前的 MySQL 连接器配置。

```
echo '{
  "name": "mysql-login-connector",
  "config": {
    "connector.class": "JdbcSourceConnector",
    "connection.url": "jdbc:mysql://127.0.0.1:3306/test?user=root",
    "mode": "timestamp",
    "table.whitelist": "login",
    "validate.non.null": "false",
    "timestamp.column.name": "login_time",
    "topic.prefix": "mysql.",
    "name": "mysql-login-connector",
    "transforms": "InsertHeader",
    "transforms.InsertHeader.type":
      "org.apache.kafka.connect.transforms.InsertHeader",
    "transforms.InsertHeader.header": "MessageSource",
    "transforms.InsertHeader.value.literal": "mysql-login-connector"
  }}' | curl -X POST -d @- http://localhost:8083/connectors --header "content-
Type:application/json"
```

现在，如果向之前创建的 MySQL 表中插入一些新数据，则会看到 `mysql.login` 主题中的新消息有了标头。（需要注意的是，只有 Kafka 2.7 或更高版本才能让控制台消费者打印标头。）

```
bin/kafka-console-consumer.sh --bootstrap-server=localhost:9092 --topic
mysql.login --from-beginning --property print.headers=true

NO_HEADERS      {"schema":{"type":"struct","fields":
[{"type":"string","optional":true,"field":"username"},
{"type":"int64","optional":true,"name":"org.apache.kafka.connect.data.Timestamp","
version":1,"field":"login_time"}],"optional":false,"name":"login"},"payload":{"
username":"tpalino","login_time":1621699816000}}
MessageSource:mysql-login-connector     {"schema":{"type":"struct","fields":

[{"type":"string","optional":true,"field":"username"},
{"type":"int64","optional":true,"name":"org.apache.kafka.connect.data.Timestamp","
version":1,"field":"login_time"}],"optional":false,"name":"login"},"payload":{"
username":"rajini","login_time":1621803287000}}
```

如你所见，旧记录显示 `NO_HEADERS`，但新记录显示 `MessageSource:mysql-login-connector`。

错误处理和死信队列

有关 SMT 的配置并非特定于某一个连接器，它可以被用在任意一个连接器中。另一个可以在任意数据池连接器中使用的配置是 `error.tolerance`，你可以让连接器静默丢弃损坏的消息，也可以将它们路由到一个叫作“死信队列”的主题上。以下博文中介绍了更多的细节：“Kafka Connect Deep Dive–Error Handling and Dead Letter Queues”。

9.3.5 深入理解 Connect

要理解 Connect 的工作原理，需要先知道 3 个基本概念以及它们之间的关系。之前的示例中演示过如何运行 worker 集群以及如何创建和移除**连接器**，不过并没有深入介绍**转换器**是如何处理数据的。转换器会把 MySQL 数据行转成 JSON 记录，然后由连接器将它们写入 Kafka。

接下来将深入介绍每一个组件以及它们之间的关系。

01. 连接器和任务

连接器插件实现了 Connector API，其中包含两部分内容。

连接器

连接器主要负责处理以下 3 件事情。

- 确定需要运行多少个任务。
- 确定如何根据任务拆分数据复制工作。
- 从 worker 获取任务配置信息并传给任务。

例如，JDBC 连接器会连接到数据库，统计需要复制的数据表，并确定需要多少个任务，然后在配置参数 `tasks.max` 和实际的数据表数量之间选择数值较小的那个作为任务数。在确定了任务数之后，连接器会为每一个任务生成一个配置，配置信息里包含了连接器的配置项（如 `connection.url`）和这个任务需要复制的数据表。`taskConfigs()` 方法会返回一个 map（包含每一个任务的配置信息）列表。worker 负责启动任务并将配置信息传给它们。每一个任务负责复制配置信息里指定的数据表。需要注意的是，当你通过 REST API 启动连接器时，它可能会在任意节点上启动，随后它启动的任务也会在这个节点上执行。

任务

任务负责将数据移入或移出 Kafka。在初始化任务时，worker 会为其分配上下文信息。源系统的上下文中包含了一个对象，任务可以将源系统记录的偏移量保存在这个对象里（例如，文件连接器的偏移量就是文件中的一个位置，JDBC 连接器的偏移量可以是表中的一个时间戳）。目标系统的上下文提供了一些方法，连接器可以用它们控制从 Kafka 接收到的数据，比如应用回压策略、进行错误重试，或者通过将偏移量保存到外部系统来实现精确一次传递。任务在初始化之后会按照连接器指定的配置（包含在一个 `Properties` 对象里）开始工作。在任务开始之后，源系统任务将对外部系统进行轮询并返回一些记录，然后由 worker 发送给 Kafka，数据池任务则通过 worker 接收来自 Kafka 的记录，然后将它们写入外部系统。

02. **worker**

worker 是执行连接器和任务的“容器”。它们负责处理用于定义连接器及其配置的 HTTP 请求、将连接器的配置保存到内部 Kafka 主题上、启动连接器和任务，并把配置信息传给任务。如果一个 worker 停止工作或发生崩溃，那么集群中的其他 worker 就会感知到（根据 Kafka 消费者协议的心跳机制来判断），并会将发生崩溃的 worker 的连接器和任务重新分配给其他 worker。如果有新的 worker 加入集群，那么其他 worker 也会感知到，并会将自己的连接器和任务分配给新 worker，确保工作负载均衡地分布到各个 worker 上。worker 还负责自动将偏移量提交到内部的 Kafka 主题上，当任务抛出异常时会进行重试。

可以这样理解：连接器和任务负责“移动数据”，worker 则负责处理 REST API 请求、配置管理、可靠性、高可用性、伸缩性和负载均衡。

这种关注点分离是 Connect API 给我们带来的最大好处，而这种好处是普通客户端 API 所不具备的。有经验的开发人员都知道，编写代码从 Kafka 读取数据并将其插入数据库只需要一两天，但要处理好配置、异常、REST API、监控、部署、伸缩性、故障处理等问题，可能需要几个月。而且，大部分的数据集成管道不止一个源系统或目标系统。再试想一下，为数据库集成付出的努力需要在其他技术

上重复很多次，这将是一场噩梦。如果使用连接器来实现数据复制，那么连接器插件会为你处理掉一大堆复杂的问题。

03. 转换器和 Connect 的数据模型

数据模型和转换器是 Connect API 需要讨论的最后一部分内容。Connect 提供了一组数据 API，包括数据对象和用于描述数据的模式。例如，JDBC 连接器会从数据库读取一个字段，并基于这个字段的数据类型创建一个 `Connect Schema` 对象。然后，连接器会基于模式创建一个包含了所有数据库字段的 `Struct`。我们保存了每一个字段的名字和它们的值。其他源连接器所做的事情都很相似：从源系统读取记录，为每条记录生成 `Schema` 和 `Value`。目标连接器正好相反，它们用 `Schema` 解析 `Value`，并把值写入目标系统。

源连接器负责基于数据 API 生成数据对象，那么 worker 是如何将这些数据对象保存到 Kafka 中的呢？这个时候，转换器就派上用场了。用户在配置 worker 或连接器时可以使用合适的转换器。目前可用的转换器有原始类型转换器、字节数组转换器、字符串转换器、Avro 转换器、JSON 转换器、JSON 模式转换器和 Protobuf 转换器。JSON 转换器既可以在转换结果中带上模式，也可以不带，这样就可以支持结构化和半结构化的数据。连接器会通过数据 API 将数据返回给 worker，然后 worker 会使用指定的转换器将数据转换成 Avro 对象、JSON 对象或字符串，转换后的数据会被写入 Kafka。

对目标连接器来说，过程刚好相反。在从 Kafka 读取数据时，worker 会使用指定的转换器将各种格式（也就是原始类型、字节数组、字符串、Avro、JSON、JSON 模式和 Protobuf）的数据转换成数据 API 格式的对象，然后将它们传给目标连接器，目标连接器再将它们插入目标系统中。

因此，Connect API 可以支持多种类型的数据，并独立于连接器的实现。（也就是说，只要有可用的转换器，连接器和数据类型可以自由组合。）

04. 偏移量管理

偏移量管理是 worker 为连接器提供的便捷服务之一（除了可以通过 REST API 进行部署和配置管理之外）。连接器需要知道哪些数据是已经处理过的，它们可以通过 Kafka 提供的 API 来维护已处理的消息的偏移量。

对源连接器来说，连接器返回给 worker 的记录里包含了一个逻辑分区和一个逻辑偏移量。它们并非 Kafka 的分区和偏移量，而是源系统的分区和偏移量。例如，对文件源来说，分区可以是一个文件，偏移量可以是文件中的一个行号或字符号。对 JDBC 源来说，分区可以是一个数据库表，偏移量可以是表中一条记录的 ID 或时间戳。在设计源连接器时，需要着重考虑如何对源系统的数据进行分区以及如何跟踪偏移量，这将影响连接器的并行能力，也决定了是否能够实现至少一次传递或精确一次传递。

源连接器将返回一些记录，记录里包含了分区和偏移量信息，worker 会将这些记录发送给 Kafka。如果 Kafka 确认记录保存成功，那么 worker 就会把偏移量保存下来。如果连接器发生崩溃，则在重启之后可以从最近保存的偏移量位置继续处理数据。偏移量的存储机制是可插拔的，通常会使用 Kafka 主题，我们可以通过 `offset.storage.topic` 参数来指定主题。另外，Connect 也使用 Kafka 主题来保存连接器的配置信息和每个连接器的运行状态，它们分别通过 `config.storage.topic` 和 `status.storage.topic` 来指定。

目标连接器的处理过程恰好相反，不过也很相似。它们会从 Kafka 上读取包含了主题、分区和偏移量信息的记录，然后调用连接器的 `put()` 方法，将记录保存到目标系统中。如果保存成功，就用消费者提交偏移量的方式将偏移量提交到 Kafka 上。

框架提供的偏移量跟踪机制简化了连接器的开发工作，并在使用不同的连接器时保证了一定程度的行为一致性。

9.4 Connect 之外的选择

现在，我们对 Connect API 有了更加深入的了解。虽然 Connect API 为我们提供了便利和可靠性，但它并非唯一的选择。下面列出了一些其他的可用框架，并说明了通常会在什么时候使用它们。

9.4.1 其他数据存储系统的数据摄入框架

虽然我们很想说 Kafka 是至高无上的“明星”，但肯定会有人不同意。有些人基于 Hadoop 或 ElasticSearch 构建他们的数据架构，这些系统都有自己的数据摄入工具，Hadoop 使用 Flume，ElasticSearch 使用 Logstash 或 Fluentd。如果架构里包含了 Kafka，并且需要连接大量的源系统和目标系统，那么建议使用 Connect API。如果你的系统是以 Hadoop 或 ElasticSearch 为中心，Kafka 只是数据的来源之一，那么使用 Flume 或 Logstash 会更合适。

9.4.2 基于图形界面的 ETL 工具

Informatica 等老古董、Talend 和 Pentaho 等开源解决方案，甚至 Apache NiFi 和 StreamSets 等较新的解决方案，都支持将 Kafka 作为数据源和数据池。如果你已经在使用这些系统（如 Pentaho），那么可能就不会为了 Kafka 而在系统中增加另一种数据集成工具。如果你已经习惯了基于图形界面的 ETL 数据管道解决方案，那么就继续使用它们。这些系统的缺点是它们的工作流比较复杂，如果你只是希望从 Kafka 获取数据或将数据写入 Kafka，那么它们就显得有点儿笨重。我们认为，在进行数据集成时，应该将注意力集中在消息的传递上，而大部分 ETL 工具太过复杂了。

建议把 Kafka 看成能够处理数据集成（通过 Connect）、应用集成（通过生产者和消费者）和流式处理的平台。Kafka 可以作为 ETL 工具的一个可行替代品。

9.4.3 流式处理框架

大部分流式处理框架具备从 Kafka 读取数据并将数据写入外部系统的能力。如果目标系统支持，并且你已经打算使用流式框架处理来自 Kafka 的数据，那么就可以使用同样的框架进行数据集成。这样就可以省掉一个处理步骤（无须把处理过的数据保存到 Kafka，只需读取数据，再把它们写入其他系统），只是在发生数据丢失或出现损坏数据时，诊断问题会变得更难一些。

9.5 小结

本章讨论了如何使用 Kafka 进行数据集成。首先解释了为什么要使用 Kafka 进行数据集成，并说明了数据集成方案的一般性考虑点。然后展示了为什么 Kafka 和 Connect API 是一种更好的选择，之后给出了一些例子，演示如何在不同的场景中使用 Connect，并深入介绍了 Connect 的工作原理。最后介绍了 Connect 之外的一些数据集成方案。

不管最终选择哪一种数据集成方案，都需要保证所有消息能够在各种恶劣条件下完成传递。我们相信，有 Kafka 的可靠性作为基础，Connect 也具备了极高的可靠性。不过，仍然需要对所选择的方案进行严格的测试，确保它们在发生进程停止、机器崩溃、网络延迟和高负载的情况下不丢失消息。毕竟，数据集成系统的核心任务是传递数据。

当然，可靠性是在集成数据系统时的一个重要考虑因素，但也只是其中的一个需求。在选择数据系统时，首先要明确需求（有关示例请参见 9.1 节），然后要确保所选择的系统能够满足这些需求。除此之外，还要深入了解自己的数据集成方案，知道怎么用它们来满足我们的需求。虽然 Kafka 支持至少一次传递，但也要小心谨慎，避免在配置上出错，影响了可靠性。

第 10 章　跨集群数据镜像

本书的大部分内容是在讨论如何配置、维护和使用单个 Kafka 集群。不过，在某些应用场景中，可能需要用到多个 Kafka 集群。

在一些应用场景中，集群相互独立，它们可能属于不同的部门或有不同的用途，不需要在集群间复制数据。有时候，因为 SLA 或工作负载的不同，很难通过调整单个集群来满足各个场景的需求，其他时候则要求具备不同等级的安全性。这些应用场景的需求其实很容易满足，就是分别为它们创建单独的集群，而管理多个集群其实就是把运行单个集群的过程重复多次。

在其他的应用场景中，不同的集群之间相互依赖，管理员需要不停地在集群间复制数据。大部分数据库支持**复制**（replication），也就是持续地在数据库服务器之间复制数据。因为“复制”一词已被用来描述在同一集群的不同节点之间移动数据，所以可以把集群间的数据复制叫作**镜像**（mirroring）。Kafka 内置的跨集群镜像工具叫作 MirrorMaker。

本章将讨论如何跨集群镜像所有或部分数据。我们首先会介绍一些常见的跨集群镜像应用场景，以及这些场景所使用的架构模式，并比较这些架构模式的优缺点。然后会介绍 MirrorMaker 以及如何使用它。接下来会分享在进行部署和性能调优时需要注意的一些事项。最后会介绍 MirrorMaker 的一些替代解决方案。

10.1 跨集群镜像的应用场景

下面列出了几种跨集群镜像的应用场景。

区域集群和中心集群

有时候，一家公司会有多个数据中心，分别分布在不同的地理区域、城市或大洲。这些数据中心都有自己的 Kafka 集群。有些应用程序只需要与本地的 Kafka 集群通信，有些应用程序则需要访问多个数据中心里的数据（否则就没必要考虑跨数据中心的复制方案了）。有很多情况会涉及跨数据中心访问数据，例如，一家公司根据供需情况修改商品价格就是一个典型的应用场景。这家公司在每个城市都有一个数据中心，用于收集每个城市的供需信息，然后根据整体的供需情况调整商品价格。这些信息会被镜像到中心集群，然后业务分析员会基于中心集群中的数据生成整个公司的收益报告。

高可用（HA）和灾备（DR）

一个 Kafka 集群已经可以满足所有应用程序的需求，但你担心集群会因某些原因变得不可用。为了实现冗余，你希望有第二个 Kafka 集群，该集群的数据与第一个集群完全相同，如果发生紧急情况，则可以将应用程序重定向到第二个集群。

监管与合规

为了符合不同国家的法律和监管要求，一家公司在不同国家的运营可能需要不同的配置和策略。例如，一些数据可能需要被保存在具有严格访问控制的独立集群中，一些数据则可以被复制到具有宽松访问权限的其他集群。为了满足不同区域对数据保留期限的合规性要求，保存在不同区域集群中的数据可以使用不同的配置。

云迁移

现如今，很多公司会将它们的业务同时部署在本地数据中心和云端。为了实现冗余，应用程序通常会运行在云供应商的多个服务区域，或使用多个云供应商提供的云服务。本地数据中心和每个云服务区域都至少有一个 Kafka 集群。本地数据中心和云服务区域中的应用程序通过这些 Kafka 集群在数据中心之间传输数据。例如，云端部署了一个应用程序，它需要访问本地数据中心里的数据。本地应用程序负责更新这些数据，并保存在本地数据库中。可以用 Connect 捕获这些数据库变更，先把它们保存在本地 Kafka 集群，然后再镜像到云端的 Kafka 集群，让云端的应用程序可以访问这些数据。这样既有助于控制跨数据中心的流量成本，也有助于提高流量的监管合规性和安全性。

聚合边缘集群的数据

一些行业，包括零售、电信、运输和医疗保健等，它们会收集小型设备生成的数据。这些设备使用的是受限的网络连接。大量的边缘集群会将收集到的数据聚合到一个高可用的聚合集群中。这些数据将被用于数据分析和其他用途。这种架构降低了对边缘集群（如物联网）在连通性、可用性和持久性方面的要求。即使边缘集群离线，高可用聚合集群仍然能够保证业务的连续性，并能够简化应用程序开发，无须直接与大量具有不稳定网络连接的边缘集群通信。

10.2 多集群架构

现在，我们已经知道有哪些场景需要用到多个 Kafka 集群，接下来将介绍几种常见的架构模式。我们已经成功使用这些模式实现了上述几种应用场景。在介绍这些架构模式之前，先简单了解一下跨数据中心通信的现实情况。下面将要讨论的解决方案都以特定的网络条件为前提。

10.2.1 跨数据中心通信的一些现实情况

以下是在进行跨数据中心通信时需要考虑的一些问题。

高延迟

Kafka 集群之间的通信延迟会随着集群间距离的增长和网络跳转次数的增加而增加。

有限的带宽

单个数据中心的广域网带宽远比我们想象的要低得多，而且可用的带宽时刻在发生变化。另外，高延迟让带宽的充分利用变得更加困难。

高成本

不管是在本地还是在云端运行 Kafka，集群之间的通信都需要更高的成本。部分原因是带宽有限，而增加带宽需要支付更高的费用。另外，这与供应商制定的在数据中心、可用区域和云平台之间传输数据的收费策略也有关系。

Kafka 服务器和客户端的设计、开发、测试以及调优都以单个数据中心为基础。假设服务器和客户端之间的延迟很低、带宽很高，在设定默认超时时间和缓冲区大小时也是基于这个前提。因此，不建议将 Kafka 集群的一部分服务器安装在一个数据中心里，另一部分安装在另一个数据中心里（不过稍后将介绍一些例外情况）。

在大多数情况下，要避免向远程的数据中心生成数据，但如果这么做了，就要忍受更高的延迟和更多的潜在网络错误。可以通过增加生产者的重试次数来应对网络错误以及增大缓冲区来降低延迟。

如果确实需要跨集群复制数据，并且不允许 broker 之间的通信以及生产者与 broker 之间的通信，那么就必须允许 broker 与消费者之间的通信。事实上，这是最安全的跨集群通信方式。在发生网络分区时，消费者无法从 Kafka 读取数据，但数据仍然驻留在 Kafka 中，当通信恢复正常时，消费者就可以继续读取数据。因此，网络分区不会造成数据丢失。不过，因为带宽有限，如果一个数据中心里的多个应用程序需要从另一个数据中心的 Kafka 服务器读取数据，那么我们倾向于在每一个数据中心里安装一个 Kafka 集群，并在这些集群间复制数据，而不是让应用程序直接通过广域网访问另一个数据中心里的数据。

在讨论跨数据中心通信的调优策略之前，需要先了解以下架构原则。

- 每个数据中心至少要有一个集群。
- 每两个数据中心之间的数据复制要做到每个事件仅复制一次（除非出现错误需要重试）。
- 如果有可能，那么尽量从远程数据中心读取数据，而不是向远程数据中心写入数据。

10.2.2 星型架构

这种架构适用于一个中心 Kafka 集群对应多个本地 Kafka 集群的情况，如图 10-1 所示。

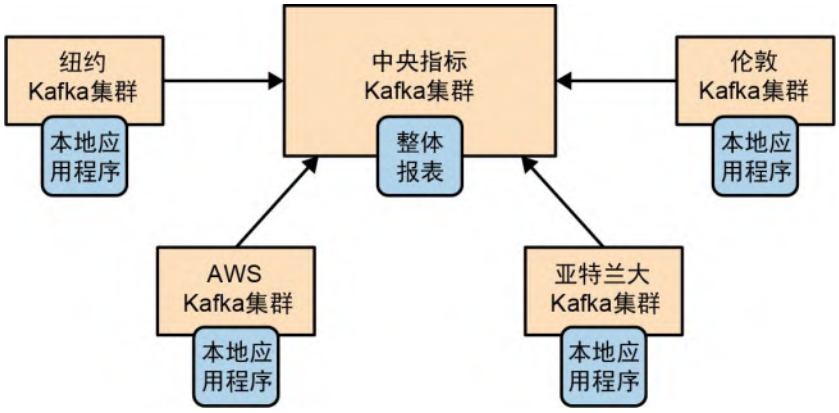

图 10-1：星型架构

这种架构的简化变种是只包含两个集群：一个首领和一个跟随者，如图 10-2 所示。

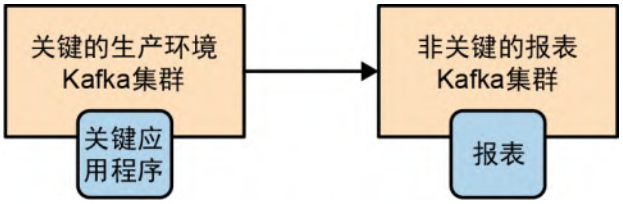

图 10-2：星型架构的简化变种

当数据分散在多个数据中心而消费者需要访问全部数据时，可以使用这种架构。如果每个数据中心的应用程序只访问自己所在数据中心的数据，那么也可以使用这种架构。但它不允许从每个数据中心访问整个数据集。

这种架构的好处在于，数据只会生成到本地的数据中心，并且每个数据中心的数据只会被镜像到中央数据中心一次。只读取单个数据中心数据的应用程序可以被部署在本地数据中心里，需要读取全部数据的应用程序则可以被部署在中央数据中心里。因为数据复制是单向的，并且消费者总是从同一个集群读取数据，所以这种架构易于部署、配置和监控。

不过这种简单的架构也有不足的地方，即一个数据中心里的应用程序无法访问另一个数据中心里的数据。为了更好地理解这种局限性，下面举一个例子来说明一下。

假设有一家银行，它在不同的城市有多家分行。每个城市的 Kafka 集群中保存了这个城市用户的账户资料和历史数据。我们把各个城市的数据复制到一个中心集群中，然后基于这些数据进行业务分析。当用户访问银行网站或去他们所属的分行办理业务时，相关的数据将被路由到本地集群中，并从本地集群读取数据。现在假设有一个用户要去另一个城市的分行办理业务，因为他的资料不在这个城市的 Kafka 集群中，所以这个分行要么从远程集群读取数据（不建议这么做），要么根本无法获取这个用户的资料（很尴尬）。因此，这种架构模式的局限在于区域数据中心之间的数据是完全独立的。

在采用这种架构时，每个区域数据中心至少要有一个镜像数据进程，用于将数据镜像到中央数据中心。镜像进程会读取每一个区域数据中心的数据，并将它们重新生成到中心集群。如果多个数据中心出现了重名的主题，则可以将这些数据写到中心集群的一个同名主题上，或者写到多个单独的主题上。

10.2.3 双活架构

当有两个或多个数据中心需要共享部分或全部数据，并且每个数据中心都可以生成和读取数据时，可以使用双活架构，如图 10-3 所示。

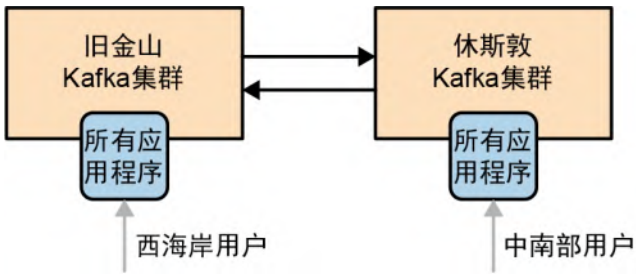

图 10-3：双活架构模型

这种架构的主要好处在于，它可以为就近用户提供速度更快的服务，具有性能方面的优势，而且不会因为数据可用性问题（在星型架构中就有这种问题）做出功能方面的牺牲。还有一个好处是冗余和弹性。因为每个数据中心都具备完整的功能，所以一旦一个数据中心发生故障，则可以把用户重定向到另一个数据中心。这种故障转移只涉及网络重定向，是一种最简单也最透明的故障转移方案。

这种架构的主要问题在于，如何在多个数据中心进行异步读写时避免冲突。比如，在镜像数据时如何确保同一条消息不会被无止境地来回镜像？更重要的是，保证跨数据中心的数据一致性会变得更加困难。以下是可能遇到的问题。

- 如果用户向一个数据中心发送数据，并从另一个数据中心读取数据，那么在用户读取数据之前，他发送的数据有可能还没有被镜像到那个数据中心。对用户来说，这就好比把一本书加到购物车里，但是当他点开购物车时，书不在里面。因此，在使用这种架构时，开发人员经常会将用户“黏”在同一个数据中心里，确保用户在大多数情况下使用的是同一个集群（除非他们从远程连接到另一个集群或那个数据中心不可用）。
- 一个用户在一个数据中心订购了图书 A，另一个数据中心几乎在同一时间收到了这个用户订购图书 B 的订单，在经过数据镜像之后，两个数据中心都有了这两个事件，也可以说两个数据中心都出现了两个冲突事件。应用程序需要知道如何处理这种情况。是否应该从中挑选一个作为“正确”的事件呢？如果是，那么就需要在两个数据中心之间定义好规则，让应用程序知道哪个事件才是正确的。或者，可以把两个事件都当成正确的事件，将两本书都发给用户，然后设立专门的部门来处理退货问题？亚马逊曾经用这种方式来处理冲突，但对券商来说，这种方案是行不通的。如何最小化和处理冲突要视具体情况而定。总而言之，如果使用了这种架构，则必然会遇到冲突，因此要想办法解决它们。

如果能够很好地处理多个数据中心在进行异步读写时发生冲突的问题，那么强烈建议使用这种架构。这种架构是我们所知道的最具伸缩性、弹性、灵活性和成本优势的解决方案。所以，它值得我们投入精力去寻找一些办法，用于避免循环复制、把相同用户的请求“黏”在同一个数据中心，以及在发生冲突时解决冲突。

双活架构（特别是当数据中心的数量超过两个）的挑战之处在于，每两个数据中心之间都需要进行镜像，而且是双向的。现在有很多镜像工具支持共享镜像进程，比如用同一个进程将所有数据镜像到目标集群。

另外，还要避免循环镜像，相同的事件不能无止境地在两个数据中心之间来回镜像。对于每一个“逻辑主题”，可以在每个数据中心里分别为它创建一个单独的主题，并确保不要从远程数据中心复制同名的主题。例如，对于逻辑主题 users，我们在一个数据中心里为其创建 SF.users 主题，在另一个数据中心里为其创建 NYC.users 主题。镜像进程会将 SF 的 SF.users 镜像到 NYC，将 NYC 的 NYC.users 镜像到 SF。这样一来，每一个事件只会被镜像一次。不过，在经过镜像之后，每个数据中心里都会有 SF.users 和 NYC.users 这两个主题，也就是说，每个数据中心将拥有所有的用户数据。消费者如果要读取所有用户的数据，就需要以 *.users 的方式订阅主题。也可以把这种方式理解为数据中心的命名空间，比如在这个例子中，NYC 和 SF 就是命名空间。一些镜像工具，比如 MirrorMaker，就是基于类似的命名空间机制来防止循环镜像。

在 Kafka 0.11.0 中引入的消息标头可以包含源数据中心的信息，我们既可以使用这些信息来避免循环镜像，也可以用它们来单独处理来自不同数据中心的数据。当然，还可以使用结构化的数据格式（如 Avro），并在消息里添加标签和标头，以此来实现同样的特性，只是在镜像数据时需要做一些额外的工作，因为现在的镜像工具并不支持自定义标头。

10.2.4 主备架构

有时候，我们使用多个集群只是为了达到灾备的目的。你可能在同一个数据中心安装了两个集群，平常只使用其中一个，第二个集群（几乎）包含了与第一个集群相同的数据，当第一个集群不可用时，就可以使用第二个集群。或者，你可能希望集群具备地理弹性。例如，你的整体业务运行在加利福尼亚州的数据中心里，并在得克萨斯州建立了第二个数据中心，这个数据中心平常不怎么用，但一旦第一个数据中心发生地震，它就可以派上用场。得克萨斯州的数据中心里有所有应用程序和数据的非活跃（“冷”）副本，一旦发生紧急情况，管理员就会启动它们，让第二个集群发挥作用（参见图 10-4）。这是一种合规性需求，业务不一定会将其纳入规划范畴，但还是有必要为其做好充分的准备。

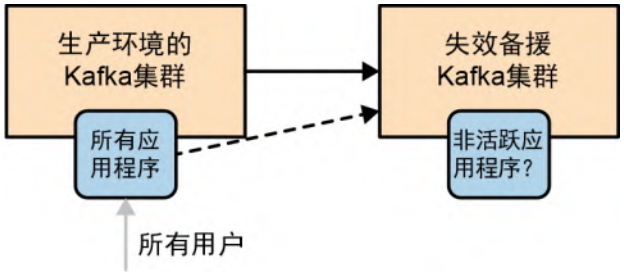

图 10-4：主备架构

这种架构的好处是易于实现，而且可以被应用于任何一种场景中。你只需要安装第二个集群，并使用镜像进程将第一个集群的数据完全镜像到第二个集群中，不需要操心如何访问数据、如何处理冲突，也不需要担心它会带来额外的架构复杂性。

这种架构的不足在于，它浪费了一个集群。Kafka 集群的故障转移比我们想象的要难得多。目前，要实现完全不丢失数据或无重复数据的 Kafka 集群故障转移是不可能的，更为常见的情况是既丢失数据，又出现重复数据。我们只能尽量减少这些问题的发生，但无法完全避免。

让一个集群什么都不做，只等待灾难的发生，明显是一种资源浪费。因为灾难是很少见的，所以在大部分时间里，灾备集群什么事也不做。为了减少浪费，有些组织会尝试减小灾备集群的规模，让它远小于生产环境集群的规模。这种做法存在一定的风险，因为你无法保证这种小规模集群能够在紧急情况下发挥应有的作用。有些组织则倾向于让灾备集群在平常也发挥作用，它们把一些只读工作负载转移到灾备集群中，也就是说，它们实际上运行的是星型架构的一个简化版本（只有一个 Spoke）。

那么问题来了：如何实现 Kafka 集群的故障转移呢？

首先，不管选择哪一种故障转移方案，SRE 团队都必须定期进行验证。今天能够正常运行的计划，在系统升级之后可能就无法正常运行了，或者现有的工具将无法满足新场景的需求。每季度进行一次故障转移验证是最低限度的要求，一个高效的 SRE 团队会更频繁地进行故障转移验证。Chaos Monkey 是 Netflix 开发的一款著名的故障注入工具，它会随机地制造灾难，有可能让任何一天都成为故障转移日。

下面来看看故障转移都包括哪些内容。

01. 灾难恢复计划

在进行灾难恢复计划时，必须考虑两个关键指标。一个是恢复时间目标（recovery time objective，RTO），它表示灾难发生后恢复所有服务需要的最长时间。另一个是恢复点目标（recovery point objective，RPO），它表示灾难发生后可能发生数据丢失的最长时间。RTO 越低，就越要避免采用人工操作流程，因为只有自动化故障转移才能实现非常低的 RTO。低 RPO 需要低延迟的实时镜像，如果要求 RPO 为零，则需要进行同步复制。

02. 没有故障转移计划导致的数据丢失和不一致性

因为 Kafka 的各种镜像解决方案都是异步的（后面将介绍一种同步的解决方案），所以灾备集群将无法及时获取主集群的最新数据。我们需要时刻注意灾备集群落后了主集群多少，并确保不要让它落后太多。但是，在一个繁忙的系统中，灾备集群与主集群之间可能有几百条甚至几千条消息的延迟。如果你的 Kafka 集群每秒处理 100 万条消息，主集群和灾备集群之间有 5 毫秒的延迟，那么在最好的情况下，灾备集群每秒会落后 5000 条消息。所以，如果没有故障转移计划，那么就要承担潜在的数据丢失风险。如果有故障转移计划，则可以先停止主集群，等待镜像进程将剩余数据镜像完毕，然后再切换到灾备集群，这样可以避免数据丢失。如果发生了非计划内的故障转移，则可能会丢失数千条消息。需要注意的是，目前镜像解决方案还不支持事务，也就是说，如果多个主题（比如保存销售数据的主题和保存产品数据的主题）之间有相关性，那么在故障转移过程中，有些数据可能可以及时到达灾备集群，有些则不能。在切换到灾备集群之后，应用程序需要知道如何处理没有相关销售信息的产品数据。

03. 故障转移之后的起始偏移量

在切换到灾备集群的过程中，最具挑战性的事情莫过于让应用程序知道该从哪里开始处理数据。下面将介绍一些常用的方法，其中一些很简单，但可能会造成数据丢失或数据重复，有些则比较复杂，但可以最小化丢失数据和出现重复数据的可能性。

偏移量自动重置

Kafka 消费者有一个配置参数，用于指定在没有前一个提交偏移量的情况下该作何处理。消费者要么从分区的起始位置开始读取数据，要么从分区的末尾开始读取。如果你的灾备计划里没有镜像偏移量，那么就需要从上述两个选项中选择一个。要么从头开始读取数据，并处理大量的重复数据，要么直接跳到末尾，放弃一些数据（希望不会太多）。如果重复处理数据或者丢失一些数据不是大问题，那么这种方案就是目前为止最为简单的。一般来说，在故障转移之后直接从主题末尾开始读取数据的方式更为常见。

复制偏移量主题

如果你使用的是 0.9.0 及以上版本的 Kafka 消费者，那么它会把偏移量提交到一个叫作 `__consumer_offsets` 的主题上。如果你将这个主题镜像到了灾备集群，那么当消费者开始读取灾备集群时，就可以获取到原先的偏移量，并从这个位置开始处理数据。这看起来很简单，但仍然有很多需要注意的地方。

第一，我们并不能保证主集群中的偏移量与灾备集群中的偏移量是完全匹配的。假设主集群中的数据只保留 3 天，但你在主题创建好一星期之后才开始镜像数据。主集群中第一个可用偏移量可能是 57 000 000（前 4 天的旧数据已经被删除了），而灾备集群中的第一个偏移量是 0。当消费者尝试从 57 000 003 位置（因为这是它要读取的下一条消息）开始读取数据时，就会失败。

第二，就算在主题创建之后立即镜像数据，并且主集群和灾备集群的偏移量都从 0 开始，当生产者需要重试发送消息时仍然会造成偏移量偏离。本章将在末尾介绍一种可以在主集群和灾备集群之间保留偏移量的镜像解决方案。

第三，就算偏移量被完美地保留下来，因为主集群和灾备集群之间存在延迟以及目前的镜像方案不支持事务，消费者提交的偏移量仍有可能在记录之前或者之后到达。在发生故障转移之后，消费者可能会找不到与偏移量匹配的记录，或者灾备集群中的偏移量会比主集群中的偏移量小，如图 10-5 所示。

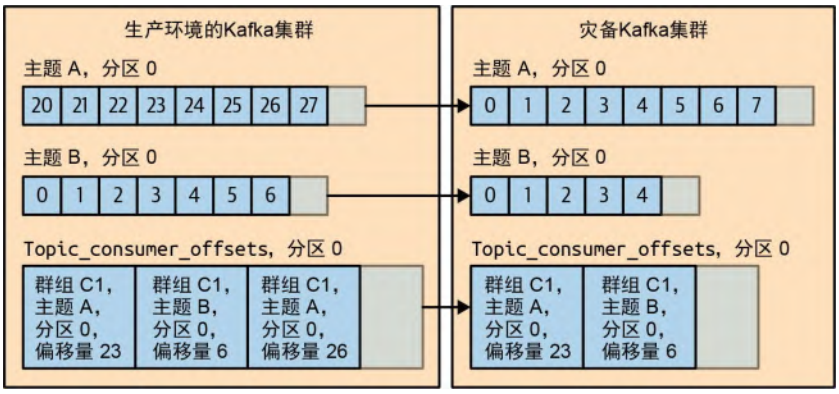

图 10-5：故障转移会导致偏移量找不到匹配的记录

在这些情况下，需要接受一定程度的数据重复。当灾备集群中的偏移量比主集群中的小，或者因生产者进行重试而导致灾备集群中的偏移量比主集群中的大时，都会造成数据重复。你还需要处理灾备集群中的偏移量找不到匹配记录的问题，即是从主题的起始位置开始读取还是从末尾开始读取？

如你所见，这种解决方案存在一些不足。但相比其他方案，它在保持简单性的前提下，降低了数据丢失或重复的可能性。

基于时间的故障转移

从 Kafka 0.10.0 开始，每条消息里都包含了一个时间戳——消息发送给 Kafka 的时间。从 Kafka 0.10.1.0 开始，broker 提供了一个索引和可根据时间戳查找偏移量的 API。如果发生了故障转移，而你知道故障是从凌晨 4:05 开始的，那么就可以让消费者从 4:03 的位置开始处理数据。虽然这两分钟时间差内会存在一些重复数据，但这种方式仍然比其他方案要好得多，而且也很容易向其他人解释：“我们将从凌晨 4:03 的位置开始处理数据”要比“我们从一个不知道是不是最新的位置开始处理数据”要好得多。所以，这是一种更好的折中。问题是，如何让消费者从凌晨 4:03 的位置开始处理数据呢？

一种方式是在应用程序中完成这件事情。我们为用户提供一个配置参数，用于指定从什么时间点开始处理数据。如果用户指定了时间，那么应用程序就可以通过新 API 获取指定时间戳对应的偏移量，然后从这个位置开始处理数据。

如果应用程序一开始就是这么设计的，那么使用这种方案就再好不过了。但如果不是呢？Kafka 提供了 `kafka-consumer-groups` 工具，我们可以用它基于一些参数来重置偏移量，包括在 Kafka 0.11.0 中引入的基于时间戳的偏移量重置。在使用这个工具时，需要先停止消费者群组，等工具运行完毕之后再立即启动。例如，下面的命令会将某个消费者群组订阅的所有主题的偏移量都重置到指定的时间。

```
bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --resetoffsets
--all-topics --group my-group --to-datetime
2021-03-31T04:03:00.000 --execute
```

这个方案适用于在发生故障转移时需要保证一定确定性的场景。

偏移量映射

镜像偏移量主题的一个最大问题在于主集群和灾备集群的偏移量会发生偏离。在过去，一些组织会选择使用外部数据存储（如 Apache Cassandra）来保存集群之间的偏移量映射关系。每当一个事件被发送到灾备集群时，镜像工具会将两个集群的偏移量都发送到外部存储。现在的镜像解决方案（包括 MirrorMaker）会使用 Kafka 主题来保存偏移量映射关系。当两个集群的偏移量差值发生变化时，就更新映射关系。如果主集群的偏移量 495 被映射到灾备集群的 500，就在外部存储上记录 (495,500)，或者把这个映射写到映射主题上。如果之后因为消息重复导致差值发生变化，偏移量 596 被映射为 600，就记录新的映射 (569,600)。没有必要保留 495 和 596 之间的所有偏移量映射，我们假

设差值都是一样的，所以主集群的偏移量 550 会被映射到灾备集群的 555。如果发生了故障转移，就使用主集群与灾备集群的偏移量映射，而不是时间戳（通常会有点儿不准确）与偏移量的映射。我们会通过之前介绍的两种技术手段之一强制消费者使用映射中的偏移量。对于那些比记录先达到的偏移量或没有及时被镜像到灾备集群的偏移量，仍然会有问题，不过这至少已经覆盖了部分场景。

04. 故障转移之后

假设故障转移进行得很顺利，灾备集群也运行得很正常，现在需要对主集群做一些改动，把它变成灾备集群。

如果能够简单地改变镜像进程的方向，让它将数据从新主集群镜像到旧主集群，那么就完美了。不过，这里有两个问题。

- 怎么知道该从哪里开始镜像呢？我们需要解决与镜像应用程序消费者所面临问题同样的问题。而且不要忘了，所有的解决方案都有可能导致数据丢失或重复，或两者兼有。
- 另外，正如之前所讨论的那样，旧主集群中可能有一些数据没有被镜像到新主集群中，如果这个时候开始镜像新主集群的数据，那么历史遗留数据会继续留在旧主集群中，导致两个集群的数据不一致。

因此，如果要保证数据的一致性和顺序，最简单的解决方案是清理旧主集群，删掉所有数据和偏移量，然后从新主集群中把数据镜像回来。这样可以获得与新主集群一致的数据状态。

05. 关于集群发现

在设计灾备集群时，需要考虑的一个很重要的问题是在发生故障转移之后，应用程序如何与灾备集群通信。如果把主集群的主机名硬编码在生产者和消费者的配置文件中，就会很麻烦。为简单起见，大多数组织使用了 DNS，并将其指向主集群，一旦发生紧急情况，则可以将其指向灾备集群。这些服务发现工具（DNS 或其他）不需要包含所有 broker 的信息，只要 Kafka 客户端能够连接到其中一个 broker，就可以获取整个集群的元数据，并发现集群中的其他 broker。一般来说，提供 3 个 broker 的信息就可以了。不管使用哪种服务发现工具，在大多数情况下需要重启消费者，这样才能找到新的偏移量，并继续读取数据。对于为降低 RTO 而使用自动故障转移的场景，需要将故障转移处理逻辑放在客户端应用程序中。

10.2.5 延展集群

在主备架构中，当 Kafka 主集群发生失效时，可以将应用程序重定向到另一个集群，以保证业务的正常运行。而当整个数据中心发生故障时，延展集群（stretch cluster）可以保证 Kafka 集群仍然可用。延展集群是指跨多个数据中心的 Kafka 集群。

延展集群与其他类型的跨数据中心集群有本质上的区别。首先，延展集群并非多个集群，而是单个集群，因此不需要使用镜像进程。延展集群使用 Kafka 内置的复制机制来保持 broker 副本的同步。这里可以使用同步复制。生产者通常会在消息成功写入 Kafka 之后收到确认。在延展集群中，也可以配置类似的参数，在消息被写入两个数据中心的 broker 之后发送确认。这还包括使用机架信息确保每个分区在其他数据中心都有相应的副本，并配置 `min.insync.replicas` 和 `acks=all`，以确保每次写入消息时都可以收到来自至少两个数据中心的确认。从 Kafka 2.4.0 开始，消费者可以基于机架信息从最近的副本获取数据。broker 会检查自己的机架信息与消费者的机架信息，以便找到最新的本地副本，如果本地副本不可用，就从首领副本读取。从本地数据中心的跟随者副本读取数据可以提高吞吐量、降低延迟和成本，因为跨数据中心的流量减少了。

同步复制是这种架构的最大优势。有些类型的业务要求灾备站点与主站点保持 100% 的同步。这是一种合规性需求，适用于公司内的任何一种数据存储，包括 Kafka。这种架构还有一个优势，即两个数据中心及集群内的所有 broker 都发挥了作用，不会浪费资源。

这种架构的不足之处是它所能应对的灾难类型是有限的。它只能应对数据中心故障，但无法应对应用程序或者 Kafka 故障。运维复杂性是它的另一个不足之处，它所需要的物理基础设施并不是所有公司都能承担得起的。

如果能够在至少 3 个具有高带宽和低延迟的数据中心里安装 Kafka（包括 ZooKeeper），那么就可以使用这种架构。如果你的公司在同一个街区有 3 栋大楼，或者你的云供应商在同一个地区有 3 个可用区域，那么就可以考虑使用这种方案。

之所以是 3 个数据中心，主要是因为 ZooKeeper 要求集群的节点个数是奇数，并且只有当大多数节点可用时，整个集群才可用。如果只有两个数据中心，并且 ZooKeeper 的节点是奇数个，那么其中一个数据中心将包含大多数节点，这就意味着如果这个数据中心不可用，那么 ZooKeeper 和 Kafka 也不可用。如果有 3 个数据中心，那么在分配节点时，可以做到每个数据中心都不会包含大多数节点。如果其中一个数据中心不可用，那么其他两个数据中心里还有大多数节点，此时 ZooKeeper 和 Kafka 仍然可用。

数据中心架构

一种比较流行的延展集群模型是 2.5 数据中心架构，即两个数据中心里都安装了 Kafka 和 ZooKeeper，然后再在另外“半”个数据中心里安装一个 ZooKeeper 节点，当其中一个数据中心发生故障时，它可以提供仲裁。

从理论上说，分别在两个数据中心运行 ZooKeeper 和 Kafka 是可能的，只要将 ZooKeeper 集群配置成允许手动进行数据中心间的故障转移。但在实际当中，这种做法并不常见。

10.3 MirrorMaker

Kafka 提供了一个叫作 MirrorMaker 的工具，用于在两个数据中心之间镜像数据。早期版本的 MirrorMaker 使用一组消费者（属于同一个消费者群组）从一组源主题读取数据，并通过 MirrorMaker 进程共享的生产者将这些数据发送到目标集群。虽然它可以满足部分应用场景，但也存在一些问题，特别是当配置发生变化或添加新主题时，再均衡会导致延迟激增。MirrorMaker 2.0 是 Kafka 的下一代多集群镜像解决方案，它基于 Connect 框架，弥补了旧版本的很多不足。我们可以很容易地配置出复杂的拓扑来支持更为广泛的场景，比如灾难恢复、备份、迁移和数据聚合。

关于 MirrorMaker

MirrorMaker 看起来很简单，但因为我们想要做到既高效又非常接近精确一次传递，所以其实现起来是很困难的。MirrorMaker 已经被重写了多次。这里以及后面的内容主要针对的是在 Kafka 2.4.0 中引入的 MirrorMaker 2.0。

MirrorMaker 使用源连接器来读取另一个 Kafka 集群而不是数据库的数据。Connect 框架的使用减轻了企业 IT 部门管理 Kafka 的负担。第 9 章在介绍 Connect 架构时提到过，每个连接器会将工作负载分配给多个任务。在 MirrorMaker 中，每个任务对应一个消费者和一个生产者。Connect 框架会根据需要将这些任务分配给不同的 Connect 工作节点，所以一台服务器上可能有多个任务，或者任务可能会被分散到多台服务器上。这样就无须手动计算每个实例要运行多少个 MirrorMaker 以及每台机器要运行多少个实例。Connect 还提供了用于集中管理连接器和任务配置的 REST API。假设大多数 Kafka 集群因为某些原因而启用了 Connect（将数据库变更事件发送到 Kafka 就是一个常见的场景），那么，在 Connect 内部运行 MirrorMaker 就可以减少需要管理的集群的数量。

为避免在添加新主题或分区时发生再均衡而导致延迟激增，在为任务分配分区时，MirrorMaker 并没有使用 Kafka 的消费者群组管理协议。源集群每一个分区的事件都会被镜像到目标集群的相同分区，保留语义分区并维护每个分区的事件顺序。如果源主题添加了新分区，那么目标主题也会自动创建对应的分区。除了数据复制，MirrorMaker 还支持迁移消费者偏移量、主题配置和主题 ACL。这是一种完整的多集群镜像解决方案。从源集群到目标集群的定向流配置叫作复制流。MirrorMaker 可以有多个复制流，从而定义出复杂的拓扑，包括前面介绍过的各种架构模式，比如星型模式、主备模式和双活模式。图 10-6 是 MirrorMaker 在主备架构中的应用。

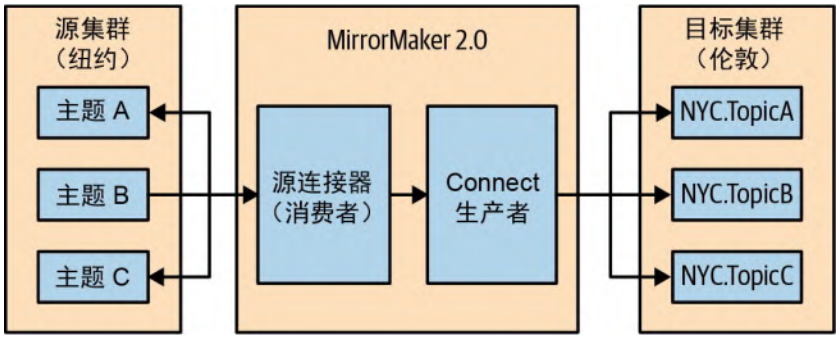

图 10-6：MirrorMaker 的镜像过程

10.3.1 配置 MirrorMaker

MirrorMaker 是高度可配置的。除了用于配置拓扑、Connect 和连接器的集群参数，MirrorMaker 所使用的生产者、消费者和 AdminClient 的每一个属性都是可配置的。接下来，我们将提供几个例子，并着重说明一些重要的配置参数。不过，MirrorMaker 的文档内容不在本书的讨论范围内。

先来看一个例子。下面的命令会使用属性文件中指定的配置参数来启动 MirrorMaker。

```
bin/connect-mirror-maker.sh etc/kafka/connect-mirror-maker.properties
```

下面将介绍 MirrorMaker 的一些配置参数。

复制流

下面的示例演示了在纽约和伦敦的两个数据中心之间建立主备复制流的配置参数。

```
clusters = NYC, LON                                    ❶
NYC.bootstrap.servers = kafka.nyc.example.com:9092     ❷
LON.bootstrap.servers = kafka.lon.example.com:9092
NYC->LON.enabled = true                                ❸
NYC->LON.topics = .*                                   ❹
```

❶ 定义在复制流中使用的集群的别名。

❷ 配置每个集群的地址，使用集群别名作为前缀。

❸ 使用 `source->target` 这样的前缀配置两个集群之间的复制流。与这个复制流相关的所有配置都使用相同的前缀。

❹ 配置复制流要镜像的主题。

要镜像的主题

对于每个复制流，可以使用正则表达式指定需要镜像的主题。在这个示例中，我们选择镜像每一个主题，但在实际当中，最好使用类似 prod.* 这样的表达式，以免镜像了测试主题。也可以指定要排除的主题或使用类似 test.* 这样的表达式来排除不需要镜像的主题。在默认情况下，目标主题会自动加上源集群别名作为前缀。例如，在双活架构中，负责将 NYC（纽约）数据中心的数据镜像到 LON（伦敦）数据中心的 MirrorMaker 会将 NYC 数据中心的 orders 主题镜像到 LON 数据中心的 NYC.orders 主题上。如果既要将主题从 NYC 镜像到 LON，又要将主题从 LON 镜像到 NYC，那么这种默认命名策略可以防止循环镜像（数据在两个集群之间无休止地来回镜像）。本地主题和远程主题之间的名称差异并不影响聚合，因为消费者既可以通过正则表达式订阅本地主题，也可以订阅远程主题，以便获得完整的数据集。

MirrorMaker 会定期检查源集群是否有新增主题，如果有，并且与指定的模式匹配，就会自动开始镜像这些主题。如果源主题新增了分区，则目标主题也会自动添加相同数量的分区，确保源主题中的事件在目标主题中以相同的顺序出现在相同的分区中。

消费者偏移量的迁移

MirrorMaker 提供了一个叫作 `RemoteClusterUtils` 的辅助类，在发生故障转移后，可以用它在灾备集群中找到最后的检查点偏移量。Kafka 2.7.0 支持定时迁移消费者偏移量，它会自动将转换后的偏移量提交到灾备集群的 `__consumer_offsets` 主题，消费者在切换到灾备集群后就可以从之前停止的位置重新开始处理数据，不会发生数据丢失，并会将重复处理消息的可能性降到最低。为安全起见，如果目标集群的消费者正在使用目标消费者群组，那么 MirrorMaker 就不会覆盖偏移量，从而能够避免出现意外冲突。

主题配置信息和 ACL 的迁移

除了数据，MirrorMaker 也可以镜像主题的配置信息和访问控制列表（ACL）信息，以便保留相同的主题行为。这个功能默认是启用的，并会以合理的时间间隔刷新。源主题的大部分配置参数会被应用于目标主题，但有一些参数在默认情况下不会被应用，比如 `min.insync.replicas`。需要排除哪些配置参数可以自行指定。

只有那些与被镜像的主题匹配的字面量 ACL 会被迁移，因此，如果你使用的是包含前缀或通配符的 ACL 或者其他身份验证机制，则需要在目标集群中显式配置。为了确保只有 MirrorMaker 能够向目标主题

写入数据，Topic:Write 权限不会被迁移。在发生故障转移时，必须显式地授予应用程序适当的访问权限，确保它们可以正常访问第二个集群。

连接器任务

参数 tasks.max 指定了与 MirrorMaker 关联的连接器可以使用的最大任务数。默认值为 1，但建议最少将其设置为 2。在复制大量的主题分区时，为了增加并行度，应该尽可能设置更大的值。

参数的前缀

MirrorMaker 支持其所有组件的配置参数定制化，包括连接器、生产者、消费者和管理客户端。Connect 和连接器的配置参数可以不带任何前缀。但是，由于 MirrorMaker 可以包含多个集群的配置，因此可以使用前缀来指定特定于集群或复制流的配置。之前的示例中是将集群别名作为集群相关配置参数的前缀。前缀可用在层级结构配置中，包含特定前缀的配置比相对不那么特定或没有前缀的配置具有更高的优先级。MirrorMaker 支持以下这些前缀。

- {cluster}.{connector_config}
- {cluster}.admin.{admin_config}
- {source_cluster}.consumer.{consumer_config}
- {target_cluster}.producer.{producer_config}
- {source_cluster}->{target_cluster}.{replication_flow_config}

10.3.2 多集群复制拓扑

前面介绍了一个简单的主备复制流的配置示例。现在，来看看如何扩展配置，以便支持其他常见的架构模式。

对于纽约和伦敦之间的双活拓扑，可以配置双向复制流。尽管 NYC 的所有主题都被镜像到了 LON，LON 的所有主题都被镜像到了 NYC，MirrorMaker 仍然可以确保相同的事件不会在两个集群之间无止境地来回镜像，因为远程主题使用集群别名作为前缀。建议让不同的 MirrorMaker 进程使用同一个包含完整复制拓扑的配置文件，这样可以避免在使用目标数据中心的内部配置主题共享配置信息时发生冲突。在启动目标数据中心的 MirrorMaker 进程时，可以使用 --clusters 选项指定目标集群。

```
clusters = NYC, LON
NYC.bootstrap.servers = kafka.nyc.example.com:9092
LON.bootstrap.servers = kafka.lon.example.com:9092
NYC->LON.enabled = true                          ❶
NYC->LON.topics = .*                             ❷
LON->NYC.enabled = true                          ❸
LON->NYC.topics = .*                             ❹
```

❶ 启用从纽约到伦敦的复制。

❷ 指定需要从纽约复制到伦敦的主题。

❸ 启用从伦敦到纽约的复制。

❹ 指定需要从伦敦复制到纽约的主题。

还可以向拓扑中添加更多包含其他源或目标集群的复制流。例如，可以添加一个 SF 复制流，这样就可以支持从 NYC 到 SF 和 LON 的扇出复制。

```
clusters = NYC, LON, SF
SF.bootstrap.servers = kafka.sf.example.com:9092
NYC->SF.enabled = true
NYC->SF.topics = .*
```

10.3.3 保护 MirrorMaker

对于生产环境中的集群，需要确保所有的跨数据中心流量都是安全的。第 11 章将介绍保护 Kafka 集群的方案。我们必须在源集群和目标集群中使用安全的 broker 监听器，必须配置集群的客户端安全选项，让 MirrorMaker 能够使用需要身份验证的安全连接。还需要使用 SSL 加密所有的跨数据中心通信。例如，可以像下面这样配置 MirrorMaker 的密钥凭证。

```
NYC.security.protocol=SASL_SSL              ❶
NYC.sasl.mechanism=PLAIN
NYC.sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule \
    required username="MirrorMaker" password="MirrorMaker-password";      ❷
```

❶ 安全协议应该与指定集群所对应的 broker 监听器的安全协议相匹配，建议使用 SSL 或 SASL_SSL。

❷ 因为使用了 SASL，所以这里需要指定 JAAS 密钥凭证。如果使用的是 SSL，并启用了双向身份验证，则还需要指定密钥存储文件。

如果在集群中启用了授权，则必须将源集群和目标集群的一些权限授予与 MirrorMaker 相关的主体。必须为 MirrorMaker 进程授予以下这些 ACL 权限。

- 源集群的 `Topic:Read`，这样就可以从源主题读取数据；目标集群的 `Topic:Create` 和 `Topic:Write`，这样就可以创建目标主题并写入数据。
- 源集群的 `Topic:DescribeConfigs`，这样就可以获取源主题的配置信息；目标集群的 `Topic:AlterConfigs`，这样就可以更新目标主题的配置。
- 目标集群的 `Topic:Alter`，如果源主题增加了新分区，就可以在目标集群中也添加相应的分区。
- 源集群的 `Group:Describe`，这样就可以获取包括偏移量在内的源消费者群组的元数据；目标集群的 `Group:Read`，这样就可以提交目标消费者群组的偏移量。
- 源集群的 `Cluster:Describe`，这样就可以获取源主题的 ACL；目标集群的 `Cluster:Alter`，这样就可以修改目标主题的 ACL。
- 源集群和目标集群的 `Topic:Create` 和 `Topic:Write`，这样就可以操作 MirrorMaker 的内部主题。

10.3.4 在生产环境中部署 MirrorMaker

在前面的示例中，我们是通过命令行让 MirrorMaker 在指定的模式下启动。也可以启动任意数量的 MirrorMaker 进程，让它们形成一个可伸缩、具备容错能力的 MirrorMaker 集群。镜像到同一集群的进程知道彼此的存在，并会自动进行负载均衡。通常，在生产环境中，你可能希望将 MirrorMaker 作为一个后台服务运行，并将控制台输出重定向到日志文件。MirrorMaker 提供了一个命令行选项 `-daemon`，我们可以用它来实现在后台运行 MirrorMaker。大多数使用 MirrorMaker 的公司有自己的启动脚本，脚本中包含了它们使用的配置参数。一些部署系统，比如 Ansible、Puppet、Chef 和 Salt，经常被用于自动化部署和管理配置参数。MirrorMaker 也可以运行在 Docker 容器中。MirrorMaker 是完全无状态的，不需要磁盘存储（所有的数据和状态都保存在 Kafka 中）。

因为 MirrorMaker 是基于 Connect 的，所以 Connect 所有的部署模式都可以应用于 MirrorMaker。在开发和测试时可以使用独立模式，MirrorMaker 将作为一个单独的 Connect worker 运行在一台机器上。MirrorMaker 也可以作为一个连接器运行在分布式 Connect 集群中。在生产环境中，建议在分布式模式下运行 MirrorMaker，既可以是专有的 MirrorMaker 集群，也可以是共享的分布式 Connect 集群。

如果有可能，那么尽量让 MirrorMaker 运行在目标数据中心里。也就是说，如果要将 NYC 的数据复制到 SF，那么 MirrorMaker 应该运行在 SF 的数据中心里，并消费 NYC 数据中心里的数据。这是因为长距离的外部网络比数据中心的内部网络更加不可靠，如果发生了网络分区，数据中心之间断开了连接，那么一个无法连接到集群的消费者要比无法连接到集群的生产者安全得多。如果消费者无法连接到集群，那么最多也就是无法读取数据，数据仍然可以在 Kafka 集群中保留很长一段时间，不会有丢失的风险。相反，在发生网络分区时，如果 MirrorMaker 已经读取了数据，但无法将数据生成到目标集群中，则会造成数据丢失。所以说，远程读取数据比远程生成数据更加安全。

那么，什么情况下必须在本地读取消息并将其生成到远程数据中心呢？答案是当你要求数据在传输过程中加密但在数据中心里不加密的时候。消费者在使用 SSL 连接时对性能有一定的影响，而且比生产者要严

重得多。这是因为消费者在使用 SSL 连接时需要复制数据并对其加密，无法享受零复制带来的性能好处。这种性能问题也会影响到 broker。如果跨数据中心流量需要加密，但本地流量不需要，那么最好把 MirrorMaker 放在源数据中心里，让它读取本地的非加密数据，然后通过 SSL 连接将数据生成到远程数据中心。这个时候，如果使用 SSL 连接的是生产者，而不是消费者，那么对性能的影响就不会太大。在采用这种方式时，需要确保 MirrorMaker 的 Connect 生产者配置了 `acks=all` 和适当的重试次数。另外，可以将 MirrorMaker 的 `errors.tolerance` 设置为 `none`，让它在出现错误时快速失效，这通常比继续发送事件更为安全。需要注意的是，新版本的 Java 显著提升了 SSL 性能，所以在本地生成数据和从远程读取数据可能也是一个可行的选择，即使是在使用了加密的情况下。

另一个可能需要在本地读取数据并将其生成到远程的场景是将数据从本地集群镜像到云端集群。出于安全考虑，本地集群可能位于防火墙后面，拒绝来自云端的连接，但在本地运行的 MirrorMaker 可以连接到云端。

在将 MirrorMaker 部署到生产环境时，一定要记得监控下面这些内容。

Connect 监控

Connect 提供了大量的指标，比如用于监控连接器状态的连接器指标、用于监控吞吐量的源连接器指标，以及用于监控再均衡延迟的 worker 指标。Connect 还提供了一个用于查看和管理连接器的 REST API。

MirrorMaker 指标监控

除了 Connect 提供的指标，MirrorMaker 也提供了用于监控镜像吞吐量和复制延迟的指标。复制延迟指标 `replication-latency-ms` 可以告诉我们消息的时间戳与消息被成功写入目标集群时的时间间隔。如果想知道目标集群是否及时跟上了源集群，那么这个指标就非常有用。在高峰时段，如果目标集群有足够的能力赶上源集群，那么即使延迟增加了也没问题，但如果延迟持续增加，则说明目标集群能力不足。其他指标，比如 `record-age-ms`（在进行镜像时消息的年龄）、`byte-rate`（镜像吞吐量）和 `checkpoint-latency-ms`（偏移量迁移延迟），也非常有用。在默认情况下，MirrorMaker 还会定时发送心跳，我们可以用它来监控 MirrorMaker 的运行状况。

延迟监控

你肯定想知道目标集群是否落后于源集群。延迟体现在源集群最新偏移量与目标集群最新偏移量的差异上，如图 10-7 所示。

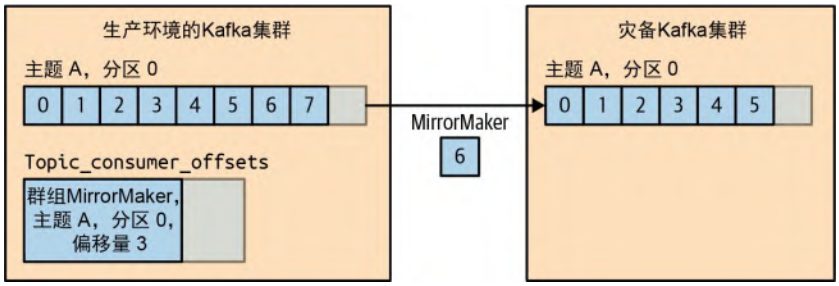

图 10-7：监控偏移量延迟

在图 10-7 中，源集群的最后一个偏移量是 7，目标集群的最后一个偏移量是 5，它们之间有两条消息的延迟。

以下是两种跟踪延迟的方式，但它们都不完美。

- 检查 MirrorMaker 提交到源集群的最新偏移量。可以用 `kafka-consumer-groups` 检查 MirrorMaker 读取的每一个分区——分区最后一条消息的偏移量、MirrorMaker 提交的最新偏移量以及它们之间的延迟。不过这个偏移量并不会 100% 准确，因为 MirrorMaker 并不会每时每刻都提交偏移量。在默认情况下，它会每分钟提交一次。所以，我们可能会在一分钟内看到延迟增加，然后又突然

下降。图 10-7 所示的延迟是 2，但用 `kafka-consumer-groups` 看到的是 5，因为 MirrorMaker 还没有提交最近的偏移量。LinkedIn 的 Burrow 也会监控这些信息，不过它使用了更为复杂的方法来识别真实的延迟，从而避免误报。
- 检查 MirrorMaker 读取的最新偏移量（即使还未提交）。内嵌在 MirrorMaker 中的消费者通过 JMX 发布关键指标，其中的一个指标是消费者最大延迟（基于它读取的所有分区计算得出）。这个延迟也不是 100% 准确，因为它只反映了消费者已读取的数据，并没有考虑生产者是否成功地将数据发送到目标集群中并收到确认。在图 10-7 中，MirrorMaker 的消费者会认为延迟是 1，而不是 2，因为它已经读取了消息 6，尽管这条消息还没有被生成到目标集群中。

需要注意的是，如果 MirrorMaker 跳过或丢弃了部分消息，则上述的两种方法是无法检测到的，因为它们只跟踪最新的偏移量。Confluent Control Center 是一款付费工具，可用于监控消息数量和校验和，弥补监控上的不足。

生产者和消费者的指标监控

MirrorMaker 所使用的 Connect 包含了一个生产者和一个消费者，它们提供了很多可用的指标。建议监控这些指标。Kafka 的文档列出了所有可用的指标。下面列出了几个对 MirrorMaker 调优非常有用的指标。

消费者

`fetch-size-avg`、`fetch-size-max`、`fetch-rate`、`fetch-throttle-time-avg` 和 `fetch-throttle-time-max`。

生产者

`batch-size-avg`、`batch-size-max`、`requests-in-flight` 和 `record-retry-rate`。

同时适用于两者

`io-ratio` 和 `io-wait-ratio`。

金丝雀验证

如果对所有东西都进行了监控，那么金丝雀（canary）验证就不是严格必需的，但为了实现多层监控，我们还是会进行金丝雀验证。我们每分钟向源集群的某个主题发送一条消息，然后再尝试从目标集群读取这条消息。如果消息在指定的时间之后才到达，就会发出告警，说明 MirrorMaker 出现了延迟或已经运行不正常了。

10.3.5 MirrorMaker 调优

MirrorMaker 是水平可伸缩的，集群的大小取决于吞吐量需求和对延迟的容忍度。如果不能容忍任何延迟，那么 MirrorMaker 的集群容量就需要满足吞吐量的上限。如果可以容忍一些延迟，则只要保证在 95%~99% 的时间里有 75%~80% 的容量可用即可。在吞吐量高峰期可以允许出现一些延迟，因为 MirrorMaker 有一些空余容量，在高峰期结束后可以很快赶上。

然后，你可能想通过设置不同的连接器任务数（通过 `tasks.max` 参数配置）来了解 MirrorMaker 的吞吐量。这主要取决于你所使用的硬件、数据中心或云服务供应商，所以需要自己进行测试。Kafka 提供了 `kafka-performance-producer` 工具，用于在源集群中制造负载，然后启动 MirrorMaker 镜像这个负载。在测试时，可以分别为 MirrorMaker 配置 1、2、4、8、16、24 和 32 个任务，并观察性能在哪个任务数开始出现下降，然后将 `tasks.max` 设置为比这个任务数小的值。如果你读取或生成的数据是压缩过的（因为网络带宽是跨集群镜像的瓶颈，所以建议将数据压缩后再传输），那么 MirrorMaker 还需要进行解压缩和再压缩。这会消耗很多 CPU 资源，所以在增加任务数时，要注意观察 CPU 的使用情况。通过这种方式你就可以知道单个 MirrorMaker 实例的最大吞吐量。如果单个实例的吞吐量还不够，那么可以测试更多的 MirrorMaker 实例和服务器。如果运行 MirrorMaker 的 Connect 集群中还有其他连接器，那么在调整集群大小时也要将这些连接器的负载考虑在内。

另外，你可能希望使用单独的 MirrorMaker 集群来镜像包含敏感数据的主题（要求低延迟并且镜像必须尽可能靠近源主题），这样可以避免主题过于臃肿或被失控的生产者拖慢数据管道。

能够对 MirrorMaker 做的调优也就是这些了。不过，还可以增加每个任务和 MirrorMaker 的吞吐量。

如果 MirrorMaker 是跨数据中心运行的，则可以通过调整 TCP 栈来增加有效带宽。第 3 章和第 4 章介绍过，可以分别用 `send.buffer.bytes` 和 `receive.buffer.bytes` 配置生产者和消费者的 TCP 缓冲区大小。类似地，可以用 `socket.send.buffer.bytes` 和 `socket.receive.buffer.bytes` 配置 broker 端的缓冲区大小。这些配置参数需要与 Linux 网络配置相结合，如下所示。

- 增加 TCP 缓冲区大小（`net.core.rmem_default`、`net.core.rmem_max`、`net.core.wmem_default`、`net.core.wmem_max` 和 `net.core.optmem_max`）。
- 启用时间窗口自动伸缩（`sysctl -w net.ipv4.tcp_window_scaling=1` 或者在 `/etc/sysctl.conf` 中添加 `net.ipv4.tcp_window_scaling=1`）。
- 减少 TCP 慢启动时间（将 `/proc/sys/net/ipv4/tcp_slow_start_after_idle` 设为 `0`）。

需要注意的是，Linux 网络调优包含了太多的内容。要了解更多的参数和细节，建议阅读相关的网络调优指南，比如由 Sandra K. Johnson 等人合著的 *PerformanceTuningforLinuxServers*。

除此之外，你可能还想对 MirrorMaker 中的生产者和消费者进行调优。首先，你可能想知道生产者或消费者是不是瓶颈所在，即生产者是否在等待消费者提供更多的数据，或者消费者在等待生产者提供更多的数据？一种办法是查看生产者和消费者的指标。如果其中一方空闲，而另一方很繁忙，那么就知道哪个需要调优了。另外一种办法是查看线程转储（可以使用 jstack 获得线程转储）。如果 MirrorMaker 的大部分时间用在轮询上，那么说明消费者是瓶颈所在；如果 MirrorMaker 的大部分时间用在发送消息上，则说明生产者是瓶颈所在。

如果需要对生产者进行调优，那么可以使用下面的配置参数。

`linger.ms` 和 `batch.size`

如果你发现生产者总是发送未被填满的批次（指标 `batch-size-avg` 和 `batch-size-max` 的值总是比 `batch.size` 小），那么可以通过增加一些延迟来提升吞吐量。可以把 `linger.ms` 设置得大一些，让生产者在发送批次之前等待几毫秒，让批次填充更多的数据。如果发送的数据都是满批的，并且还有空余内存，则可以配置更大的 `batch.size`，以便发送更大的批次。

`max.in.flight.requests.per.connection`

目前，如果某些消息需要多次重试才能确认发送成功，那么 MirrorMaker 保证消息有序的唯一方法是将处理中的请求数量限制为 1。但这意味着在发送下一条消息之前，生产者当前发送的消息必须得到目标集群的确认。这可能会对吞吐量造成限制，特别是如果在 broker 确认消息之前存在显著的延迟。如果消息顺序对你来说不是很关键，那么保留 `max.in.flight.requests.per.connection` 的默认值 5 可以显著增加吞吐量。

下面的配置参数可用于提升消费者的吞吐量。

`fetch.max.bytes`

如果指标 `fetch-size-avg` 和 `fetch-size-max` 的数值与 `fetch.max.bytes` 很接近，那么说明消费者读取的数据已经接近 broker 允许的上限。如果有更多的可用内存，则可以配置更大的 `fetch.max.bytes`，这样消费者就可以在每个请求中读取更多的数据。

`fetch.min.bytes` 和 `fetch.max.wait.ms`

如果指标 `fetch-rate` 的数值很高，那么说明消费者发送了太多请求，但每个请求都获取不到足够的数据。这个时候可以配置更大的 `fetch.min.bytes` 和 `fetch.max.wait.ms`，这样消费者的每个请求就可以获取到更多数据，broker 会等到有足够可用数据时才将响应返回。

10.4 其他跨集群镜像方案

我们已经深入了解了 MirrorMaker。不过，在实际使用当中，MirrorMaker 也存在一些局限性。除了 MirrorMaker，我们还有其他一些替代方案，它们弥补了 MirrorMaker 的不足，避开了 MirrorMaker 的复杂性。下面将介绍几个来自 Uber 和 LinkedIn 的开源解决方案，以及 Confluent 的商业解决方案。

10.4.1 Uber 的 uReplicator

Uber 大规模使用了旧版的 MirrorMaker，随着主题和分区数量的增加以及集群吞吐量的增长，它开始面临一些问题。旧版 MirrorMaker 使用一个消费者群组中的消费者从源主题读取数据，在增加 MirrorMaker 线程和实例、重启 MirrorMaker 实例或添加新主题时，都会触发消费者再均衡。正如第 4 章所述，再均衡会停止所有消费者，直到新分区被分配给每一个消费者。如果主题和分区的数量很大，那么整个过程将需要很长时间，在使用了旧版消费者的情况下则更是如此。有时候可能会出现 5~10 分钟的停顿，导致镜像进度滞后，堆积了大量的待镜像数据，需要更长时间才能恢复。这导致从目标集群读取数据的消费者出现了很大延迟。为了避免因为有人添加新主题而发生再均衡，Uber 决定把每一个需要镜像的主题都列出来，而不是使用正则表达式。但这给维护带来了麻烦，因为要添加一个新主题，所有 MirrorMaker 实例都必须重新配置和重启。如果配置错误，则可能会导致无止境的再均衡，因为消费者无法就它们订阅的主题达成一致。

为了解决上述问题，Uber 开发了 MirrorMaker 的克隆版，叫作 uReplicator。Uber 决定将 Apache Helix（以下简称 Helix）作为中心控制器（具有高可用性），用于管理主题列表和分配给每个 uReplicator 实例的分区。管理员通过 REST API 添加新主题，uReplicator 负责将分区分配给不同的消费者。Uber 用自己开发的 Helix 消费者替换了 MirrorMaker 中的 Kafka 消费者。Helix 消费者的分区由 Helix 控制器负责分配，不需要在消费者之间进行协调（更多细节请参考第 4 章），这样就可以避免再均衡，它们只需监听来自 Helix 控制器的分配变更事件即可。

Uber 工程部门写了一篇博文“uReplicator: Uber Engineering's Robust Apache Kafka Replicator”，分享了 uReplicator 的架构细节和他们所经历的改进过程。uReplicator 依赖于 Helix，所以又多了一个需要管理的组件，这给部署带来了额外的复杂性。MirrorMaker 2.0 解决了旧版 MirrorMaker 的伸缩性和容错性问题，并且不需要依赖外部组件。

10.4.2 LinkedIn 的 Brooklin

和 Uber 一样，LinkedIn 也使用旧版 MirrorMaker 在 Kafka 集群之间传输数据。随着数据规模的增长，它也遇到了类似的伸缩性问题和运维难题。因此，LinkedIn 在它的数据流系统 Brooklin 基础上构建了一种镜像解决方案。Brooklin 是一个分布式服务，可以在异构数据源和目标系统（包括 Kafka）之间传输数据。作为一个可以用来构建数据管道的通用数据摄取框架，Brooklin 支持多种场景。

- 数据桥，将来自不同数据源的数据送入流式处理系统。
- 将来自不同数据存储系统的变更数据捕获（CDC）事件流化。
- Kafka 跨集群镜像解决方案。

Brooklin 是一个可扩展、具备高可靠性的分布式系统，并已针对 Kafka 进行了大规模测试。它每天可镜像数万亿条消息，并在稳定性、性能和可操作性方面进行了优化。Brooklin 还提供了一个管理 REST API。它是一个共享服务，可以处理大量的数据管道和跨多个 Kafka 集群镜像数据。

10.4.3 Confluent 的跨数据中心镜像解决方案

在 Uber 开发 uReplicator 的同时，Confluent 也开发了 Confluent Replicator。尽管名字有点儿相似，但这两个项目几乎没有共同点，因为它们是为解决不同的 MirrorMaker 问题而开发的。与后来推出的 MirrorMaker 2.0 一样，Replicator 也是基于 Connect。开发 Replicator 的目的是为了解决 Confluent 的企业客户在使用旧版 MirrorMaker 进行多集群镜像时遇到的问题。

对于那些部署了延展集群的用户，Confluent Server（Confluent 平台的一个商业组件）提供了多区域集群（multi-region cluster，MRC）特性。MRC 使用了异步复制，降低了对延迟和吞吐量的影响。与延展集群

一样，这种解决方案适用于延迟低于 50 毫秒的多地域之间的数据复制，并可以享受到透明的客户端故障转移所带来的好处。对于网络不太可靠的远程集群，Confluent Server 提供了一个叫作集群链接（cluster linking）的特性。集群链接对 Kafka 的集群内复制协议进行了扩展，将其用于集群之间的数据镜像。

下面来看看这些解决方案都支持哪些特性。

Confluent Replicator

Confluent Replicator 是一个镜像工具，与 MirrorMaker 类似，它也依赖于 Connect 框架，并可以在 Connect 集群中运行。它们都支持各种拓扑的数据复制以及消费者偏移量和主题配置信息的迁移。它们在特性方面存在一些差异。例如，MirrorMaker 支持 ACL 迁移和各种客户端的偏移量转换，但 Replicator 不迁移 ACL，而且只支持 Java 客户端的偏移量转换（使用时间戳拦截器）。与 MirrorMaker 不同的是，Replicator 没有本地主题和远程主题的概念，但它支持聚合主题。与 MirrorMaker 相同的是，Replicator 也可以避免循环复制，但它使用的是来源标头。Replicator 提供了一系列指标，比如复制延迟，我们可以使用它提供的 REST API 或 Control Center UI 来监控这些指标。它还支持集群之间的模式迁移和模式转换。

多区域集群（**MRC**）

如前所述，延展集群为客户端提供了简单透明的故障转移和故障回退能力。但是，延展集群要求数据中心之间靠得比较近并提供稳定的低延迟网络，以支持数据中心之间的同步复制。MRC 也只适用于延迟在 50 毫秒以内的数据中心，但它组合使用了同步复制和异步复制，以此来降低对生产者性能的影响，并提供更高的网络容错性。

Kafka 允许客户端从跟随者副本获取数据，这样客户端就可以根据机架 ID 从最近的 broker 获取数据，从而减少跨数据中心的流量。Confluent Server 加入了观察者的概念。观察者是不属于 ISR 的异步副本，即使设置了 `acks=all` 也不对生产者产生影响，但又能够将消息发送给消费者。可以同时使用区域内的同步复制和区域间的异步复制，以实现低延迟和高持久性。Confluent Server 有副本放置约束，你可以基于机架 ID 指定每个区域最少放置多少个副本，以确保副本分布在多个区域，从而保证持久性。Confluent Platform 6.1 可以根据配置的条件自动提升观察者，确保在不丢失数据的情况下自动实现快速故障转移。当 `min.insync.replicas` 低于配置的最小同步副本数量时，同步的观察者将自动被提升，并可以加入 ISR，这样 ISR 的数量就可以恢复到配置的最小数量。被提升的观察者使用的是同步复制，可能会影响吞吐量，但集群在整个过程中会保持运行状态，即使某个区域发生故障，也不会有数据丢失。当发生故障的区域恢复之后，观察者会自动降级，集群就可以恢复到正常的性能水平。

集群链接

集群链接是 Confluent Platform 6.0 的一个预览特性，它直接将集群间复制功能放在了 Confluent Server 中。集群链接将集群内的 broker 间复制协议用在了集群间复制上，可以跨集群执行带偏移量迁移的数据复制，无须偏移量转换就可以实现客户端无缝迁移。主题配置信息、分区、消费者偏移量和 ACL 都可以在两个集群之间保持同步，以便在发生灾难时实现低 RTO 的故障转移。集群链接定义了从一个源集群到一个目标集群的定向流。目标集群的首领 broker 从对应的源首领那里获取分区数据，目标集群的跟随者使用 Kafka 的标准复制机制从本地首领那里复制数据。目标集群的镜像主题会被标记为只读，以防止本地生产者将数据生成到这些主题上，从而确保镜像主题在逻辑上与源主题保持一致。

集群链接提供了运维上的便利性，无须用到其他集群（如 Connect 集群），并且比外部工具更加高效，因为它避免了镜像期间的解压缩和再压缩。与 MRC 不同，集群链接没有同步复制选项，而且在进行客户端故障转移时需要手动重启客户端。不过，集群链接可用于网络不可靠且高延迟的远程数据中心，它只在数据中心之间复制一次数据，从而减少了跨数据中心的流量。它适用于迁移集群和需要共享主题的场景。

10.5 小结

从解释为什么需要多个 Kafka 集群开始，本章首先介绍了几种从简单到复杂的多集群架构。然后介绍了 Kafka 故障转移的实现细节，并比较了当前几种可用的解决方案。接下来介绍了一些可用的工具，从 MirrorMaker 开始，说明了在生产环境中使用 MirrorMaker 需要注意的细节问题。最后介绍了 MirrorMaker 之外的替代方案，它们弥补了 MirrorMaker 的不足。

不管最终选择哪一种架构和工具，总少不了要对多集群进行配置和对镜像管道进行测试。因为 Kafka 多集群管理比关系数据库要简单得多，所以很多组织在设计、规划、测试、自动化部署、监控和维护方面疏于投入。重视多集群的管理，并把它作为组织全盘灾备计划或多区域计划的一部分，才能更好地管理多个 Kafka 集群。

第 11 章　保护 Kafka

Kafka 被应用在各种各样的场景中，从网站活动跟踪和指标管道到病人记录管理和在线支付。每一种场景对安全性、性能、可靠性和可用性都有不同的需求。虽然使用最强大且最新的安全特性总是最好的，但通常也难免要做出权衡，因为安全性的提升是以影响性能、成本和用户体验为代价的。Kafka 支持几种标准的安全技术，并为各种应用场景提供了一系列安全配置选项。

与性能和可靠性一样，我们也需要从系统的整体层面考虑安全性，而不是只考虑单个组件。系统的安全性取决于最薄弱的环节，必须在整个系统（包括底层平台）层面执行安全流程和策略。Kafka 提供了可配置的安全特性，可以与现有的安全基础设施集成，构建出一个适用于整个系统的安全模型。

在本章中，我们首先会介绍 Kafka 的安全特性，看看它们如何解决不同的安全问题，并为 Kafka 集群提供整体的安全保护。然后会分享最佳实践、潜在的安全威胁以及用于抵御这些威胁的安全技术。最后还会讨论其他可用于保护 ZooKeeper 和平台其余组件的安全措施。

11.1 锁住 Kafka

Kafka 采用了一系列安全措施来建立和维护数据的机密性、完整性和可用性。

- 身份验证特性用于识别和确定用户身份。
- 授权特性决定了用户可以做什么。
- 加密特性保护数据不被窃取和篡改。
- 审计特性用于跟踪用户已经做了什么或试图做什么。
- 配额特性控制用户可以使用多少资源。

为了了解如何锁住 Kafka，先来看看数据是如何在 Kafka 集群中流动的。数据流经集群的主要步骤如图 11-1 所示。本章将用这个示例来说明可以通过哪些方式配置 Kafka，为流经每一个步骤的数据提供保护，从而保证整个系统的安全。

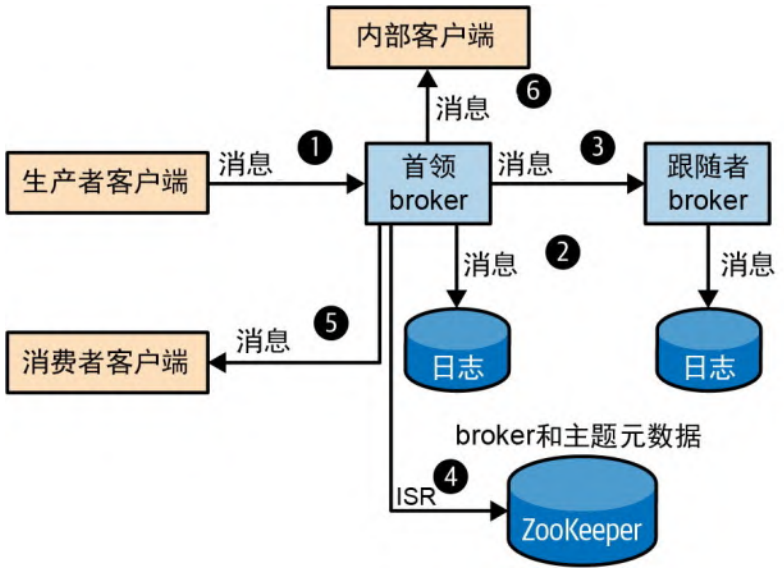

图 11-1：数据在 Kafka 集群中流动

01. Alice 的客户端会向 `customerOrders` 主题的一个分区写入一条客户订单记录。这条记录会被发送给分区的首领。
02. 首领 broker 会将记录写入它的本地日志文件。
03. 跟随者 broker 会从首领那里获取消息，并将其写入它的本地日志文件。
04. 如果需要更新同步副本信息，那么首领 broker 会直接更新 ZooKeeper 中的分区状态。
05. Bob 的客户端会读取 `customerOrders` 主题中的客户订单记录。Bob 会收到由 Alice 发送的消息。
06. 一个内部应用程序会负责处理 `customerOrders` 主题的所有消息，并实时生成最受欢迎的产品的指标。

一个安全的部署必须保证做到以下几点。

客户端真实性

当 Alice 的客户端连接到 broker，broker 需要验证客户端，确保消息确实来自 Alice。

服务器端真实性

在向首领 broker 发送消息之前，Alice 的客户端需要验证自己连接的确实是真实的 broker。

数据隐私

消息流经的所有连接通道以及保存消息的磁盘都应该进行加密或物理保护，以防止窃听者读取数据，确保数据不被窃取。

数据完整性

在不安全网络上传输的数据需要包含消息摘要，以用于检测数据是否被篡改。

访问控制

在将消息写入日志之前，首领 broker 需要验证 Alice 的客户端是否有写入 `customerOrders` 主题的权限。在将数据返回给 Bob 之前，broker 需要验证 Bob 的客户端是否有读取主题消息的权限。如果 Bob 使用了消费者群组，则 broker 还需要验证 Bob 的客户端是否有权访问这个消费者群组。

可审核性

broker、Alice 的客户端、Bob 的客户端和其他客户端执行的所有操作都需要被审计跟踪。

可用性

broker 需要启用配额限制，避免某些用户占用所有可用的带宽或借助拒绝服务攻击来拖垮 broker。为了保证 Kafka 集群的可用性，需要锁住 ZooKeeper，因为 broker 的可用性依赖于 ZooKeeper 的可用性和保存在 ZooKeeper 中的元数据的完整性。

在接下来的内容中，我们将探讨可以提供上述这些保证的 Kafka 安全特性。我们首先会介绍 Kafka 的连接模型，以及与连接（客户端与 broker 之间的连接）相关的安全协议。然后会详细研究每一个安全协议，并介绍每个协议用于确保客户端真实性和服务器端真实性的身份验证特性。之后还会介绍不同阶段的加密选项，包括一些安全协议内置的用于解决数据隐私和数据完整性问题的数据传输加密特性。接下来会探讨 Kafka 提供的用来管理访问控制和审计日志的可定制授权模型。最后会介绍系统其余部分的安全性，包括 ZooKeeper 和平台的安全性，因为它们也是维持 Kafka 可用性所必需的组件。如果想了解更多有关配额（通过在用户之间公平分配资源来提高服务可用性）的内容，请参阅第 3 章。

11.2 安全协议

我们在 broker 的一个或多个端点上配置了监听器，这些监听器负责接收来自客户端的连接。每个监听器可以有自己的安全设置。受物理保护且仅允许授权人员访问的内部监听器的安全需求与可通过公共互联网访问的外部监听器的安全需求是不一样的。安全协议的选择决定了数据传输的身份验证和加密级别。

Kafka 使用两种标准技术（TLS 和 SASL）支持 4 种安全协议。传输层安全（transport layer security，TLS）通常以安全套接字层（secure sockets layer，SSL，TLS 的前身）作为代称，支持加密以及客户端和服务器端的身份验证。简单身份验证和安全层（simple authentication and security layer，SASL）是一个在面向连接的协议中使用不同的机制实现身份验证的框架。每一个 Kafka 安全协议都结合了传输层安全（PLAINTEXT 或 SSL）和可选的认证层安全（SSL 或 SASL）。

PLAINTEXT

没有身份验证的 PLAINTEXT 传输层，只适用于在私有网络内传输不敏感的数据，因为它没有使用身份验证或加密。

SSL

带有可选 SSL 客户端身份验证的 SSL 传输层，适用于不安全网络，因为它支持客户端和服务器端身份验证以及加密。

SASL_PLAINTEXT

带有 SASL 客户端身份验证的 PLAINTEXT 传输层。一些 SASL 机制也支持服务器端身份验证。它不支持加密，因此只适用于私有网络。

SASL_SSL

带有 SASL 身份验证的 SSL 传输层，适用于不安全网络，因为它支持客户端和服务器端身份验证以及加密。

TLS/SSL

TLS 是在公共互联网上使用最为广泛的加密协议之一。一些应用程序协议（比如 HTTP、SMTP 和 FTP）都依赖 TLS 提供数据传输的隐私性和完整性。TLS 用公钥（PKI）创建、管理和分发可用于非对称加密的数字证书，避免在服务器端和客户端之间共享密钥。在 TLS 握手过程中生成的会话密钥可用于对称加密，这样后续的数据传输就有了更高的性能。

可以通过 `inter.broker.listener.name` 或 `security.inter.broker.protocol` 来配置用于 broker 间通信的监听器。用于 broker 间通信的安全协议必须配置在 broker 级别（包括服务器端和客户端的配置），因为 broker 需要为监听器建立客户端连接。下面的例子演示了如何为 broker 间通信和内部监听器配置 SSL，以及为外部监听器配置 SASL_SSL。

```
listeners=EXTERNAL://:9092,INTERNAL://10.0.0.2:9093,BROKER://10.0.0.2:9094
advertised.listeners=EXTERNAL://broker1.example.com:9092,INTERNAL://
broker1.local:9093,BROKER://broker1.local:9094
listener.security.protocol.map=EXTERNAL:SASL_SSL,INTERNAL:SSL,BROKER:SSL
inter.broker.listener.name=BROKER
```

我们为客户端配置了一个安全协议和可以用来确定 broker 监听器的引导服务器地址。返回给客户端的元数据只包含 broker 监听器对应的端点。

```
security.protocol=SASL_SSL
bootstrap.servers=broker1.example.com:9092,broker2.example.com:9092
```

下一节将介绍每一个安全协议的 broker 和客户端配置选项。

11.3 身份验证

身份验证是通过建立客户端和服务器端身份来验证客户端和服务器端真实性的过程。当 Alice 的客户端连接到首领 broker 时，可以通过服务器端身份验证来确定自己连接的就是真实的 broker。在进行客户端身份验证时，服务器通过验证 Alice 的凭证（比如密码或数字证书）来确定 Alice 的身份，确保连接是来自 Alice 而不是冒充者。一旦通过身份验证，Alice 的身份就与连接相关联，并在整个连接生命周期中起作用。Kafka 用 `KafkaPrincipal` 实例表示客户端身份，并用它授予资源访问权限，以及为具有这个客户端身份的连接分配配额。每个连接的 `KafkaPrincipal` 实例都是在身份验证过程中基于某种身份验证协议创建的。如果使用了基于密码的身份验证，那么 Alice 的主体就是 `User:Alice`。可以通过配置 broker 的 `principal.builder.class` 来自定义 `KafkaPrincipal`。

匿名连接

`User:ANONYMOUS` 这个主体被用于未经身份验证的连接，包括 PLAINTEXT 监听器接受的客户端连接和 SSL 监听器接受的未经身份验证的客户端连接。

11.3.1 SSL

如果监听器的安全协议配置的是 SSL 或 SASL_SSL，那么监听器连接的安全传输层就会使用 TLS。建立 TLS 连接需要一个握手过程，在这个过程中会执行身份验证、协商加密参数，并生成用于加密的共享密钥。客户端通过验证服务器的数字证书来确定服务器的身份。如果启用了 SSL 客户端身份验证，则服务器也会通过验证客户端数字证书来确定客户端的身份。所有的 SSL 流量都是加密的，适合用在不安全的网络中。

SSL 的性能

因为 SSL 通道是加密的，所以会增加 CPU 方面的开销。SSL 目前不支持零复制传输。根据流量模式的不同，增加的开销可能会高达 20%~30%。

01. **配置 TLS**

如果配置了 SSL 或 SASL_SSL 的 broker 监听器启用了 TLS，则还需要为 boker 配置一个密钥存储，其中包含 broker 的私有密匙和证书。同时，客户端也需要配置一个信任存储，其中包含 broker 证书或签署了 broker 证书的证书颁发机构（CA）的证书。broker 证书需要包含作为主体别名（subject alternative name，SAN）扩展或公用名称（common name，CN）的 broker 主机名，客户端可以用它验证服务器的主机名。也可以使用通配符证书，为同一域名下的所有 broker 使用相同的密钥存储，以此来简化配置。

服务器主机名验证

在默认情况下，Kafka 客户端会验证保存在服务器证书中的服务器主机名与客户端连接的主机名是否匹配。连接用的主机名既可以是在客户端配置的引导服务器地址，也可以是 broker 通过元数据响应返回给客户端的监听器主机名。主机名验证是服务器端身份验证的一个关键部分，可以防止中间人攻击，因此在生产环境中不应该被禁用。

可以在 broker 端配置 `ssl.client.auth=required`，让 broker 对 SSL 监听器接受的客户端连接进行身份验证。客户端需要配置一个密钥存储；broker 端需要配置一个信任存储，其中包含客户端证书或客户端证书的 CA。如果 broker 间通信也启用了 SSL，那么 broker 端的信任存储需要包含 broker 证书的 CA 和客户端证书的 CA。在默认情况下，客户端证书中的可识别名称（distinguished name，DN）会被作为 `KafkaPrincipal`，用于授权和配额。可以通过配置

ssl.principal.mapping.rules 来自定义主体。配置了 SASL_SSL 的监听器将禁用 TLS 客户端身份验证，其安全性主要依赖 SASL 身份验证和由 SASL 创建的 KafkaPrincipal。

SSL 客户端身份验证

如果配置了 ssl.client.auth=requested，那么 SSL 客户端身份验证就会变成可选的。在这种情况下，没有配置密钥存储的客户端可以完成 TLS 握手，但其主体是 User:ANONYMOUS。

下面的例子演示了如何使用自签名 CA 为服务器端和客户端身份验证创建密钥存储和信任存储。

为 broker 生成自签名 CA 密钥对。

```
$ keytool -genkeypair -keyalg RSA -keysize 2048 -keystore server.ca.p12        \
  -storetype PKCS12 -storepass server-ca-password -keypass server-ca-password  \
  -alias ca -dname "CN=BrokerCA" -ext bc=ca:true -validity 365      ❶
$ keytool -export -file server.ca.crt -keystore server.ca.p12 \
  -storetype PKCS12 -storepass server-ca-password -alias ca -rfc    ❷
```

❶ 为 CA 创建一个密钥对，并将其保存在 PKCS12 格式的 server.ca.p12 文件中。我们将使用它来签署证书。

❷ 将 CA 公共证书导出到 server.ca.crt，它将被包含在信任存储和证书链中。

为使用自签名 CA 的 broker 创建密钥存储。如果使用了通配符主机名，那么同一个密钥存储就可以用于所有的 broker，否则就用完全限定域名（fully qualified domain name，FQDN）为每一个 broker 创建单独的密钥存储。

```
$ keytool -genkey -keyalg RSA -keysize 2048 -keystore server.ks.p12          \
  -storepass server-ks-password -keypass server-ks-password -alias server    \
  -storetype PKCS12 -dname "CN=Kafka,O=Confluent,C=GB" -validity 365       ❶
$ keytool -certreq -file server.csr -keystore server.ks.p12 -storetype PKCS12 \
  -storepass server-ks-password -keypass server-ks-password -alias server ❷
$ keytool -gencert -infile server.csr -outfile server.crt                    \
  -keystore server.ca.p12 -storetype PKCS12 -storepass server-ca-password    \
  -alias ca -ext SAN=DNS:broker1.example.com -validity 365                 ❸
$ cat server.crt server.ca.crt > serverchain.crt
$ keytool -importcert -file serverchain.crt -keystore server.ks.p12          \
  -storepass server-ks-password -keypass server-ks-password -alias server    \
  -storetype PKCS12 -noprompt                                              ❹
```

❶ 为 broker 生成私钥，并将其保存在 PKCS12 格式的 server.ks.p12 文件中。

❷ 生成证书签名请求。

❸ 用 CA 密钥存储对 broker 证书进行签名。签名后的证书保存在 server.crt 文件中。

❹ 将 broker 证书链导入到 broker 密钥存储中。

如果 broker 间通信启用了 TLS，就用 broker 的 CA 证书为 broker 创建一个信任存储，用于 broker 间的身份验证。

```
$ keytool -import -file server.ca.crt -keystore server.ts.p12 \
  -storetype PKCS12 -storepass server-ts-password -alias server -noprompt
```

用 broker 的 CA 证书为客户端生成一个信任存储。

```
$ keytool -import -file server.ca.crt -keystore client.ts.p12 \
  -storetype PKCS12 -storepass client-ts-password -alias ca -noprompt
```

如果启用了 TLS 客户端身份验证，则客户端必须配置密钥存储。下面的脚本将为客户端生成自签名 CA，并用客户端 CA 签名的证书为客户端创建密钥存储。客户端 CA 会被添加到 broker 的信任存储中，这样 broker 就可以验证客户端的真实性了。

```
# 生成客户端自签名CA密钥对
keytool -genkeypair -keyalg RSA -keysize 2048 -keystore client.ca.p12          \
  -storetype PKCS12 -storepass client-ca-password -keypass client-ca-password \
  -alias ca -dname CN=ClientCA -ext bc=ca:true -validity 365  ❶
keytool -export -file client.ca.crt -keystore client.ca.p12  -storetype PKCS12 \
  -storepass client-ca-password -alias ca -rfc

# 生成客户端密钥存储
keytool -genkey -keyalg RSA -keysize 2048 -keystore client.ks.p12           \
  -storepass client-ks-password -keypass client-ks-password -alias client   \
  -storetype PKCS12 -dname "CN=Metrics App,O=Confluent,C=GB" -validity 365 ❷
keytool -certreq -file client.csr -keystore client.ks.p12 -storetype PKCS12 \
  -storepass client-ks-password -keypass client-ks-password -alias client
keytool -gencert -infile client.csr -outfile client.crt                     \
  -keystore client.ca.p12 -storetype PKCS12 -storepass client-ca-password   \
  -alias ca -validity 365
cat client.crt client.ca.crt > clientchain.crt
keytool -importcert -file clientchain.crt -keystore client.ks.p12           \
  -storepass client-ks-password -keypass client-ks-password -alias client   \
  -storetype PKCS12 -noprompt                                               ❸

# 将客户端CA证书添加到broker的信任存储中
keytool -import -file client.ca.crt -keystore server.ts.p12 -alias client \
   -storetype PKCS12 -storepass server-ts-password -noprompt  ❹
```

❶ 在这个例子中为客户端创建一个新的 CA。

❷ 在默认情况下，客户端身份验证会将 `User:CN=Metrics App,O=Confluent,C=GB` 作为主体。

❸ 将客户端证书链添加到客户端密钥存储中。

❹ broker 的信任存储应该包含客户端的 CA。

有了密钥和信任存储后，就可以为 broker 配置 TLS 了。只有当 broker 间通信启用了 TLS 或客户端身份验证时，broker 才需要配置信任存储。

```
ssl.keystore.location=/path/to/server.ks.p12
ssl.keystore.password=server-ks-password
ssl.key.password=server-ks-password
ssl.keystore.type=PKCS12
ssl.truststore.location=/path/to/server.ts.p12
ssl.truststore.password=server-ts-password
ssl.truststore.type=PKCS12
ssl.client.auth=required
```

为客户端配置信任存储。如果需要进行客户端身份验证，则还要为客户端配置密钥存储。

```
ssl.truststore.location=/path/to/client.ts.p12
ssl.truststore.password=client-ts-password
ssl.truststore.type=PKCS12
ssl.keystore.location=/path/to/client.ks.p12
ssl.keystore.password=client-ks-password
ssl.key.password=client-ks-password
ssl.keystore.type=PKCS12
```

信任存储

如果使用了由知名的受信任权威机构签署的证书，就可以省略 broker 和客户端的信任存储配置。在这种情况下，将使用 Java 提供的默认信任存储来建立信任。第 2 章介绍过相关的安装步骤。

必须在证书过期前更新密钥和信任存储，避免 TLS 握手失败。可以通过直接修改存储文件或将配置参数指向新的存储文件来更新 broker 的 SSL 存储。在这两种情况下，都可以使用 Admin API 或 Kafka 配置工具来更新存储。下面的例子使用了 Kafka 配置工具来更新 broker 0 外部监听器的密钥存储。

```
$ bin/kafka-configs.sh --bootstrap-server localhost:9092     \
  --command-config admin.props                               \
  --entity-type brokers --entity-name 0 --alter --add-config \
  'listener.name.external.ssl.keystore.location=/path/to/server.ks.p12'
```

02. 安全方面的考虑

TLS 广泛用于为多种协议（包括 HTTPS）提供传输层安全性。在关键应用程序中采用安全协议时，不管是哪一种协议，最重要的是要了解潜在的威胁和缓解策略。Kafka 默认只启用了较新的 TLSv1.2 和 TLSv1.3，因为旧协议（如 TLSv1.1）存在已知的漏洞。由于不安全重协商存在已知漏洞，因此 Kafka 不支持 TLS 连接重协商。为了防止中间人攻击，Kafka 默认启用了主机名验证。可以通过限制加密机制进一步加强安全性。在不安全网络中传输数据时，可以使用至少 256 位的加密密钥来防止加密攻击，并确保数据的完整性。为达到 FIPS 140-2 等安全标准，一些组织要求使用 TLS 协议和限制加密机制。

在默认情况下，包含私钥的密钥存储直接保存在文件系统中，所以非常有必要对文件系统的访问权限进行限制。如果私钥被泄露，那么可以使用标准的 Java TLS 特性来吊销证书。还可以使用寿命较短的密钥来降低泄露的概率。

TLS 握手过程开销巨大，并占用了 broker 大量的网络线程时间。对于不安全网络中的 TLS 监听器，要用连接配额来保护它们不受拒绝服务攻击，从而保证 broker 的可用性。broker 的配置参数 `connection.failed.authentication.delay.ms` 可以用来延迟发送身份验证失败响应，以降低客户端对身份验证失败的重试速率。

11.3.2 SASL

Kafka 协议支持使用 SASL 进行身份验证，并内置支持几种常用的 SASL 机制。SASL 可以结合 TLS 作为传输层，提供一个有身份验证和加密的安全通道。SASL 身份验证通过一系列服务器质询（challenge）和客户端响应来实现，其中 SASL 机制定义了质询和响应的序列和连接格式。broker 支持以下几种 SASL 机制，并通过回调机制集成现有的安全基础设施。

GSSAPI

SASL/GSSAPI 支持 Kerberos 身份验证，并可以与 Active Directory 或 OpenLDAP 等 Kerberos 服务器集成。

PLAIN

用户名和密码身份验证通常与自定义服务器回调一起使用，用于验证保存在外部密码存储系统中的密码。

SCRAM-SHA-256 和 SCRAM-SHA-512

Kafka 提供的用户名和密码身份验证，不需要额外的密码存储。

OAUTHBEARER

使用 OAuth 承载令牌进行身份验证，通过自定义回调来获取和验证标准 OAuth 服务器授予的令牌。

可以通过 sasl.enabled.mechanisms 参数为启用了 SASL 的 broker 监听器配置一个或多个 SASL 机制。客户端可以通过配置 sasl.mechanism 参数来选择任何一种已被启用的机制。

Kafka 使用 Java 认证和授权服务（JAAS）来配置 SASL。配置参数 sasl.jaas.config 包含了一个 JAAS 配置条目，其中指定了登录模块及相关参数。我们在配置 sasl.jaas.config 时会将 listener 和 mechanism 作为前缀。例如，listener.name.external.gssapi.sasl.jaas.config 表示为监听器 EXTERNAL 的 SASL/GSSAPI 机制配置 JAAS 条目。broker 和客户端的登录过程将使用 JAAS 配置来确定用于身份验证的公共和私有凭证。

JAAS 配置文件

JAAS 也可以配置在文件中，并通过 Java 的系统属性 java.security.auth.login.config 来指定配置文件路径。不过，还是推荐使用 Kafka 的 sasl.jaas.config 配置参数，因为它支持密码保护，而且如果监听器启用了多种机制，则可以对每种 SASL 机制进行单独的配置。

Kafka 支持的 SASL 机制可以通过回调机制与第三方认证服务器集成。我们可以为 broker 或客户端提供一个登录回调，以此来自定义登录过程，例如，获取用于身份验证的凭证。也可以提供一个服务器端回调，用它来验证客户端凭证，例如，通过一台外部密码服务器来验证密码。还可以提供一个客户端回调，将客户端凭证注入而不是包含在 JAAS 配置中。

下面将更详细地探讨 Kafka 支持的 SASL 机制。

01. **SASL/GSSAPI**

Kerberos 是一种被广泛使用的网络身份验证协议，它使用了强加密技术，支持在不安全网络中进行安全的双向身份验证。通用安全服务应用程序编程接口（GSS-API）是一个为使用了不同身份验证机制的应用程序提供安全服务的框架。RFC-4752 将 GSS-API 的 Kerberos V5 身份验证机制引入到了 SASL 中。Kerberos 服务器的开源和企业级实现让 Kerberos 成为很多对安全性有严格要求的行业的选择。Kafka 通过 SASL/GSSAPI 支持 Kerberos 身份验证。

配置 SASL/GSSAPI。Kafka 使用包含在 Java 运行时环境中的 GSSAPI 提供程序来支持 Kerberos 身份验证。GSSAPI 的 JAAS 配置包含了一个密钥表（keytab）文件路径，这个文件包含了主体和加密密钥的映射。要为 broker 配置 GSSAPI，需要用包含 broker 主机名的主体为每一个 broker 创建一个密钥表。客户端会验证 broker 主机名，以确保服务器的真实性，并防止中间人攻击。在进行身份验证时，Kerberos 需要一个安全的 DNS 服务来查找主机名。如果正向和反向查找不匹配，那么可以在客户端 Kerberos 配置文件 krb5.conf 中设置 rdns=false，以此来禁用反向查找。每个 broker 的 JAAS 配置需要包含 Java 运行时提供的 Kerberos V5 登录模块、密钥表文件的路径和完整的 broker 主体。

```
sasl.enabled.mechanisms=GSSAPI
listener.name.external.gssapi.sasl.jaas.config=\ ❶
  com.sun.security.auth.module.Krb5LoginModule required \
    useKeyTab=true storeKey=true     \
    keyTab="/path/to/broker1.keytab" \ ❷
    principal="kafka/broker1.example.com@EXAMPLE.COM"; ❸
```

❶ 使用带有监听器前缀的 sasl.jaas.config，前缀中包含了小写形式的监听器名称和 SASL 机制。

❷ 密钥表文件对 broker 进程来说必须是可读的。

❸ broker 的服务主体需要包含 broker 主机名。

如果 broker 间通信启用了 SASL/GSSAPI，则也需要配置 broker 间的 SASL 机制和 Kerberos 服务名。

```
sasl.mechanism.inter.broker.protocol=GSSAPI
sasl.kerberos.service.name=kafka
```

客户端需要在 JAAS 配置中指定它们自己的密钥表和主体，并用 `sasl.kerberos.service.name` 指定它们想要连接的服务名。

```
sasl.mechanism=GSSAPI
sasl.kerberos.service.name=kafka ❶
sasl.jaas.config=com.sun.security.auth.module.Krb5LoginModule required \
    useKeyTab=true storeKey=true   \
    keyTab="/path/to/alice.keytab" \
    principal="Alice@EXAMPLE.COM"; ❷
```

❶ 为客户端指定 Kafka 服务名。

❷ 客户端可以使用没有主机名的主体。

在默认情况下，主体的短名称将作为客户端标识。例如，例子中的 `User:Alice` 就是客户端主体，`User:kafka` 则是 broker 主体。可以用 broker 的 `sasl.kerberos.principal.to.local.rules` 参数来指定一组规则，对主体进行自定义转换。

安全方面的考虑。如果在生产环境中使用了 Kerberos，那么建议使用 SASL_SSL 来保护身份验证流程和经过身份验证之后的数据流量。如果不使用 TLS 来提供安全传输层，则网络窃听者有可能可以获取到足够的信息来发动字典攻击或暴力破解攻击，并窃取客户端凭证。使用随机生成的密钥比使用基于密码生成的密钥更为安全。应该避免使用较弱的加密算法（如 DES-MD5），并通过文件系统权限来限制对密钥表文件的访问，因为能够访问这个文件的用户都可能成为冒充者。

SASL/GSSAPI 需要一个安全的 DNS 服务来进行服务器身份验证。由于针对 KDC 服务或 DNS 服务的拒绝服务攻击有可能会导致客户端身份验证失败，因此有必要对这些服务的可用性进行监控。Kerberos 还依赖可配置的松散同步时钟来检测重放攻击，所以要确保时钟同步是安全的。

02. **SASL/PLAIN**

RFC-4616 定义了一个简单的用户名和密码身份验证机制，可以与 TLS 一起，为我们提供安全的身份验证。在进行身份验证期间，客户端会向服务器发送用户名和密码，服务器则会根据它的密码存储来验证密码。Kafka 内置了 SASL/PLAIN 支持，我们可以通过一个自定义回调与外部密码数据库集成。

配置 SASL/PLAIN。SASL/PLAIN 的默认实现会将 broker 的 JAAS 配置作为密码存储。所有的客户端用户名和密码都包含在登录选项中，broker 会验证客户端提供的密码是否与其中的一个条目相匹配。如果 broker 间通信启用了 SASL/PLAIN，那么还需要提供 broker 的用户名和密码。

```
sasl.enabled.mechanisms=PLAIN
sasl.mechanism.inter.broker.protocol=PLAIN
listener.name.external.plain.sasl.jaas.config=\
  org.apache.kafka.common.security.plain.PlainLoginModule required \
    username="kafka" password="kafka-password" \ ❶
    user_kafka="kafka-password" \
    user_Alice="Alice-password"; ❷
```

❶ broker 间连接使用的用户名和密码。

❷ 当 Alice 的客户端开始连接 broker 时，Alice 提供的密码将与 broker 配置的密码进行对比。

客户端也需要配置用户名和密码。

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

在 JAAS 配置文件中保存密码既不安全也不灵活，因为在添加或删除用户后需要重启所有的 broker。如果是在生产环境中使用 SASL/PLAIN，那么可以使用自定义服务器回调将 broker 与安全的第三方密码服务器集成在一起。也可以使用自定义回调实现密码轮换。服务器端的回调需要在轮换密码的重叠时间段内同时支持旧密码和新密码，直到所有客户端都切换到新密码。下面是一个使用自定义回调的例子，我们对 `htpasswd` 生成的密码进行了验证。

```
public class PasswordVerifier extends PlainServerCallbackHandler {

  private final List<String> passwdFiles = new ArrayList<>(); ❶

  @Override
  public void configure(Map<String, ?> configs, String mechanism,
                        List<AppConfigurationEntry> jaasEntries) {
    Map<String,?> loginOptions = jaasEntries.get(0).getOptions();
    String files = (String) loginOptions.get("password.files"); ❷
    Collections.addAll(passwdFiles, files.split(","));
  }

  @Override
  protected boolean authenticate(String user, char[] password) {
    return passwdFiles.stream()                                  ❸
       .anyMatch(file -> authenticate(file, user, password));
  }

  private boolean authenticate(String file, String user, char[] password) {
    try {
      String cmd = String.format("htpasswd -vb %s %s %s",      ❹
              file, user, new String(password));
      return Runtime.getRuntime().exec(cmd).waitFor() == 0;
    } catch (Exception e) {
      return false;
    }
  }
}
```

❶ 使用了多个密码文件，以便支持密码轮换。

❷ 将密码文件的路径作为 broker 的 JAAS 配置选项传进去。也可以使用自定义的 broker 配置选项。

❸ 检查密码是否与文件中的密码匹配，并允许在一段时间内同时使用旧密码和新密码。

❹ 为简单起见，这里使用了 `htpasswd`。可以在生产环境中使用更安全的密码数据库。

broker 配置了密码验证回调处理器和其他选项。

```
listener.name.external.plain.sasl.jaas.config=\
    org.apache.kafka.common.security.plain.PlainLoginModule required \
    password.files="/path/to/htpassword.props,/path/to/oldhtpassword.props";
listener.name.external.plain.sasl.server.callback.handler.class=\
    com.example.PasswordVerifier
```

在客户端，一个实现了 `org.apache.kafka.common.security.auth.AuthenticateCallbackHandler` 的客户端回调处理器会在建立连接时动态加载密码，而不是在启动时从 JAAS 配置中加载静态密码。为了提高安全性，可以将密码保存在加密的文件或外部的安全服务器中。下面的例子使用配置类从文件中动态加载密码。

```
@Override
public void handle(Callback[] callbacks) throws IOException {
  Properties props = Utils.loadProps(passwdFile);❶
  PasswordConfig config = new PasswordConfig(props);
  String user = config.getString("username");
  String password = config.getPassword("password").value();❷
  for (Callback callback: callbacks) {
    if (callback instanceof NameCallback)
```

```
      ((NameCallback) callback).setName(user);
    else if (callback instanceof PasswordCallback) {
      ((PasswordCallback) callback).setPassword(password.toCharArray());
    }
  }
}

private static class PasswordConfig extends AbstractConfig {
  static ConfigDef CONFIG = new ConfigDef()
    .define("username", STRING, HIGH, "User name")
    .define("password", PASSWORD, HIGH, "User password");❸
  PasswordConfig(Properties props) {
    super(CONFIG, props, false);
  }
}
```

❶ 在回调中加载配置文件，确保使用最新的密码来进行密码轮换。

❷ 即使密码被存储在外部，底层的配置库也会返回实际的密码。

❸ 将密码定义成 `PASSWORD` 类型，确保密码不会被包含在日志中。

客户端和为 broker 间通信启用了 SASL/PLAIN 的 broker 都可以配置客户端回调。

```
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule \
  required file="/path/to/credentials.props";
sasl.client.callback.handler.class=com.example.PasswordProvider
```

安全方面的考虑。由于 SASL/PLAIN 会通过网络传输明文密码，因此 PLAIN 机制应该与 SASL_SSL 一起使用，以此来提供安全传输层。在 broker 和客户端的 JAAS 配置文件中以明文形式保存密码是不安全的，所以要考虑将这些密码加密或外部化到安全的密码存储中。不建议使用内置的密码存储（将所有客户端密码保存在 broker 的 JAAS 配置文件中），相反，应该使用安全的外部密码服务器，它们不仅可以安全地保存密码，而且有强大的密码保护策略。

明文密码

要避免在配置文件中使用明文密码，即使可以通过文件系统权限来保护这些文件。可以考虑将密码外部化或加密，确保不会意外泄露。本章将在后面部分介绍 Kafka 的密码保护特性。

03. SASL/SCRAM

RFC-5802 中引入了一种安全的用户名和密码身份验证机制，解决了普通密码身份验证机制（如 SASL/PLAIN）的安全问题。加盐质询响应认证机制（salted challenge response authentication mechanism，SCRAM）不传输明文密码，而是以一种无法被客户端冒充的格式保存密码。密码会与一些随机数组合在一起（也就是所谓的加盐），并对其应用单向加密哈希函数。Kafka 提供了一个内置的 SCRAM 实现，可以与安全的 ZooKeeper 部署在一起，不需要额外的密码服务器。Kafka 的 SCRAM 机制支持 SCRAM-SHA-256 和 SCRAM-SHA-512。

配置 SASL/SCRAM。可以在启动 ZooKeeper 之后创建初始用户，然后再启动 broker。broker 会在启动期间将 SCRAM 用户元数据加载到内存中，确保所有用户（包括用于 broker 间通信的 broker 用户）都能成功地进行身份验证。我们可以随时添加或删除用户。broker 会使用基于 ZooKeeper 监听器的通知机制来更新缓存。下面的这个例子创建了一个用户，主体是 `User:Alice`，密码是 `Alice-password`，使用的机制是 SCRAM-SHA-512。

```
$ bin/kafka-configs.sh --zookeeper localhost:2181 --alter --add-config \
  'SCRAM-SHA-512=[iterations=8192,password=Alice-password]' \
  --entity-type users --entity-name Alice
```

一个监听器可以配置一个或多个 SCRAM 机制。只有当监听器用于 broker 间通信时，才需要用到 broker 的用户名和密码。

```
sasl.enabled.mechanisms=SCRAM-SHA-512
sasl.mechanism.inter.broker.protocol=SCRAM-SHA-512
listener.name.external.scram-sha-512.sasl.jaas.config=\
  org.apache.kafka.common.security.scram.ScramLoginModule required \
    username="kafka" password="kafka-password"; ❶
```

❶ broker 间连接需要用到用户名和密码。

客户端必须配置一个在 broker 端启用的 SASL 机制，而且客户端 JAAS 配置中必须包含用户名和密码。

```
sasl.mechanism=SCRAM-SHA-512
sasl.jaas.config=org.apache.kafka.common.security.scram.ScramLoginModule \
  required username="Alice" password="Alice-password";
```

可以用配置管理工具的 `--add-config` 选项添加新 SCRAM 用户，用 `--delete-config` 选项删除用户。当现有用户被删除时，我们无法为这个用户建立新连接，但这个用户的现有连接可以继续使用。可以为 broker 配置一个重新认证时间间隔，用于限制用户被删除后可以继续使用现有连接的时间。下面的例子中删除了 Alice 的 SCRAM-SHA-512 机制的凭证。

```
$ bin/kafka-configs.sh --zookeeper localhost:2181 --alter --delete-config \
  'SCRAM-SHA-512' --entity-type users --entity-name Alice
```

安全方面的考虑。SCRAM 在密码中加入了随机数，并对其应用了单向加密哈希函数，以避免通过网络传输或在数据库中保存真实的密码。然而，任何基于密码的系统的安全性仅取决于密码本身。必须采用强密码策略，保护系统免受暴力或字典攻击。Kafka 只支持强哈希算法 SHA-256 和 SHA-512，不采用较弱的 SHA-1，从而提供了安全保障。这与默认的 4096 迭代次数和密钥随机盐相结合，限制了当 ZooKeeper 安全遭破坏时给 Kafka 造成的影响。

我们还是要为握手过程中传输的密钥和保存在 ZooKeeper 中的密钥提供额外的保护，以防止暴力破解攻击。SCRAM 必须与 SASL_SSL 一起使用，避免窃听者在身份验证过程中获取哈希密钥。ZooKeeper 也必须启用 SSL，同时 ZooKeeper 数据必须进行磁盘加密，确保存储的密钥即使在遭受攻击时也不会被获取到。如果没有与安全的 ZooKeeper 部署在一起，则可以使用 SCRAM 回调，并与安全的外部凭证存储系统集成在一起。

04. SASL/OAUTHBEARER

OAuth 是一个用于限制应用程序访问 HTTP 服务的授权框架。RFC-7628 定义了 OAUTHBEARER SASL 机制，可以用通过 OAuth 2.0 获取的凭证访问受保护的非 HTTP 协议资源。OAUTHBEARER 使用的是寿命较短且资源访问受限的 OAuth 2.0 不记名令牌，避免了在长期密码机制中存在的安全漏洞。Kafka 支持 SASL/OAUTHBEARER 客户端身份验证，并可以与第三方 OAuth 服务器集成。OAUTHBEARER 的内置实现使用了不安全的 JSON Web Token（JWT），不适合被用在生产环境中。可以通过自定义回调将其与标准的 OAuth 服务器集成，以便在生产环境中使用 OAUTHBEARER 机制提供安全的身份验证。

配置 SASL/OAUTHBEARER。Kafka 内置的 SASL/OAUTHBEARER 实现不验证令牌，因此只需要在 JAAS 配置中指定登录模块。如果监听器被用于 broker 间通信，则还需要提供 broker 在发起客户端连接时需要用到的令牌的相关信息。`unsecuredLoginStringClaim_sub` 指定了默认的主体声明。

```
sasl.enabled.mechanisms=OAUTHBEARER
sasl.mechanism.inter.broker.protocol=OAUTHBEARER
listener.name.external.oauthbearer.sasl.jaas.config=\
```

```
  org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule \
    required unsecuredLoginStringClaim_sub="kafka"; ❶
```

❶ broker 间连接使用的令牌的主体声明。

客户端必须配置主体声明选项 `unsecuredLoginStringClaim_sub`。也可以配置其他声明和令牌生命周期。

```
sasl.mechanism=OAUTHBEARER
sasl.jaas.config=\
  org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule \
    required unsecuredLoginStringClaim_sub="Alice"; ❶
```

❶ `User:Alice` 是默认的 `KafkaPrincipal`。

为了将 Kafka 与第三方 OAuth 服务器集成，以便在生产环境中使用不记名令牌，Kafka 客户端必须配置 `sasl.login.callback.handler.class`，通过长期密码或刷新令牌从 OAuth 服务器获取令牌。如果 broker 间通信启用了 OAUTHBEARER，则 broker 也需要配置一个登录回调处理器，用于获取 broker 在发起客户端连接时需要用到的令牌。

```
@Override
public void handle(Callback[] callbacks) throws UnsupportedCallbackException {
  OAuthBearerToken token = null;
  for (Callback callback : callbacks) {
    if (callback instanceof OAuthBearerTokenCallback) {
      token = acquireToken(); ❶
      ((OAuthBearerTokenCallback) callback).token(token);
    } else if (callback instanceof SaslExtensionsCallback) { ❷
      ((SaslExtensionsCallback) callback).extensions(processExtensions(token));
    } else
      throw new UnsupportedCallbackException(callback);
  }
}
```

❶ 客户端必须从 OAuth 服务器获取一个令牌，并将其传给回调。

❷ 客户端可能还用了其他回调。

broker 还必须配置 `listener.name.<listener-name>.oauthbearer.sasl.server.callback.handler.class`，用于验证客户端提供的令牌。

```
@Override
public void handle(Callback[] callbacks) throws UnsupportedCallbackException {
  for (Callback callback : callbacks) {
    if (callback instanceof OAuthBearerValidatorCallback) {
      OAuthBearerValidatorCallback cb = (OAuthBearerValidatorCallback) callback;
      try {
        cb.token(validatedToken(cb.tokenValue()));   ❶
      } catch (OAuthBearerIllegalTokenException e) {
        OAuthBearerValidationResult r = e.reason();
        cb.error(errorStatus(r), r.failureScope(), r.failureOpenIdConfig());
      }
    } else if (callback instanceof OAuthBearerExtensionsValidatorCallback) {
      OAuthBearerExtensionsValidatorCallback ecb =
          (OAuthBearerExtensionsValidatorCallback) callback;
      ecb.inputExtensions().map().forEach((k, v) ->
        ecb.valid(validateExtension(k, v)));   ❷
    } else {
      throw new UnsupportedCallbackException(callback);
    }
  }
}
```

❶ `OAuthBearerValidatorCallback` 包含来自客户端的令牌，broker 会验证这个令牌。

❷ broker 会验证来自客户端的可选扩展。

安全方面的考虑。由于 SASL/OAUTHBEARER 客户端通过网络发送 OAuth 2.0 不记名令牌，这些令牌可能会被用于冒充客户端，因此必须启用 TLS 来加密身份验证流量。如果令牌遭到破坏，那么可以使用短期令牌来限制泄露范围。可以开启 broker 的重新认证，防止连接的存活时间超过令牌有效期。为 broker 配置重新认证时间间隔，并与可撤销令牌相结合，有效限制了已有连接被撤销后可以继续使用令牌的时间。

05. 委托令牌

委托令牌是一种在 broker 和客户端之间共享的密钥，它提供了一种轻量级的配置方式，无须向客户端应用程序分发 SSL 密钥存储或 Kerberos 密钥表。可以用委托令牌来减少身份验证服务器（如 Kerberos 密钥分发中心，KDC）的负载。一些框架（如 Kafka Connect）可以用委托令牌来简化 worker 的安全配置。一个已经通过 broker 身份验证的客户端可以为同一个用户主体创建委托令牌，并将它们分发给 worker，worker 可以直接用这些令牌通过 broker 的身份验证。每个委托令牌由令牌标识符和作为共享秘密使用的哈希消息验证码（HMAC）组成。基于委托令牌的客户端身份验证使用的是 SASL/SCRAM 机制，令牌标识符就是用户名，HMAC 就是密码。

可以使用 Kafka Admin API 或 `delegation-tokens` 命令创建或更新委托令牌。要为主体 `User:Alice` 创建委托令牌，必须先用 Alice 的凭据对客户端进行身份验证（除委托令牌之外的任何一种身份验证协议）。使用委托令牌进行身份验证的客户端不能创建其他委托令牌。

```
$ bin/kafka-delegation-tokens.sh --bootstrap-server localhost:9092 \
  --command-config admin.props --create --max-life-time-period -1 \
  --renewer-principal User:Bob  ❶
$ bin/kafka-delegation-tokens.sh --bootstrap-server localhost:9092 \ ❷
  --command-config admin.props --renew --renew-time-period -1 --hmac c2VjcmV0
```

❶ 如果 Alice 运行了这个命令，那么生成的令牌就可以用来冒充 Alice。这个令牌的所有者是 `User:Alice`。我们将 `User:Bob` 配置为令牌的续订者。

❷ 更新命令可以由令牌所有者（Alice）或令牌续订者（Bob）来执行。

配置委托令牌。要创建和验证委托令牌，所有 broker 都必须使用 `delegation.token.master.key` 参数配置相同的主密钥。这个主密钥只能通过重启所有 broker 进行轮换。所有已有的令牌都必须在更新主密钥之前删除，因为它们不会再被使用，应该在所有 broker 都更新了密钥之后再创建新令牌。

要用委托令牌进行身份验证，broker 至少要启用一种 SASL/SCRAM 机制。客户端也需要配置 SCRAM 机制，将令牌标识符作为用户名，HMAC 作为密码。`KafkaPrincipal` 就是与令牌相关联的原始主体，比如 `User:Alice`。

```
sasl.mechanism=SCRAM-SHA-512
sasl.jaas.config=org.apache.kafka.common.security.scram.ScramLoginModule \
  required tokenauth="true" username="MTIz" password="c2VjcmV0"; ❶
```

❶ 用带有 `tokenauth` 的 SCRAM 机制配置委托令牌。

安全方面的考虑。与内置的 SCRAM 实现一样，委托令牌只适合用于有安全 ZooKeeper 的环境中。SCRAM 的所有安全注意事项也都适用于委托令牌。

用于生成令牌的主密钥必须通过加密或外部化到安全密码存储的方式进行保护。如果令牌遭到破坏，则可以用短期委托令牌来限制泄露范围。可以开启 broker 的重新认证，防止连接使用过期的令牌，并

限制已有连接在令牌被删除后继续存在的时间。

11.3.3 重新认证

当客户端向 broker 发起连接时，broker 会对客户端进行身份验证。broker 会验证客户端凭证，如果凭证有效，验证就通过。一些安全机制（比如 Kerberos 和 OAuth）使用的凭证的生存期是有限的。Kafka 的一个后台登录线程负责在旧凭证过期之前获取新凭证，但在默认情况下，新凭证只用于验证新连接。使用旧凭证验证的旧连接将继续处理请求，直到因为某些原因（比如请求超时、空闲超时或网络错误）断开连接。在用于验证连接的凭证过期之后，长寿命连接可能会继续处理请求。可以通过配置 `connections.max.reauth.ms` 参数让 broker 进行重新认证。如果这个参数的值被设置为正整数，那么 broker 就会在进行 SASL 握手期间将 SASL 连接的会话生存期告诉客户端。会话生存期是凭证剩余生存期和 `connections.max.reauth.ms` 二者当中的较小者。任何在这个间隔内没有进行重新认证的连接都将被 broker 终止。客户端可以使用后台登录线程获取的最新凭证或自定义回调注入的最新凭据来执行重新认证。重新认证可在以下几种场景中加强安全性。

- 对使用了有限生存期凭证的 SASL 机制（比如 GSSAPI 和 OAUTHBEARER）来说，重新认证可以保证所有活动连接都与有效凭证相关联。短期凭证可以在遭到破坏时限制泄露范围。
- 基于密码的 SASL 机制（比如 PLAIN 和 SCRAM）可以通过定期登录来实现密码轮换。重新认证可以限制使用旧密码进行身份验证的连接在密码轮换后继续处理请求的时间。自定义服务器回调允许在一段时间内同时使用新密码和旧密码，这样就可以避免在所有客户端迁移到新密码之前造成停机。
- 设置 `connections.max.reauth.ms` 参数，强制所有 SASL 机制进行重新认证，包括那些凭证未过期的机制。这样就可以限制活动连接在凭证被撤销后继续存在的时间。
- 不支持 SASL 重新认证的客户端连接会在会话过期时被终止，客户端必须重新连接并进行身份验证，从而提供与凭证过期或凭证被撤销时同样的安全保证。

被泄露的用户身份

如果用户身份被泄露，则必须尽快将其从系统中移除。一旦用户被从认证服务器上移除，该用户所有的新连接都将无法通过 broker 的身份验证。已有连接将继续处理请求，直到执行下一次重新认证。如果没有配置 `connections.max.reauth.ms`，就不会进行重新认证，已有连接可能会在很长一段时间内继续使用被泄露的用户身份。Kafka 不支持 SSL 重新协商，因为旧 SSL 协议的重新协商过程存在已知的漏洞。较新的协议（如 TLSv1.3）不支持重新协商。因此，已有 SSL 连接可以继续使用已撤销或过期的证书。可以用用户主体的 `DenyACL` 来阻止这些连接继续执行操作。因为 ACL 变更可以很快被应用于所有 broker，所以这是禁用被泄露用户访问权限最快的方法。

11.3.4 安全更新不停机

Kafka 需要定期轮换密钥、应用安全补丁以及更新到最新的安全协议。大部分维护任务是通过滚动更新的方式进行的，也就是用新的配置逐个重启 broker，而像更新 SSL 密钥存储和信任存储这类任务则可以通过动态配置更新来执行，无须重启 broker。

在向已有集群添加新的安全协议时，可以在保留旧监听器的同时加入新监听器，确保在更新安全协议期间客户端可以继续使用旧监听器。例如，可以按照下面的顺序将已有集群的 PLAINTEXT 切换到 SASL_SSL。

01. 使用 Kafka 配置工具为每一个 broker 的新端口添加一个新监听器。使用单独的配置更新命令更新 `listeners` 和 `advertised.listeners`，让它们同时包含旧监听器和新监听器，并用监听器前缀为新的 SASL_SSL 监听器提供所有的配置参数。
02. 修改所有的客户端应用程序，让它们使用新的 SASL_SSL 监听器。
03. 如果 broker 间通信也使用了新的 SASL_SSL 监听器，则需要用新的 `inter.broker.listener.name` 滚动更新所有 broker。
04. 使用配置工具删除 `listeners` 和 `advertised.listeners` 中的旧监听器，并删除未被使用的旧监听器配置参数。

在向已有 SASL 监听器添加 SASL 机制或从已有 SASL 监听器中删除 SASL 机制时，可以在同一个监听器端口上进行滚动更新，以此来避免停机。可以按照下面的顺序将 PLAIN 切换到 SCRAM-SHA-256。

01. 使用 Kafka 配置工具将所有已有用户添加到 SCRAM 存储中。
02. 设置 `sasl.enabled.mechanisms=PLAIN,SCRAM-SHA-256`，并为监听器配置 `listener.name.<_listener-name_>.scram-sha-256.sasl.jaas.config`，然后进行 broker 滚动更新。
03. 修改所有的客户端应用程序，设置 `sasl.mechanism=SCRAM-SHA-256`，并使用 SCRAM 更新 `sasl.jaas.config`。
04. 如果监听器被用于 broker 间通信，则需要设置 `sasl.mechanism.inter.broker.protocol=SCRAM-SHA-256`，并进行 broker 滚动更新。
05. 移除 PLAIN 机制，再进行一次 broker 滚动更新。设置 `sasl.enabled.mechanisms=SCRAM-SHA-256`，并移除 `listener.name.<listener-name>.plain.sasl.jaas.config` 以及与 PLAIN 相关的配置参数。

11.4 加密

加密被用于保护数据的隐私和完整性。启用了 SSL 和 SASL_SSL 安全协议的 Kafka 监听器将 TLS 作为传输层，提供了安全的加密通道，以用来保护在不安全网络上传输的数据。可以对 TLS 加密套件进行限制，以此来加强安全性，并达到一些安全标准[如联邦信息处理标准（Federal Information Processing Standard，FIPS）] 所要求的安全需求水平。

还需要采取额外的措施来保护静态数据，确保有 Kafka 日志存储磁盘物理访问权限的用户也无法检索敏感数据。物理磁盘可以采用全磁盘加密或数据卷加密，即使磁盘被盗，也可以避免数据泄露。

虽然传输层和数据存储加密已经提供了足够的保护，但仍然需要提供额外的保护措施，避免将自动数据访问权限授予平台管理员。broker 内存中未加密的数据可能会出现在堆转储中，有磁盘访问权限的管理员可以直接访问这些数据和包含潜在敏感数据的 Kafka 日志。如果集群中有高度敏感的数据或 PII，则需要采取额外措施来保护数据隐私。为了符合监管合规性，特别是在云端，需要确保平台管理员或云供应商不能以任何方式访问机密数据。可以将自定义加密提供程序内嵌到 Kafka 客户端中，以此来实现端到端加密，保证整个数据流都是加密的。

端到端加密

在第 3 章介绍 Kafka 生产者时，我们使用**序列化器**将消息转换成了存储在 Kafka 日志中的字节数组。在第 4 章介绍 Kafka 消费者时，我们使用**反序列化器**将字节数组转换成了消息。可以将序列化器和反序列化器与加密库集成，在序列化期间进行消息加密，在反序列化期间进行消息解密。消息加密通常使用对称加密算法，比如 AES。生产者可以用保存在密钥管理系统（KMS）中的共享加密密钥加密消息，消费者可以用它解密消息。broker 不需要访问加密密钥，并且永远看不到消息里未加密的内容，因此在云端使用这种方法是安全的。解密消息所需的加密参数可以保存在消息标头或消息体（针对不支持消息标头的旧客户端）中。还可以在消息标头中包含数字签名，用于验证消息的完整性。

图 11-2 演示了一个端到端加密的 Kafka 数据流。

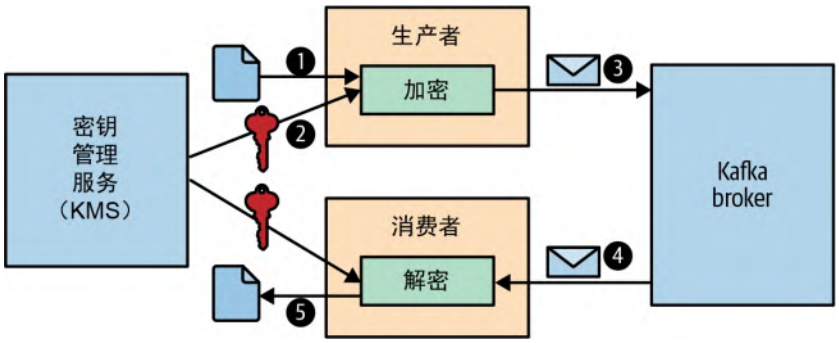

图 11-2：端到端加密

01. 使用 Kafka 生产者发送了一条消息。
02. 生产者会使用来自 KMS 的加密密钥加密消息。
03. 加密的消息被发送给 broker。broker 会将加密的消息保存在分区日志中。
04. broker 会将加密的消息发送给消费者。
05. 消费者会使用来自 KMS 的加密密钥解密消息。

生产者和消费者必须配置凭证才能从 KMS 获取共享密钥。建议定期轮换密钥，以此来加强安全性，因为频繁轮换密钥可以限制被泄露的消息的数量，还可以防止暴力破解攻击。如果使用旧密钥加密的消息仍然符合保留策略，那么新旧密钥都必须可用。有很多 KMS 系统支持优雅的对称加密密钥轮换，无须在 Kafka 客户端做任何特殊处理。对于压实型主题，使用旧密钥加密的消息可能会被保留很长时间，而且可能需要对旧消息进行重新加密。为了避免新消息对其造成干扰，生产者和消费者在这个过程中需要离线。

压缩加密的消息

与加密前压缩消息相比，加密后压缩消息并不会在存储空间方面提供任何优势。可以通过配置让序列化器在加密消息之前压缩消息，或者让应用程序在生产消息之前压缩消息。不管是哪一种情况，最好都禁用 Kafka 的压缩，因为它们增加了开销却不会带来任何额外的好处。对于在不安全传输层上传输的消息，还需要考虑已压缩的加密消息的安全传输问题。

在很多环境中，特别是在使用 TLS 作为传输层时，消息键不需要加密，因为它们通常不会像消息体那样包含敏感数据。但在某些情况下，明文的消息键可能不符合监管要求。由于在分区和压实时会用到消息的键，因此在对键进行转换时必须保留等价哈希，确保即使加密参数发生变化，键的哈希值也不会变。一种方法是将原始消息键的安全哈希作为消息键，并将加密的消息键保存在消息体或消息标头中。因为消息键和消息体的序列化是独立进行的，所以可以用生产者拦截器来执行这个转换。

11.5 授权

授权是决定你可以对哪些资源执行哪些操作的过程。broker 会使用一个可定制的授权器进行访问控制管理。前面讲过，每当客户端向 broker 发起连接时，broker 会对客户端进行身份验证，并将代表客户端身份的 KafkaPrincipal 与连接关联起来。在处理请求时，broker 会验证与当前连接关联的主体是否被授权执行这个请求。例如，当 Alice 的生产者试图向主题 customerOrders 写入一个新的客户订单记录时，broker 会验证 User:Alice 是否有写入这个主题的权限。

Kafka 提供了一个内置的授权器 AclAuthorizer，我们可以通过下面的配置来启用它。

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

SimpleAclAuthorizer

AclAuthorizer 是在 Kafka 2.3 中引入的。从 Kafka 0.9.0.0 开始的旧版本中有一个内置的授权器 kafka.security.auth.SimpleAclAuthorizer，虽然该授权器已被弃用，但仍受支持。

11.5.1 AclAuthorizer

AclAuthorizer 支持使用访问控制列表（ACL）对 Kafka 资源进行细粒度的访问控制。ACL 保存在 ZooKeeper 中，broker 也会将它们缓存在内存中，以便在授权请求时提供高性能的查找。broker 在启动时会将 ACL 加载到缓存中，并通过 ZooKeeper 的 watcher 通知机制来更新缓存。在授权请求时，broker 会验证与当前连接关联的 KafkaPrincipal 是否有权限操作被请求的资源。

每个 ACL 由以下几部分组成。

- 资源类型：Cluster|Topic|Group|TransactionalId|DelegationToken。
- 模式类型：Literal|Prefixed。
- 资源名称：资源的名称、前缀或通配符 *。
- 操作：Describe|Create|Delete|Alter|Read|Write|DescribeConfigs|AlterConfigs。
- 权限类型：Allow|Deny，其中 Deny 的优先级更高。
- 主体：主体可以表示为 < 主体类型 >:< 主体名称 >，例如，User:Bob 或 Group:Sales。如果要授予所有用户访问权限，那么可以使用 User:*。
- 主机：客户端连接的源 IP 地址。如果要为所有主机授权，那么可以使用 *。

例如，一个 ACL 可能会指定如下内容。

```
User:Alice has Allow permission for Write to Prefixed Topic:customer from
192.168.0.1
```

如果一个动作没有匹配的 DenyACL，并且至少有一个匹配的 AllowACL，那么 AclAuthorizer 将对这个动作进行授权。如果授予了 Read 权限、Write 权限、Alter 权限或 Delete 权限，那么也会隐含授予 Describe 权限；如果授予了 AlterConfigs 权限，那么也会隐含授予 DescribeConfigs 权限。

通配符 ACL

模式类型为 Literal 和资源名称为 * 的 ACL 将作为通配符，它将匹配某个资源类型的所有资源名称。

broker 必须被授予 Cluster:ClusterAction 权限才能对控制器请求和获取请求进行授权。生产者必须被授予 Topic:Write 权限才能向主题写入数据。没有启用事务的幂等生产者必须被授予 Cluster:IdempotentWrite 权限。事务性生产者必须被授予 TransactionalId:Write 权限和 Group:Read 权限，这样才能访问事务 ID 和提交消费者群组的偏移量。消费者必须被授予 Topic:Read 权限才能从主题读取数据，如果启用了群组管理或偏移管理，还需要被授予 Group:Read 权限。管理操作需要被授予 Create 权限、Delete 权限、Describe 权限、Alter 权限、DescribeConfigs 权限或 AlterConfigs 权限。表 11-1 列出了每一个 ACL 可以被应用在哪些请求上。

表 11-1：每一个 Kafka ACL 被授予的权限

ACL	Kafka 请求类型	说明
Cluster:ClusterAction	broker 间请求，包括控制器请求和来自跟随者的获取请求	只授权给 broker
Cluster:Create	CreateTopics 和自动创建主题	可以用 Topic:Create 授权给特定的创建主题请求
Cluster:Alter	CreateAcls、DeleteAcls、AlterReplicaLogDirs、ElectReplicaLeader、AlterPartitionReassignments	
Cluster:AlterConfigs	针对 broker 和 broker 日志的 AlterConfigs 和 IncrementalAlterConfigs、AlterClientQuotas	
Cluster:Describe	DescribeAcls、DescribeLogDirs、ListGroups、ListPartitionReassignments、获取 broker 已授权操作的元数据请求	可以用 Group:Describ 授权给 ListGroups
Cluster:DescribeConfigs	针对 broker 和 broker 日志的 DescribeConfigs、DescribeClientQuotas	
Cluster:IdempotentWrite	幂等的 InitProducerId 请求和 Produce 请求	只有非事务性的幂等生产者需要这个权限
Topic:Create	CreateTopics 和自动创建主题	
Topic:Delete	DeleteTopics、DeleteRecords	
Topic:Alter	CreatePartitions	
Topic:AlterConfigs	AlterConfigs 和针对主题的 IncrementalAlterConfigs	
Topic:Describe	主题元数据请求、OffsetForLeaderEpoch、ListOffset、OffsetFetch	
Topic:DescribeConfigs	针对主题的 DescribeConfigs 和在 CreateTopics 响应中返回配置信息	
Topic:Read	Consumer Fetch、OffsetCommit、TxnOffsetCommit、OffsetDelete	应该授权给消费者

ACL	Kafka 请求类型	说明
`Topic:Write`	`Produce`、`AddPartitionToTxn`	应该授权给生产者
`Group:Read`	`JoinGroup`、`SyncGroup`、`LeaveGroup`、`Heartbeat`、`OffsetCommit`、`AddOffsetsToTxn`、`TxnOffsetCommit`	使用消费者群组管理或偏移量管理的消费者以及事务性生产者需要这个权限
`Group:Describe`	`FindCoordinator`、`DescribeGroup`、`ListGroups`、`OffsetFetch`	
`Group:Delete`	`DeleteGroups`、`OffsetDelete`	
`TransactionalId:Write`	带有事务的 `Produce` 和 `InitProducerId`、`AddPartitionToTxn`、`AddOffsetsToTxn`、`TxnOffsetCommit`、`EndTxn`	事务性生产者需要这个权限
`TransactionalId:Describe`	事务协调器的 `FindCoordinator`	
`DelegationToken:Describe`	`DescribeTokens`	

Kafka 提供了一个使用 broker 中的授权器管理 ACL 的工具。也可以直接在 ZooKeeper 中创建 ACL，如果需要在启动 broker 之前创建 ACL，那么这就非常有用。

```
$ bin/kafka-acls.sh --add --cluster --operation ClusterAction \
  --authorizer-properties zookeeper.connect=localhost:2181    \ ❶
  --allow-principal User:kafka
$ bin/kafka-acls.sh --bootstrap-server localhost:9092       \
  --command-config admin.props --add --topic customerOrders \ ❷
  --producer --allow-principal User:Alice
$ bin/kafka-acls.sh --bootstrap-server localhost:9092       \
  --command-config admin.props --add --resource-pattern-type PREFIXED \ ❸
  --topic customer --operation Read --allow-principal User:Bob
```

❶ 直接在 ZooKeeper 中创建 broker 用户的 ACL。

❷ 在默认情况下，ACL 命令会授予字面量 ACL。`User:Alice` 会被授予写入 `customerOrders` 主题的权限。

❸ 带前缀的 ACL 会授予 Bob 读取所有以 `customer` 开头的主题的权限。

`AclAuthorizer` 提供了两个配置选项，可用于对资源或主体进行大范围的授权，以此来简化 ACL 管理，特别是在第一次为已有集群添加授权时。

```
super.users=User:Carol;User:Admin
allow.everyone.if.no.acl.found=true
```

超级用户可以访问所有资源，不受任何限制，而且不能用 `Deny`ACL 禁止超级用户的权限。如果 Carol 的凭证被泄露，则必须将 Carol 从 `super.users` 中移除，然后重启 broker，让变更生效。在生产环境中，通过 ACL 为用户授予特定权限会更为安全，因为在必要时可以轻松将其撤销。

超级用户分隔符

与 Kafka 中其他使用逗号分隔的配置参数不同，`super.users` 使用分号来分隔多个用户，因为用户主体（比如 SSL 证书的可识别名称）通常会包含逗号。

如果 `allow.everyone.if.no.acl.found` 被设置为 `true`，那么所有用户都可以访问没有配置 ACL 的资源。如果是第一次在集群中启用授权或在开发过程中启用授权，那么这个可能很有用，但不适合用在生产环境中，因为这样有可能在无意中授予了访问新资源的权限。而且，如果添加了与前缀或通配符匹配的 ACL，则不再满足 `no.acl.found` 条件，访问权限有可能被意外移除。

11.5.2 自定义授权

在 Kafka 中，可以自定义授权，这样就可以实现额外的控制或增加新的访问控制类型，比如基于角色的访问控制。

下面的自定义授权器限制了只有内部监听器可以处理某些请求。为简单起见，请求和监听器名称都是硬编码的。当然，如果想灵活一点儿，则可以通过自定义授权器的属性来配置它们。

```
public class CustomAuthorizer extends AclAuthorizer {
  private static final Set<Short> internalOps =
    Utils.mkSet(CREATE_ACLS.id, DELETE_ACLS.id);
  private static final String internalListener = "INTERNAL";

  @Override
  public List<AuthorizationResult> authorize(
          AuthorizableRequestContext context, List<Action> actions) {
    if (!context.listenerName().equals(internalListener) && ❶
        internalOps.contains((short) context.requestType()))
      return Collections.nCopies(actions.size(), DENIED);
    else
      return super.authorize(context, actions); ❷
  }
}
```

❶ 这里为授权器提供了带有元数据的请求上下文，其中包括监听器的名称、安全协议、请求类型等，让授权器能够根据上下文添加或删除授权限制。

❷ 使用公共 API 调用内置的授权器功能。

Kafka 授权器也可以与外部系统集成，实现基于组或角色的访问控制。可以使用不同的主体类型为组主体或角色主体创建 ACL。例如，在下面的示例中，我们定期根据 LDAP 服务器的角色和组计算出 Scala 类中的 `groups` 和 `roles`，用于支持不同级别的 `AllowACL`。

```
class RbacAuthorizer extends AclAuthorizer {

  @volatile private var groups = Map.empty[KafkaPrincipal, Set[KafkaPrincipal]]
    .withDefaultValue(Set.empty) ❶
  @volatile private var roles = Map.empty[KafkaPrincipal, Set[KafkaPrincipal]]
    .withDefaultValue(Set.empty) ❷

  override def authorize(context: AuthorizableRequestContext,
      actions: util.List[Action]): util.List[AuthorizationResult] = {
    val principals = groups(context.principal) + context.principal
    val allPrincipals = principals.flatMap(roles) ++ principals ❸
    val contexts = allPrincipals.map(authorizeContext(context, _))
    actions.asScala.map { action =>
      val authorized = contexts.exists(
        super.authorize(_, List(action).asJava).get(0) == ALLOWED)
      if (authorized) ALLOWED else DENIED                          ❹
    }.asJava
  }
```

```
  private def authorizeContext(context: AuthorizableRequestContext,
      contextPrincipal: KafkaPrincipal): AuthorizableRequestContext = {
    new AuthorizableRequestContext {                                    ❺
      override def principal() = contextPrincipal
      override def clientId() = context.clientId
      override def requestType() = context.requestType
      override def requestVersion() = context.requestVersion
      override def correlationId() = context.correlationId
      override def securityProtocol() = context.securityProtocol
      override def listenerName() = context.listenerName
      override def clientAddress() = context.clientAddress
    }
  }
}
```

❶ 用户所属的组，根据外部源（如 LDAP）计算得出。

❷ 与每个用户关联的角色，根据外部源（如 LDAP）计算得出。

❸ 根据用户以及用户的所有组和角色进行授权。

❹ 如果授权成功就返回 `ALLOWED`。需要注意的是，这个例子不支持组或角色的 `Deny` ACL。

❺ 用与原始上下文相同的元数据为每个主体创建授权上下文。

可以用标准的 Kafka ACL 工具为 `Sales` 组或 `Operator` 角色分配 ACL。

```
$ bin/kafka-acls.sh --bootstrap-server localhost:9092            \
  --command-config admin.props --add --topic customer --producer \
  --resource-pattern-type PREFIXED --allow-principal Group:Sales ❶
$ bin/kafka-acls.sh --bootstrap-server localhost:9092 \
  --command-config admin.props --add --cluster --operation Alter \
  --allow-principal=Role:Operator  ❷
```

❶ 用主体 `Group:Sales` 和自定义主体类型 `Group` 创建一个 ACL，将其应用于属于 `Sales` 组的用户。

❷ 用主体 `Role:Operator` 和自定义主体类型 `Role` 创建一个 ACL，将其应用于具有 `Operator` 角色的用户。

11.5.3 安全方面的考虑

因为 `AclAuthorizer` 将 ACL 保存在 ZooKeeper 中，所以要对 ZooKeeper 的访问进行安全限制。如果集群中没有安全的 ZooKeeper，则可以实现一个自定义授权器，将 ACL 保存在安全的外部数据库中。

在拥有大量用户的大型组织中，管理单个资源的 ACL 可能会非常烦琐。可以为不同的部门预留不同的资源前缀，这样就可以使用前缀 ACL，以此来减少 ACL 的数量。还可以结合使用基于组或角色的 ACL（如前面的例子所示），以进一步简化大型组织的访问控制配置。

当用户受到攻击时，基于最小权限原则来限制用户访问可以降低暴露风险。也就是说，我们只为每个用户主体授予执行其操作所需的资源的访问权限，并在 ACL 使用完毕之后将其移除。当用户主体不再被使用（例如，当一个员工离开组织）时，应该立即移除对应的 ACL。对于长时间运行的应用程序，可以使用服务凭证（而不是与特定用户关联的凭证），以避免员工离开组织时出现任何中断。由于一个用户主体的长寿命连接有可能在用户被删除之后继续处理请求，因此可以用 `Deny` ACL 来确保主体不会因通配符 ACL 而无意中被授予了访问权限。如果可能，要避免重用主体，防止将访问权限授予与旧版本主体关联的连接。

11.6 审计

broker 可以生成用于审计和调试的 log4j 日志。可以在 log4j.properties 文件中指定日志级别、日志记录器及其配置选项。用于授权日志的 `kafka.authorizer.logger` 和用于请求日志的 `kafka.request.logger` 可以单独配置日志级别和保留策略。在生产环境中，可以用一些框架（如 Elastic Stack）来分析和可视化这些日志。

授权器会记录 `INFO` 级别的拒绝访问日志和 `DEBUG` 级别的授权访问日志。

```
DEBUG Principal = User:Alice is Allowed Operation = Write from host = 127.0.0.1
on resource = Topic:LITERAL:customerOrders for request = Produce with resourceRefCount
= 1 (kafka.authorizer.logger)
INFO Principal = User:Mallory is Denied Operation = Describe from host =
10.0.0.13 on resource = Topic:LITERAL:customerOrders for request = Metadata
with resourceRefCount = 1 (kafka.authorizer.logger)
```

`DEBUG` 级别的请求日志中包含了用户主体和客户端主机的相关信息。如果请求日志设置了 `TRACE` 级别，那么日志中也会包含请求细节。

```
DEBUG Completed request:RequestHeader(apiKey=PRODUCE, apiVersion=8,
clientId=producer-1, correlationId=6) --
{acks=-1,timeout=30000,partitionSizes=[customerOrders-0=15514]},response:
{responses=[{topic=customerOrders,partition_responses=[{partition=0,error_code=0
,base_offset=13,log_append_time=-1,log_start_offset=0,record_errors=[],error_mes
sage=null}]}],throttle_time_ms=0} from connection
127.0.0.1:9094-127.0.0.1:61040-0;totalTime:2.42,requestQueueTime:0.112,local-
Time:2.15,remoteTime:0.0,throttleTime:0,responseQueueTime:0.04,sendTime:
0.118,securityProtocol:SASL_SSL,principal:User:Alice,listener:SASL_SSL,clientInf
ormation:ClientInformation(softwareName=apache-kafka-java,
softwareVersion=2.7.0-SNAPSHOT) (kafka.request.logger)
```

可以通过分析授权器和请求日志来检测可疑活动。身份验证失败率指标和授权失败日志对审计来说非常有用，而且在遭到攻击或发生未授权访问时可以为我们提供有价值的信息。为了能够进行端到端的消息审计和追踪，可以在生成消息时将审计元数据放在消息标头中。端到端加密可用于保护元数据的完整性。

11.7 保护 ZooKeeper

用于维护 Kafka 集群可用性的元数据就保存在 ZooKeeper 中，所以，除了保护 Kafka，也很有必要保护 ZooKeeper。ZooKeeper 支持基于 SASL/GSSAPI 的 Kerberos 身份验证和基于 SASL/DIGEST-MD5 的用户名和密码身份验证。ZooKeeper 3.5.0 中加入了对 TLS 的支持，可以进行双向身份验证和传输数据加密。需要注意的是，SASL/DIGEST-MD5 应该与 TLS 加密一起使用，并且不适合用在生产环境中，因为它存在已知的漏洞。

11.7.1 SASL

ZooKeeper 的 SASL 是通过 Java 系统属性 `java.security.auth.login.config` 来配置的。这个属性必须指向 JAAS 配置文件，其中包含了一个登录模块以及与 ZooKeeper 服务器相关的配置参数。broker 必须配置 ZooKeeper 客户端登录模块，以便与启用了 SASL 的 ZooKeeper 服务器通信。下面的 ZooKeeper 服务器端 JAAS 配置启用了 Kerberos 身份验证。

```
Server {
  com.sun.security.auth.module.Krb5LoginModule required
  useKeyTab=true storeKey=true
  keyTab="/path/to/zk.keytab"
  principal="zookeeper/zk1.example.com@EXAMPLE.COM";
};
```

要启用 ZooKeeper 服务器端的 SASL 身份验证，需要在 ZooKeeper 配置文件中配置身份验证提供程序。

```
authProvider.sasl=org.apache.zookeeper.server.auth.SASLAuthenticationProvider
kerberos.removeHostFromPrincipal=true
kerberos.removeRealmFromPrincipal=true
```

broker 主体

在默认情况下，ZooKeeper 用完整的 Kerberos 主体（如 `kafka/broker1.example.com@EXAMPLE.COM`）来识别客户端。如果 ZooKeeper 的身份验证启用了 ACL，那么 ZooKeeper 服务器端需要配置 `kerberos.removeHostFromPrincipal=true` 和 `kerberos.removeRealmFromPrincipal=true`，确保所有的 broker 都有相同的主体。

Kafka broker 需要启用到 ZooKeeper 的 SASL 身份验证，为此需要使用包含客户端凭证的 JAAS 配置文件。

```
Client {
  com.sun.security.auth.module.Krb5LoginModule required
  useKeyTab=true storeKey=true
  keyTab="/path/to/broker1.keytab"
  principal="kafka/broker1.example.com@EXAMPLE.COM";
};
```

11.7.2 SSL

ZooKeeper 的每一个端点都应该启用 SSL，包括使用 SASL 身份验证的那些。与 Kafka 一样，启用客户端身份验证需要配置 SSL，但与 Kafka 不同，ZooKeeper 会用 SASL 和 SSL 两种协议进行客户端身份验证，并将多个主体与连接关联起来。如果任意一个与某个连接关联的主体有访问某个资源的权限，那么 ZooKeeper 就会授予对这个资源的访问权限。

要为 ZooKeeper 服务器配置 SSL，需要提供一个包含服务器主机名或者通配符主机名的密钥存储。如果启用了客户端身份验证，则还需要提供一个用于验证客户端证书的信任存储。

```
secureClientPort=2181
serverCnxnFactory=org.apache.zookeeper.server.NettyServerCnxnFactory
authProvider.x509=org.apache.zookeeper.server.auth.X509AuthenticationProvider
ssl.keyStore.location=/path/to/zk.ks.p12
ssl.keyStore.password=zk-ks-password
ssl.keyStore.type=PKCS12
ssl.trustStore.location=/path/to/zk.ts.p12
ssl.trustStore.password=zk-ts-password
ssl.trustStore.type=PKCS12
```

要为 Kafka 配置 SSL 以便连接到 ZooKeeper，需要提供一个用于验证 ZooKeeper 证书的信任存储。如果启用了客户端身份验证，则还需要提供一个密钥存储。

```
zookeeper.ssl.client.enable=true
zookeeper.clientCnxnSocket=org.apache.zookeeper.ClientCnxnSocketNetty
zookeeper.ssl.keystore.location=/path/to/zkclient.ks.p12
zookeeper.ssl.keystore.password=zkclient-ks-password
zookeeper.ssl.keystore.type=PKCS12
zookeeper.ssl.truststore.location=/path/to/zkclient.ts.p12
zookeeper.ssl.truststore.password=zkclient-ts-password
zookeeper.ssl.truststore.type=PKCS12
```

11.7.3 授权

可以通过设置路径 ACL 来启用 ZooKeeper 节点授权。如果 broker 配置了 `zookeeper.set.acl=true`，就会在创建节点时设置 ZooKeeper 节点 ACL。在默认情况下，元数据节点是可公开访问的，但只有 broker 能修改。如果内部管理员用户需要直接通过 ZooKeeper 修改元数据，那么需要添加额外的 ACL。敏感路径（比如包含 SCRAM 凭证的节点）在默认情况下是不公开的。

11.8 保护平台

前文讨论了如何保护 Kafka 和 ZooKeeper。生产环境的系统所采用的安全威胁模型不仅要涵盖单个组件的安全威胁，还要涵盖整个系统的安全威胁。威胁模型提供了一个系统抽象，用于识别潜在的威胁和相关风险。在评估、记录并基于风险等级确定了威胁优先级之后，必须实施针对每个潜在威胁的缓解策略，以确保整个系统的安全。在评估潜在威胁时，需要考虑外部威胁和内部威胁。对于存储 PII 或其他敏感数据的系统，还必须实施符合监管政策的措施。不过，与威胁模型建模技术相关的内容超出了本章的讨论范围。

除了用安全的身份验证、授权和加密来保护 Kafka 中的数据和 ZooKeeper 中的元数据，还必须采取额外的措施来确保平台的安全性。可以用网络防火墙来保护网络，用加密来保护物理存储，用文件系统权限来保护包含身份验证凭证的密钥存储、信任存储和 Kerberos 密钥表文件。必须对包含安全关键信息（如凭证）的配置文件进行访问限制。由于在配置文件中保存明文密码是不安全的（即使对访问权限进行了限制），因此 Kafka 支持将密码外部化到安全存储中。

保护密码

可以为 broker 和客户端配置提供程序，用于从安全的第三方密码存储库获取密码。也可以将密码加密后保存在配置文件中，并提供用于解密的提供程序。

下面的提供程序使用 gpg 来解密保存在文件中的 broker 或客户端属性。

```
public class GpgProvider implements ConfigProvider {

  @Override
  public void configure(Map<String, ?> configs) {}

  @Override
  public ConfigData get(String path) {
    try {
      String passphrase = System.getenv("PASSPHRASE");  ❶
      String data = Shell.execCommand(                  ❷
        "gpg", "--decrypt", "--passphrase", passphrase, path);
      Properties props = new Properties();
      props.load(new StringReader(data));               ❸
      Map<String, String> map = new HashMap<>();
      for (String name : props.stringPropertyNames())
        map.put(name, props.getProperty(name));
      return new ConfigData(map);
    } catch (IOException e) {
    throw new RuntimeException(e);                      ❹
    }
  }

  @Override
  public ConfigData get(String path, Set<String> keys) { ❺
    ConfigData configData = get(path);
    Map<String, String> data = configData.data().entrySet()
      .stream().filter(e -> keys.contains(e.getKey()))
      .collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue));
    return new ConfigData(data, configData.ttl());
  }

  @Override
  public void close() {}
}
```

❶ 通过环境变量 PASSPHRASE 来提供用于解密密码的密码短语。

❷ 用 gpg 解密配置文件。返回值中包含了全部的解密配置参数。

❸ 将配置数据解析成 Java 属性。

❹ 如果遇到错误，就抛出 RuntimeException 异常。

❺ 调用者可能想要获取指定文件中的部分属性，我们在这里加载所有属性，并返回调用者请求的部分。

你可能还记得，11.3.2 节中使用了标准的 Kafka 配置类来从外部文件加载凭证。现在可以用 gpg 来加密这个文件。

```
gpg --symmetric --output credentials.props.gpg \
  --passphrase "$PASSPHRASE" credentials.props
```

现在在原始属性文件中添加间接配置和提供程序，这样 Kafka 客户端就可以从被加密的文件中加载它们的证书了。

```
username=${gpg:/path/to/credentials.props.gpg:username}
password=${gpg:/path/to/credentials.props.gpg:password}
config.providers=gpg
config.providers.gpg.class=com.example.GpgProvider
```

也可以用 Kafka 配置工具将加密过的敏感 broker 配置信息保存在 ZooKeeper 中。可以在启动 broker 之前执行如下命令，以将加密过的 SSL 密钥存储密码保存到 ZooKeeper 中。必须在每个 broker 的配置文件中配置用于解密的加密密钥。

```
$ bin/kafka-configs.sh --zookeeper localhost:2181 --alter \
  --entity-type brokers --entity-name 0 --add-config      \
  'listener.name.external.ssl.keystore.password=server-kspassword,
password.encoder.secret=encoder-secret'
```

11.9 小结

在过去十年中，随着网络攻击手段变得越来越复杂，数据泄露的频率和规模都在不断增长。除了隔离和解决漏洞需要付出的成本和在应用安全修复补丁之前因数据泄露造成的损失，数据泄露还可能导致监管处罚和对品牌声誉的长期损害。本章探讨了大量可用于保证 Kafka 数据机密性、完整性和可用性的保护措施。

针对本章开头的示例数据流，我们探讨了不同安全方面的可用选项。

客户端的真实性

当 Alice 的客户端向 broker 发起连接时，启用了 SASL 或 SSL 的监听器通过客户端身份验证来验证连接是来自 Alice 而不是冒充者。可以开启重新认证，以便在用户受到攻击时限制暴露范围。

服务器的真实性

Alice 的客户端可以用带有主机名验证的 SSL 或带有双向身份验证的 SASL 机制（比如 Kerberos 或 SCRAM）来验证它连接的是真实的 broker。

数据隐私

可以用 SSL 加密传输中的数据，保护数据不被窃取。也可以通过磁盘或数据卷加密保护静态数据。对于高度敏感的数据，可以通过端到端加密提供细粒度的数据访问控制，确保有网络和磁盘物理访问权限的云供应商和平台管理员也无法访问这些数据。

数据完整性

可以用 SSL 检测发生在不安全网络中的数据篡改。在使用端到端加密时，可以在消息中包含数字签名，用于验证数据完整性。

访问控制

Alice、Bob 以及 broker 执行的每一个操作都可以用一个可定制的授权器进行授权。Kafka 提供了一个内置的授权器，可以使用 ACL 进行细粒度的访问控制。

可审计性

可以通过授权日志和请求日志对操作进行审计和异常检测。

可用性

可以结合使用配额和配置选项来管理连接，保护 broker 免受拒绝服务攻击。还可以用 SSL、SASL 和 ACL 来保护 ZooKeeper，确保与 broker 可用性相关的元数据是安全的。

提升安全性有大量可用的选项，因此，为每一种应用场景选择合适的选项可能是一项艰巨的任务。本章探讨了每一种安全机制需要考虑的问题，以及可用来限制潜在攻击面的措施和策略。还探讨了保护 ZooKeeper 和平台其他组件所需的额外措施。Kafka 对标准安全技术的支持以及与已有安全基础设施集成的各种扩展点，让我们能够构建出一致的安全解决方案来保护整个平台。

第 12 章　管理 Kafka

管理 Kafka 集群意味着需要使用额外的工具对主题、配置等做出修改。Kafka 提供了一些命令行工具，以用于对集群做出变更。这些工具使用 Java 类实现，Kafka 提供了一些脚本来调用它们。不过，这些工具只提供了基本的功能，无法完成复杂的操作，难以被用于管理大规模的集群。本章只介绍 Kafka 项目提供的基础工具。Kafka 社区也开发了很多高级工具，虽然不属于 Kafka 核心项目，但是可以在 Kafka 网站上找到它们。

操作授权

虽然 Kafka 为主题操作提供了身份验证和授权机制，但默认配置并没有对这些工具的使用进行严格的限制。也就是说，不需要经过身份验证也可以使用这些工具，并可以在没有安全检查和审计的情况下执行诸如修改主题之类的操作。为防止发生未经授权的变更，需要确保只有管理员可以使用这些工具。

12.1 主题操作

可以用 kafka-topics.sh 执行大部分与主题相关的操作。可以用它创建、修改、删除和查看主题。虽然这个工具也可用于配置主题的一些参数，但现在已被弃用，建议使用更健壮的 kafka-config.sh 来配置主题参数。要使用 kafka-topics.sh，需要通过 --bootstrap-server 参数来指定集群的连接串和端口。下面的例子中使用的是本地 Kafka 集群，所以连接串是 localhost:9092。

本章假设所有工具都位于 /usr/local/kafka/bin/ 目录。本节示例中的命令都假设这个目录为当前工作目录，或者这个目录已经被添加到 $PATH 环境变量中。

版本检查

Kafka 的很多命令行工具对 Kafka 版本有所依赖，一些命令可能会将数据保存在 ZooKeeper 中而不是 broker 中。因此，你需要确保命令行工具的版本与集群 broker 的版本相匹配。直接使用 broker 自带的命令行工具是最保险的。

12.1.1 创建新主题

在使用 --create 命令创建新主题时，必须提供几个参数，即使其中一些参数可能已经有了 broker 级别的默认值。还可以通过 --config 选项指定其他参数和覆盖默认值，本章将在后面部分讨论这方面的内容。下面是必须提供的 3 个参数。

--topic

想要创建的主题的名字。

--replication-factor

主题的副本数量。

--partitions

主题的分区数量。

主题命名最佳实践

主题的名字可以包含字母数字字符、下划线、破折号和点号，但不建议使用点号。Kafka 的内部指标会将点号转成下划线（例如，“topic.1”在指标中会变成“topic_1”），这可能会导致主题名称冲突。

另外，避免使用双下划线作为主题名字的前缀。根据约定，Kafka 内部主题的名字以双下划线开头（例如，用于保存消费者组偏移量的 __consumer_offsets 主题）。因此，为了避免冲突，不建议使用以双下划线开头的主题名字。

创建一个新主题非常简单，只需像下面这样运行 kafka-topics.sh。

```
# kafka-topics.sh --bootstrap-server <connection-string>:<port> --create --
topic <string>
--replication-factor <integer> --partitions <integer>
#
```

这个命令将创建一个新主题，主题名字为指定的值，并包含了指定数量的分区。集群会为每个分区创建指定数量的副本。如果为集群指定了基于机架信息的副本分配策略，那么分区的副本就会被放置在不同的机架上。如果不需要这种分配策略，则可以指定 --disable-rack-aware 选项。

例如，使用下面的命令创建一个名为“my-topic”的主题，该主题包含 8 个分区，每个分区有两个副本。

```
# kafka-topics.sh --bootstrap-server localhost:9092 --create
--topic my-topic --replication-factor 2 --partitions 8
Created topic "my-topic".
#
```

使用 `if-exists` 参数和 `if-not-exists` 参数

在自动化脚本中调用 `kafka-topics.sh` 时，可以指定 `--if-not-exists` 选项，这样即使要创建的主题已经存在也不会抛出异常。

虽然 Kafka 为 `--alter` 命令提供了 `--if-exists` 参数，但不建议使用它。这是因为如果指定了这个选项，并且要修改的主题不存在，那么 `--alter` 命令就不会返回任何错误，从而掩盖了本应创建这个不存在的主题的错误。

12.1.2 列出集群中的所有主题

可以用 `--list` 命令列出集群中的所有主题。每个主题占用一行输出，主题之间没有特定的顺序。

下面是用 `--list` 命令列出集群中所有主题的示例。

```
# kafka-topics.sh --bootstrap-server localhost:9092 --list
__consumer_offsets
my-topic
other-topic
#
```

这里也列出了内部主题 `__consumer_offsets`。可以用 `--exclude-internal` 参数将名字以双下划线开头的主题排除。

12.1.3 列出主题详情

还可以获取一个或多个主题的详细信息。详细信息里包含了分区数量、主题的覆盖配置以及每个分区的副本清单。也可以指定 `--topic` 参数，只列出指定主题的详细信息。

例如，下面的命令列出了我们最近创建的“my-topic”主题的详细信息。

```
# kafka-topics.sh --boostrap-server localhost:9092 --describe --topic my-topic
Topic: my-topic PartitionCount: 8       ReplicationFactor: 2    Configs: seg
ment.bytes=1073741824
        Topic: my-topic Partition: 0    Leader: 1       Replicas: 1,0   Isr: 1,0
        Topic: my-topic Partition: 1    Leader: 0       Replicas: 0,1   Isr: 0,1
        Topic: my-topic Partition: 2    Leader: 1       Replicas: 1,0   Isr: 1,0
        Topic: my-topic Partition: 3    Leader: 0       Replicas: 0,1   Isr: 0,1
        Topic: my-topic Partition: 4    Leader: 1       Replicas: 1,0   Isr: 1,0
        Topic: my-topic Partition: 5    Leader: 0       Replicas: 0,1   Isr: 0,1
        Topic: my-topic Partition: 6    Leader: 1       Replicas: 1,0   Isr: 1,0
        Topic: my-topic Partition: 7    Leader: 0       Replicas: 0,1   Isr: 0,1
#
```

`--describe` 命令也有一些用于过滤输出结果的参数，它们在诊断集群问题时非常有用。我们通常不会为这些命令指定 `--topic` 参数，因为我们的目的是要找出集群中所有满足条件的主题和分区。这个参数对 `list` 命令不管用。下面是其他一些有用的参数。

`--topics-with-overrides`

这个参数只会列出配置参数与集群默认值不同的主题。

--exclude-internal

这是之前提到的参数，它可以将所有名字以双下划线开头的主题排除。下面的参数可用于找出有问题的分区。

--under-replicated-partitions

这个参数可以找出一个或多个副本与首领不同步的分区。这并不一定是坏事，因为集群维护、部署和再均衡都会导致分区副本不同步，但还是要注意一下。

--at-min-isr-partitions

这个参数可以找出副本数量（包括首领在内）与配置的最少同步副本（ISR）数完全匹配的分区。这些主题对生产者客户端或消费者客户端来说仍然可用，但已无冗余，有会变得不可用的风险。

--under-min-isr-partitions

这个参数可以找出 ISR 数低于配置的最小值的分区。这些分区实际上处于只读模式，不能向其写入数据。

--unavailable-partitions

这个参数可以找出所有没有首领的分区。这种情况很严重，说明分区已离线，对生产者客户端或消费者客户端来说已经是不可用的。

下面是一个找出具有最少 ISR 数的主题的例子。在这个例子中，主题的最少 ISR 数为 1，复制系数（RF）为 2。主机 0 在线，主机 1 停机维护。

```
# kafka-topics.sh --bootstrap-server localhost:9092 --describe --at-min-isrpartitions
        Topic: my-topic Partition: 0    Leader: 0       Replicas: 0,1   Isr: 0
        Topic: my-topic Partition: 1    Leader: 0       Replicas: 0,1   Isr: 0
        Topic: my-topic Partition: 2    Leader: 0       Replicas: 0,1   Isr: 0
        Topic: my-topic Partition: 3    Leader: 0       Replicas: 0,1   Isr: 0
        Topic: my-topic Partition: 4    Leader: 0       Replicas: 0,1   Isr: 0
        Topic: my-topic Partition: 5    Leader: 0       Replicas: 0,1   Isr: 0
        Topic: my-topic Partition: 6    Leader: 0       Replicas: 0,1   Isr: 0
        Topic: my-topic Partition: 7    Leader: 0       Replicas: 0,1   Isr: 0
#
```

12.1.4 增加分区

有时候需要增加主题的分区数量。分区是主题进行伸缩和复制的基础，增加分区通常是为了通过降低单个分区的吞吐量来扩展主题容量。如果要在单个消费者群组内运行更多的消费者，则分区数量也需要相应增加，因为一个分区只能被同一个群组里的一个消费者读取。

下面的例子会先使用 `--alter` 命令将“my-topic”主题的分区数量增加到 16，然后再验证是否修改成功。

```
# kafka-topics.sh --bootstrap-server localhost:9092
--alter --topic my-topic --partitions 16

# kafka-topics.sh --bootstrap-server localhost:9092 --describe --topic my-topic
Topic: my-topic PartitionCount: 16      ReplicationFactor: 2    Configs: seg
ment.bytes=1073741824
        Topic: my-topic Partition: 0    Leader: 1       Replicas: 1,0   Isr: 1,0
        Topic: my-topic Partition: 1    Leader: 0       Replicas: 0,1   Isr: 0,1
        Topic: my-topic Partition: 2    Leader: 1       Replicas: 1,0   Isr: 1,0
        Topic: my-topic Partition: 3    Leader: 0       Replicas: 0,1   Isr: 0,1
        Topic: my-topic Partition: 4    Leader: 1       Replicas: 1,0   Isr: 1,0
        Topic: my-topic Partition: 5    Leader: 0       Replicas: 0,1   Isr: 0,1
        Topic: my-topic Partition: 6    Leader: 1       Replicas: 1,0   Isr: 1,0
        Topic: my-topic Partition: 7    Leader: 0       Replicas: 0,1   Isr: 0,1
        Topic: my-topic Partition: 8    Leader: 1       Replicas: 1,0   Isr: 1,0
        Topic: my-topic Partition: 9    Leader: 0       Replicas: 0,1   Isr: 0,1
        Topic: my-topic Partition: 10   Leader: 1       Replicas: 1,0   Isr: 1,0
```

```
        Topic: my-topic Partition: 11   Leader: 0       Replicas: 0,1   Isr: 0,1
        Topic: my-topic Partition: 12   Leader: 1       Replicas: 1,0   Isr: 1,0
        Topic: my-topic Partition: 13   Leader: 0       Replicas: 0,1   Isr: 0,1
        Topic: my-topic Partition: 14   Leader: 1       Replicas: 1,0   Isr: 1,0
        Topic: my-topic Partition: 15   Leader: 0       Replicas: 0,1   Isr: 0,1
#
```

调整包含键的主题

从消费者角度来看，为包含键的主题添加分区是很困难的。这是因为如果改变了分区数量，那么键与分区之间的映射也会发生变化。因此，对于包含键的主题，建议一开始就设置好分区数量，避免后续对其进行调整。

12.1.5 减少分区

我们不可能减少主题的分区数量。如果删除了主题的一个分区，那么这个分区里的数据也会被删除，导致客户端看到的数据不一致。此外，将数据重新分布到其余的分区中是非常困难的，即使能够做到，也无法保证消息的顺序。如果要减少分区数量，则建议删除整个主题并重新创建，或者（如果不能删除的话）创建一个新主题（比如叫作“my-topic-v2”），并将所有流量重定向到新主题。

12.1.6 删除主题

即使一个主题里没有任何消息，仍然会占用集群资源（比如磁盘空间、文件句柄和内存）。控制器中也有一些垃圾元数据，这些垃圾元数据在大型集群中会影响集群性能。如果一个主题不再被使用，那么可以将其删除，以便释放出被占用的资源。要删除主题，broker 的 `delete.topic.enable` 参数必须被设置为 `true`，如果设置为 `false`，那么删除主题的请求将被忽略，也就无法删除主题。

删除主题是异步操作，也就是说，要删除的主题将被打上删除标记，但可能不会立即被删除，具体取决于数据量和清理策略。控制器会尽快通知 broker 有挂起的删除操作（在现有的控制器任务完成之后），然后 broker 会让主题的元数据失效，并从磁盘上删除相应的文件。由于控制器删除主题的方式受到了限制，建议不要一次性删除多个主题，而且在开始删除其他主题之前要给当前的删除操作留有足够的时间。本书中的示例使用的是小型集群，主题删除几乎会立即执行，但在大型集群中可能需要更长的时间。

小心丢失数据

删除一个主题也就删除了主题的所有数据。这是一个不可逆操作，所以在删除时要十分小心。

下面的例子会使用 `--delete` 命令删除“my-topic”主题。根据 Kafka 版本的不同，可能会有一个提示，告诉你如果没有设置其他一些参数，这个参数将不起作用。

```
# kafka-topics.sh --bootstrap-server localhost:9092
--delete --topic my-topic

Note: This will have no impact if delete.topic.enable is not set
to true.
#
```

主题删除是否成功并没有可视的反馈，不过可以通过 `--list` 命令或 `--describe` 命令来验证。如果要删除的主题已经不在集群中，则说明删除成功。

12.2 消费者群组

消费者群组用于协调消费者从多个主题或单个主题的多个分区读取消息。可以用 `kafka-consumer-groups.sh` 来管理和查看消费者群组信息，比如列出所有消费者群组、查看特定的消费者群组、删除多个或特定的消费者群组以及重置消费者群组偏移量。

基于 ZooKeeper 的消费者群组

在旧版 Kafka 中，可以直接通过访问 ZooKeeper 来管理和维护消费者群组，Kafka 0.11.0.* 及后续版本弃用了这种方式，也不再使用旧版消费者群组。一些脚本可能仍然会显示已被弃用的 `--zookeeper` 连接串命令，但不建议继续使用它，除非你的旧环境中还有一些消费者群组没有升级到更高版本。

12.2.1 列出并描述消费者群组信息

要列出消费者群组，需要指定 `--bootstrap-server` 参数和 `--list` 参数。使用 `kafka-consumer-groups.sh` 创建的临时消费者将以 `console-consumer-<generated_id>` 的格式列出来。

```
# kafka-consumer-groups.sh --bootstrap-server localhost:9092 --list
console-consumer-95554
console-consumer-9581
my-consumer
#
```

如果用 `--describe` 代替 `--list`，并用 `--group` 指定特定的群组，就可以获取这个群组的详细信息。它会列出消费者群组读取的所有主题和分区的信息以及每个分区的偏移量信息。表 12-1 对列出的信息字段做了详细说明。

表 12-1：输出结果中的字段释义

字段	描述
GROUP	消费者群组名称
TOPIC	正在读取的主题名称
PARTITION	正在读取的分区 ID
CURRENT-OFFSET	消费者群组要读取的下一条消息的偏移量，也就是消费者在当前分区中的位置
LOG-END-OFFSET	分区的当前高水位标记，也就是下一条要写入这个分区的消息的偏移量
LAG	CURRENT-OFFSET 和 LOG-END-OFFSET 之间的差值
CONSUMER-ID	基于 CLIENT-ID 生成的消费者 ID
HOST	消费者群组读取的主题所在的主机名
CLIENT-ID	客户端提供的用于标识客户端的 ID

下面是获取临时消费者群组“my-consumer”的详细信息。

```
# kafka-consumer-groups.sh --bootstrap-server localhost:9092
--describe --group my-consumer
GROUP          TOPIC          PARTITION  CURRENT-OFFSET  LOG-END-OFFSET
LAG        CONSUMER-ID
HOST                           CLIENT-ID
my-consumer     my-topic       0          2               4
2          consumer-1-029af89c-873c-4751-a720-cefd41a669d6   /
127.0.0.1 consumer-1
my-consumer     my-topic       1          2               3
1          consumer-1-029af89c-873c-4751-a720-cefd41a669d6   /
127.0.0.1 consumer-1
my-consumer     my-topic       2          2               3
1          consumer-2-42c1abd4-e3b2-425d-a8bb-e1ea49b29bb2   /
127.0.0.1 consumer-2
#
```

12.2.2 删除消费者群组

可以用 `--delete` 命令来删除消费者群组。它将删除整个群组，包括所有已保存的偏移量。在删除群组之前，必须将群组里所有的消费者都关闭。如果你试图删除一个非空的群组，那么它将抛出“群组不为空”异常。也可以用这个命令删除单个主题的偏移量，只是需要额外提供 `--topic` 参数，并指定要删除的偏移量。

下面是删除消费者群组“my-consumer”的示例。

```
# kafka-consumer-groups.sh --bootstrap-server localhost:9092 --delete --group
my-consumer
Deletion of requested consumer groups ('my-consumer') was successful.
#
```

12.2.3 偏移量管理

除了显示和删除消费者群组的偏移量，还可以批量获取和保存消费者群组的偏移量，这在重置消费者偏移量时非常有用。当消费者因为某些原因需要重新读取消息或因为无法正常处理某些消息（比如格式错误的消息）需要跳过这些消息时就可以进行偏移量重置。

01. 导出偏移量

要将消费者群组的偏移量导出到 CSV 文件，可以使用 `--reset-offsets` 参数和 `--dry-run` 参数。这样就可以将当前的偏移量导出到文件，并在需要的时候用于导入或回滚偏移量。导出的 CSV 文件格式如下所示。

<topic-name>,<partition-number>,<offset>

如果没有指定 `--dry-run` 参数，那么偏移量将被重置，所以在执行这个命令时要十分小心。

下面的例子会将消费者群组“my-consumer”读取的“my-topic”主题的偏移量导出到一个叫作 offsets.csv 的文件中。

```
# kafka-consumer-groups.sh --bootstrap-server localhost:9092
--export --group my-consumer --topic my-topic
--reset-offsets --to-current --dry-run > offsets.csv

# cat offsets.csv
my-topic,0,8905
my-topic,1,8915
my-topic,2,9845
my-topic,3,8072
my-topic,4,8008
my-topic,5,8319
my-topic,6,8102
```

```
my-topic,7,12739
#
```

02. 导入偏移量

与导出偏移量刚好相反，导入偏移量会用之前导出的文件来重置消费者群组的偏移量。一般情况下，我们会先导出消费者群组的当前偏移量，并将导出的文件复制一份（作为备份），然后将文件中的偏移量修改成想要的值。

先关闭消费者

在导入偏移量之前，必须先关闭所有的消费者。如果消费者群组处于活跃状态，那么当消费者正要写入偏移量时，不仅不会读取到导入的偏移量，反而有可能将它们覆盖掉。

在下面的例子中导入上一个例子中导出的 offsets.csv 文件，以此来重置消费者群组“my-consumer”的偏移量。

```
# kafka-consumer-groups.sh --bootstrap-server localhost:9092
--reset-offsets --group my-consumer
--from-file offsets.csv --execute
  TOPIC                          PARTITION  NEW-OFFSET
  my-topic                        0          8905
  my-topic                        1          8915
  my-topic                        2          9845
  my-topic                        3          8072
  my-topic                        4          8008
  my-topic                        5          8319
  my-topic                        6          8102
  my-topic                        7          12739
#
```

12.3 动态配置变更

主题、客户端、broker 等都有大量的配置参数可以在运行时动态更新，无须关闭或重新部署集群。可以使用 `kafka-config.sh` 来修改这些配置参数。目前，可以进行动态变更的配置参数主要有 4 种类型：主题、broker、用户和客户端。对于每一种类型，都有一些可以覆盖的配置。随着 Kafka 不断发布新版本，新的动态配置参数也会不断被添加进来，所以最好确保你使用的工具版本与 Kafka 版本相匹配。为了便于自动化管理动态配置参数，可以通过 `--add-config-file` 参数来指定包含了你想要配置的参数的文件。

12.3.1 覆盖主题的默认配置

在默认情况下，静态的 broker 配置文件中已经提供了主题的一些默认设置（比如数据保留策略）。可以通过动态配置覆盖个别主题的默认值，以满足不同应用场景的需求。表 12-2 列出了可以被动态修改的主题配置参数。

表 12-2：可用的主题配置参数

配置项	描述
`cleanup.policy`	如果被设置为 `compact`，则只有包含了键的最新消息会被保留下来（日志被压实），其他消息会被丢弃
`compression.type`	broker 在将消息批次写入磁盘时所使用的压缩类型
`delete.retention.ms`	墓碑消息能够保留多久，以毫秒为单位。这个参数只对压缩日志类型的主题有效
`file.delete.delay.ms`	从磁盘上删除日志片段和索引之前可以等待多长时间，以毫秒为单位
`flush.messages`	在冲刷到磁盘之前可以接收多少条消息
`flush.ms`	在将消息冲刷到磁盘之前可以等待多长时间，以毫秒为单位
`follower.replication.throttled.replicas`	在复制日志时需要根据跟随者副本进行节流的副本清单
`index.interval.bytes`	日志片段索引之间能够容纳的消息字节数
`leader.replication.throttled.replica`	在复制日志时需要根据首领副本进行节流的副本清单
`max.compaction.lag.ms`	一条消息可以不被压实的最长时间
`max.message.bytes`	消息的最大字节数
`message.downconversion.enable`	如果启用了这个参数，则消息格式可以被转换成之前的版本
`message.format.version`	broker 将消息写入磁盘时所使用的消息格式版本，必须是有效的 API 版本号

配置项	描述
message.timestamp.difference.max.ms	消息自带的时间戳和 broker 收到消息时的时间戳之间最大的差值，以毫秒为单位。这个参数只在 messsage.timestamp.type 被设为 CreateTime 时有效
message.timestamp.type	在将消息写入磁盘时使用哪一种时间戳。目前支持两个值：一个是 CreateTime，指客户端指定的时间戳；一个是 LogAppendTime，指消息被写入分区时的时间戳
min.cleanable.dirty.ratio	压实分区的频率，表示为未压缩日志片段数与总日志分段数之间的比例。这个参数只对压缩日志类型的主题有效
min.compaction.lag.ms	一条消息不被压实的最短时间
min.insync.replicas	可用分区的最少 ISR
preallocate	如果被设置为 true，那么需要为新的日志片段预分配空间
retention.bytes	主题能够保留多少消息，以字节为单位
retention.ms	主题需要保留消息多长时间，以毫秒为单位
segment.bytes	分区的单个日志片段可以保存的消息字节数
segment.index.bytes	单个日志片段的最大索引字节数
segment.jitter.ms	在滚动日志片断时，在 segment.ms 基础上随机增加的毫秒数
segment.ms	日志片段多长时间滚动一次，以毫秒为单位
unclean.leader.election.enable	如果被设置为 false，就不进行不彻底的首领选举

修改主题配置的命令格式如下。

```
kafka-configs.sh --bootstrap-server localhost:9092
--alter --entity-type topics --entity-name <topic-name>
--add-config <key>=<value>[,<key>=<value>...]
```

下面的例子将主题“my-topic”的数据保留时间设置为 1 小时（3 600 000 毫秒）。

```
# kafka-configs.sh --bootstrap-server localhost:9092
--alter --entity-type topics --entity-name my-topic
--add-config retention.ms=3600000
Updated config for topic: "my-topic".
#
```

12.3.2 覆盖客户端和用户的默认配置

对 Kafka 客户端和用户配置来说，只有少数参数可以覆盖，而且基本上都与配额有关。常见的两个需要修改的参数是生产者的生产速率和消费者的消费速率，以字节 / 秒为单位。表 12-3 列出了可以被修改的客户端和用户的配置参数。

表 12-3：客户端和用户配置参数

配置项	描述
consumer_bytes_rate	单个消费者每秒可以从单个 broker 读取的消息字节数
producer_bytes_rate	单个生产者每秒可以向单个 broker 生成的消息字节数
controller_mutations_rate	可接受的创建主题请求、创建分区请求和删除主题请求的速率。这个速率是根据创建或删除的分区数量累计计算出来的
request_percentage	用户请求或客户端请求的配额窗口百分比（(num.io.threads + num.network.threads) × 100%）

不均衡集群的不均衡节流

因为节流是以 broker 为基础，所以集群中的分区首领需要均匀分布才能更好地实现节流策略。假设你在一个集群中有 5 个 broker，并为一个客户端指定了 10 MBps 的生产者配额。如果分区首领均匀地分布在这 5 个 broker 上，那么这个客户端就可以同时以 10 MBps 的速率向每个 broker 生成数据，总共 50 MBps。但是，如果所有分区的首领都在同一个 broker 上，那么同一个生产者的生产速率最多只能是 10 MBps。

客户端 ID 与消费者群组

客户端 ID 可以与消费者群组的名字不一样。消费者可以有自己的客户端 ID，因此不同群组里的消费者可以有相同的客户端 ID。在为消费者设置客户端 ID 时，最好可以使用能够表明它们所属群组的标识符，让群组可以共享配额，以后在日志里查找哪个请求是被哪个群组处理的也更容易一些。

也可以指定同时适用于用户和客户端的配置参数。下面的例子在一个配置步骤中同时修改了用户的控制器突变率和客户端的控制器突变率。

```
# kafka-configs.sh --bootstrap-server localhost:9092
--alter --add-config "controller_mutations_rate=10"
--entity-type clients --entity-name <client ID>
--entity-type users --entity-name <user ID>
#
```

12.3.3 覆盖 broker 的默认配置

broker 和集群级别的配置主要被放在静态的集群配置文件中，但仍然有大量的参数可以在运行时修改，也就是说修改这些参数无须重新部署 Kafka。我们可以用 `kafka-config.sh` 修改 80 多个 broker 配置参数，不过本书不打算在这里一一罗列出来。可以通过 `--help` 命令或在开源文档中找到它们，下面是其中几个比较重要的。

`min.insync.replicas`

当生产者的 `acks` 被设置为 `all` 或 `-1` 时，用于确认消息写入成功所需的最少 ISR 数。

`unclean.leader.election.enable`

允许一个副本被选举为首领，即使可能会导致数据丢失。当允许丢失一些数据，或者在发生不可恢复的数据丢失后需要快速恢复 Kafka 集群时，可以短暂启用这个功能。

`max.connections`

broker 允许的最大连接数。还可以用 `max.connections.per.ip` 和 `max.connections.per.ip.override` 进行更细粒度的节流。

12.3.4 查看被覆盖的配置

可以用 `kafka-config.sh` 列出所有被覆盖的配置，这样就可以知道主题、broker 或客户端中都有哪些参数被覆盖。与其他工具类似，这是通过 `--describe` 命令实现的。

下面的例子列出了主题“my-topic”中所有被覆盖的配置。可以看到，被覆盖的只有“数据保留时间”这个参数。

```
# kafka-configs.sh --bootstrap-server localhost:9092
--describe --entity-type topics --entity-name my-topic
Configs for topics:my-topic are
retention.ms=3600000
#
```

只显示主题的覆盖配置

这个命令只显示被覆盖的配置，并不会包含集群的默认配置。我们现在无法动态发现 broker 级别的配置。所以，如果用户想在自动化流程中使用这个工具来获取主题或客户端配置，则必须对集群的默认配置做到心中有数。

12.3.5 移除被覆盖的配置

也可以移除动态配置，将其恢复到集群的默认配置。可以用 `--alter` 命令和 `--delete-config` 参数来移除被覆盖的配置。

例如，移除主题“my-topic”的 `retention.ms` 覆盖配置。

```
# kafka-configs.sh --bootstrap-server localhost:9092
--alter --entity-type topics --entity-name my-topic
--delete-config retention.ms
Updated config for topic: "my-topic".
#
```

12.4 生产和消费

在使用 Kafka 时，为了验证应用程序的逻辑，经常需要手动生成或消费一些示例消息。Kafka 为此提供了两个工具，即 `kafka-console-consumer.sh` 和 `kafka-console-producer.sh`。在第 2 章中，我们用它们验证过安装好的 Kafka 是否可以正常运行。这些工具对 Java 客户端库进行了包装，让我们可以在不编写代码的情况下与 Kafka 主题发生交互。

将输出作为其他应用程序的输入

虽然可以编写应用程序将控制台消费者或控制台生产者包装起来（例如，用它读取消息，再把消息传给另一个应用程序），但这种应用程序太过脆弱，所以应该尽量避免这么做。我们很难保证控制台消费者不丢失数据。同样，我们也无法充分利用控制台生产者的所有特性，正确地发送字节是很困难的。建议直接使用 Java 客户端，或其他实现了 Kafka 协议的第三方客户端。

12.4.1 控制台生产者

可以用 `kafka-console-producer.sh` 向 Kafka 主题写入消息。在默认情况下，一行输入就是一条消息，消息的键和值以 Tab 字符分隔（如果没有出现 Tab 字符，那么键就是 `null`）。与控制台消费者一样，生产者使用默认的序列化器（`DefaultEncoder`）生成原始字节。

控制台生产者要求至少提供两个参数，以便知道要连接到哪个 Kafka 集群，以及向哪个主题生成数据。第一个是我们经常使用的 `--bootstrap-server` 连接串。在生成数据结束后，需要发送一个文件结束符（EOF）来关闭客户端。对于大多数常见的终端，可以使用 Control-D 组合键来发送 EOF。

在下面的例子中，我们向主题“my-topic”生成了 4 条消息。

```
# kafka-console-producer.sh --bootstrap-server localhost:9092 --topic my-topic
>Message 1
>Test Message 2
>Test Message 3
>Message 4
>^D
#
```

01. 生产者配置参数

也可以将普通生产者的配置参数传给控制台生产者。传递参数有两种方式，使用哪一种取决于需要传多少个参数以及个人偏好。第一种方式是通过 `--producer.config` *`<config-file>`* 指定生产者配置文件，其中 *`<config-file>`* 是包含了配置参数的文件的完整路径。第二种方式是在命令行直接指定参数，格式为 `--producer-property` *`<key>=<value>`*，其中 *`<key>`* 是参数名，*`<value>`* 是参数值。这在配置生产者的一些参数 [比如消息批次的相关参数（`linger.ms` 或 `batch.size`）] 时非常有用。

容易混淆的命令行参数

控制台生产者和消费者有一个共同的命令行参数 `--property`，但不要将其与 `--producer-property` 或 `--consumer-property` 相混淆。我们通过 `--property` 将配置参数传给消息格式化器，而不是客户端本身。

控制台生产者有很多命令行参数，我们可以通过 `--producer-property` 来指定它们。以下是其中一些有用的参数。

`--batch-size`

指定在采用非同步发送方式时单个批次发送的消息数量。

--timeout

指定生产者在采用异步发送模式时等待批次填满消息的最长时间，以避免其在低吞吐量的情况下等待太长时间。

--compression-codec <string>

指定生成消息所使用的压缩类型，可以是 none、gzip、snappy、zstd 或 lz4，默认值是 gzip。

--sync

指定以同步的方式发送消息。也就是说，在发送下一条消息之前需要等待当前消息被确认。

02. 文本行读取器配置参数

kafka.tools.ConsoleProducer$LineMessageReader 类负责读取标准输入，并基于读取到的内容创建要发送给 Kafka 的消息。它也有一些非常有用的配置参数，我们可以通过 --property 把这些配置参数传给控制台生产者。

ignore.error

如果设置为 false，那么在 parse.key 被设置为 true 而标准输入中没有包含键分隔符时就会抛出异常。默认值为 true。

parse.key

如果设置为 false，那么生成的消息的键就为空。默认值为 true。

key.separator

消息的键和值之间的分隔符。默认使用 Tab 字符。

改变命令行的读取行为

如果有必要，那么可以使用自定义类来读取命令行输入。自定义类必须继承 kafka.common.MessageReader，并负责创建 ProducerRecord 对象。然后，我们通过 --line-reader 参数来指定这个类，并确保包含这个类的 JAR 包已经被加入类路径中。默认的类是 kafka.tools.Console Producer$LineMessageReader。

在生成消息时，LineMessageReader 将使用第一个出现的 key.separator 作为分隔符来拆分输入内容。如果在分隔符之后没有其他字符，那么消息的值就为空。如果输入中没有包含键分隔符，或者 parse.key 被设置为 false，那么消息的键就为空。

12.4.2 控制台消费者

kafka-console-consumer.sh 为我们提供了另外一种从 Kafka 集群的一个或多个主题读取消息的方式。它读取的消息会被打印在标准输出中，并用换行符分隔。在默认情况下，它将输出消息的原始字节，没有键，也不进行格式化（使用 DefaultFormatter）。与生产者类似，它也有一些必选的配置参数：集群连接串、要读取的主题以及要读取的消息的位置。

检查工具版本

控制台消费者的版本要与 broker 的版本相同，这一点非常重要。旧版控制台消费者与集群或 ZooKeeper 之间不恰当的交互行为可能对集群造成不良影响。

和其他命令一样，可以通过 `--bootstrap-server` 来指定集群连接串，不过要读取的主题的名字可以通过以下两个参数来指定。

`--topic`

这个参数会直接指定要读取的主题的名字。

`--whitelist`

这个参数是一个正则表达式，其匹配所有要读取的主题（要记得转义正则表达式，以免命令行解析错误）。

上述两个参数只需提供一个即可。控制台消费者在启动后会持续尝试读取消息，直到遇到退出命令（在下面的例子中使用了 Ctrl-C 组合键）。下面的例子读取的是集群中所有前缀与 `my` 匹配的主题（在这个例子中只有一个匹配的主题“my-topic”）。

```
# kafka-console-consumer.sh --bootstrap-server localhost:9092
--whitelist 'my.*' --from-beginning
Message 1
Test Message 2
Test Message 3
Message 4
^C
#
```

01. 消费者配置参数

除了这些基本的命令行参数，也可以将普通消费者的配置参数传给控制台消费者。与 `kafka-console-producer.sh` 类似，传递参数有两种方式，使用哪一种取决于需要传多少个参数以及个人偏好。第一种方式是通过 `--consumer.config <config-file>` 指定消费者配置文件，其中 *`<config-file>`* 是包含了配置参数的文件的完整路径。第二种方式是在命令行中直接指定参数，格式为 `--consumer-property =`，其中 *`<key>`* 是参数名，*`<value>`* 是参数值。

控制台消费者还有一些常用的其他参数，如果能了解和熟悉它们，则会非常有用。

`--formatter <classname>`

指定用于解码消息的消息格式化器的类名，默认是 `kafka.tools.DefaultMessageFormatter`。

`--from-beginning`

指定从最旧的偏移量开始读取数据。如果不指定这个参数，就从最新的偏移量开始读取。

`--max-messages <int>`

指定在退出之前最多读取多少条消息。

`--partition <int>`

只读取指定 ID 的分区。

`--offset`

如果提供的是整数，就从指定位置开始读取数据。其他有效的值为 earliest（将从起始位置开始读取）和 latest（将从最近的位置开始读取）。

--skip-message-on-error

如果在处理消息时出现错误就跳过消息，而不是一直挂起，这在调试问题时会非常有用。

02. 消息格式化器配置参数

除了默认的消息格式化器，还有另外 3 种可用的格式化器。

kafka.tools.LoggingMessageFormatter

将消息输出到日志而不是标准输出。对应的日志级别为 INFO，打印内容包含消息的时间戳、键和值。

kafka.tools.ChecksumMessageFormatter

只打印消息的校验和。

kafka.tools.NoOpMessageFormatter

读取但不打印消息。

下面的例子会使用 kafka.tools.ChecksumMessageFormatter 格式化器来读取之前的消息。

```
# kafka-console-consumer.sh --bootstrap-server localhost:9092
--whitelist 'my.*' --from-beginning
--formatter kafka.tools.ChecksumMessageFormatter
checksum:0
checksum:0
checksum:0
checksum:0
#
```

kafka.tools.DefaultMessageFormatter 也有一些非常有用的参数，这些参数可以通过 --property 传递，如表 12-4 所示。

表 12-4：消息格式化器的配置参数

配置项	描述
print.timestamp	如果被设置为 true，那么将打印每条消息的时间戳（如果有的话）
print.key	如果被设置为 true，那么除了打印消息的值，还会打印消息的键
print.offset	如果被设置为 true，那么除了打印消息的值，还会打印消息的偏移量
print.partition	如果被设置为 true，那么将打印消息来自哪个分区
key.separator	指定打印消息的键和值时所使用的分隔符
line.separator	指定消息之间的分隔符
key.deserializer	指定打印消息的键所使用的反序列化器的类名
value.deserializer	指定打印消息的值所使用的反序列化器的类名

反序列化器类必须实现 org.apache.kafka.common.serialization.Deserializer 接口，控制台消费者会调用它们的 toString() 方法来获取输出结果。一般来说，在调用 kafka_console_consumer.sh 之前，需要设置环境变量 CLASSPATH 以将实现类添加到类路径中。

03. 读取偏移量主题

有时候，需要查看集群的消费者群组提交了哪些偏移量。例如，你可能想知道某个消费者群组是否在提交偏移量，或者在以怎样的频率提交偏移量。这可以通过使用控制台消费者读取 `consumer_offsets` 这个特殊的内部主题来实现。所有的消费者偏移量都会被写入这个主题。要解码这个主题中的消息，必须使用 `kafka.coordinator.group.GroupmetadataManager$OffsetsMessageFormatter` 这个格式化器。

下面的例子会从最早的偏移量位置开始读取 `consumer_offsets` 主题。

```
# kafka-console-consumer.sh --bootstrap-server localhost:9092
--topic __consumer_offsets --from-beginning --max-messages 1
--formatter "kafka.coordinator.group.GroupMetadataManager\$OffsetsMessageFormat
ter"
--consumer-property exclude.internal.topics=false
[my-group-name,my-topic,0]::[OffsetMetadata[1,NO_METADATA]
CommitTime 1623034799990 ExpirationTime 1623639599990]
Processed a total of 1 messages
#
```

12.5 分区管理

Kafka 提供了一些用于管理分区的脚本，其中一个用于重新选举首领，另一个用于将分区分配给 broker。有了这两个工具，就可以通过手动的方式让消息流量均衡地分布在集群的 broker 上。

12.5.1 首选首领选举

第 7 章讲过，为了保证可靠性，分区可以有多个副本。在任意时刻，这些副本当中只能有一个可以成为分区首领，所有的写入操作和读取操作都发生在分区首领所在的 broker 上。为了保证负载均衡地分布在整个 Kafka 集群中，需要保持分区首领均衡地分布在 broker 上。

Kafka 会将分区副本清单中的第一个 ISR 定义为首选首领。当 broker 停止运行或与集群中的其他 broker 断开连接时，分区领导权将被转移给另一个 ISR，原始副本就自动丧失了分区领导权。如果不启用自动首领均衡，那么在进行跨集群部署后可能会出现非常低效的均衡。因此，建议启用这个功能，或者使用其他开源工具（如 Cruise Control）来确保在任何时候都能保持良好的均衡。

如果发现 Kafka 集群变得不均衡了，则可以考虑进行**首选首领选举**，这是一个轻量级的首领选举过程，一般来说不会造成负面影响。集群控制器会为分区选择理想的首领。客户端可以自动跟踪首领变更，如果领导权发生了变更，则它们会转移到新首领所在的 broker。这个过程可以用 `kafka-leader-election.sh` 来手动触发。旧版本的 `kafka-preferred-replica-election.sh` 也可以继续使用，但已被弃用。新版本支持更多的自定义选项，比如指定是进行“首选”还是“不彻底”的选举。

例如，可以使用下面的命令为集群中的所有主题启动一个首选首领选举。

```
# kafka-leader-election.sh --bootstrap-server localhost:9092
--election-type PREFERRED --all-topic-partitions
#
```

也可以为特定的分区或主题启动选举，直接用 `--topic` 参数指定主题名，用 `--partition` 参数指定分区。还可以指定一个清单，其中包含了多个参与选举的分区，这个清单包含在一个名为 partitions.json 的 JSON 文件中。

```
{
    "partitions": [
        {
            "partition": 1,
            "topic": "my-topic"
        },
        {
            "partition": 2,
            "topic": "foo"
        }
    ]
}
```

在这个例子中，我们将通过 partitions.json 文件指定的分区清单启动一个首选首领选举。

```
# kafka-leader-election.sh --bootstrap-server localhost:9092
--election-type PREFERRED --path-to-json-file partitions.json
#
```

12.5.2 修改分区的副本

在某些情况下，可能需要手动修改分区的副本，下面是需要这样做的几种场景。

- broker 的负载分布不均衡，自动首领选举也无法解决这个问题。
- broker 离线，造成分区不同步。
- 新加了 broker，你想快速给它分配分区。
- 你想修改主题的复制系数。

可以用 kafka-reassign-partitions.sh 来调整分区的副本。这个过程包含了多个步骤，先是生成迁移清单，然后再根据迁移清单执行调整。具体步骤如下：首先，需要基于 broker 和主题生成一个迁移清单。要生成迁移清单，需要一个 JSON 文件，其中包含了要调整的主题。然后，执行调整。最后，这个工具可用于跟踪和验证分区调整的进度或完成情况。

假设你有一个包含 4 个 broker 的 Kafka 集群。最近，你又添加了 2 个新 broker，现在总数是 6 个，你希望将其中的 2 个主题移动到 broker 5 和 broker 6 上。

你需要先创建一个包含了主题清单的 JSON 文件（假设文件名是 topics.json），格式如下（目前的版本号都是 1）。

```
{
    "topics": [
        {
            "topic": "foo1"
        },
        {
            "topic": "foo2"
        }
    ],
    "version": 1
}
```

有了这个 JSON 文件，就可以用它生成移动清单，以便在后面的步骤中将 topics.json 中的主题移动到 broker 5 和 broker 6 上。

```
# kafka-reassign-partitions.sh --bootstrap-server localhost:9092
--topics-to-move-json-file topics.json
--broker-list 5,6 --generate
  {"version":1,
  "partitions":[{"topic":"foo1","partition":2,"replicas":[1,2]},
                {"topic":"foo1","partition":0,"replicas":[3,4]},
                {"topic":"foo2","partition":2,"replicas":[1,2]},
                {"topic":"foo2","partition":0,"replicas":[3,4]},
                {"topic":"foo1","partition":1,"replicas":[2,3]},
                {"topic":"foo2","partition":1,"replicas":[2,3]}]
  }

  Proposed partition reassignment configuration

  {"version":1,
  "partitions":[{"topic":"foo1","partition":2,"replicas":[5,6]},
                {"topic":"foo1","partition":0,"replicas":[5,6]},
                {"topic":"foo2","partition":2,"replicas":[5,6]},
                {"topic":"foo2","partition":0,"replicas":[5,6]},
                {"topic":"foo1","partition":1,"replicas":[5,6]},
                {"topic":"foo2","partition":1,"replicas":[5,6]}]
  }
#
```

可以分别将两个 JSON 片段保存成两个 JSON 文件，一个叫作 revert-reassignment.json，另一个叫作 expand-cluster-reassignment.json。如果因为某些原因需要回滚，则可以用第一个文件将分区移回原来的位置。第二个文件用于在下一步执行分区调整。这一步只是生成建议，还没有执行任何操作。你可能会注意到，调整建议里的首领是不均衡的，因为所有的首领都在 broker 5 上。不过可以暂且忽略这一点，并假定集群启用了自动首领均衡，稍后将会自动进行均衡。如果你知道要将分区移到哪里，并手动创建了 JSON 主题清单文件，那么就可以跳过第一步。

可以通过下面的命令基于 expand-cluster-reassignment.json 文件执行分区调整。

```
# kafka-reassign-partitions.sh --bootstrap-server localhost:9092
--reassignment-json-file expand-cluster-reassignment.json
--execute
  Current partition replica assignment

  {"version":1,
  "partitions":[{"topic":"foo1","partition":2,"replicas":[1,2]},
                {"topic":"foo1","partition":0,"replicas":[3,4]},
```

```
                {"topic":"foo2","partition":2,"replicas":[1,2]},
                {"topic":"foo2","partition":0,"replicas":[3,4]},
                {"topic":"foo1","partition":1,"replicas":[2,3]},
                {"topic":"foo2","partition":1,"replicas":[2,3]}]
  }

  Save this to use as the --reassignment-json-file option during rollback
  Successfully started reassignment of partitions
  {"version":1,
  "partitions":[{"topic":"foo1","partition":2,"replicas":[5,6]},
                {"topic":"foo1","partition":0,"replicas":[5,6]},
                {"topic":"foo2","partition":2,"replicas":[5,6]},
                {"topic":"foo2","partition":0,"replicas":[5,6]},
                {"topic":"foo1","partition":1,"replicas":[5,6]},
                {"topic":"foo2","partition":1,"replicas":[5,6]}]
  }
#
```

这样就开始将指定的分区副本重新分配给新 broker 了。输出结果与之前生成的建议一样。集群控制器会将新副本添加到每个分区的副本清单中，这些主题的复制系数会临时变大。然后，新副本将从当前首领复制已有的消息。根据磁盘上分区的大小，当数据通过网络复制到新副本时，可能需要花费一些时间。复制完成之后，控制器会将旧副本从副本清单中移除，让复制系数恢复到原来的大小。

下面是这个工具的一些有用的参数。

`--additional`

如果已经有进行中的重分配过程，就加入其中，这样就不会出现中断，也不需要等待已有的重分配完毕之后再启动一个新的批次。

`--disable-rack-aware`

有时候，因为启用了机架感知，可能无法实现想要的重分配状态。如果有必要，那么可以用这个参数禁用机架感知。

`--throttle`

这个参数以字节 / 秒为单位。分区重分配对集群性能有很大的影响，因为它们会导致内存缓存页的一致性发生变化，并占用额外的网络带宽和磁盘 IO。对分区移动流量进行节流可以有效缓解这个问题。这个参数可以与 `--additional` 结合使用，以便对进行中的可能导致上述问题的重分配过程进行节流。

为副本重分配优化网络的使用

如果要移除一个 broker 的多个分区（将这个 broker 从集群中删除），那么最好可以先移除这个 broker 所有的分区首领。这可以通过手动的方式实现，但要用之前介绍的工具来完成这个操作有点儿困难。一些开源工具（如 Cruise Control）提供了 broker“降级”等功能，它们可以安全地将 broker 的领导权转移出去。这可能是最简单的方法。

如果不能使用这些工具，那么就重启 broker。在 broker 被关闭的过程中，位于这个 broker 上的所有分区首领都将被转移到其他 broker 上。这样可以显著提高副本重分配的性能，并减少对集群的影响，因为复制流量被其他 broker 分摊掉了。但是，如果启用了自动首领重分配，则 broker 在重启后有可能会重新获得领导权。所以，在重启期间最好临时禁用这个功能。

也可以用这个工具来验证重分配的完成情况。它可以显示哪些重分配正在进行当中、哪些已经完成，以及（如果出错的话）哪些已经失败。这个时候，必须提供用于执行重分配的 JSON 迁移清单文件。

下面是通过 `--verify` 参数基于 expand-cluster-reassignment.json 文件对之前的分区重分配进行验证的例子。

```
# kafka-reassign-partitions.sh --bootstrap-server localhost:9092
--reassignment-json-file expand-cluster-reassignment.json
--verify
Status of partition reassignment:
  Reassignment of partition [foo1,0] completed successfully
  Reassignment of partition [foo1,1] is in progress
  Reassignment of partition [foo1,2] is in progress
  Reassignment of partition [foo2,0] completed successfully
  Reassignment of partition [foo2,1] completed successfully
  Reassignment of partition [foo2,2] completed successfully
#
```

01. 修改复制系数

也可以用 kafka-reassign-partitions.sh 来增加或减少一个分区的 RF。如果在创建分区时没有选择合适的 RF，那么当你在扩展集群时想要增加冗余或为了节约成本想要减少冗余，就需要修改 RF。一个很明显的例子是，如果集群的默认 RF 被修改了，但已有主题的 RF 不会自动随之发生变化，那么这个时候就可以用这个工具增加已有分区的 RF。

假设你想将之前示例中“foo1”主题的 RF 从 2 增加到 3，那么可以创建一个类似于之前用于执行分区调整的 JSON 文件，然后再向副本集中加入额外的 broker ID。例如，可以创建一个叫作 increase-foo1-RF.json 的 JSON 文件，并将 broker 4 添加到已有的 broker 5 和 broker 6 集合中。

```
{
  {"version":1,
  "partitions":[{"topic":"foo1","partition":1,"replicas":[5,6,4]},
                {"topic":"foo1","partition":2,"replicas":[5,6,4]},
                {"topic":"foo1","partition":3,"replicas":[5,6,4]},
  ]
}
```

然后，使用之前的命令执行这个文件。在执行完成之后，可以用 --verify 参数或 kafka-topics.sh 脚本列出主题信息，以此来验证 RF 是否已经修改成功。

```
# kafka-topics.sh --bootstrap-server localhost:9092 --topic foo1 --describe
  Topic:foo1    PartitionCount:3        ReplicationFactor:3     Configs:
    Topic: foo1 Partition: 0    Leader: 5       Replicas: 5,6,4 Isr: 5,6,4
    Topic: foo1 Partition: 1    Leader: 5       Replicas: 5,6,4 Isr: 5,6,4
    Topic: foo1 Partition: 2    Leader: 5       Replicas: 5,6,4 Isr: 5,6,4
#
```

02. 取消分区重分配

在以前，取消副本重分配是一个危险的过程，需要手动删除 ZooKeeper 的 /admin/reassign_partitions 节点。幸运的是，kafka-reassign-partitions.sh 脚本（对 AdminClient 进行了包装）现在支持 --cancel 选项，我们可以用它来取消集群中正在进行的重分配过程。如果指定了 --cancel 选项，那么副本集将会被恢复到重分配之前的状态。如果正在从已经失效或过载的 broker 中移除副本，则取消分区重分配有可能会导致集群处于非预期的状态，也不能保证恢复后的副本集的顺序与之前相同。

12.5.3 转储日志片段

有时候，你可能需要查看消息的内容，比如主题中出现了已损坏的“毒药”消息，而用户无法处理它们。可以用 kafka-dump-log.sh 来解码分区的日志片段，这样就可以在不读取和解码消息的情况下查看消息的内容。这个工具以日志片段文件列表（以逗号分隔）作为参数，可以输出消息的摘要信息或详细内容。

在下面的例子中，我们对只包含 4 条消息的“my-topic”主题进行了日志转储。首先，简单地解码日志片段文件 00000000000000000000.log，并获取每条消息的基本元数据信息，但不打印消息内容。Kafka 的数据目录是 /tmp/kafka-logs。因此，可以在 /tmp/kafka-logs/<topic-name>-<partition> 目录下找到转储文件，对这个例子来说就是 /tmp/kafka-logs/my-topic-0/。

```
# kafka-dump-log.sh --files /tmp/kafka-logs/my-topic-0/00000000000000000000.log
Dumping /tmp/kafka-logs/my-topic-0/00000000000000000000.log
Starting offset: 0
baseOffset: 0 lastOffset: 0 count: 1 baseSequence: -1 lastSequence: -1
  producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0
  isTransactional: false isControl: false position: 0
  CreateTime: 1623034799990 size: 77 magic: 2
  compresscodec: NONE crc: 1773642166 isvalid: true
baseOffset: 1 lastOffset: 1 count: 1 baseSequence: -1 lastSequence: -1
  producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0
  isTransactional: false isControl: false position: 77
  CreateTime: 1623034803631 size: 82 magic: 2
  compresscodec: NONE crc: 1638234280 isvalid: true
baseOffset: 2 lastOffset: 2 count: 1 baseSequence: -1 lastSequence: -1
  producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0
  isTransactional: false isControl: false position: 159
  CreateTime: 1623034808233 size: 82 magic: 2
  compresscodec: NONE crc: 4143814684 isvalid: true
baseOffset: 3 lastOffset: 3 count: 1 baseSequence: -1 lastSequence: -1
  producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0
  isTransactional: false isControl: false position: 241
  CreateTime: 1623034811837 size: 77 magic: 2
  compresscodec: NONE crc: 3096928182 isvalid: true
#
```

在下一个例子中，我们提供了 `--print-data-log` 参数，这样就可以打印出消息的内容和其他更多的信息。

```
# kafka-dump-log.sh --files /tmp/kafka-logs/my-topic-0/00000000000000000000.log
--print-data-log
Dumping /tmp/kafka-logs/my-topic-0/00000000000000000000.log
Starting offset: 0
baseOffset: 0 lastOffset: 0 count: 1 baseSequence: -1 lastSequence: -1
  producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0
  isTransactional: false  isControl: false position: 0
  CreateTime: 1623034799990 size: 77 magic: 2
  compresscodec: NONE crc: 1773642166 isvalid: true
| offset: 0 CreateTime: 1623034799990 keysize: -1 valuesize: 9
    sequence: -1 headerKeys: [] payload: Message 1
baseOffset: 1 lastOffset: 1 count: 1 baseSequence: -1 lastSequence: -1
  producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0
  isTransactional: false isControl: false position: 77
  CreateTime: 1623034803631 size: 82 magic: 2
  compresscodec: NONE crc: 1638234280 isvalid: true
| offset: 1 CreateTime: 1623034803631 keysize: -1 valuesize: 14
    sequence: -1 headerKeys: [] payload: Test Message 2
baseOffset: 2 lastOffset: 2 count: 1 baseSequence: -1 lastSequence: -1
  producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0
  isTransactional: false isControl: false position: 159
  CreateTime: 1623034808233 size: 82 magic: 2
  compresscodec: NONE crc: 4143814684 isvalid: true
| offset: 2 CreateTime: 1623034808233 keysize: -1 valuesize: 14
    sequence: -1 headerKeys: [] payload: Test Message 3
baseOffset: 3 lastOffset: 3 count: 1 baseSequence: -1 lastSequence: -1
  producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0
  isTransactional: false isControl: false position: 241
  CreateTime: 1623034811837 size: 77 magic: 2
  compresscodec: NONE crc: 3096928182 isvalid: true
| offset: 3 CreateTime: 1623034811837 keysize: -1 valuesize: 9
    sequence: -1 headerKeys: [] payload: Message 4
#
```

这个工具还提供了其他一些有用的参数，比如验证日志片段的索引文件。索引用于查找日志片段中的消息，如果索引出现损坏，则会导致读取错误。如果 broker 在启动时处于“不干净”的状态（之前被非正常关闭），就会对索引文件进行验证。也可以手动进行索引验证。验证索引有两种方式，使用哪一种取决于你想验证到什么程度。如果指定了 `--index-sanity-check`，就会只检查索引是否处于可用状态；如果指定了 `--verify-index-only`，则会检查不匹配的索引，但不会打印出所有的索引条目。还有一个有用的选项是 `--value-decoder-class`，可以用它指定用于反序列化消息的解码器。

12.5.4 副本验证

分区复制的原理与普通的消费者客户端类似：跟随者 broker 会定期将上一个偏移量到当前偏移量之间的数据复制到磁盘上。如果复制过程停止，那么在重启之后将从上一个检查点继续复制。这个时候，如果之前复制的日志片段被删除，那么跟随者将不会填补这个缺口。

可以用 `kafka-replica-verification.sh` 来验证集群分区副本的一致性。它会从指定分区的副本读取消息，检查所有副本是否包含了相同的消息，并打印出指定分区的最大延迟。这个过程会一直重复，直到被取消。要使用这个工具，必须提供 broker 的连接串（如果有多个 broker 地址，就用逗号隔开）。在默认情况下，它会验证所有的主题。还可以用正则表达式匹配想要验证的主题。

副本验证对集群的影响

与分区重分配一样，副本验证也会对集群造成影响，因为它需要从最旧的位置读取所有消息，以此来验证数据的一致性。另外，它可以并行地从分区的多个副本读取消息，所以在进行副本验证时要非常小心。

例如，下面的例子会对 broker 1 和 broker 2（包含“my-topic”主题的分区 0）上以 my 开头的主题的副本进行验证。

```
# kafka-replica-verification.sh --broker-list kafka.host1.domain.com:
9092,kafka.host2.domain.com:9092
--topic-white-list 'my.*'

2021-06-07 03:28:21,829: verification process is started.
2021-06-07 03:28:51,949: max lag is 0 for partition my-topic-0 at offset 4
among 1 partitions
2021-06-07 03:29:22,039: max lag is 0 for partition my-topic-0 at offset 4
among 1 partitions
...
#
```

12.6 其他工具

Kafka 发行版还提供了一些其他工具，本书并没有深入介绍它们，但在管理 Kafka 集群时它们会非常有用。有关这些工具的更多信息可以在 Kafka 官方网站上找到。

客户端 ACL

`kafka-acls.sh` 是一个命令行工具，可用于设置 Kafka 客户端 ACL，包括所有的授权器属性、拒绝或允许策略的设置、集群或主题级别的限制、ZooKeeper TLS 文件配置等。

轻量级 MirrorMaker

一个轻量级的 `kafka-mirror-maker.sh` 脚本，可用于镜像数据。有关数据镜像的更多内容可以参考第 10 章。

测试工具

还有其他几个可用于测试或升级 Kafka 的脚本。当从一个 Kafka 版本升级到另一个版本时，可以用 `kafka-broker-api-versions.sh` 来识别不同版本的可用 API 元素和检查兼容性问题。生产者和消费者也有对应的性能测试脚本。还有几个脚本可用于管理 ZooKeeper。`trogdor.sh` 是一个测试框架，主要用于运行基准测试和为系统制造测试工作负载。

12.7 不安全的操作

虽然有一些操作在技术上是可行的，但如果不是在极端情况下则不建议执行。比如，你正在诊断一个问题，但已经没有其他可行的办法，或者你发现了一个 bug，需要一个临时修复方案。这些操作一般在文档中不会有相关的说明，而且不受支持，有可能会给应用程序带来风险。

本节会列举一些常见的操作，紧急情况下可以考虑将它们作为潜在的解决方案。不过，一般情况下不建议执行这些操作，或者在执行之前要经过慎重的考虑。

此处有危险

本节介绍的操作将涉及直接访问保存在 ZooKeeper 中的元数据。这些可能是非常危险的操作，所以要非常小心。最好不要直接修改 ZooKeeper 中的数据，除非有特别说明。

12.7.1 移动集群控制器

每个 Kafka 集群都有一个被指定为控制器的 broker。控制器有一个特殊的线程，除了处理普通的任务，它还负责监督集群操作。正常情况下，控制器选举是通过自动监听 ZooKeeper 临时节点来完成的。当一个控制器被关闭或变为不可用时，其他 broker 会尽快尝试让自己成为新控制器，因为一旦控制器被关闭，ZooKeeper 临时节点就会被删除。

有时候，在对问题集群或 broker 进行故障诊断时，需要在不关闭主机的情况下强制将控制器角色转移给另一个 broker。例如，当控制器遇到异常或其他问题时，虽然还在运行，但功能已经变得不正常。在这些情况下转移控制器通常不会有很高的风险，但这不是一个普通的操作，所以不应该经常这么做。

如果要强制转移控制器，则需要手动删除 /admin/controller 下的 ZooKeeper 节点，让当前控制器退出，而集群会随机选择一个新的控制器。目前，Kafka 还不支持直接将指定的 broker 作为控制器。

12.7.2 移除待删除的主题

在尝试删除一个主题时，broker 会请求 ZooKeeper 创建一个删除节点。当每一个副本都完成删除并进行了确认时，ZooKeeper 节点将被移除。在正常情况下，集群可以非常快地执行完这个过程。但是，有时候也会出问题。以下是删除请求可能被卡住的场景。

01. 请求者无法知道集群是否启用了主题删除功能，而要删除的主题所在的集群可能刚好禁用了主题删除功能。
02. 请求删除一个非常大的主题，但在处理这个请求之前，一个或多个副本集因发生硬件故障而离线，控制器无法对删除操作做出确认，也就无法完成删除操作。

要解决这个问题，需要先删除 /admin/delete_topic/<topic> 节点。删除主题的 ZooKeeper 节点（不是 /admin/delete_topic 父节点）等于删除了挂起的请求。如果控制器中有缓存的请求，它就会重新排队，所以在删除主题的节点后要立即强制转移控制器，确保控制器中没有缓存的请求。

12.7.3 手动删除主题

如果集群禁用了主题删除功能，或者需要通过非正式的途径删除某些主题，那么可以进行手动删除。但是，这需要完全关闭集群中所有的 broker，只要集群中有一个运行中的 broker，就无法执行这个操作。

先关闭 broker

在集群还在运行的时候修改 ZooKeeper 中的元数据是很危险的，这会造成集群不稳定。所以，在集群还在运行的时候不要删除或修改 ZooKeeper 中的元数据。

从集群中手动删除主题的过程如下。

01. 关闭集群中所有的 broker。
02. 删除 ZooKeeper 中的 /brokers/topics/ 路径，注意要先删除路径下的子节点。
03. 删除每一个 broker 的日志目录下的分区目录，这些目录的名字可能是 <topic>-<int>，其中 <int> 是分区的 ID。
04. 重启所有的 broker。

12.8 小结

运行一个 Kafka 集群需要付出很大的努力，为了让 Kafka 保持巅峰状态，需要做大量的配置和维护工作。本章介绍了 Kafka 的很多日常操作，比如经常会用到的主题管理和客户端配置。也介绍了一些用于诊断问题的复杂操作，比如检查日志片段。最后还介绍了一些不安全的操作，这些操作在特殊情况下可以帮你解决问题。通过执行这些操作，你就可以更好地管理 Kafka 集群。当你开始扩展 Kafka 集群规模时，这些工具会越来越难以满足你的需求。建议与 Kafka 开源社区合作，并利用这个生态系统中的其他开源项目来自动化集群的管理任务。

现在，我们已经有了管理集群所需的工具，但如果没有进行适当的监控，那么管理好集群仍然是一个不可能完成的任务。第 13 章将讨论如何监控 broker 和集群的健康状况，确保 Kafka 始终在正常的状态下运行。我们也会分享一些监控客户端（包括生产者和消费者）的最佳实践。

第 13 章　监控 Kafka

Kafka 应用程序提供了大量的指标，以至于很多人搞不清楚哪些是重要的，哪些可以置之不理。它们所涉及的范围很广，从简单的流量速率指标到各种详细的请求时间指标，再到主题级别的和分区级别的指标。它们为 broker 的每一种行为都提供了详细信息，但也成了 Kafka 集群管理员的“噩梦”。

本章将详细介绍一些常用的关键性指标，以及如何根据这些指标采取相应的行动。我们也会介绍一些用于调试问题的指标。不过，这不是一个完整的指标清单，因为指标清单经常会发生变化，并且很多指标只对经验丰富的 Kafka 开发人员有参考价值。

13.1 指标基础

在介绍 broker 和客户端的指标之前，先来了解一下有关 Java 应用程序监控的基础知识，以及监控和告警的一些最佳实践。它们有助于我们理解如何监控应用程序，以及为什么本章介绍的指标是最为重要的指标。

13.1.1 指标来自哪里

Kafka 提供的所有指标都可以通过 Java 管理扩展（Java management extension，JMX）接口访问。要在外部监控系统中使用这些指标，最简单的方法是将监控系统中负责收集指标的代理连接到 Kafka 上。代理可以是运行在监控系统中的一个独立进程，并用 Nagios XI `check_jmx` 插件或 `jmxtrans` 连接到 Kafka 的 JMX 接口上。你也可以直接在 Kafka 中运行一个 JMX 代理，然后通过 HTTP 连接（比如 Jolokia 或 MX4J）获取指标。

本章不会深入介绍如何设置监控代理，因为还有很多其他形式的代理可供选择。如果你所在的组织没有监控 Java 应用程序的经验，那么可以考虑使用第三方提供的监控服务。很多

第三方供应商的服务包已经包含了监控代理、指标收集点、存储、图形可视化和告警。它们还能进一步帮你搭建符合要求的监控代理。

找到 JMX 端口

JMX 端口是 broker 配置信息的一部分，保存在 ZooKeeper 中，如果监控系统要直接连到 Kafka 的 JMX 端口，那么可以先从 ZooKeeper 获取端口信息。`/brokers/ids/<ID>` 节点包含了 broker 的配置信息（JSON 格式），其中就有 `hostname` 和 `jmx_port`。不过需要注意的是，出于安全方面的考虑，Kafka 默认禁用了远程 JMX。如果要启用它，则必须保护好端口。这是因为 JMX 不仅可以作为查看应用程序状态的窗口，它还允许执行代码。强烈建议使用嵌入应用程序中的 JMX 指标代理。

非应用程序指标

并不是所有的指标都来自 Kafka。我们能够获取到的指标通常可以分为 5 类，表 13-1 列出了这 5 种类别。

表 13-1：指标来源

类别	描述
应用程序指标	这些是可以通过 JMX 接口获取到的 Kafka 指标
日志	来自 Kafka 的另一种监控数据。因为日志是文本或结构化数据，不仅仅是数字，所以需要进行额外的处理
基础设施指标	这些指标来自部署在 Kafka 前面的系统，这些系统也处在请求路径中，并在你的可控范围之内，比如负载均衡器就属于这一类
外部工具指标	这些数据来自 Kafka 以外的工具，类似于客户端，但在你的直接控制之下，它们的用途与普通的 Kafka 客户端不一样，比如 Kafka Monitor 就属于这一类
客户端指标	这些指标来自与集群相连的 Kafka 客户端

本章稍后将介绍 Kafka 日志和客户端指标，还会简要介绍一下外部工具指标。因为基础设施指标依赖于用户的特定环境，所以不在本书的讨论范围之内。随着不断深入使用 Kafka，你会发现这些指标对了解应用程序运行状况起到的作用越来越大。在上面的清单中，越是排在后面的指标提供的 Kafka 视图越客观。例如，一开始你可能只需要 broker 指标，后来则希望对应用程序运行状况有一个更客观的了解。一个常见的例子是监控网站的运行状况。假设 Web 服务器运行正常，所有指标都显示它没有问题。但是，Web 服务器和外部用户之间出现了网络问题，导致用户请求无法到达 Web 服务器。这个时候可以使用外部工具检查网站的可访问性，它运行在你的网络之外，可以检测到这一问题，并向你发送告警。

13.1.2 需要哪些指标

哪些指标对你来说更为重要？这个问题就好比哪一款编辑器对你来说是最好的。这很大程度上取决于你打算用它们做什么、你有哪些收集指标的工具、你使用 Kafka 多长时间以及你有多少时间可以用在构建 Kafka 基础设施上。一般来说，broker 开发人员对这些指标的需求与 SRE 的需求有很大的不同。

01. 告警还是调试

你需要问自己的第一个问题是，当 Kafka 出现问题时，你的主要目的是收到告警还是调试问题。答案通常是二者兼而有之。不管怎样，如果你知道哪些指标可用于告警、哪些指标可用于调试问题，那么在收集完之后就可以区别对待它们。

用于告警的指标在短时间内（通常不会比对问题做出响应所需的时间长很多）非常有用。你可以以小时或天为单位，将这些指标发送给能够对已知问题做出响应的自动化工具，或者如果没有自动化工具，则可以发送给运维人员。这些指标要尽量保持客观，因为不会影响客户端的问题远没有会影响客户的问题那么重要。

用于调试问题的数据需要保留较长的时间，因为你可能会频繁诊断已经存在一段时间的问题或对比较复杂的问题进行更深入的探究。这些数据在被收集到之后需要保留几天或几周。它们通常比较客观，或者是来自 Kafka 应用程序本身。需要注意的是，并不一定要将这些数据收集到监控系统中。如果主要目的是调试问题，那么在调试问题时能够获取它们就可以了。不需要持续不断地收集数以万计的指标，从而让监控系统不堪重负。

历史指标

你可能还需要第三种类型的数据，也就是应用程序的历史数据。通常，历史数据被用于容量管理，它们一般会包含资源的使用信息，比如计算资源、存储和网络。这些指标需要保存很长一段时间，一般以年为单位。你可能还需要收集额外的元数据，比如一个 broker 是何时加入或被移出集群的，这些元数据将作为指标的上下文信息。

02. 自动化工具还是人工介入

另一个需要考虑的问题是，指标的使用者是谁。如果指标被用在自动化系统中，那么它们就应该非常具体。我们可以收集大量的指标，并且每个指标都只描述一些小细节，这没有问题，因为这就是计算机存在的原因：处理大量的数据。指标越具体，就越容易对其进行自动化，因为越具体就越没有太多的解释空间。如果指标是给人类看的，那么太多的指标会让人类不堪重负。在基于指标定义告警时，这一点尤为重要。人们很容易陷入“告警疲劳”，因为告警太多，很难知道问题究竟有多严重。我们也很难为每个指标定义合理的阈值并持续更新它们。如果告警过多或经常误报，我们就会怀疑告警是否正确地反映了应用程序的状态。

以驾驶汽车为例。为了能够在汽车行驶过程中适当地调整空气与燃料的比例，计算机需要对空气密度、燃料、排气和发动机的运行细节做一些测量。这些测量指标对驾驶员来说太过复杂了。因此，我们设计了一个“检查发动机”指示灯。这个指示灯会告诉你车子出问题了，然后你可以通过其他方法找出究竟是什么问题。本章将介绍一些指标，这些指标能够提供最高的覆盖范围，以降低告警的复杂性。

13.1.3 应用程序健康检测

不管使用哪一种方式收集来自 Kafka 的指标，都要确保有其他更为简单的健康检测方式来监控应用程序的整体健康状况。这可以通过两种方式来实现。

- 使用外部进程来报告 broker 的运行状态（健康检测）。
- 在 broker 停止发送指标时发出告警（有时也叫作**过时指标**）。

虽然第二种方式也是可行的，但有时候很难区分是 broker 出了问题还是监控系统本身出了问题。

如果监控的是 broker，那么可以直接连接到它的外部端口（也就是客户端连接 broker 所使用的端口），看看是否可以获得响应；如果监控的是 Kafka 客户端，则会比较复杂，我们需要检查客户端进程是否处于运行状态或通过内部方法确定应用程序的健康状况。

13.2 服务级别目标

对基础设施服务（如 Kafka）来说，服务级别目标（service level objective，SLO）是一个非常重要的监控领域。服务供应商可以基于这些指标向客户宣称他们能够获得怎样的基础设施服务级别。客户希望将基础设施服务视为一个不透明的系统：他们不想也不需要了解它的内部原理，只需要了解他们正在使用的接口，并且知道这些接口会完成需要它们完成的事情。

13.2.1 服务级别定义

在讨论 Kafka 的 SLO 之前，需要先统一一下涉及的术语。你可能经常听到工程师、经理、高管等错误地使用“服务级别”这个术语，导致人们对他们正在谈论的东西困惑重重。

服务级别指标（service level indicator，SLI）是一种用于描述服务可靠性的指标。因为它与用户体验紧密相关，所以通常来说越客观越好。在一个处理请求的系统（如 Kafka）中，这个指标通常用正常事件数量与总事件数量之间的比率来表示，比如 Web 服务器在处理请求时返回 2xx、3xx 或 4xx 响应的比例。

服务级别目标（SLO）也叫作**服务级别阈值**（service level threshold，SLT），它将 SLI 与目标值组合在一起。服务级别目标的一种常见表示方法是使用数字 9（99.9% 就是 3 个 9），尽管这不是必需的。SLO 还需要包含一个时间窗口，通常以天为单位。例如，7 天内 99%发送给 Web 服务器的请求必须返回一个 2xx、3xx 或 4xx 响应。

服务级别协议（service level agreement，SLA）是服务供应商和客户之间的一种契约。它通常会包含几个 SLO，以及如何度量和报告它们、客户如何向服务供应商寻求支持、服务供应商如果不按照 SLA 执行将受到怎样的惩罚。例如，之前的 SLO 所对应的 SLA 可能是这样的：如果服务供应商在 SLO 服务期内暂停运营，那么就需要退还客户为此期间支付的所有费用。

运营级别协议

运营级别协议（operational level agreement，OLA）这个术语很少被用到。它描述了 SLA 整体交付过程中的多个内部服务或供应商之间的协议。它的目标是确保在日常运营中有实现 SLA 所需的活动在进行。

我们经常听到人们谈论 SLA，但他们并不知道自己谈论的其实是 SLO。虽然那些为付费客户提供服务的供应商可能与客户之间存在 SLA，但运营工程师很少会负责服务性能以外的事情。此外，那些只有内部用户（也就是将 Kafka 作为大型服务的内部数据基础设施）的供应商与内部用户之间通常不存在 SLA。但这并不影响你设定 SLO，因为这样可以减少用户对 Kafka 的行为做出假设。

13.2.2 哪些指标是好的 SLI

通常，需要使用外部工具或系统来收集适用于 SLI 的指标。SLO 可以用来描述你服务的用户的满意度，我们无法主观地得出这些指标。客户并不关心你是否认为你的服务运行正常，他们的体验才是关键。也就是说，基础设施指标没什么问题，外部工具指标也不错，但客户端指标可能是大多数 SLI 的最佳指标。

表 13-2 列出了在请求和响应系统以及数据存储系统中常见的 SLI。

表 13-2：SLI 的类型

类型	描述
可用性	客户端是否能够发起请求并获得响应？

类型	描述
延迟	返回响应的速度有多快？
质量	响应是否包含了正确的内容？
安全性	请求和响应是否得到适当的保护？是否经过了授权和加密？
吞吐量	客户端是否能够快速获取足够的数据？

客户总是想要更多

你的客户可能对某些 SLO 感兴趣，这些 SLO 对他们来说很重要，但不在你的控制范围之内。例如，他们可能关心发送给 Kafka 的数据的正确性或新鲜度。如果你不负责某些 SLO，就不要同意为它们提供支持，因为这样会导致你承担更多额外的工作。你可以让客户与对应的团队联系，在他们之间达成对这些额外需求的一致理解并建立协议。

通常来说，SLI 最好是以 SLO 阈值内的事件计数器为基础。也就是说，在理想情况下，可以单独检查每个事件是否满足 SLO 阈值。所以，分位数指标就不是好的 SLI，因为它们只会告诉你 90% 的事件低于给定的值，而你无法控制这个值。不过，在你还不确定该使用什么样的阈值时，可以按照区间来聚合事件（比如“小于 10 毫秒”“10~50 毫秒”“50~100 毫秒”等）。你可以获得 SLO 范围内的事件分布视图，并设置区间的边界，这些边界就是合理的 SLO 阈值。

13.2.3 将 SLO 用于告警

SLO 应该为你提供主要的告警。因为 SLO 是从客户的角度描述问题，而这些问题应该是你首先要关注的。一般来说，如果一个问题不会对你的客户造成影响，就不会重要到要在夜里把你叫醒的程度。SLO 还会告诉你一些你不知道该如何检测的问题，因为你以前可能从未遇到过。它们不会告诉你这些问题是什么，但会告诉你这些问题是存在的。

问题在于，我们很难直接将 SLO 作为告警。SLO 的时间范围通常较长（比如一周），因为我们希望提供给高层或客户的 SLO 是可重复查看的。此外，在发出 SLO 告警时，可能已经太迟了，因为你已经在 SLO 之外采取了行动。有些人用派生值来设置预警，但在告警中使用 SLO 的最佳方式是观察 SLO 的燃烧率。

假设你的 Kafka 集群每周会收到 100 万个请求，其中一个 SLO 要求 99.9% 的请求必须在 10 毫秒内发送第一个响应字节。也就是说，在一周内，你可能会有多达 1000 个响应速度较慢的请求，但一切都还算正常。通常，你每小时会看到一个这样的请求，一周大约有 168 个。你有一个表示 SLO 燃烧率的指标：在每周有 100 万个请求的情况下，一个请求的燃烧率为 0.1%/ 小时。

周二上午 10 点，这个指标变了，现在显示燃烧率是 0.4%/ 小时。虽然有点儿不妙，但仍然不是问题，因为在当周可能不会违反 SLO。你开了一个工单诊断问题，然后又回到了其他更重要的事情上。周三下午 2 点，燃烧率跳到了 2%/ 小时，系统向你发起告警。你知道，按照这样的速度，将会在周五午餐时间违反 SLO。于是，你放下手头的工作，开始诊断问题。大约 4 小时后，燃烧率回落到了 0.4%/ 小时，并在当周接下来的几天里一直保持在这个水平。通过观察燃烧率，你避免了违反当周的 SLO。

要了解更多有关在告警中使用 SLO 和燃烧率的内容，建议阅读 Betsy Beyer 等人合著的 *SiteReliabilityEngineering* 和 *TheSiteReliabilityWorkbook*。

13.3 broker 的指标

broker 提供了很多指标。它们大部分是底层的度量信息，是 Kafka 开发者为诊断特定问题或预计在未来调试问题时需要用到这些信息而添加的。broker 的每一项功能都有相应的指标，其中一些最为常见的指标为 Kafka 的日常运行提供了必要的信息。

示例：谁来看着看门人？

很多组织使用 Kafka 收集应用程序和系统的指标和日志，并将收集到的数据聚合到中心监控系统。这样可以很好地解耦应用程序和监控系统，但对 Kafka 来说存在一个问题。如果使用监控系统来监控 Kafka，那么当 Kafka 崩溃时，我们很可能无法感知到，因为流入监控系统的数据流也随之停止了。

这个问题有很多种解决方法。一种方法是，使用一个单独的监控系统来监控 Kafka，它不依赖 Kafka 提供的数据。另一种方法是，如果有多个数据中心，则可以将数据中心 A 的 Kafka 集群指标生成到数据中心 B 的 Kafka 集群中，反之亦然。不管怎样，你要确保 Kafka 的监控和告警不依赖 Kafka 本身。

本节将从介绍如何根据一些有用的指标诊断集群问题开始。本章将在后面部分详细介绍这些指标。我们不会列出所有的 broker 指标，但会列出那些在检测 broker 和集群运行状况时“必须要有”的指标。在介绍客户端指标之前，本节还会针对日志展开详细的讨论。

13.3.1 诊断集群问题

Kafka 集群一般会出现 3 类问题。

- 单个 broker 的问题
- 集群过载
- 控制器的问题

到目前为止，单个 broker 的问题是最容易诊断和应对的。这些问题会通过异常的集群指标显现出来，通常与缓慢或发生故障的存储设备或者计算资源限制有关。要检测这些问题，需要根据操作系统的指标监控每个服务器的可用性和存储设备的运行状态。

如果问题不在操作系统或硬件层面，那么很可能是 Kafka 集群负载不均衡导致的。尽管 Kafka 尽量保持集群数据能够均匀地分布在所有的 broker 上，但并不能保证访问这些数据的客户端是均匀分布的。Kafka 也无法检测出诸如热分区之类的问题。强烈建议使用外部工具来保持集群均衡，比如 Cruise Control，它会持续监控集群，并在集群不均衡时对分区进行再均衡。它还提供了很多其他的管理功能，比如添加和删除 broker。

示例：首选副本选举

在进一步诊断问题之前，需要确保最近进行过首选副本选举（参见第 12 章）。broker 在释放分区领导权（例如，当 broker 发生崩溃或被关闭时）之后不会自动收回（除非启用了自动首领再均衡），集群很容易会因此变得不均衡。首选副本选举既安全又容易运行，所以在诊断问题之前最好先执行一下，看看问题能不能得到解决。

集群过载也是一个很容易检测的问题。如果集群是均衡的，并且多个 broker 的请求延迟一直在增加或者请求处理器空闲率较低，那么说明集群 broker 处理流量的能力达到了极限。你可能会发现一个客户端改变了它的请求模式，正在给你的集群制造麻烦。但即使发生了这种情况，你可能也无法对客户端做些什么。你能做的要么是降低集群负载，要么是增加 broker。

集群控制器的问题较难诊断，它们通常属于 Kafka 本身的 bug。这些问题表现为 broker 元数据不同步、broker 看起来正常但副本是离线的，以及主题操作问题（比如创建主题的操作没能被正常执行）。如果一个集群问题让你抓狂，让你不禁觉得“这真的很奇怪”，那么很有可能是因为控制器做了一些不可预知的事情。监控控制器的方法并不多，不过，只要监控好活动控制器的计数和控制器队列大小，当出现问题时，它们也是很有用的。

13.3.2 非同步分区的艺术

在监控 Kafka 时，非同步分区是最为常见的监控指标。这个指标会告诉我们首领 broker 有多少个分区处于非同步状态。这个指标可以反映 Kafka 的很多内部问题，从 broker 发生崩溃到资源的过度消耗。因为这个指标可以说明很多问题，所以当它的值大于零时，就要想办法采取相应的行动。稍后本章将介绍更多用于诊断这类问题的指标。表 13-3 列出了非同步分区指标的详细信息。

表 13-3：指标和其对应的非同步分区

指标名称	非同步分区
JMX MBean	`kafka.server:type=ReplicaManager,name=UnderReplicatedPartitionis`
取值范围	非负整数

非同步分区告警陷阱

在本书第 1 版和很多技术会议讨论中，作者们都提到了将非同步分区（URP）指标作为主要的告警指标，因为它可以说明很多问题。这种做法存在大量问题，因为在大部分良性的情况下，URP 指标也会是非零值。也就是说，当有人操作 Kafka 集群时，你可能会收到错误的告警，从而导致真正的告警被忽略。你还需要掌握大量的知识才能理解这个指标要告诉你的是什么信息。因此，本书不再建议将 URP 用在告警中。应该使用基于 SLO 的告警来检测未知的问题。

如果集群中多个 broker 的非同步分区数量一直很稳定（保持不变），则说明集群中的某个 broker 已经离线了。整个集群的非同步分区数量等于离线 broker 的分区数量，而离线 broker 不会生成任何指标。在这种情况下，需要检查这个 broker 出了什么问题，并解决问题。通常可能是硬件问题，也可能是操作系统问题或 Java 的问题。

如果非同步分区的数量是波动的，或者虽然很稳定但没有 broker 离线，那么说明集群出现了性能问题。这类问题繁复多样，难以诊断，不过还是可以通过一些步骤来缩小问题的范围。第一步，先确认问题是与单个 broker 有关还是与整个集群有关。有时候这个也难有定论。如果非同步分区属于单个 broker（如下面的例子所示），那么这个 broker 就是问题的根源，因为错误显示其他 broker 无法从它那里复制消息。

如果多个 broker 出现了非同步分区，那么有可能是集群的问题，但也有可能是单个 broker 的问题。这时候有可能是因为一个 broker 无法从其他 broker 那里复制数据。要找出这个 broker，可以列出集群的所有非同步分区，并检查哪个 broker 出现在所有的非同步分区中。可以用 `kafka-topics.sh`（第 12 章已经详细介绍过）获取非同步分区清单。

例如，列出集群的非同步分区。

```
# kafka-topics.sh --bootstrap-server kafka1.example.com:9092/kafka-cluster-
--describe --under-replicated
    Topic: topicOne    Partition: 5    Leader: 1    Replicas: 1,2 Isr: 1
    Topic: topicOne    Partition: 6    Leader: 3    Replicas: 2,3 Isr: 3
    Topic: topicTwo    Partition: 3    Leader: 4    Replicas: 2,4 Isr: 4
    Topic: topicTwo    Partition: 7    Leader: 5    Replicas: 5,2 Isr: 5
    Topic: topicSix    Partition: 1    Leader: 3    Replicas: 2,3 Isr: 3
    Topic: topicSix    Partition: 2    Leader: 1    Replicas: 1,2 Isr: 1
```

```
    Topic: topicSix   Partition: 5    Leader: 6    Replicas: 2,6 Isr: 6
    Topic: topicSix   Partition: 7    Leader: 7    Replicas: 7,2 Isr: 7
    Topic: topicNine  Partition: 1    Leader: 1    Replicas: 1,2 Isr: 1
    Topic: topicNine  Partition: 3    Leader: 3    Replicas: 2,3 Isr: 3
    Topic: topicNine  Partition: 4    Leader: 3    Replicas: 3,2 Isr: 3
    Topic: topicNine  Partition: 7    Leader: 3    Replicas: 2,3 Isr: 3
    Topic: topicNine  Partition: 0    Leader: 3    Replicas: 2,3 Isr: 3
    Topic: topicNine  Partition: 5    Leader: 6    Replicas: 6,2 Isr: 6
#
```

在上面的示例中，broker 2 出现在了所有的副本里，这说明这个 broker 在复制消息时出了问题，所以要将注意力放在这个 broker 上。如果没有发现这样的 broker，那么问题有可能出在集群级别。

01. 集群级别的问题

集群一般会出现以下两类问题。

- 负载不均衡
- 资源过度消耗

虽然分区或首领的不均衡问题解决起来有点儿麻烦，但定位问题是很容易的。为了定位这个问题，需要用到 broker 的以下几个指标。

- 分区数量
- 首领分区数量
- 所有主题的消息流入速率
- 所有主题的字节流入速率
- 所有主题的字节流出速率

在一个均衡的集群中，这些指标的数值在整个集群范围内是均衡的，如表 13-4 所示。

表 13-4：资源使用情况指标

broker	分区	首领	消息流入	字节流入	字节流出
1	100	50	13 130msg/s	3.56MBps	9.45MBps
2	101	49	12 842msg/s	3.66MBps	9.25MBps
3	100	50	13 086msg/s	3.23MBps	9.82MBps

可以看出，所有的 broker 处理的流量几乎是相等的。假设在运行了首选副本选举之后，这些指标出现了很大的偏差，而这说明集群的流量出现了不均衡。要解决这个问题，需要将负载较重的 broker 分区移动到负载较轻的 broker 上，这可以使用第 12 章中介绍的 `kafka-reassign-partitions.sh` 来实现。

实现集群负载均衡的辅助工具

broker 本身无法自动实现整个集群的分区重分配。Kafka 集群的流量均衡是一个十分费劲的过程，需要手动检查一大堆指标，然后再进行副本重分配。一些组织开发了自动化工具来协助完成这个任务。例如，LinkedIn 发布了一个叫作 `kafka-assigner` 的工具，该工具可以在 LinkedIn 的 GitHub 代码仓库 kafka-tools 中找到。一些供应商提供的 Kafka 服务也包含了这一功能。

Kafka 集群的另一个性能问题是请求数量超过了 broker 的处理能力。有很多潜在的瓶颈会拖慢整个集群：CPU、磁盘 IO 和网络吞吐量是其中最为常见的。磁盘使用率不在其列，因为 broker 会一直运行，直到磁盘被填满然后“挂掉”。为了诊断这类问题，可以监控以下这些操作系统级别的指标。

- CPU 使用
- 网络输入吞吐量
- 网络输出吞吐量

- 磁盘平均等待时间
- 磁盘使用百分比

上述任何一种资源出现过度消耗都会表现为分区不同步。需要注意的是，broker 的复制过程与其他 Kafka 客户端做的事情是一样的。如果集群的数据复制出了问题，那么集群的用户在发送或读取消息时也会有问题。所以，有必要为这些指标定义一个基线，并设定相应的阈值，以便在容量告急之前就能定位到正在发生的问题。随着集群流量的增长，也有必要对这些指标的走势进行持续观察。在这些指标中，“所有主题的字节流入速率”是查看集群资源使用情况的一个最好的指标。

02. 主机级别的问题

如果性能问题不是出现在集群级别，而是出现在一两个 broker 中，就要检查一下 broker 所在的主机，看看是什么导致它与集群中的其他 broker 不一样。主机级别的问题可以分为以下几类。

- 硬件故障
- 网络
- 进程冲突
- 本地配置不一致

典型的服务器和典型的问题

一台服务器和它的操作系统承载了数千个组件，十分复杂，任何一个组件都可能出问题，从而导致整个系统崩溃或部分性能衰退。本书不可能覆盖所有的故障场景，关于这个话题的图书已经有很多了。不过，本书会讨论一些最为常见的问题。本节将着重讨论在装有 Linux 操作系统的服务器上可能发生的问题。

有时候，硬件问题表现得非常明显，比如服务器直接停止工作，而那些引起性能衰退的硬件问题就不那么明显了。当出现这类问题时，系统仍然在运行，只是性能下降了。例如，内存出现了坏点，系统虽然可以检测到，但选择直接跳过（可用的内存变少了）。类似的问题也会发生在 CPU 上。对于这类问题，可以使用硬件工具 [比如智能平台管理接口（IPMI）] 来监控硬件的健康状况。当问题出现时，可以通过 `dmesg` 查看输出到系统控制台的内核缓冲区日志。

磁盘故障是导致 Kafka 性能衰退的一个比较常见的硬件问题。Kafka 使用磁盘来存储消息，生产者的性能与磁盘的写入速度有直接的关系。这里出现的任何偏差都会表现为生产者和消费者的性能问题，消费者的性能问题会导致分区不同步。因此，应该持续地监控磁盘，并在出现问题时马上进行修复。

一粒老鼠屎

一个 broker 的磁盘问题可能会影响到整个集群的性能。这是因为生产者客户端会连接到所有包含了主题首领分区的 broker。如果你已经遵循了最佳实践，那么分区的首领几乎能够均衡地分布在整个集群中。如果一个 broker 的性能出现衰退并拖慢了请求的处理速度，就会导致生产者开启回压，从而拖慢发给所有 broker 的请求。

假设你正在通过 IPMI 或其他硬件管理接口监控磁盘状态，同时还在操作系统中运行了 SMART（self-monitoring, analysis and reporting technology，自行监控、分析和报告技术），以便对磁盘进行持续的监控和测试。在故障即将发生时，它们会发出告警。除此之外，还要注意查看磁盘控制器，特别是如果启用了 RAID（不管是否使用了 RAID 硬件都要注意）。很多磁盘控制器有板载缓存，这个缓存只在控制器和电池备份单元（BBU）正常时才会被使用。如果 BBU 发生故障，那么缓存就会被禁用，磁盘性能就会衰退。

在网络方面，局部故障也会造成很大的问题。有些问题是硬件引起的，比如糟糕的光缆或连接器。有些问题是配置不当造成的，比如修改了服务器或上游网络硬件的网络连接速度或双工设置。网络配置

问题还有可能在操作系统中出现，比如网络缓冲区太小，或者太多的网络连接占用了大量的系统内存。在这方面，网络接口报告的错误数量是一个最为关键的指标。如果这个数字一直在增长，则说明出现了亟待解决的硬件问题。

如果硬件没有问题，那么就需要注意系统中的其他应用程序，因为它们也在消耗硬件资源，而且有可能给 broker 造成压力。它们可能是没有被正确安装的软件，或者是一个运行有问题的进程，比如监控代理。这个时候，可以用操作系统提供的工具（如 `top`）来识别那些过度消耗 CPU 或内存的进程。

如果经过上述检查还是找不出问题根源，则可以检查一下 broker 或系统的配置，看看是否存在配置不一致的问题。一台服务器上可能运行着很多个应用程序，每个应用程序又有很多个配置参数，要找出它们的差别真是一项艰巨的任务。这就是为什么要使用配置管理工具（比如 Chef 或 Puppet）来保持操作系统和应用程序（包括 Kafka）的配置一致性。

13.3.3 broker 指标

除了非同步分区的数量，还要监控其他很多 broker 级别的指标。虽然不一定会对所有的指标设定告警阈值，但它们确实为我们提供了有价值的信息。你创建的任何一个监控仪表盘都应该包含这些指标。

01. 活跃控制器数量

活跃控制器数量指标可以告诉我们一个 broker 是否就是当前的集群控制器。指标的值可以是 0 或者 1，如果是 1，那么表示这个 broker 就是当前的控制器。在任何时候，集群都应该只有一个控制器。如果出现了两个控制器，则说明有一个本该退出的控制器线程被阻塞了。这会导致一些管理操作（比如移动分区）无法正常执行。要解决这个问题，需要重启这两个 broker。当集群中出现两个控制器时，可能无法通过正常的方式安全重启 broker，所以要强制关闭它们。表 13-5 列出了活跃控制器数量指标的详细信息。

表 13-5：活跃控制器数量指标

指标名称	活跃控制器数量
JMX MBean	`kafka.controller:type=KafkaController,name=ActiveControllerCount`
取值范围	0 或 1

如果集群中没有控制器，就无法对状态变化（比如创建主题或分区或者 broker 故障）做出正确的响应。这个时候，要仔细检查为什么控制器线程出现了异常。例如，ZooKeeper 集群的网络分区就会导致这个问题。在解决了这些底层问题之后，最好重启所有的 broker，以便重置控制器线程的状态。

02. 控制器队列大小

控制器队列大小指标会告诉我们控制器当前有多少个等待处理的请求，它的值可以是 0 或正整数。因为新请求不断在涌入，管理操作（比如创建分区、移动分区和首领变更）也不断在发生，所以这个指标的值会频繁波动。这个指标会出现可预料的峰值，但如果持续增长，或保持在一个较高的位置不下降，则说明控制器可能被卡住了。这可能导致管理任务无法被正常执行。要解决这个问题，需要关闭当前的控制器，并将控制器角色交给另一个 broker。但是，当控制器被卡住时，我们无法通过正常的方式关闭它。表 13-6 列出了控制器队列大小指标的详细信息。

表 13-6：控制器队列大小指标

指标名称	控制器队列大小
JMX MBean	`kafka.controller:type=ControllerEventManager,name=EventQueueSize`
取值范围	非负整数

03. 请求处理器空闲率

Kafka 使用两个线程池来处理客户端请求：**网络线程**和**请求处理线程**（也叫 **IO 线程**）。网络线程负责通过网络读入数据和写出数据。这里不涉及太多的处理工作，所以不用太过担心这些线程会被耗尽。请求处理线程负责处理来自客户端的请求，包括从磁盘读取消息和向磁盘写入消息。因此，broker 负载的增长对这个线程池有很大的影响。表 13-7 列出了请求处理器空闲率指标的详细信息。

表 13-7：请求处理器空闲率指标

指标名称	请求处理器平均空闲百分比
JMX MBean	`kafka.server:type=KafkaRequestHandlerPool,name=RequestHandlerAvgIdlePercent`
取值范围	从 0 到 1 的浮点数（包括 1 在内）

智能地使用线程

似乎我们需要数百个请求处理线程，但在实际当中，请求处理线程数不需要超过 broker 的 CPU 核数。Kafka 在使用请求处理线程方面是非常智能的，它会将需要很长时间才能处理完的请求放到缓冲区（“炼狱”）中。例如，当请求的配额被限定或生产请求需要得到多个副本的确认时，Kafka 就会这么做。

请求处理器平均空闲百分比这个指标的数值越低，说明 broker 的负载越高。经验表明，如果空闲百分比低于 20%，那么说明存在潜在的问题；如果低于 10%，则说明出现了性能问题。除了集群的规模太小，还有其他两个方面的原因会导致这个线程池被重度使用。一个原因是线程池里没有足够的线程。一般来说，请求处理线程的数量应该等于系统的处理器核数（包括多线程处理器）。

另一个常见的原因是线程做了不该做的事。在 Kafka 0.10 之前，请求处理线程负责解压缩消息批次、验证消息、分配偏移量，并在写入磁盘之前重新压缩消息。更糟糕的是，压缩方法使用了同步锁。从 0.10 版本开始，Kafka 引入了一种新的消息格式，允许在消息批次里使用相对偏移量。新版生产者可以在发送消息批次之前设置相对偏移量，这样 broker 就可以避免解压缩和重新压缩消息批次了。如果使用了支持 0.10 版本消息格式的生产者客户端和消费者客户端，并把 broker 的消息格式也升级到了 0.10 版本，你就会发现性能有了显著的改进，并减少了对请求处理线程的消耗。

04. 主题流入字节

主题流入字节速率使用字节 / 秒来表示，用于度量 broker 从生产者客户端接收到的消息流量。可以基于这个指标决定何时对集群进行扩展或开展其他与流量增长相关的工作。这个指标也可用于评估一个 broker 是否比集群中的其他 broker 接收了更多的流量，如果出现了这种情况，则需要对分区进行再均衡。表 13-8 列出了这个指标的详细信息。

表 13-8：主题流入字节指标

指标名称	每秒流入字节
JMX MBean	`kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec`
取值范围	速率为双精度浮点数，计数为整数

因为这是我们介绍的第一个速率指标，所以有必要对它的属性进行简短的描述。所有的速率指标都有 7 个属性，使用哪些属性取决于你想获得哪方面的信息。有些属性是离散的事件计数，有些是基于时间段的事件平均数。务必在合适的地方使用合适的指标，否则你将得到一个错误的 broker 视图。

前两个属性不是度量值，但有助于你理解这些速率指标。

`EventType`

速率属性的度量单位，在这里是“字节”。

`RateUnit`

速率的时间段，在这里是“秒”。

这两个属性表明，速率是通过字节 / 秒来表示的，不管它的值是基于多长时间段计算出来的平均值。速率还有其他 4 个不同粒度的属性。

`OneMinuteRate`

前 1 分钟的平均值。

`FiveMinuteRate`

前 5 分钟的平均值。

`FifteenMinuteRate`

前 15 分钟的平均值。

`MeanRate`

从 broker 启动到现在的平均值。

`OneMinuteRate` 波动很快，提供了“时间点”粒度的度量，适合用于观察短期的流量走势。`MeanRate` 一般不会有太大变化，其提供了整体的流量走势视图。虽然 `MeanRate` 也有一定的作用，但一般不需要对它设置告警。`FiveMinuteRate` 和 `FifteenMinuteRate` 是前面两种属性的折中。

除了速率属性，速率指标还有一个 `Count` 属性，它在 broker 启动之后就一直保持增长。`Count` 表示从 broker 启动以来接收的流量的字节总数。将这个属性用在一个支持计数指标的监控系统中，就可以获得完整的度量视图（我们看到的不仅仅是平均速率）。

05. 主题流出字节

主题流出字节速率是另一个与流量增长有关的指标，与流入字节速率类似。流出字节速率也就是消费者从 broker 读取消息的速率。流出速率的扩展方式与流入速率不一样。这是因为 Kafka 支持多消费者客户端。很多 Kafka 集群的流出速率可以达到流入速率的 6 倍！这就是为什么要单独对流出速率进行观察和分析。表 13-9 列出了这个指标的详细信息。

表 13-9：主题流出字节指标

指标名称	每秒流出字节
JMX MBean	`kafka.server:type=BrokerTopicMetrics,name=BytesOutPerSec`
取值范围	速率为双精度浮点数，计数为整数

将复制消费者也包括在内

流出速率也包含了副本的流量。也就是说，如果所有主题的复制系数都是 2，那么在没有消费者客户端的情况下，流出速率等于流入速率。如果有一个消费者客户端读取了集群所有的消息，那么流出速率就是流入速率的两倍。如果不知道这一点，那么光是看这些指标你可能会感到疑惑。

06. 主题流入消息

前面介绍的字节速率使用字节来表示 broker　流量，而**消息**速率则使用每秒生成消息条数（不考虑消息的大小）来表示流量。这也是一个很有用的生产者流量增长指标。它还可以与字节速率结合在一起使用，用于计算消息的平均大小。与字节速率一样，这个指标也能反映出集群流量是否均衡。表 13-10 列出了这个指标的详细信息。

表 13-10：主题流入消息指标

指标名称	每秒流入消息
JMX MBean	kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec
取值范围	速率为双精度浮点数，计数为整数

为什么没有消息流出速率？

经常会有人问，为什么没有 broker 的消息流出指标？这是因为在消息被读取时，broker 会将整个消息批次发送给消费者，并不会展开计算每个批次包含了多少条消息。所以，broker 也就不知道发送了多少条消息。broker 提供了一个每秒获取请求次数指标，不过它是一种请求速率，不是消息计数。

07. 分区数量

broker 的**分区数量**就是分配给 broker 的分区总数，一般不会经常发生变化。它包括 broker 的每一个分区副本，不管是首领副本还是跟随者副本。如果一个集群启用了自动创建主题的功能，那么监控这个指标就会很有趣，因为主题的创建不受集群管理员的控制。表 13-11 列出了这个指标的详细信息。

表 13-11：分区数量指标

指标名称	分区数量
JMX MBean	kafka.server:type=ReplicaManager,name=PartitionCount
取值范围	非负整数

08. 首领数量

首领数量指标表示一个 broker 的首领分区数量。与 broker 的其他指标一样，这个指标也应该在整个集群的 broker 之间保持均衡。我们需要定期检查这个指标，并适时地发出告警，因为即使在副本的数量和大小看起来都很均衡的时候，它仍然能够显示出集群的不均衡。broker 有可能会因为各种原因释放掉一个分区的领导权（比如 ZooKeeper 会话过期），但在会话恢复之后并不会自动收回领导权（除非启用了自动首领再均衡）。在这些情况下，这个指标会显示较少的首领分区数，或通常为零。这时候需要进行一次首选副本选举，重新均衡集群的首领。表 13-12 列出了这个指标的详细信息。

表 13-12：首领数量指标

指标名称	首领数量
JMX MBean	kafka.server:type=ReplicaManager,name=LeaderCount
取值范围	非负整数

可以将这个指标与分区数量指标结合在一起，计算出以该 broker 作为首领的分区的百分比。一个均衡的集群，如果它的复制系数是 2，那么所有的 broker 都应该差不多是 50% 分区的首领。如果复制系

数是 3，则这个百分比应该降到 33%。

09. 离线分区数量

与非同步分区数量一样，**离线分区数量**也是一个关键的监控指标（参见表 13-13）。只有集群控制器会提供这个指标（其他 broker 提供的值为零），它会告诉我们集群中有多少个分区是没有首领的。以下两个原因会导致分区没有首领。

- 包含这个分区副本的所有 broker 都关闭了。
- 由于消息数量不匹配，没有同步副本能够获得领导权（并禁用了不彻底首领选举）。

表 13-13：离线分区数量指标

指标名称	离线分区数量
JMX MBean	`kafka.controller:type=KafkaController,name=OfflinePartiionsCount`
取值范围	非负整数

在生产环境中，离线分区会影响生产者客户端，导致消息丢失或应用程序回压。这属于“站点停机”问题，需要立即解决。

10. 请求指标

第 6 章介绍过 Kafka 协议，它有多种请求，每种请求都有相应的指标。截至 Kafka 2.5.0，表 13-14 中的请求类型都有相应的指标。

表 13-14：请求指标的名字

`AddOffsetsToTxn`	`AddPartitionsToTxn`	`AlterConfigs`
`AlterPartitionReassignments`	`AlterReplicaLogDirs`	`ApiVersions`
`ControlledShutdown`	`CreateAcls`	`CreateDelegationToken`
`CreatePartitions`	`CreateTopics`	`DeleteAcls`
`DeleteGroups`	`DeleteRecords`	`DeleteTopics`
`DescribeAcls`	`DescribeConfigs`	`DescribeDelegationToken`
`DescribeGroups`	`DescribeLogDirs`	`ElectLeaders`
`EndTxn`	`ExpireDelegationToken`	`Fetch`
`FetchConsumer`	`FetchFollower`	`FindCoordinator`
`Heartbeat`	`IncrementalAlterConfigs`	`InitProducerId`
`JoinGroup`	`LeaderAndIsr`	`LeaveGroup`
`ListGroups`	`ListOffsets`	`ListPartitionReassignments`
`Metadata`	`OffsetCommit`	`OffsetDelete`
`OffsetFetch`	`OffsetsForLeaderEpoch`	`Produce`
`RenewDelegationToken`	`SaslAuthenticate`	`SaslHandshake`
`StopReplica`	`SyncGroup`	`TxnOffsetCommit`
`UpdateMetadata`	`WriteTxnMarkers`	

每一种请求类型都有 8 个指标，它们分别提供了不同处理阶段的细节。例如，`Fetch` 请求有如表 13-15 所示的指标。

表 13-15：`Fetch` 请求指标

名字	JMX MBean
总时间	`kafka.network:type=RequestMetrics,name=TotalTimeMs,request=Fetch`

请求队列时间	kafka.network:type=RequestMetrics,name=RequestQueueTimeMs,request=Fetch
本地时间	kafka.network:type=RequestMetrics,name=LocalTimeMs,request=Fetch
远程时间	kafka.network:type=RequestMetrics,name=RemoteTimeMs,request=Fetch
节流时间	kafka.network:type=RequestMetrics,name=ThrottleTimeMs,request=Fetch
响应队列时间	kafka.network:type=RequestMetrics,name=ResponseQueueTimeMs,request=Fetch
响应发送时间	kafka.network:type=RequestMetrics,name=ResponseSendTimeMs,request=Fetch
每秒请求数	kafka.network:type=RequestMetrics,name=RequestsPerSec,request=Fetch

每秒请求数是一个速率指标，也就是在单位时间内接收并处理的请求个数。这个指标可以让我们知道每种请求类型的请求频率，不过需要注意的是，很多请求类型并不会频繁发生，比如 `StopReplica` 和 `UpdateMetadata`。

与速率指标类似，7 个**时间**指标分别提供了一组百分比属性和一个离散的计数属性。这些指标都是从 broker 启动之后累计计算得出的。broker 运行的时间越长，这些指标就越稳定。它们所代表的含义如下。

总时间

表示 broker 花在处理请求（从收到请求到将响应返回给请求者）上的时间。

请求队列时间

表示请求停留在队列里（从收到请求到开始处理请求）的时间。

本地时间

表示首领分区花在处理请求上的时间，包括把消息写入磁盘（但不一定要冲刷）。

远程时间

表示在请求处理完毕之前，用于等待跟随者的时间。

节流时间

表示暂停返回响应的时间，这样做是为了降低请求者的速率，把客户端限定在配额范围内。

响应队列时间

表示响应被发送给请求者之前停留在队列里的时间。

响应发送时间

表示实际用于发送响应的时间。

每个指标的属性如下。

Count

从 broker 启动到目前为止已处理的请求数量。

Min

所有请求的最小值。

Max

所有请求的最大值。

Mean

所有请求的平均值。

StdDev

整体请求时间标准偏差。

Percentiles

`50thPercentile`、`75thPercentile`、`95thPercentile`、`98thPercentile`、`99thPercentile`、`999thPercentile`。

什么是百分位？

百分位是一种常见的时间度量方式。`99` 百分位表示整组样本（这里指请求时间）中有 `99%` 的值小于指标，那么就有 `1%` 的值大于指标。一般情况下，需要查看平均值和 `99%` 或 `99.9%` 的值，这样就可以知道平均请求处理情况和异常值。

在这些指标和属性中，哪些对监控来说是比较重要的？对于每一种请求类型，至少需要收集总时间指标的平均值和较高的百分位（99% 或 99.9%），以及每秒请求次数，这样就可以知道 broker 处理请求的整体性能。如果有可能，那么也尽量收集每种请求类型的其他 6 种时间指标，这样就可以将性能问题细分到请求的各个阶段。

为这些时间指标设定告警阈值有一定的难度。例如，`Fetch` 请求时间会受到各种因素的影响，包括客户端等待消息的时间、主题的繁忙程度，以及客户端和 broker 之间的网络连接速度。一般来说，可以为总时间指标设定 99.9 百分位基线并设置告警，对 `Produce` 请求类型来说更是如此。与非同步分区类似，`Produce` 请求的 99.9 百分位快速增长说明集群出现了大范围的性能问题。

13.3.4 主题的指标和分区的指标

除了用于描述 broker 一般行为的指标，还有很多主题级别的指标和分区级别的指标。在较大的集群中，这样的指标很多，一般来说不太可能将它们全部收集到一个指标系统中，但在调试与客户端相关的问题时，它们会很有用。例如，主题指标可用于识别造成集群流量大量增长的主题。我们需要提供这些指标，并将它们提供给 Kafka 生产者和消费者。不管你是否能够定期收集这些指标，都有必要知道它们的用处。

表 13-16 所列的指标将使用主题名称 *`TOPICNAME`* 和分区 0 作为示例。在实际当中，要把主题名称和分区 ID 替换成真实的主题名称和分区 ID。

表 13-16：个体主题指标

名字	JMX MBean
字节流入速率	`kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec,topic=`*`TOPICNAME`*
字节流出速率	`kafka.server:type=BrokerTopicMetrics,name=BytesOutPerSec,topic=`*`TOPICNAME`*
失败的获取速率	`kafka.server:type=BrokerTopicMetrics,name=FailedFetchRequestsPerSec,topic=`*`TOPICNAME`*
失败的生产速率	`kafka.server:type=BrokerTopicMetrics,name=FailedProduceRequestsPerSec,topic=`*`TOPICNAME`*
消息流入速率	`kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec,topic=`*`TOPICNAME`*

名字	JMX MBean
获取请求速率	kafka.server:type=BrokerTopicMetrics,name=TotalFetchRequestsPerSec,topic=*TOPICNAME*
生产请求速率	kafka.server:type=BrokerTopicMetrics,name=TotalProduceRequestsPerSec,topic=*TOPICNAME*

01. 个体主题指标

个体主题指标与前面介绍的 broker 指标非常相似。事实上，它们之间唯一的区别在于，个体主题指标指定了特定的主题名称，也就是说，这些指标属于某个特定的主题。因为个体主题指标的数量可能非常巨大（取决于集群主题的数量），所以我们极有可能不会监控它们或为它们设置告警。它们通常会被提供给客户端，客户端将用它们来评估和调试自己对 Kafka 的使用。

02. 个体分区指标

个体分区指标不如个体主题指标有用。另外，它们的数量更加庞大，因为几百个主题可能包含数千个分区。不过，在某些情况下，它们还是有一定用处的，特别是用于表示分区当前在磁盘上保留的数据量（参见表 13-17）的分区大小指标。如果把它们组合在一起，则可以知道主题保留了多少数据。如果同一主题的两个分区大小存在差异，则说明消息分布不均衡。日志片段数量指标表示分区保存在磁盘上的日志片段文件的数量。它可以与分区大小指标结合起来使用，用于跟踪磁盘资源的使用情况。

表 13-17：个体分区指标

名称	JMX MBean
分区大小	kafka.log:type=Log,name=Size,topic=*TOPICNAME*,partition=0
日志片段数量	kafka.log:type=Log,name=NumLogSegments,topic=*TOPICNAME*,partition=0
日志结束偏移量	kafka.log:type=Log,name=LogEndOffset,topic=*TOPICNAME*,partition=0
日志起始偏移量	kafka.log:type=Log,name=LogStartOffset,topic=*TOPICNAME*,partition=0

日志结束偏移量和日志起始偏移量这两个指标分别表示消息在分区中的最大偏移量和最小偏移量。然而，需要注意的是，不能用这两个指标来推算分区包含的消息数量，因为在压实日志时，具有相同键的新消息会覆盖掉旧消息，从而导致偏移量“丢失”。但它们在某些情况下还是很有用的，比如在进行时间戳和偏移量映射时，就可以得到更精确的映射，消费者客户端可以很容易地回滚到与某个时间点对应的偏移量。（不过 Kafka 0.10.1 引入了基于时间的索引搜索，所以这些指标就显得没有那么重要了。）

非同步分区指标

在个体分区指标中，有一个指标用于表示分区是否处于非同步状态。一般来说，这个指标对日常运维来说不是很有用，因为我们需要为此收集和查看非常多的指标。更为简单的做法是直接监控 broker 的非同步分区数量，然后用命令行工具（参见第 12 章）确定哪些分区处于非同步状态。

13.3.5 Java 虚拟机监控

除了 broker 的指标，还需要监控服务器的指标和 Java 虚拟机（JVM）的指标。JVM 频繁发生垃圾回收会影响 broker 的性能，这个时候应该收到告警。JVM 的指标还能告诉我们为什么 broker 下游组件的指标会发生变化。

01. 垃圾回收

在 JVM 方面，最需要监控的是垃圾回收（GC）状态。需要监控哪些 MBean 取决于你使用的 Java 运行时（JRE）和垃圾回收器配置。如果你使用的是 Oracle Java 1.8 和 G1 垃圾回收器，那么需要监控的 MBean 如表 13-18 所示。

表 13-18：G1 垃圾回收器指标

名称	JMX MBean
完全垃圾回收周期	`java.lang:type=GarbageCollector,name=G1 Old Generation`
年轻代垃圾回收周期	`java.lang:type=GarbageCollector,name=G1 Young Generation`

需要注意的是，在垃圾回收语义中，“Old”和“Full”代表的是一个意思。对于上述的两个指标，我们需要收集 `CollectionCount` 和 `CollectionTime` 这两个属性。`CollectionCount` 表示从 JVM 启动开始算起的垃圾回收次数，`CollectionTime` 表示从 JVM 启动开始算起的垃圾回收时间（以毫秒为单位）。因为这些属性都是计数值，所以它们可以告诉我们单位时间内发生垃圾回收的绝对次数和时间。还可以用它们计算出平均每次垃圾回收花费的时间，尽管这在日常运维中没有多大用处。

这些指标还有一个 `LastGcInfo` 属性。这是由 5 个属性组成的组合值，为我们提供了最后一次垃圾回收的信息。5 个属性中最重要的一个属性是 `duration`（以毫秒为单位），表示最后一次垃圾回收花费了多长时间。其他几个属性（`GcThreadCount`、`id`、`startTime` 和 `endTime`）虽然也会提供一些信息，但用处不大。不过需要注意的是，我们无法通过这些属性查看到每一次垃圾回收的信息，因为年轻代垃圾回收发生得非常频繁。

02. **Java 操作系统监控**

JVM 通过 `java.lang:type=OperatingSystem` 这个 MBean 提供了有关操作系统的一些信息，但这些信息很有限，并不能告诉我们操作系统的所有情况。有两个比较有用但在操作系统层面难以收集到的属性是 `MaxFileDescriptorCount` 和 `OpenFileDescriptorCount`。`MaxFileDescriptorCount` 告诉我们 JVM 最多能打开多少个文件描述符（FD），`OpenFileDescriptorCount` 告诉我们目前已经打开了多少个文件描述符。每个日志片段和网络连接都需要一个文件描述符，所以它们的数量增长得很快。如果网络连接不能被正常关闭，那么 broker 很快就会把文件描述符耗尽。

13.3.6 操作系统监控

JVM 并不能告诉我们所有与操作系统相关的信息。因此，我们不仅要收集 broker 的指标，也需要收集操作系统的指标。大多数监控系统会提供代理，这些代理将收集有关操作系统的信息。我们需要收集的信息包括 CPU 的使用情况、内存的使用情况、磁盘的使用情况、磁盘 IO 和网络的使用情况。

在 CPU 方面，至少需要监控系统平均负载。系统负载是一个数字，表示处理器的相对使用率。另外，按照使用类型收集到的 CPU 使用百分比信息也很有用。根据收集方法和操作系统的不同，我们可以得到如下这些 CPU 百分比细分（使用了缩写）中的一部分或全部。

us

用户空间使用的时间。

sy

内核空间使用的时间。

ni

低优先级进程使用的时间。

id

空闲时间。

wa

等待（磁盘）时间。

hi

处理硬件中断的时间。

si

处理软件中断的时间。

st

等待监视器的时间。

什么是系统负载？

尽管很多人知道系统负载是对 CPU 使用情况的度量，但大多数人误解了度量的方法。平均负载是指等待处理器执行的可执行进程数。Linux 系统中还有一些处于不可中断睡眠状态的进程，比如磁盘等待。负载用 3 个数值来表示，分别是前 1 分钟的平均数、前 5 分钟的平均数和前 15 分钟的平均数。在单 CPU 系统中，数值 1 表示系统负载达到了 100%。此时总是会存在一个等待执行的进程。在多 CPU 系统中，如果负载达到 100%，那么它的数值就等于 CPU 核数。如果系统的 CPU 的核数是 24，那么负载达到 100% 时它的数值就是 24。

跟踪 broker 的 CPU 使用情况是很有必要的，因为它们在处理请求时占用了大量的 CPU 时间。内存使用情况的跟踪就显得没那么重要了，因为运行 Kafka 并不需要太大的内存。broker 会使用堆外的一小部分内存来压缩数据，剩下的大部分会被用作缓存。尽管如此，还是要跟踪内存的使用情况，确保 broker 的内存不会受到其他应用程序的影响。可以通过监控总内存空间和可用交换内存空间来确保内存交换空间没有被其他应用程序占用。

对 Kafka 来说，磁盘是最重要的子系统。所有的消息都保存在磁盘上，所以 Kafka 的性能很大程度上依赖于磁盘的性能。我们需要监控磁盘空间和索引节点（**索引节点**是 Unix 文件系统中文件和目录的元数据对象），确保磁盘空间不会被用光。对保存数据的分区来说更是如此。还需要监控磁盘 IO 的统计数据，它们可以告诉我们磁盘是否被有效利用。至少，我们要监控磁盘的每秒读写速度、平均读写队列大小、平均等待时间和磁盘使用百分比。

最后，还需要监控 broker 的网络使用情况。简单地说，就是流入和流出的网络流量，一般用每秒多少位来表示。需要注意的是，在没有消费者的情况下，流入一个位就有复制系数个位流出，如果再算上消费者，那么流出流量很容易比流入流量高出一个数量级。在设置告警阈值时要切记这一点。

13.3.7 日志

没有日志的监控是不完整的。与其他应用程序一样，broker 也会将日志写到磁盘上。为了能够从日志中获得有用的信息，需要配置恰当的日志级别。可以通过记录 `INFO` 级别的日志捕捉到很多有关 broker 运行状态的重要信息。为了获得一系列清晰的日志文件，需要将不同的日志分开。

有两个日志记录器，它们会分别将日志写到单独的磁盘文件中。一个是 `kafka.controller`，我们可以将它的级别设置为 `INFO`。这个日志记录了集群控制器的信息。在任何时候，集群中都只有一个控制器，所以只有一个 broker 会使用这个日志。日志中包含了主题的创建和修改操作信息、broker 的状态变更信息，以及集群活动的信息，比如首选副本选举和分区移动。另一个是 `kafka.server.ClientQuotaManager`，我们也可以将它的级别设置为 `INFO`。这个日志用于记录与生产和消费配额活动相关的信息。因为这些信息很有用，所以最好不要把它们记录在 broker 的主日志文件中。

日志压实线程的运行状态也是有用的信息。不过，Kafka 并没有为这些线程提供单独的指标。一个分区压实失败可能导致整个压实线程崩溃，而且这是静默发生的。可以启用 kafka.log.LogCleaner、kafka.log.Cleaner 和 kafka.log.LogCleanerManager 这些日志，并将它们的级别设置为 DEBUG，这样就可以输出这些线程的运行状态信息。这些日志包含了每个被压实的分区的大小和消息条数。正常情况下，这些日志的数量不会很大，所以默认启用这些日志并不会给我们造成什么麻烦。

在调试问题时，还有一些日志也很有用，比如 kafka.request.logger，我们可以将它的级别设置为 DEBUG 或 TRACE。这个日志包含了发送给 broker 的每一个请求的详细信息。如果级别被设置为 DEBUG，那么它将包含连接端点、请求时间和概要信息。如果级别被设置为 TRACE，那么它将包含主题和分区的信息，以及除消息体之外的所有与请求相关的信息。不管被设置成什么级别，这个日志都会产生大量的数据，所以如果不是出于调试的目的，则不建议启用这个日志。

13.4 客户端监控

所有的应用程序都需要被监控。Kafka 客户端，不管是生产者还是消费者，都有特定的指标需要监控。本节将主要介绍如何监控官方的 Java 客户端，其他第三方客户端应该也提供了自己的指标。

13.4.1 生产者指标

Kafka 生产者客户端的指标经过精简只提供了少量的 JMX MBean。相反，旧版本客户端（不再受支持）提供了大量的 MBean，其中有很多指标包含了更多的细节（包括大量的百分位和各种移动平均数）。它们有更大的覆盖面，却让异常情况跟踪变得更加困难。

所有的生产者指标在 MBean 名字里都有对应的客户端 ID。在下面的示例中，客户端 ID 用 *CLIENTID* 表示，broker ID 用 *BROKERID* 表示，主题名字用 *TOPICNAME* 表示，如表 13-19 所示。

表 13-19：Kafka 生产者指标 MBean

名称	JMX MBean
生产者整体	kafka.producer:type=producer-metrics,client-id=*CLIENTID*
个体 broker	kafka.producer:type=producer-node-metrics,client-id=*CLIENTID*,node-id=node-*BROKERID*
个体主题	kafka.producer:type=producer-topic-metrics,client-id=*CLIENTID*,topic=*TOPICNAME*

表 13-19 中列出的每一个 MBean 都提供了多个属性用于描述生产者的状态。下面列出了用处最大的几个属性。在了解这些属性之前，请确保已经阅读了第 3 章，了解了生产者的工作原理。

01. 生产者整体指标

生产者整体指标的属性描述了生产者各个方面的信息，从消息批次的大小到内存缓冲区的使用情况。虽然这些指标在调试时都有一定用处，但只有少数几个会经常用到，而在这几个指标当中，也只有一部分需要监控和告警。下面将讨论一些平均数指标（以 `-avg` 结尾），它们都有相应的最大值（以 `-max` 结尾），不过这些最大值的用处很有限。

`record-error-rate` 是一个很有必要对其设置告警的属性。这个属性的值一般情况下都是零，如果大于零，则说明生产者正在丢弃无法发送的消息。生产者配置了重试次数和回退策略，如果重试次数达到了上限，那么消息就会被丢弃。也可以跟踪另外一个属性 `record-retry-rate`，不过这个属性不如 `record-error-rate` 重要，因为重试是很正常的行为。

也可以对 `request-latency-avg` 属性设置告警。这个属性是指生产者向 broker 发送请求时平均花费的时间。应该为这个指标设置一个基线，并设定告警阈值。请求延迟的增加说明生产者请求速率变慢，有可能是网络出了问题，也有可能是 broker 出了问题。不管是哪一种原因导致的，都是性能问题，并将导致生产者应用程序出现回压和其他问题。

除了这些关键指标，还需要知道生产者发送了多大的消息量。有 3 个属性为我们提供了 3 个不同的视图：`outgoing-byte-rate` 表示每秒发送的消息字节数，`record-send-rate` 表示每秒发送的消息数量，`request-rate` 表示每秒生产者发送给 broker 的请求数。一个请求可以包含一个或多个批次，一个批次可以包含一条或多条消息，一条消息由多个字节组成。把这些指标都显示在仪表盘上会很有用。

有一些指标描述了消息、请求和批次的大小（都以字节为单位）。`request-size-avg` 表示生产者发送给 broker 的请求的平均大小。`batch-size-avg` 表示单个消息批次的平均大小（根据定义，批

次包含了属于同一个分区的多条消息）。record-size-avg 表示单条消息的平均大小。对单个主题生产者来说，这些属性提供了有关消息发送的有用信息。对多主题生产者（如 MirrorMaker）来说，这些属性就没有那么有用了。除了这 3 个指标，还有另外一个指标 records-per-request-avg，它表示单个生产请求包含的平均消息条数。

最后一个值得推荐的生产指标是 record-queue-time-avg，它表示消息在发送给 Kafka 之前在生产者客户端的平均等待时间（以毫秒为单位）。应用程序在使用生产者客户端发送消息（调用 send 方法）之后，生产者会等待以下任一情况发生。

- 有足够多的消息填满批次（根据配置的 batch.size）。
- 距离上一批次发送已经有足够长的时间（根据配置的 linger.ms）。

这两种情况都会促使生产者客户端关闭当前批次，并把它发送给 broker。对于较为繁忙的主题，一般会发生第一种情况；对于吞吐量较低的主题，一般会发生第二种情况。record-queue-time-avg 指标会告诉我们发送消息用了多长时间，因此，如果希望通过配置上面的两个参数来降低应用程序延迟，那么这个指标会很有用。

02. 个体 **broker** 指标和个体主题指标

除了生产者整体指标，还有一些 MBean 为每个 broker 连接和每个主题提供了一系列属性。这些指标在调试问题时很有用，但在平常不需要对它们进行常规的监控。这些 MBean 属性的名字与整体指标的名字一样，表示的含义也一样（只是它们是应用在个体 broker 或个体主题上）。

request-latency-avg 是最为有用的个体 broker 指标，因为它比较稳定（在消息批次比较稳定的情况下），而且能够显示与某个 broker 之间的连接是否有问题。其他属性，比如 outgoing-byte-rate 和 request-latency-avg，它们会因为 broker 所包含分区的不同而有所不同。也就是说，这些指标会随时间发生变化，完全取决于集群的状态。

个体主题指标比个体 broker 指标要有趣一些，但它们只有在生产者向多个主题生成消息时才有用。当然，也只有在生产者不会向过多主题生成消息的情况下才有必要对这些指标进行常规的监控。例如，MirrorMaker 有可能向成百上千个主题生成消息。我们无法逐个检查这些指标，也几乎不可能为它们设置合理的告警阈值。与个体 broker 指标一样，个体主题指标一般用于诊断问题。例如，可以根据 record-send-rate 和 record-error-rate 这两个属性将丢弃的消息隔离到特定的主题上（或者作为跨所有主题的可用消息）。另外还有一个指标 byte-rate，它表示主题整体的消息速率（以字节 / 秒表示）。

13.4.2 消费者指标

与生产者客户端类似，消费者客户端将大量的指标属性整合到少数的几个 MBean 里，去掉了延迟百分位和速率移动平均数指标。因为消费者读取消息的逻辑比生产者发送消息的逻辑复杂，所以消费者的指标会更多，如表 13-20 所示。

表 13-20：Kafka 消费者指标 MBean

名称	JMX MBean
消费者整体	kafka.consumer:type=consumer-metrics,client-id=*CLIENTID*
获取请求管理器	kafka.consumer:type=consumer-fetch-manager-metrics,client-id=*CLIENTID*
个体主题	kafka.consumer:type=consumer-fetch-manager-metrics,client-id=*CLIENTID*,topic=*TOPICNAME*

名称	JMX MBean
个体 broker	kafka.consumer:type=consumer-node-metrics,client-id=*CLIENTID*,node-id=node-*BROKERID*
协调器	kafka.consumer:type=consumer-coordinator-metrics,client-id=*CLIENTID*

01. 获取请求管理器指标

在消费者客户端，消费者整体指标 MBean 的作用并不是很大，因为有用的指标都在**获取请求管理器** MBean 里。消费者整体指标 MBean 包含了与底层网络相关的指标，而获取请求管理器 MBean 则包含了与字节、请求和消息速率相关的指标。与生产者客户端不同，我们可以查看消费者客户端提供的指标，但对它们设置告警并没有太大意义。

对于获取请求管理器，需要监控并设置告警的一个指标是 `fetch-latency-avg`。与生产者客户端的 `request-latency-avg` 类似，这个指标表示消费者向 broker 发送请求所需要的时间。为这个指标设置告警的问题在于，请求延迟是通过消费者的 `fetch.min.bytes` 和 `fetch.max.wait.ms` 这两个参数来控制的。一个低吞吐量的主题会出现不稳定的延迟，有时候 broker 响应很快（有可用消息），有时候却无法在 `fetch.max.wait.ms` 指定的时间内完成响应（没有可用消息）。只有当主题有相对稳定和足够的消息流量时，这个指标才更有用。

为什么不监控滞后指标？

我们建议监控消费者的滞后情况，但为什么不建议监控获取请求管理器 MBean 的 `records-lag-max` 属性呢？这个属性表示分区滞后的最大消息条数（消费者偏移量和 broker 日志结束偏移量之间的最大差值）。

这里有两方面的原因：一是因为这个指标只显示了单个分区的滞后，二是因为它依赖了消费者的某些特定功能。如果没有其他选择，那么可以考虑监控这个指标，并为其设置告警。不过，最好的办法应该是使用外部滞后监控工具，13.5 节将介绍更多相关内容。

要想知道消费者客户端正在处理多少消息流量，可以查看 `bytes-consumed-rate` 或 `records-consumed-rate` 这两个指标，最好是两个都查看。它们分别表示客户端每秒读取的消息字节数和每秒读取的消息条数。有些用户为它们设置了最小值告警阈值，当消费者工作负载不足时，他们就会收到告警。不过需要注意的是，Kafka 会试图解开消费者客户端和生产者客户端之间的耦合，但消费者读取消息的速率经常会依赖于生产者是否在正常运行，所以监控消费者的这些指标实际上是对生产者做了某些假设，这样可能会导致误报。

获取请求管理器也提供了一些指标，有助于我们理解字节、消息和请求之间的关系。`fetch-rate` 表示消费者每秒发出的请求数量，`fetch-size-avg` 表示这些请求的平均大小（字节），`records-per-request-avg` 表示每个请求的平均消息条数。需要注意的是，消费者并没有提供与生产者 `record-size-avg` 相对应的指标，所以我们无法知道消息的平均大小。如果这个对你来说很重要，那么可以用其他指标来推算，或者在应用程序接收到消息之后获取消息的大小。

02. 个体 broker 指标和个体主题指标

与生产者客户端类似，消费者客户端也为每个 broker 连接和主题提供了很多指标，我们可以用它们诊断问题，但可能不需要进行常规的监控。与获取请求管理器一样，个体 broker 的 `request-latency-avg` 指标的用处也是有限的，具体取决于主题的消息流量。`incoming-byte-rate` 和 `request-rate` 分别表示从个体 broker 每秒读取的消息字节数和每秒请求数。我们可以用它们诊断消费者与 broker 之间的连接问题。

如果读取的是多个主题，那么消费者客户端提供的个体主题指标就会很有用，否则它们就与获取请求管理器的指标一样，我们也就没有必要去收集它们。如果客户端（如 MirrorMaker）读取了大量主题，那么查看这些指标就会变得很困难。如果你真的打算收集这些指标，那么可以考虑收集最重要的 3 个：`bytes-consumed-rate`、`records-consumed-rate` 和 `fetch-size-avg`。`bytes-consumed-rate` 表示从某个主题每秒读取的消息字节数，`records-consumed-rate` 表示每秒读取的消息条数，`fetch-size-avg` 表示每个请求的平均大小。

03. 消费者协调器指标

如第 4 章所述，多个消费者客户端组成了消费者群组，群组中会发生一些需要协调的活动，比如新成员加入，或者通过向 broker 发送心跳来维持群组的成员关系。消费者协调器负责处理这些协调工作，并提供了一组指标。与其他指标一样，协调器指标也提供了很多数字，但只有一小部分需要进行常规的监控。

消费者群组在执行同步操作时可能会导致消费者出现停顿。此时，群组中的消费者正在协商哪些分区应该由哪些消费者读取。停顿的时间长短取决于分区的数量。协调器提供了 `sync-time-avg` 指标，表示同步活动发生的平均时长（以毫秒为单位）。`sync-rate` 属性也很有用，表示消费者群组每秒发生同步的次数。在一个稳定的消费者群组中，这个数字大多数时候是零。

Kafka 会将消费者提交的偏移量作为读取进度的检查点。消费者既可以基于固定的时间间隔自动提交偏移量，也可以在代码中手动提交偏移量。提交偏移量也是一种生成消息的请求（它们有自己的请求类型），提交的偏移量会作为消息被发送到一个特定的主题上。协调器提供的 `commit-latency-avg` 属性表示提交偏移量所需的平均时长。我们也需要监控这个指标，就像监控生产者请求延迟一样。还要为它设定一个预期的基线和合理的告警阈值。

协调器指标的最后一个属性是 `assigned-partitions`，表示分配给消费者客户端（群组中的单个消费者）的分区数量。这个属性之所以有用，是因为我们可以通过比较整个群组各个消费者分配到的分区数量来判断群组的负载是否均衡。可以使用这个属性来识别因协调器分区分配算法所导致的负载不均衡问题。

13.4.3 配额

Kafka 可以对客户端请求进行节流，防止某个客户端拖垮整个集群。对消费者客户端和生产者客户端来说，这都是可配的，并用每秒允许单个客户端从单个 broker 读取多少字节或写入多少字节来表示。它有一个 broker 级别的默认值，客户端可以动态覆盖。当 broker 发现客户端的流量已经超出配额时，它会暂缓向客户端返回响应，直到客户端流量降到配额以下。

broker 并不会将客户端被节流的错误码放在响应消息里。也就是说，对应用程序来说，如果不监控这些指标，就不会知道发生了节流。我们需要监控的指标如表 13-21 所示。

表 13-21：需要监控的指标

客户端	MBean
消费者	`kafka.consumer:type=consumer-fetch-manager-metrics,client-id=CLIENTID` 的属性 `fetch-throttle-time-avg`
生产者	`kafka.producer:type=producer-metrics,client-id=CLIENTID` 的属性 `produce-throttle-time-avg`

在默认情况下，broker 不会开启配额限制。但不管有没有开启，监控这些指标总是安全无害的。况且，配额限制有可能在未来某个时刻被启用，从一开始就监控这些指标要比在后期添加更容易。

13.5 滞后监控

对消费者来说，最需要被监控的指标是滞后消息数量，也就是分区生成的最后一条消息和消费者读取的最后一条消息之间的差值。前面讲过，对于消费者滞后，使用外部监控工具比使用客户端提供的指标要好得多。虽然消费者提供了滞后指标，但该指标存在一个问题，即它只表示单个分区（也就是具有最大滞后的那个分区）的滞后，所以不能准确地告诉我们消费者的整体滞后情况。另外，它需要消费者做一些额外的操作，因为这个指标是由消费者对每个发出的获取请求进行计算得出的。如果消费者发生崩溃或离线，那么这个指标要么不准确，要么不可用。

监控消费者滞后最好的办法是使用外部工具，它们既能观察 broker 的分区状态（通过跟踪最近生成的消息的偏移量），也能观察消费者的状态（通过跟踪消费者群组提交的最新偏移量）。这种监控方式提供了一种不依赖消费者状态的客观视图。我们需要监控消费者群组读取的每一个分区。对于大型的消费者，比如 MirrorMaker，这意味着需要监控成千上万个分区。

第 12 章介绍过如何使用命令行工具获取消费者群组的信息，包括提交的偏移量和滞后的消息数量。如果直接监控由命令行工具提供的信息，那么也存在一些问题。首先，需要为每个分区定义一个合理的滞后阈值。每小时接收 100 条消息的主题和每秒接收 10 万条消息的主题的滞后阈值显然不同。其次，需要将滞后指标导入监控系统，并为它们设置告警。如果一个消费者群组读取了 1500 个主题，这些主题的分区数量超过了 10 万个，那么这将是一项令人望而生畏的任务。

可以使用 Burrow 来降低这项任务的复杂性。Burrow 是一个开源的应用程序，最初由 LinkedIn 开发。它会收集集群消费者群组的滞后信息，并为每个群组计算出一个独立的状态，告诉我们群组是否运行正常、是否出现了滞后、速度是否变慢或者是否已经停止工作。它不需要通过监控消费者群组的处理进度来获得阈值，不过我们仍然可以从中获得滞后的消息绝对数量。LinkedIn 工程博客“Burrow: Kafka Consumer Monitoring Reinvented”介绍了 Burrow 的工作原理。Burrow 既可用于监控一个集群中的消费者，也可用于监控多个集群，而且可以很容易被集成到已有的监控和告警系统中。

如果没有其他选择，则可以考虑查看消费者客户端的 `records-lag-max` 指标，它提供了消费者消费状态的部分视图。不管怎样，仍然建议使用像 Burrow 这样的外部监控系统。

13.6 端到端监控

我们推荐的另一种外部监控解决方案是端到端监控，它为 Kafka 集群的健康状态提供了一种客户端视图。消费者客户端和生产者客户端的一些指标能够说明集群可能出现了问题，但这里有猜想的成分，因为延迟的增加有可能是由客户端、网络或 Kafka 本身引起的。另外，你原本的任务可能只是管理 Kafka 集群，但现在也需要监控客户端。你需要知道下面这两个问题的答案。

- 可以向 Kafka 集群写入消息吗？
- 可以从 Kafka 集群读取消息吗？

理想情况下，你可能希望对每一个主题都执行这些操作。但是，在大多数情况下，向每一个主题都注入人为制造的流量是不合理的。所以，可以考虑将问题提升到 broker 级别，而这正是 Xinfra Monitor（之前叫作 Kafka Monitor）要做的事情。这个工具最初由 LinkedIn 的 Kafka 团队开发并开源。它会持续地向一个横跨集群所有 broker 的主题生成消息，并读取这些消息。然后会监控每个 broker 生产请求和读取请求的可用性，以及生产消息和读取消息之间的整体延迟。这种从外部验证 Kafka 集群运行状态的监控方式是非常有价值的，因为与监控消费者滞后一样，broker 本身无法告知客户端集群是否运行正常。

13.7 小结

监控是运行 Kafka 的一个重要组成部分，这也是为什么有那么多团队在这上面花费了那么多时间。很多组织使用 Kafka 处理千万亿字节级别的数据流。确保数据的持续性和不丢失消息是一个关键性的业务需求。我们有责任为 Kafka 用户提供他们需要的监控指标，帮助他们更好地监控应用程序。

本章介绍了如何监控 Java 应用程序，特别是 Kafka 应用程序。我们首先介绍了 broker 的一些指标，然后介绍了 Java 和操作系统的监控和日志，接下来详细介绍了 Kafka 客户端的监控，包括配额的监控，最后讨论了如何使用外部监控系统进行消费者滞后监控以及如何进行端到端集群可用性监控。虽然本章没有列出所有可用的指标，但已经涵盖了最为关键的部分。

第 14 章　流式处理

传统上，Kafka 会被看成一个强大的消息总线，能够传递事件流，但没有处理和转换事件的能力。Kafka 可靠的传递能力让它成了流式处理系统完美的数据来源。很多基于 Kafka 构建的流式处理系统将 Kafka 作为唯一可靠的数据来源，比如 Apache Storm、Apache Spark Streaming、Apache Flink、Apache Samza 等。

随着 Kafka 越来越流行，最初作为简单的消息总线，后来成为一种数据集成系统，很多公司的系统中保存了大量有价值的数据流，它们井然有序地存在了很长时间，好像在等待一个流式处理框架的出现。就像在数据库出现之前数据处理是一项艰巨的任务，流式处理的发展也因为缺少流式处理平台而停滞不前。

从 0.10.0 版本开始，除了被用作流式处理框架可靠的数据来源，Kafka 还提供了一个强大的流式处理开发库，作为其客户端开发库的一部分，叫作 Kafka Streams（有时也叫 Streams API，以下简称 Streams）。借助这个开发库，开发人员可以直接在应用程序中读取、处理和生成事件，无须再依赖外部处理框架。

本章将从解释什么是流式处理开始（因为这个概念经常被人误解）。然后会介绍流式处理的一些基本概念和流式处理系统常用的设计模式。接下来会深入讲解 Kafka 的流式处理开发库，介绍它的设计目标和架构。之后会提供一个示例，介绍如何使用 Streams 计算股价移动平均数。最后会介绍流式处理的其他应用场景，并列出选择流式处理框架（如果有的话）的一些参考标准。

本章的主要目的是让读者对流式处理和 Streams 这个庞大而迷人的领域有一个快速的了解。

关于这方面的图书很多，其中一些从数据架构的角度解释了流式处理的基本概念。

- 由 Martin Kleppmann 所著的 *MakingSenseofStreamProcessing*，该书讨论了从传统应用程序到流式处理应用程序的思维转变将为我们带来哪些好处，以及如何用事件流思想重塑数据架构。
- 由 Tyler Akidau、Slava Chernyak 和 Reuven Lax 合著的 *StreamingSystems*，该书对流式处理和该领域的一些基本思想做了很好的概括介绍。
- 由 James Urquhart 所著的 *FlowArchitectures*，该书主要面向 CTO，讨论了流式处理对业务的影响。

以下是一些详细介绍了流式处理框架细节的图书。

- 由 Mitch Seymour 所著的 *MasteringKafkaStreamsandksqlDB*。
- 由 William P. Bejeck Jr. 所著的 *KafkaStreamsinAction* 和 *EventStreamingwithKafkaStreamsandksqlDB*。
- 由 Fabian Hueske 和 Vasiliki Kalavri 合著的 *StreamProcessingwithApacheFlink*。
- 由 Gerard Maas 和 Francois Garillot 合著的 *StreamProcessingwithApacheSpark*。

Streams 还在不断演化当中。每一个主版本都会对语义做出修改并弃用一些 API。本章涉及的 API 和语义截至 Kafka 2.8。我们会避免使用计划在 Kafka 3.0 中弃用的 API，本书讨论的连接语义和时间戳处理不会包含任何计划在 Kafka 3.0 中发生的变更。

14.1 什么是流式处理

人们对流式处理的理解非常混乱。有太多关于流式处理的定义，它们混淆了实现细节、性能需求、数据模型和软件工程很多方面的东西。在关系数据库领域也面临类似的窘境，关系模型的抽象定义总是夹杂了数据库引擎的实现细节和特定局限性。

流式处理还处在发展阶段，一些流行的实现方案的处理方式可能很特别，或者有特定的局限性，但这并不能说明它们的实现细节就是流式处理的固有组成部分。

让我们从头开始：什么是数据流（也被称为**事件流**或**流数据**）？首先，**数据流**是无边界数据集的抽象表示。**无边界**意味着无限和持续增长。无边界数据集之所以是无限的，是因为随着时间的推移，会有新记录不断加入。谷歌和亚马逊等大多数公司采用了这个定义。

注意，这个简单的模型（事件流）几乎可以用来表示任何一种业务活动，比如信用卡交易、股票交易、包裹递送、流经交换机的网络事件、制造商设备传感器发出的事件、发送出去的邮件、游戏场景中的物体移动，等等。这样的例子不胜枚举，因为大部分事情可以被看成一个事件序列。

除了无边界，事件流模型还有其他一些属性。

事件流是有序的

事件的发生都有先后顺序。以金融活动事件为例，先把钱存进账户，然后再把钱花掉与先把钱花掉，然后再把钱存入账户的顺序是完全不一样的。后者会出现透支，前者则不会。这是事件流与数据库表的一个区别。需要注意的是，数据库表记录的数据是无序的，SQL 语句中的“order by”并不是关系模型的组成部分，它是为了方便查询数据而添加的。

不可变的数据记录

事件一旦发生，就不能被改变。一次金融交易被取消，并不是说它消失了，相反，表示前一个交易操作被取消的事件将被添加到事件流中。顾客向商店退货，之前的销售事实并不会消失，退货行为将被视为一个额外的事件。这是数据流与数据表之间的另一个区别——可以删除和修改数据表中的记录，但这些操作都是发生在数据库中的事务，也可以将这些事务当成事件记录到事件流中。如果你熟悉数据库的二进制日志（bin log）、预写式日志（WAL）或重做日志（redo log），就应该知道，如果向数据库表插入一条记录，再将其删除，那么表中就不会包含这条记录，但重做日志中会有两个事务：插入事务和删除事务。

事件流是可重放的

这是事件流非常有价值的一个属性。我们都知道不可重放的流（流经套接字的 TCP 数据包通常是不可重放的），但对大多数业务应用程序来说，能够重放发生在几个月前（甚至几年前）的原始事件流是非常关键的。可能是为了能够使用新的分析方法纠正过去的错误，或者是为了达到审计的目的。这也是为什么我们相信 Kafka 能够让现代业务领域的流式处理大获成功——Kafka 可以被用来捕获和重放事件流。如果没有这项能力，那么流式处理充其量只是数据科学实验室里的一个玩具而已。

需要注意的是，事件流定义和我们列出的事件流属性都没有提到事件所包含的数据和每秒事件数。不同系统的数据是不一样的，事件可以很小（有时候只有几字节），也可以很大（包含很多消息头的 XML 消息），它们可以是完全非结构化的键–值对，可以是半结构化的 JSON，也可以是结构化的 Avro 消息或 Protobuf 消息。虽然数据流经常被视为“大数据”，并且包含了每秒数百万个事件，但这里所讨论的技术同样适用（通常是更加适用）于小事件流，可能每秒甚至每分钟只有几个事件。

现在，我们已经知道了什么是事件流，接下来是时候了解“流式处理”的真正含义了。流式处理是指实时地处理一个或多个事件流。流式处理是一种编程范式，就像请求与响应范式和批处理范式一样。下面将对这 3 种范式进行比较，以便更好地理解如何在软件架构中应用流式处理。

请求与响应

这是延迟最小的一种范式，响应时间在亚毫秒和毫秒之间，通常也比较稳定。这种处理模式一般是阻塞的，即应用程序会向处理系统发出请求，然后等待响应。在数据库领域，这种范式就是**联机事务处理**（OLTP）。销售点（POS）系统、信用卡处理系统和基于时间的追踪系统通常都使用这种范式。

批处理

这种范式具有高延迟和高吞吐量的特点。处理系统按照设定的时间启动处理进程，比如每天凌晨两点开始启动、每小时启动一次，等等。它会读取所有的输入数据（自上一次处理之后的所有可用数据，或者从月初开始的所有可用数据，等等），输出结果，然后等待下一次启动。处理时间从几分钟到几小时不等，并且用户看到的结果都是旧数据。在数据库领域，数据仓库或商业智能系统就使用了这种范式。它们每天一次性加载大量数据，然后生成报告，在下一次加载新数据之前，用户看到的都是相同的报告。这种范式通常既高效又具备规模经济效益。但近几年，为了能够更及时、高效地做出决策，企业要求在更短的时间内提供可用数据，这就给那些为探索规模经济而开发却无法提供低延迟报告的系统带来了巨大压力。

流式处理

这种范式是连续的、非阻塞的。流式处理填补了请求与响应范式和批处理范式之间的空白。在请求与响应范式世界里，处理一个事件可能只需要 2 毫秒，而在批处理范式世界里，可能每天只处理一次数据，并且需要 8 小时才能完成。大多数业务不要求亚毫秒级的响应，但也不能等到第二天。大多数业务流程是持续进行的，只要业务报告保持更新，业务产品线应用程序能够持续响应，处理流程就可以进行下去，不一定需要毫秒级的响应。具有持续性和非阻塞特点的业务流程，比如针对可疑信用卡交易或网络发送告警、根据供应关系实时调整价格、跟踪快递包裹，都可以选择这种范式。

需要注意的是，流式处理的定义不依赖于任何一个特定的框架、API 或特性。只要持续地从一个无边界的数据集读取数据，处理它们并生成结果，就是流式处理。重点是，整个处理过程必须是持续的。每天凌晨两点启动，从流里读取 500 条记录，生成结果，然后结束，这样的处理流程算不上是流式处理。

14.2 流式处理相关概念

流式处理与其他数据处理非常相似——写一些代码来接收数据，对数据做一些处理（转换、聚合、增强等），然后把生成的结果输出到某个地方。不过，流式处理有一些特有的概念，那些有数据处理经验但刚开始尝试开发流式处理应用程序的人很容易混淆它们。下面我们将试着澄清这些概念。

14.2.1 拓扑

一个流式处理应用程序包含一个或多个处理拓扑。处理拓扑从一个或多个源数据流开始，经过满是流处理器的图路径，直到结果被写入一个或多个目标数据流。每个流处理器都是一个应用在事件流上的事件转换计算步骤。本书示例中使用的一些流处理器有过滤器、计数、分组和左连接。为了可视化流式处理应用程序，我们通常会画出处理器节点，并用箭头将它们连接起来，这样就可以知道应用程序在处理数据时，事件如何从一个节点流到下一个节点。

14.2.2 时间

时间可能是流式处理中最为重要的概念，也是最让人感到困惑的概念。要了解时间在分布式系统中会变得多么复杂，可以阅读 Justin Sheehy 的论文“There Is No Now”。在流式处理中，形成一个通用的时间概念非常重要，因为大部分流式应用程序的操作是基于时间窗口的。例如，我们可能有一个计算股价 5 分钟移动平均数的流式应用程序。如果一个生产者因为网络问题离线 2 小时，并在重新连线后返回 2 小时的数据，那么我们就需要知道该如何处理这些数据。这些数据大多与过去了很久的 5 分钟时间窗口有关，而且已经计算并保存结果了。

流式处理系统一般包含以下几种时间。

事件时间

事件时间是指事件的发生时间和消息的创建时间，比如指标的生成时间、商店里商品的出售时间、网站用户访问网页的时间，等等。在 Kafka 0.10.0 和更高版本中，生产者会自动在消息里添加消息的创建时间。如果这个时间戳与应用程序对**事件时间**的定义不一样（例如，Kafka 消息是在事件发生以后基于数据库记录而创建的），那么建议将事件时间作为一个单独的字段添加到消息里，这样在后续处理事件时两个时间戳将都可用。在处理数据流时，事件时间是非常重要的。

日志追加时间

日志追加时间是指事件到达并保存到 broker 的时间，也叫**摄取时间**。在 Kafka 0.10.0 和更高版本中，如果启用了自动添加时间戳的功能，或者记录是用旧版本生产者客户端生成的，并且不包含时间戳，那么 broker 就会在收到记录时自动添加时间戳。这个时间戳通常与流式处理没有太大关系，因为我们一般只对事件的发生时间感兴趣。如果要计算每天生产了多少台设备，就需要计算在那一天实际生产的设备数量，尽管这些事件有可能因为网络问题第二天才进入 Kafka。不过，如果事件时间没有被记录下来，则也可以考虑使用日志追加时间，因为它在记录创建之后就不会发生变化，而且如果事件在数据管道中没有延迟，那么就可以将其作为事件发生的近似时间。

处理时间

处理时间是指应用程序在收到事件之后要对其进行处理的时间。这个时间可以是在事件发生之后的几毫秒、几小时或几天。同一个事件可能会被分配不同的处理时间戳，具体取决于应用程序何时读取这个事件。即使是同一个应用程序中的两个处理线程，它们为事件分配的时间戳也可能不一样。所以，这个时间戳非常不可靠，最好避免使用它。

Streams 基于 `TimestampExtractor` 接口为事件分配时间。Streams 应用程序开发人员可以使用这个接口的不同实现，既可以使用前面介绍的 3 种时间语义，也可以使用完全不同的时间戳，包括从事件中提取的时间戳。

当 Streams 将输出的消息写入 Kafka 主题时，它会根据以下规则为每个事件分配时间戳。

- 如果输出记录直接映射到输入记录，那么将使用输入记录的时间戳作为输出记录的时间戳。
- 如果输出记录是一个聚合结果，那么将使用聚合中最大的时间戳作为输出记录的时间戳。
- 如果输出记录是两个流的连接结果，那么将使用被连接的两个记录中较大的时间戳作为输出记录的时间戳。如果连接的是一个流和一张表，那么将使用流记录的时间戳作为输出记录的时间戳。
- 如果输出记录是由 Streams 函数（如 `punctuate()`）生成的，该函数会在一个特定的时间调度内生成数据，而不管输入是什么，那么输出记录的时间戳将取决于流式处理应用程序的当前内部时间。

如果开发人员使用的是 Streams 底层处理 API 而不是 DSL，那么就可以使用直接操作记录时间戳的 API，从而实现符合应用程序业务逻辑的时间戳语义。

注意时区问题

在处理与时间相关的问题时，需要注意时区问题。整个数据管道应该使用同一个时区，否则得到的结果会令人感到困惑。如果不可避免地要处理不同时区的数据，那么可以在处理事件之前将它们转换成同一个时区，所以需要将时区信息保存在记录里。

14.2.3 状态

如果只是单独处理每一个事件，那么流式处理会非常简单。如果你要做的只是从 Kafka 读取在线购物交易事件流，找出金额超过 10 000 美元的交易，并将结果通过邮件发送给销售人员，那么可以使用 Kafka 消费者客户端和 SMTP 开发库，几行代码就可以搞定。

但如果处理流程中包含了多个事件，那么流式处理就会变得很有意思，例如，按照类型计算事件的数量、移动平均数、合并两个流以便生成更丰富的信息流，等等。在这些情况下，只看单个事件是不够的，需要跟踪更多的信息，比如这个小时内每种类型的事件有多少个，需要连接、求和、求平均值的所有事件，等等。我们把这些信息叫作**状态**。

这些状态通常会被保存在应用程序的本地变量里，比如使用哈希表保存移动计数器。事实上，本书的很多例子就是这么做的。但是，在流式处理应用程序中，这种保存状态的方法是不可靠的，因为如果应用程序停止或发生崩溃，那么状态就会丢失，导致结果发生变化。这通常不是我们所期望的，所以要小心地持久化状态，如果应用程序重启，则要将其恢复。

流式处理通常涉及以下几种状态。

本地状态或内部状态

这种状态只能被单个应用程序实例访问，通常通过内嵌在应用程序中的数据库来维护和管理。本地状态的优点是速度快，缺点是受可用内存的限制。所以，流式处理的很多设计模式会将数据拆分成多个子流，以便使用有限的本地状态来处理它们。

外部状态

这种状态通过使用外部数据存储来维护和管理，通常使用 NoSQL 系统，比如 Cassandra。外部状态的优点是几乎没有大小限制，而且可以被应用程序的多个实例甚至是不同的应用程序访问。缺点是使用额外的系统会增加延迟和复杂性，还可能对可用性造成影响，而且外部系统也存在变得不可用的可能性。大部分流式处理应用程序会尽量避免使用外部存储，或者将信息缓存在本地，减少与外部存储发生交互，以此来降低延迟，而如何维护内部状态与外部状态的一致性就成了一个问题。

14.2.4 流和表

大家对数据库的表都很熟悉。**表**是记录的集合，每条记录都有一个主键标识，并包含了一组由模式定义的属性。表的记录是可变的（可以执行更新和删除操作）。可以通过查询表获知数据在某一时刻的状态。例如，通过查询 `CUSTOMERS_CONTACTS` 这张表，就可以获取所有客户当前的联系信息。如果表中不包含历史数据，那么就找不到客户过去的联系信息。

与表不同，流包含了历史变更数据。**流**是一系列事件，每个事件就是一个变更。表表示的是世界的当前状态，是发生多个变更后的结果。可见，表和流是同一枚硬币的两面——世界总是在发生变化，我们有时候对导致发生变化的事件感兴趣，有时候对世界的当前状态感兴趣。如果一个系统允许通过这两种方式来看待数据，那么它就比只支持一种方式的系统更强大。

要将表转化成流，需要捕获所有对表做出的变更。要将 `insert` 事件、`update` 事件和 `delete` 事件保存到流里。大多数数据库提供了 CDC 解决方案，有很多 Kafka 连接器可以将这些变更发送到 Kafka，用于后续的流式处理。

要将流转化成表，需要应用流里所有的变更。这也叫作流的**物化**。我们需要在内存、内部状态存储或外部数据库中创建一张表，然后从头到尾遍历流里所有的事件，逐个修改状态。在完成这个过程之后，就得到了一张表，它代表了某个时间点的状态。

假设我们有一家鞋店，店里的零售活动可以用一个事件流来表示。

“红色鞋子、蓝色鞋子和绿色鞋子到货。”
“蓝色鞋子卖出。”
“红色鞋子卖出。”
“蓝色鞋子退货。”
“绿色鞋子卖出。”

如果想知道现在仓库里还有哪些库存或到目前为止赚了多少钱，就需要对视图进行物化。从图 14-1 可以看出，我们目前还有 299 双红色鞋子。如果想知道鞋店的繁忙程度，那么可以查看整个事件流，可以看到总共发生了 4 个顾客事件。我们可能还想知道为什么蓝色鞋子被退货了。

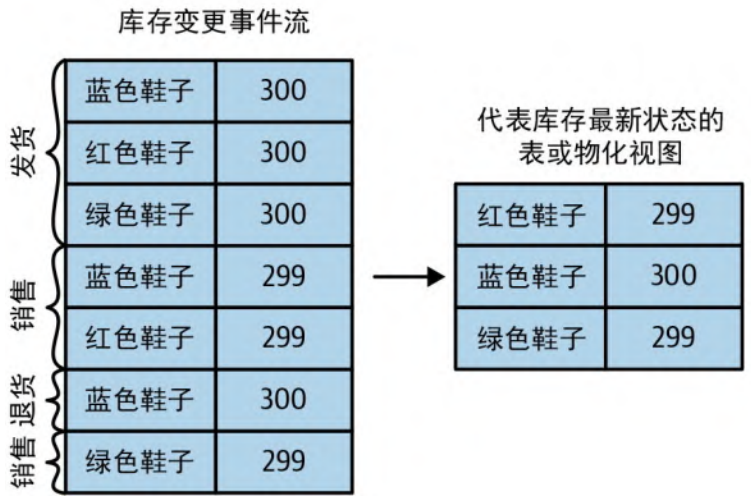

图 14-1：物化库存变更事件流

14.2.5 时间窗口

大部分针对流的操作是基于时间窗口的，比如移动平均数、一周内销量最好的产品、99 百分位系统负载，等等。两个流的连接操作也是基于时间窗口的——我们会连接发生在相同时间片段内的事件。但是，很少会有人停下来仔细想想他们需要哪些时间窗口。例如，在计算移动平均数时，我们想要知道如下信息。

窗口大小

你想要计算 5 分钟内，还是 15 分钟，抑或一天的平均数？窗口越大就越平滑，但滞后也越多。如果价格涨了，则需要更长的时间才能看出来。Streams 提供了一种**会话窗口**，其大小是通过不活跃的时间段来定义的。开发人员会定义一个会话间隙，所有连续到达且间隙小于这个会话间隙的事件都属于同一个会

话。一个大的间隙将开始一个新会话，在这个间隙之后但在下一个间隙之前到达的所有事件都属于这个会话。

窗口移动频率（移动间隔）

5 分钟平均数可以每分钟或每秒变化一次，或者在有新事件到达时发生变化。时间间隔固定的窗口叫作**跳跃窗口**（hopping window），移动间隔与窗口大小相等的窗口叫作**滚动窗口**（tumbling window）。

窗口可更新时间（宽限期）

假设我们已经计算出了 00:00 和 00:05 之间的 5 分钟移动平均数，一小时后，又收到了一些**事件时间**为 00:02 的事件，那么需要更新 00:00~00:05 这个窗口的结果吗？或者就这么算了？理想情况下，可以定义一个时间段，在这个时间段内，事件可以被添加到与它们对应的时间片段里。可以规定如果事件延迟不超过 4 小时，就重新计算并更新结果，否则就忽略它们。

窗口可以与时间对齐，也就是说，如果 5 分钟的窗口每分钟移动一次，那么第一个分片可以是 00:00~00:05，第二个分片可以是 00:01~00:06。窗口也可以不与时间对齐，可以随应用程序在任意时刻启动，所以第一个分片可以是 03:17~03:22。图 14-2 展示了这两种时间窗口的不同之处。

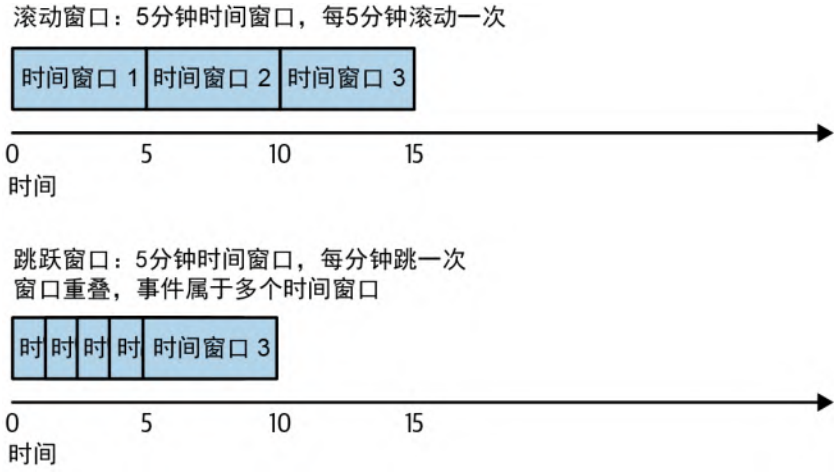

图 14-2：滚动窗口与跳跃窗口

14.2.6 处理保证

无论是否出现故障，都能够一次且仅一次处理每一条记录，这是流式处理应用程序的一个关键需求。如果没有精确一次性保证，那么流式处理就不能被用在要求精确结果的场景中。正如第 8 章介绍的那样，Kafka 支持精确一次性语义。Kafka 生产者支持事务和幂等性。Streams 借助 Kafka 的事务特性为流式处理应用程序提供精确一次性保证。使用 Streams 库的应用程序可以将 `processing.guarantee` 设置为 `exactly_once`，以此来启用精确一次性保证。Streams 2.6 或更高版本提供了更高效的精确一次性实现，它需要 broker 2.5 或更高版本。可以将 `processing.guarantee` 设置为 `exactly_once_beta`，以此来启用这种更高效的实现。

14.3 流式处理设计模式

每一种流式处理系统都不一样——从基本的消费者、处理逻辑和生产者的组合，到使用了 Spark Streaming 和机器学习软件包的复杂集群，以及其他很多介于二者之间的系统。但不管怎样，还是有一些基本的设计模式和解决方案，它们是解决流式处理架构常见需求的解决方案。下面将介绍一些众所周知的模式，并举例说明如何使用它们。

14.3.1 单事件处理

处理单个事件是流式处理最基本的模式。这种模式也叫映射（map）模式或过滤器（filter）模式，因为它经常被用于过滤无用的事件或对事件进行转换。（map 这个术语是从 map-reduce 模式中来的，在 map 阶段转换事件，在 reduce 阶段聚合事件。）

在这种模式中，应用程序会读取流中的事件，修改它们，再把它们生成到另一个流中。一个例子是，一个应用程序从流中读取日志消息，然后把 ERROR 级别的消息写到高优先级流中，把其他消息写到低优先级流中。另一个例子是，一个应用程序从流中读取事件，然后把事件从 JSON 格式转换成 Avro 格式。这类应用程序不需要在程序内部维护状态，因为每一个事件都是独立处理的。这也意味着，从故障或负载均衡中恢复都是非常容易的，因为不需要恢复状态，只需将事件交给另一个实例去处理即可。

这种模式可以使用一个生产者和一个消费者来实现，如图 14-3 所示。

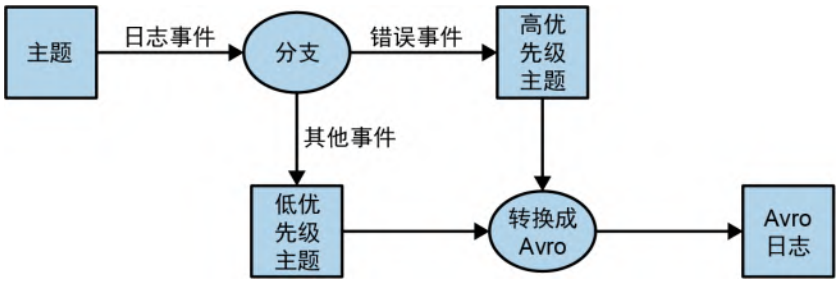

图 14-3：单事件处理拓扑

14.3.2 使用本地状态

大多数流式处理应用程序需要用到聚合信息，特别是基于时间窗口的聚合。例如，找出每天最低和最高的股票交易价格，并计算移动平均数。

要实现这些聚合操作，需要维护流的**状态**。在我们的例子中，为了计算股票每天的最小价格和平均价格，需要将到当前时间为止的最小值、总和以及记录数量保存下来。

这些操作可以通过**本地**状态（而不是共享状态）来实现，因为示例中的每一个操作都是分组聚合操作。也就是说，我们是对各只股票进行聚合，而不是对整个股票市场进行聚合。我们使用 Kafka 分区器来确保具有相同股票代码的事件总是被写入相同的分区。然后，应用程序的每个实例会从分配给自己的分区读取事件（这是 Kafka 的消费者保证），并维护一个股票代码子集的状态。如图 14-4 所示。

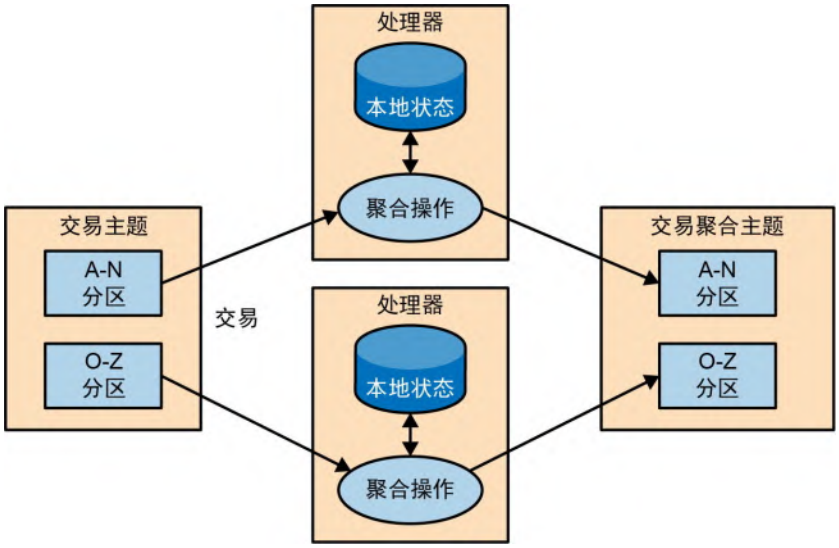

图 14-4：使用了本地状态的事件处理拓扑

如果流式处理应用程序包含了本地状态，那么情况就会变得非常复杂。流式处理应用程序必须解决下面这些问题。

内存使用

应用程序实例必须有可用的内存来保存本地状态。一些本地存储允许溢出到磁盘，但这对性能有很大影响。

持久化

要确保在应用程序关闭时状态不会丢失，并且在应用程序重启后或切换到另一个应用实例时可以恢复状态。Streams 可以很好地处理这些问题，它使用内嵌的 RocksDB 将本地状态保存在内存里，同时持久化到磁盘上，以便在重启后恢复。同时，本地状态的变更也会被发送到 Kafka 主题上。如果一个 Streams 节点发生崩溃，那么本地状态并不会丢失，因为可以通过重新读取 Kafka 主题上的事件来重建本地状态。如果本地状态包含“IBM 当前最小价格 167.19”，那么就将其保存到 Kafka 中，以便通过读取这些数据来重建本地缓存。这些 Kafka 主题使用了压缩日志，以确保它们不会无限量地增长，方便我们重建状态。

再均衡

有时候，分区会被重新分配给不同的消费者。在这种情况下，失去分区的实例必须把最后的状态保存起来，而获得分区的实例必须知道如何恢复到正确的状态。

不同的流式处理框架为开发者提供了不同的本地状态支持。如果你的应用程序需要维护本地状态，那么一定要知道框架是否提供了支持。本章将在末尾部分对一些框架进行简单的比较，但我们都知道，软件发展变化的速度非常快，而流式处理框架更是如此。

14.3.3 多阶段处理和重分区

本地状态对按组聚合的操作起到了非常大的作用。但如果需要基于所有可用信息来获得一个结果呢？假设我们每天要公布排名“前 10”的股票，也就是每天从开盘到收盘收益最高的 10 只股票。很显然，只是在每个应用程序实例中执行操作是不够的，因为排名前 10 的股票可能分布在分配给其他实例的分区中。我们需要一个两阶段解决方案。首先，计算出每只股票当天的涨跌，这个可以在每个实例中进行，并保存本地状态。然后，将结果写到一个包含单个分区的新主题中。另一个独立的应用程序实例会读取这个分区，找出当天排名前 10 的股票。新主题只包含了每只股票当日的概要信息，比其他包含交易信息的主题要小很多，所以流量很小，使用单个应用程序实例就足以应付。不过，有时候需要更多的步骤才能生成结果。如图 14-5 所示。

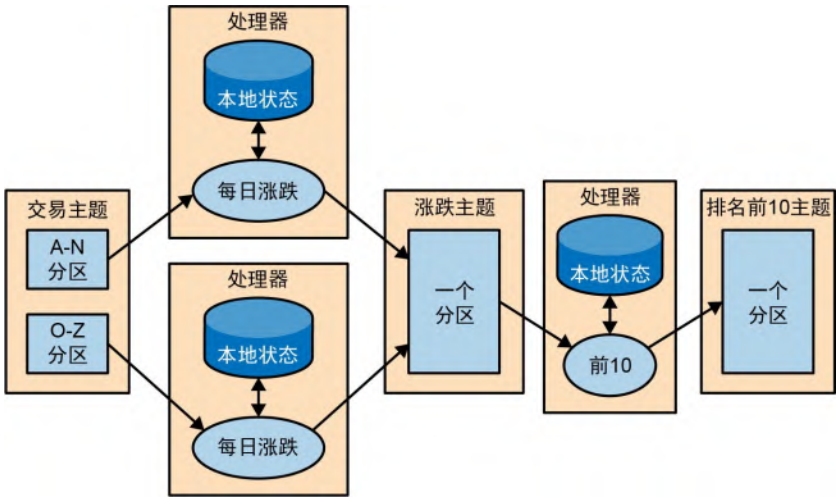

图 14-5：包含本地状态和重分区步骤的拓扑

这种多阶段处理对写过 map-reduce 代码的人来说应该很熟悉，因为他们经常使用多个 reduce 步骤。如果你写过 map-reduce 代码，就应该知道，每一个 reduce 步骤都需要单独的应用程序来处理。与 map-reduce 不同，大多数流式处理框架可以将多个步骤放在同一个应用程序中，框架会负责调配哪一个应用程序实例（或 worker）执行哪一个步骤。

14.3.4 使用外部查找：流和表的连接

有时候，流式处理需要将外部数据和流集成在一起，比如根据保存在外部数据库中的规则来验证事务，或者将用户信息填充到点击事件流中。

要使用外部查找来实现数据填充，可以这样做：对于事件流中的每一个点击事件，从用户信息表中查找相关的用户信息，生成一个新事件，其中包含原始事件以及用户的年龄和性别信息，然后将新事件发布到另一个主题上，如图 14-6 所示。

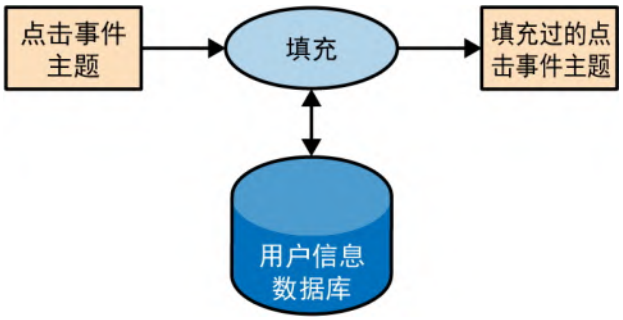

图 14-6：包含外部数据源的流式处理

这种方式最大的问题在于，外部查找会严重增加处理每条记录的延迟，通常为 5~15 毫秒。这在很多情况下是不可行的。另外，给外部数据存储造成的额外负担也是不可接受的——流式处理系统每秒可以处理 10~50 万个事件，而数据库正常情况下每秒只能处理 1 万个事件。这也增加了可用性方面的复杂性，因为当外部存储不可用时，应用程序需要知道该作何处理。

为了获得更好的性能和伸缩性，需要在流式处理应用程序中缓存从数据库读取的信息。不过，管理缓存也是一个大问题。例如，该如何保证缓存中的数据是最新的？如果刷新太过频繁，则仍然会对数据库造成压力，缓存也就失去了应有的作用。如果刷新不及时，那么流式处理用的就是过时的数据。

如果能够捕获数据库变更事件，并形成事件流，那么就可以让流式处理作业监听事件流，然后根据变更事件及时更新缓存。捕获数据库变更事件并形成事件流的过程叫作 CDC，Connect 提供了一些连接器用于执

行 CDC 任务，并把数据库表转成变更事件流。这样我们就拥有了数据库表的私有副本，一旦数据库发生变更，我们就会收到通知，并根据变更事件更新私有副本里的数据，如图 14-7 所示。

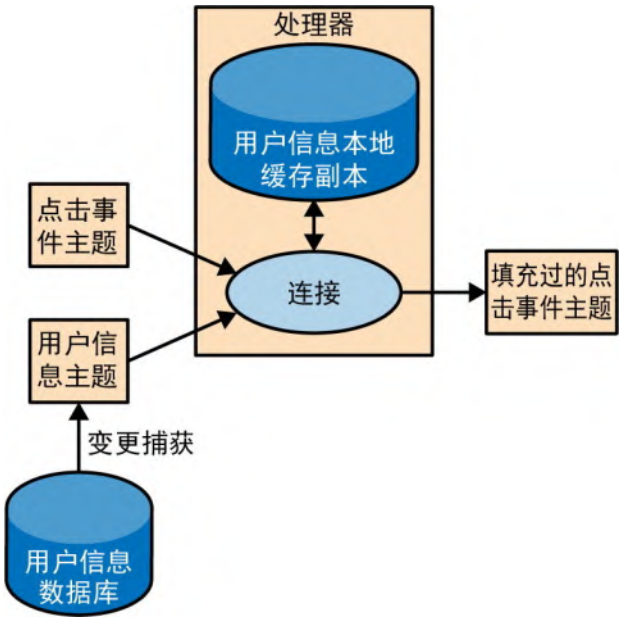

图 14-7：连接流和表的拓扑，不依赖外部数据源

这样一来，每当收到点击事件，我们就从本地缓存里查找 `user_id`，并将其填充到点击事件中。因为使用的是本地缓存，所以具有更强的伸缩性，而且不会影响数据库和其他使用数据库的应用程序。

之所以将这种方式叫作**流和表的连接**，是因为其中一个流代表了本地缓存表的变更。

14.3.5 表与表的连接

上一节介绍了表和变更事件流之间的等价性，以及如何连接流和表。那么，我们也完全有理由在连接操作的两边都使用物化表。

连接两张表不是基于时间窗口，在执行连接操作时，连接的是两张表的当前状态。Streams 可以实现等价连接（`equi-join`），也就是说，两张表具有相同的键，并且分区方式也一样，这样我们就可以在大量的应用程序实例和机器之间执行高效的连接操作。

Streams 还支持两张表的外键连接（`foreign-key join`），即一个流或表的键与另一个流或表的任意字段连接。你可以从 2020 年 Kafka 峰会的“Crossing the Streams”演讲或博文“Crossing the Streams: The New Streaming Foreign-Key Join Feature in Kafka Streams”中深入了解它的原理。

14.3.6 流与流的连接

有时候，我们需要连接两个真实的事件流，而不是一个流和一张表。什么是“真实”的流？本章开头说过，流是无边界的。如果用一个流来表示一张表，那么就可以忽略流的大部分历史事件，因为我们只关心表的当前状态。但是，如果要连接两个流，则要连接所有的历史事件，也就是将两个流里具有相同键和发生在相同时间窗口内的事件匹配起来。这就是为什么流和流的连接也叫作**基于时间窗口的连接**。

假设我们有一个用户搜索事件流和一个用户点击搜索结果事件流。我们想要匹配用户的搜索和用户对搜索结果的点击，以便知道哪个搜索的热度更高。很显然，我们需要基于搜索关键词进行匹配，而且只能匹配一个时间窗口内的事件。假设用户会在输入搜索关键词几秒之后点击搜索结果。因此，我们为每一个流保留了几秒的时间窗口，并对每个时间窗口内的事件进行匹配，如图 14-8 所示。

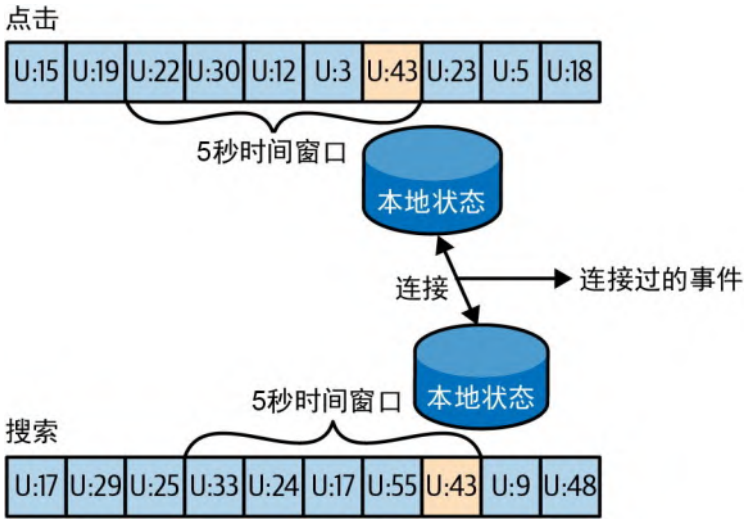

图 14-8：连接两个事件流，总是包含一个移动时间窗口

Streams 支持等价连接（`equi-joins`），流、查询、点击事件都是通过相同的键来进行分区的，而且这些键就是连接用的键。这样一来，`user_id:42` 所有的点击事件都会被保存到点击主题的分区 5 中，而 `user_id:42` 所有的搜索事件都会被保存到搜索主题的分区 5 中。然后，Streams 会确保这两个主题的分区 5 被分配给同一个任务，这样这个任务就可以看到所有与 `user_id:42` 相关的事件。Streams 在内嵌的 RocksDB 中维护了两个主题的连接时间窗口，所以能执行连接操作。

14.3.7 乱序事件

无论是流式处理系统还是传统的 ETL 系统，处理乱序事件对它们来说都是一个挑战。物联网领域经常出现乱序事件，这也是意料之中的（参见图 14-9）。例如，一个移动设备断开 WiFi 连接几小时，在重新连上后将几小时以来累积的事件一起发送出去。这种情况在监控网络设备时（发生故障的交换机在修复之前不会发送任何诊断数据）或在生产车间（车间的网络连接非常不可靠）里也时有发生。

图 14-9：乱序事件

要让流处理应用程序处理好这些场景，需要做到以下几点。

- 识别乱序事件。应用程序需要检查事件的时间，并将其与当前时间对比。
- 规定一个时间段用于重排乱序事件。例如，3 小时以内的事件可以重排，但 3 周以外的事件可以直接丢弃。
- 能够带内（in-band）重排乱序事件。这是流式处理与批处理作业的一个主要不同点。如果我们有一个每天运行的作业，一些事件在作业结束之后才到达，那么可以重新运行昨天的作业并更新事件。而在流式处理中，“重新运行昨天的作业”这种事情是不存在的，乱序事件和新到达的事件必须一起处理。
- 能够更新结果。如果处理结果是保存到数据库中，那么可以通过 put 或 update 更新结果。如果处理结果是通过邮件发送的，则需要用到一些巧妙的方法。

有些流式处理框架（比如 Google Dataflow 和 Kafka Streams）支持处理独立于处理时间的事件，能够处理比当前时间更晚或更早的事件。它们在本地状态里维护了多个可更新的聚合时间窗口，为开发人员提供了配置时间窗口可更新时长的能力。当然，时间窗口可更新时间越长，维护本地状态需要的内存就越大。

Streams API 通常会将聚合结果写到主题中。这些主题一般是压缩日志主题（`compacted topics`），也就是说，它们只保留每个键的最新值。如果一个聚合时间窗口的结果因为晚到事件需要被更新，那么

Streams 会直接为这个聚合时间窗口写入一个新结果，将前一个覆盖掉。

14.3.8 重新处理

最后一个重要的模式是重新处理事件，它有两个变种。

- 我们对流式处理应用程序做了改进，用它处理同一个事件流，生成新的结果流，并比较两种版本的结果，然后在某个时间点将客户端切换到新的结果流中。
- 现有的流式处理应用程序有很多 bug，修复完 bug 之后，我们用它重新处理事件流并计算结果。

第一种情况很简单，Kafka 会将事件流保存在一个可伸缩的数据存储里很长一段时间。要使用两个版本的流式处理应用程序来生成结果，只需满足如下条件。

- 将新版本应用程序作为一个新的消费者群组。
- 让新版本应用程序从输入主题的第一个偏移量开始读取数据（这样它就有了属于自己的输入流事件副本）。
- 让新版本应用程序继续处理事件，等它赶上进度时，将客户端应用程序切换到新的结果流中。

第二种情况具有一定的挑战性。它要求“重置”现有的应用程序，让它回到输入流的起始位置开始处理，同时重置本地状态（这样就不会将两个版本应用程序的处理结果混淆起来了），还可能需要清理之前的输出流。尽管 Streams 提供了一个用于重置应用程序状态的工具，但我们的建议是，如果有条件运行两个应用程序并生成两个结果流，那么还是使用第一种方案。第一种方案更加安全，因为它可以在多个版本之间来回切换，比较不同版本的结果，而且不会造成数据丢失，也不会在清理过程中引入错误。

14.3.9 交互式查询

如前所述，流式处理应用程序是有状态的，并且状态可以分布在多个应用程序实例中。大多数时候，流式处理应用程序的用户会从输出主题获取处理结果，但在某些情况下也可以用更简便的办法直接从状态存储里读取结果。如果处理结果是一张表（例如，10 本最畅销的图书 ），而结果流又是这张表的更新流，那么直接从流式处理应用程序的状态存储中读取表数据要快得多，也容易得多。

Streams 为此提供了灵活的 API，用于查询流式处理应用程序的状态。

14.4 Streams 示例

为了演示如何在实际当中实现这些模式，本节将给出一些使用 Streams API 的例子。之所以使用这个 API，是因为它相对简单，而且是与 Kafka 一起发布的。需要注意的是，可以用任意的流式处理框架和开发库来实现这些模式，这些模式具有通用性。

Kafka 有两个基于流的 API，一个是底层的 Processor API，另一个是高级的 Streams DSL。下面的例子中将使用 Streams DSL。我们通过为事件流定义转换链来实现流式处理。转换既可以是简单的过滤器，也可以是复杂的流与流的连接。还可以用底层 API 来实现自己的转换。要了解有关底层 Processor API 的更多信息，可以参考开发者指南和演示文稿“Beyond the DSL”。

在使用 DSL 时，通常会先用 `StreamBuilder` 创建一个**拓扑**，这是一个有向无环图（DAG），图中包含的转换步骤会被应用在事件流上。然后，我们会用拓扑创建一个 `KafkaStreams` 执行对象。`KafkaStreams` 对象将启动多个线程，每个线程都将拓扑应用到事件流上。当 `KafkaStreams` 对象关闭，处理也随之结束。

接下来将演示一些使用 Streams 来实现上述模式的例子。“字数统计”这个例子用于演示 map 与filter 模式以及简单的聚合，“计算股票交易市场统计信息”这个例子用于演示基于时间窗口的聚合，“填充点击事件流”这个例子用于演示流的连接。

14.4.1 字数统计

先来看一个使用 Streams 进行字数统计的例子。完整的示例代码可以在 GitHub 网站上找到。

创建流式处理应用程序的第一件事情是配置 Streams。Streams 有很多配置参数，这里就不展开讨论了，感兴趣的读者可以在官方文档里查看。另外，也可以将生产者和消费者内嵌到 Streams 中，只需把生产者或消费者的配置信息添加到 `Properties` 对象里即可。

```
public class WordCountExample {

    public static void main(String[] args) throws Exception{

        Properties props = new Properties();
        props.put(StreamsConfig.APPLICATION_ID_CONFIG,
          "wordcount"); ❶
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
          "localhost:9092"); ❷
        props.put(StreamsConfig.KEY_SERDE_CLASS_CONFIG,
          Serdes.String().getClass().getName()); ❸
        props.put(StreamsConfig.VALUE_SERDE_CLASS_CONFIG,
          Serdes.String().getClass().getName());
```

❶ 每个 Streams 应用程序都必须有一个应用程序 ID。这个 ID 即可用于协调应用程序实例，也可用于命名内部的本地存储和相关主题。对于同一个 Kafka 集群中的每一个 Streams 应用程序，这个 ID 必须是唯一的。

❷ Streams 应用程序通常会从 Kafka 主题上读取数据，并将结果写到 Kafka 主题。稍后我们还会看到，Streams 应用程序也会将 Kafka 作为协调工具。所以，应用程序需要知道怎样连接到 Kafka。

❸ 在读写数据时，应用程序需要对消息进行序列化和反序列化，所以我们提供了默认的序列化类和反序列化类。如果有必要，则可以在稍后创建拓扑时覆盖这些默认值。

配置好之后，开始创建拓扑。

```
StreamBuilder builder = new StreamBuilder(); ❶

KStream<String, String> source =
  builder.stream("wordcount-input");
```

```
final Pattern pattern = Pattern.compile("\\W+");

KStream<String,String> counts  = source.flatMapValues(value->
  Arrays.asList(pattern.split(value.toLowerCase()))) ❷
        .map((key, value) -> new KeyValue<String,
           String>(value, value))
        .filter((key, value) -> (!value.equals("the"))) ❸
        .groupByKey() ❹
        .count("CountStore").mapValues(value->
           Long.toString(value)).toStream(); ❺
counts.to("wordcount-output"); ❻
```

❶ 创建一个 StreamBuilder 对象，并定义一个流，将它指向一个输入主题。

❷ 从源主题上读取的每一个事件都是一行文本。先用正则表达式将它拆分成一系列单词，然后将每个单词（事件的值）放到事件的键里，稍后就可以用它们执行分组操作了。

❸ 将单词 the 过滤掉。可见过滤操作有多么简单。

❹ 根据键执行分组操作，这样就得到了每一个单词的事件集合。

❺ 计算每个集合中的事件数。计算的结果是 Long 数据类型，我们将它转成 String 类型，方便其他用户从 Kafka 中读取结果。

❻ 最后把结果写回 Kafka。

定义好转换流程后，接下来要做的就是运行它。

```
KafkaStreams streams = new KafkaStreams(builder.build(), props); ❶

streams.start(); ❷

// 一般情况下，Streams应用程序会一直运行下去
// 在这里，我们只让它运行一段时间，然后将其关闭
Thread.sleep(5000L);

streams.close(); ❸
```

❶ 基于拓扑和配置属性定义一个 KafkaStreams 对象。

❷ 启动 Streams。

❸ 过一段时间后将它关闭。

就是这么简单！这个例子只用几行代码就演示了如何实现单事件处理模式（将 map 与filter 应用在事件上）。然后，通过分组操作对数据进行重新分区，并为每一个单词的计数维护了一个简单的本地状态。

建议读者运行完整的示例，GitHub 代码库的 README 文件中包含了如何运行完整示例的说明。

需要注意的是，除了 Kafka，无须在机器上安装任何其他软件就可以运行完整的示例。如果你的输入主题包含了多个分区，则可以运行多个 WordCount 应用程序实例（在不同的命令行终端运行），这样你就有了第一个 Streams 集群。多个 WordCount 应用程序实例之间会互相通信，协调处理任务。一些流式处理框架的壁垒在于，本地模式使用起来非常简单，但要在生产环境运行集群，需要安装 YARN 或者 Mesos，然后在所有机器上安装流式处理框架，最后还要学习如何将应用程序提交到集群上。而如果使用的是 Streams API，则只需启动几个应用程序实例就可以得到一个集群。在开发机上和生产环境中运行的是同样的应用程序。

14.4.2 股票市场统计

接下来的这个例子会复杂一些，我们将从一个股票交易事件流里读取事件，这些事件包含了股票行情、卖出价和卖价数量。在股票市场交易中，**卖出价**是指卖方的出价，**买入价**是指买方建议支付的价格，**卖价数量**是指卖方愿意在这个价格出售的股数。为简单起见，我们直接忽略了竞价。数据里不包含时间戳，这里将使用由 Kafka 生产者填充的事件时间。

我们将创建包含若干时间窗口统计信息的输出流。

- 每 5 秒内最好的（也即最低的）卖出价。
- 每 5 秒内的交易股数。
- 每 5 秒内的平均卖出价。

这些统计信息每秒会更新一次。

为简单起见，假设交易所只有 10 只不同的股票。应用程序的参数配置与 14.4.1 节的“字数统计”示例差不多。

```
Properties props = new Properties();
props.put(StreamsConfig.APPLICATION_ID_CONFIG, "stockstat");
props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, Constants.BROKER);
props.put(StreamsConfig.KEY_SERDE_CLASS_CONFIG,
  Serdes.String().getClass().getName());
props.put(StreamsConfig.VALUE_SERDE_CLASS_CONFIG,
  TradeSerde.class.getName());
```

主要的区别在于这次使用的 `Serde` 类不一样。在 14.4.1 节的“字数统计”示例中，键和值的类型都是字符串，所以我们使用 `Serdes.String()` 类作为序列化器和反序列化器。而在这个例子中，键仍然是字符串，但值是 `Trade` 对象，它包含了股票代码、卖出价和卖价数量。为了序列化和反序列化这个对象（还包括应用程序中用到的其他几种对象），我们使用谷歌的 Gson 开发库从 `Java` 对象中生成了 JSon 序列化器和反序列化器。然后创建了一个包装类，用于生成 `Serde` 对象。

```
static public final class TradeSerde extends WrapperSerde<Trade> {
    public TradeSerde() {
        super(new JsonSerializer<Trade>(),
            new JsonDeserializer<Trade>(Trade.class));
    }
}
```

这里没什么特别的，只是要记得为存储在 Kafka 中的每一个对象提供一个 `Serde` 对象——输入、输出和中间结果。为了简化这个过程，建议使用 GSon、Avro、Protobuf 等框架来生成 `Serde`。

配置好以后，开始构建拓扑。

```
KStream<Windowed<String>, TradeStats> stats = source
    .groupByKey() ❶
    .windowedBy(TimeWindows.of(Duration.ofMillis(windowSize))
                           .advanceBy(Duration.ofSeconds(1))) ❷
    .aggregate( ❸
        () -> new TradeStats(),
        (k, v, tradestats) -> tradestats.add(v), ❹
        Materialized.<String, TradeStats, WindowStore<Bytes, byte[]>>
            as("trade-aggregates") ❺
           .withValueSerde(new TradeStatsSerde())) ❻
    .toStream() ❼
    .mapValues((trade) -> trade.computeAvgPrice()); ❽

stats.to("stockstats-output",
    Produced.keySerde(
      WindowedSerdes.timeWindowedSerdeFrom(String.class, windowSize))); ❾
```

❶ 从输入主题上读取事件并执行 `groupByKey()` 操作。这个方法实际上并不会执行任何分组操作，但它会确保事件流按照记录的键进行分区。因为在写数据时使用了键，而且在调用 `groupByKey()` 方法之前

不会修改键，所以数据仍然是按照它们的键进行分区的。也就是说，这个方法什么事情都没做。

❷ 定义窗口。这里是 5 分钟的窗口，每秒移动一次。

❸ 在确保有了正确的分区并定义好时间窗口之后，开始聚合操作。aggregate 方法会将事件流拆分成相互叠加的时间窗口（每秒出现一个 5 秒的时间窗口），然后将聚合方法应用在时间窗口内的所有事件上。这个方法的第一个参数是一个新对象，用于存放聚合结果，在这个例子中是 TradeStats。我们用这个对象存放每个时间窗口的统计信息，包括最低价格、平均价格和交易数量。

❹ 我们提供了一个方法对记录进行聚合，这里，TradeStats 的 add 方法用于更新时间窗口内的最低价格、交易数量和总价。

❺ 14.3 节曾经讲过，基于时间窗口的聚合需要用到保存在本地存储中的状态。聚合方法的最后一个参数就是本地状态存储的配置信息。Materialized 是本地存储配置对象，它的名字是 trade-aggregates，也可以是任意具有唯一性的名字。

❻ 作为状态存储配置的一部分，我们还提供了 Serde 对象，用于序列化和反序列化聚合结果（TradeStats 对象）。

❼ 聚合结果是一张表，包含了股票行情，并使用时间窗口作为键、聚合结果作为值。我们将表转成事件流。

❽ 更新平均价格。现在，聚合结果中包含了总价和交易数量。接下来遍历所有的记录，并使用现有的统计信息计算出平均价格，然后把它写到输出流里。

❾ 最后，将结果写到 stockstats-output 流里。因为结果是窗口操作的一部分，所以我们创建了一个 WindowedSerde 对象，将结果保存成窗口数据格式，其中包含了窗口时间戳。窗口大小会作为 Serde 的一个参数被传递进去，尽管它没有被用在序列化中（反序列化需要用到窗口大小，因为输出主题中只有窗口的开始时间）。

定义好流程之后，用它生成 KafkaStreams 对象，并运行这个对象，就像 14.4.1 节中的“字数统计”示例一样。

这个例子演示了如何对一个流执行基于时间窗口的聚合操作，这可能是流式处理最为流行的应用场景。大家可以发现，为聚合维护一个本地状态是一件多么简单的事情，只需提供一个 Serde 对象和状态存储的名字即可。这个应用程序还可以扩展到多个实例，如果有实例发生故障，则它的分区将被分配给其他活跃的实例，从而实现自动故障恢复。14.5 节将介绍它的原理。

完整的示例和运行说明可以在 GitHub 网站上找到。

14.4.3 填充点击事件流

最后一个例子将通过填充网站点击事件流来演示如何连接流。我们将生成一个模拟点击事件流、一个虚拟用户信息数据库表的更新事件流和一个网站搜索事件流。然后再将这 3 个流连接起来，得到用户活动的 360 度视图。例如，用户搜索的是什么？他们点击了哪些搜索结果？他们是否修改了“感兴趣”的内容？这些连接操作为数据分析提供了丰富的数据集。产品推荐通常就是基于这些信息。如果用户搜索了自行车，点击了“Trek”的链接，并且爱好旅游，那么就可以向其推荐 Trek 自行车、头盔和骑行活动（比如去充满异国情调的内布拉斯加州）。

应用程序的配置与前一个例子相似，所以我们跳过这部分，直接进入构建拓扑这一步。

```
KStream<Integer, PageView> views =
    builder.stream(Constants.PAGE_VIEW_TOPIC,
      Consumed.with(Serdes.Integer(), new PageViewSerde())); ❶
KStream<Integer, Search> searches =
    builder.stream(Constants.SEARCH_TOPIC,
      Consumed.with(Serdes.Integer(), new SearchSerde()));
```

```
KTable<Integer, UserProfile> profiles =
    builder.table(Constants.USER_PROFILE_TOPIC,
      Consumed.with(Serdes.Integer(), new ProfileSerde())); ❷


KStream<Integer, UserActivity> viewsWithProfile = views.leftJoin(profiles, ❸
                (page, profile) -> {
                    if (profile != null)
                        return new UserActivity(
                          profile.getUserID(), profile.getUserName(),
                          profile.getZipcode(), profile.getInterests(),
                          "", page.getPage()); ❹
                    else
                      return new UserActivity(
                        -1, "", "", null, "", page.getPage());
                    });

KStream<Integer, UserActivity> userActivityKStream =
    viewsWithProfile.leftJoin(searches, ❺
      (userActivity, search) -> {
          if (search != null)
              userActivity.updateSearch(search.getSearchTerms()); ❻
          else
              userActivity.updateSearch("");
          return userActivity;
      },
      JoinWindows.of(Duration.ofSeconds(1)).before(Duration.ofSeconds(0)), ❼
                    StreamJoined.with(Serdes.Integer(), ❽
                                      new UserActivitySerde(),
                                      new SearchSerde()));
```

❶ 首先，为要连接的点击事件流和搜索事件流创建流对象。在创建流对象时，我们指定了输入主题和键值的 Serde，用于从主题读取记录或将记录反序列化成输入对象。

❷ 为用户信息定义一个 `KTable`。`KTable` 是一种物化存储，可以通过变更流对其进行更新。

❸ 然后，将点击事件流与用户信息表连接起来，将用户信息填充到点击事件里。在连接流和表时，每一个事件都会收到来自用户信息表缓存副本里的信息。这是一个左连接操作，所以那些没有匹配的用户信息的事件仍然可以被保留下来。

❹ 这是 `join` 方法，它接受两个值，一个来自事件流，一个来自表记录，并会返回一个新值。与数据库不同，我们可以决定如何将两个值合并成一个结果。这里创建了一个 `activity` 对象，该对象包含用户信息和浏览过的页面。

❺ 接下来，连接用户点击信息和用户搜索。这也是一个左连接，不过现在连接的是两个流，而不是流和表。

❻ 这是 `join` 方法，我们只是简单地将搜索关键词添加到与之匹配的页面浏览事件中。

❼ 这部分很有意思。**流和流的连接**是基于时间窗口的。如果只是把每个用户所有的点击事件和所有的搜索事件连接起来，那么并没有什么意义。我们要把具有相关性的搜索事件和点击事件连接起来，也就是说，具有相关性的点击事件应该发生在搜索之后的一小段时间内。所以，我们定义了一个 1 秒的连接时间窗口。首先调用 `of` 方法创建了一个在搜索之前一秒和搜索之后一秒的时间窗口，然后再调用 `before` 方法，间隔时间为 0 秒，确保只连接了发生在搜索之后而不是搜索之前一秒的单击事件。结果中将包含相关的点击、搜索关键词和用户信息。这样有助于对搜索和其结果进行全面的分析。

❽ 这里定义了连接结果的 Serde，包括键和值的 Serde。在这个例子中，键是用户 ID，所以我们使用了简单的 `IntegerSerde`。

定义好流程之后，用它生成 `KafkaStreams` 对象，并运行这个对象，就像 14.4.1 节中的“字数统计”示例一样。

这个例子演示了不同类型的两种连接模式，一种是连接流和表，用于将表里的信息填充到流的事件里，这与在数据仓库中运行查询时连接事实表与维度表有点儿相似；另一种是基于时间窗口连接两个流，这种操作只会在流式处理中出现。

完整的例子和运行说明可以在 GitHub 网站上找到。

14.5 Streams 架构概览

前面演示了如何使用 Streams API 实现众所周知的流式处理设计模式。为了更好地理解 Streams 的工作原理，需要深入了解并理解 API 背后的一些设计原则。

14.5.1 构建拓扑

每个流式处理应用程序都会实现和执行一个**拓扑**。拓扑（在其他流式处理框架中叫作 DAG，即有向无环图）是一组操作和转换的集合，事件从输入到输出都会流经这个集合。14.4.1 节的“字数统计”示例的拓扑如图 14-10 所示。

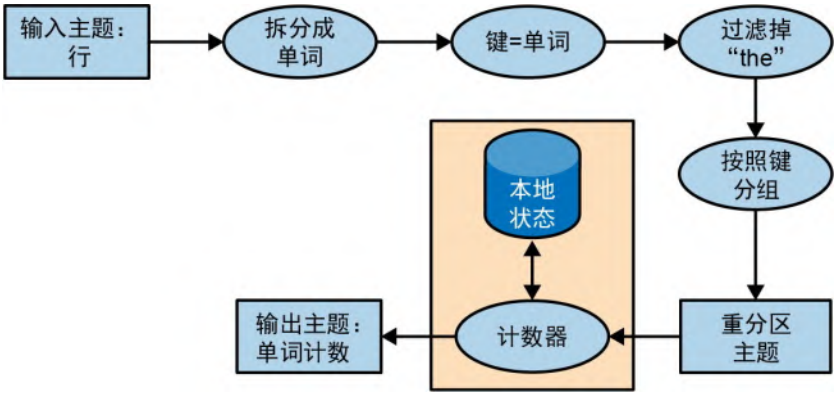

图 14-10：“字数统计”示例的拓扑

即使是一个简单的应用程序也会有不简单的拓扑。拓扑由处理器组成，处理器是拓扑图中的节点（在图中用椭圆表示）。大部分处理器实现了一个数据操作——过滤、映射、聚合等。数据源处理器从主题读取数据，并将数据传给其他组件，数据池处理器从处理器接收数据，并将数据生成到主题上。拓扑总是从一个或多个数据源处理器开始，并以一个或多个数据池处理器结束。

14.5.2 优化拓扑

默认情况下，在执行使用 DSL API 构建的应用程序时，Streams 会将每个 DSL 方法独立映射到一个底层的等价对象。因为每个 DSL 方法都是独立计算的，所以错失了优化整体拓扑的机会。

Streams 应用程序的执行分为 3 个步骤。

01. 通过创建 `KStream` 对象和 `KTable` 对象并对它们执行 DSL 操作（比如过滤和连接）来定义逻辑拓扑。
02. 调用 `StreamsBuilder.build()`，从逻辑拓扑生成物理拓扑。
03. 调用 `KafkaStreams.start()` 执行拓扑，这是读取、处理和生成数据的步骤。

在第 2 个步骤中，也就是从逻辑拓扑生成物理拓扑这一步，可以对执行计划进行整体优化。

目前，Kafka 只提供了一部分优化，主要与重用主题有关。可以通过将 `StreamsConfig.TOPOLOGY_OPTIMIZATION` 设置成 `StreamsConfig.OPTIMIZE` 并调用 `build(props)` 来启用这些优化。如果只调用 `build()` 但没有传入配置，则仍然无法启用优化。建议对启用了优化和没有启用优化的应用程序进行测试，比较执行时间和写入 Kafka 的数据量，当然，也要验证各种已知场景中的结果是否相同。

14.5.3 测试拓扑

一般来说，在正式运行应用程序之前，需要对它进行测试。自动化测试被认为是黄金标准，每次修改应用程序或开发库都会自动进行测试。这种可重复的评估方式可以实现快速迭代，并让问题诊断变得更容易。

我们也想对我们的 Streams 应用程序进行同样的测试。除了自动化端到端测试（使用生成的数据在 staging 环境中运行流式处理应用程序），我们还想进行更快、更轻量级且更容易调试的单元测试和集成测试。

Streams 应用程序的主要测试工具是 `TopologyTestDriver`。自发布 1.1.0 版本以来，它的 API 经过了重大改进。从 2.4 版本开始，它变得越来越容易使用。这些测试看起来就像是普通的单元测试。我们定义输入数据，将其生成到模拟输入主题，然后用这个工具运行拓扑，从模拟输出主题读取结果，并将结果与期望值进行对比。

建议使用 `TopologyTestDriver` 来测试流式处理应用程序，但因为没有模拟 Streams 的缓存行为（未在本书中提及的一种优化），所以有一类错误它无法检测到。

除了单元测试，还需要进行集成测试。对 Streams 来说，有两个流行的集成测试框架：`EmbeddedKafkaCluster` 和 `Testcontainers`。前者的 broker 和测试代码运行在同一个 JVM 中，后者的 broker 运行 Docker 容器中（还有其他需要用到的组件）。建议使用 `Testcontainer`，因为它使用了 Docker，可以将 Kafka、依赖项和要用到的资源与要测试的应用程序完全隔离开。

以上这些只是对 Streams 测试方法的一个简单概述。如果想更深入地了解拓扑测试，并获得更详细的代码示例，那么建议阅读“Testing Kafka Streams—A Deep Dive”这篇博文。

14.5.4 扩展拓扑

Streams 的伸缩方式有两种，一种是在一个应用程序实例中运行多个线程，一种是在分布式实例之间进行负载均衡。我们可以在一台机器上使用多线程或在多台机器上运行 Streams 应用程序，不管是哪一种，应用程序中的所有活动线程都将均衡地分摊数据处理工作。

Streams 引擎会将拓扑分为多个并行执行的任务。任务数由 Streams 引擎决定，也取决于应用程序处理的主题有多少个分区。每个任务负责处理一部分分区：它们会订阅这些分区并从分区中读取事件。对于所读取的每一个事件，任务都会在将最终结果写入目标主题之前按顺序执行所有的处理步骤。这些任务是 Streams 最基本的并行执行单元，每个任务都可以独立执行，如图 14-11 所示。

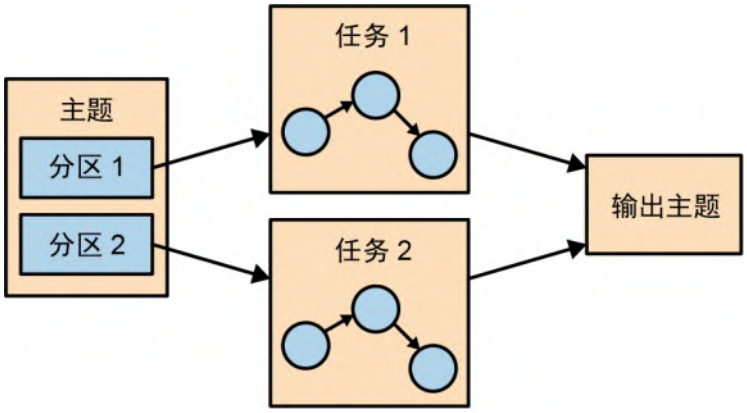

图 14-11：一个拓扑中的两个任务，每个负责读取输入主题的一个分区

应用程序开发人员可以选择每个应用程序使用的线程数。如果有多个线程可用，则每个线程将执行一部分任务。如果有多个应用程序实例运行在多台服务器上，那么每台服务器上的每一个线程都将执行一部分不同的任务。这就是流式处理应用程序的伸缩方式：主题有多少分区，就有多少个任务。如果想处理得更快，则可以添加更多的线程。如果一台服务器的资源被耗尽，就在另一台服务器上启动另一个应用程序实例。Kafka 会自动协调任务，也就是说，它会为每个任务分配属于它们的分区，让每个任务独自处理属于自己的分区，并在必要时维护与聚合相关的本地状态，如图 14-12 所示。

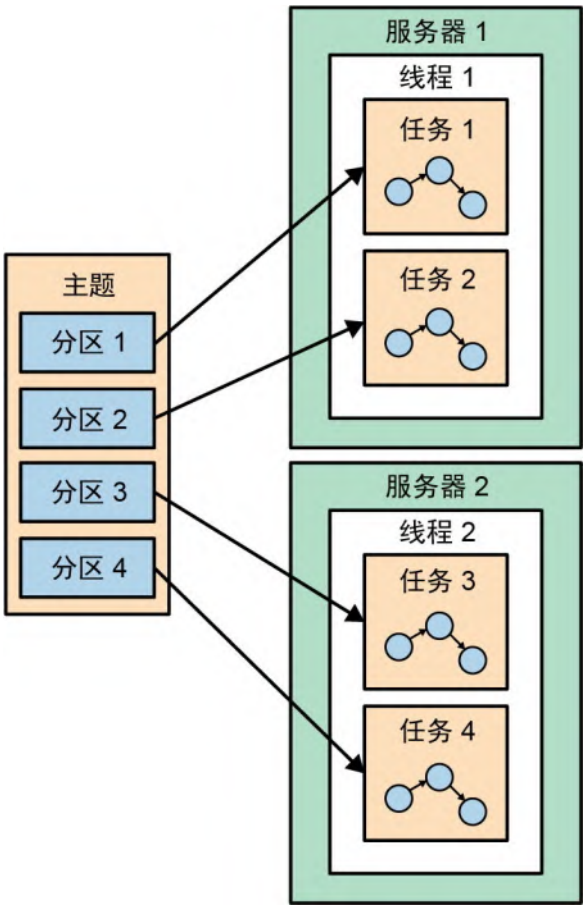

图 14-12：流式处理任务可以运行在多个线程和多台服务器上

有时候，一个处理步骤可能需要来自多个分区的输入，这会导致任务之间产生依赖关系。如果要连接两个流，就像 14.4.3 节在“填充点击事件流”示例中所做的那样，就需要从每个流中读取一个分区的数据才能生成结果。Streams 会将连接所需的所有分区分配给同一个任务，这样任务就可以读取所有相关分区并独立执行连接操作。这就是为什么 Streams 要求所有参与连接操作的主题都要有相同的分区数，并基于连接所使用的键进行分区。

应用程序重新分区也会导致任务间产生依赖关系。例如，在“填充点击事件流”示例中，所有事件都使用用户 ID 作为键。如果想基于页面或邮政编码生成统计信息该怎么办？Streams 需要用邮政编码对数据进行重新分区，并对新分区执行聚合操作。假设任务 1 在处理分区 1 的数据，这时遇到一个对数据进行重新分区（`groupBy` 操作）的处理器，它需要 shuffle 数据，或者把数据发送给其他任务。与其他流式处理框架不同，Streams 会将事件写到新主题上，并使用新的键和分区，以此来实现重新分区。然后，另一组新任务会从新主题上读取和处理事件。重分区会将拓扑拆分成两个子拓扑，每个子拓扑都有自己的任务集。第二个任务集依赖于第一个任务集，因为它们处理的是第一个子拓扑的结果。不过，这两组任务仍然可以独立并行执行，因为第一个任务集会按照自己的速率将数据写到一个主题上，第二个任务集也会按照自己的速率从这个主题读取和处理数据。两个任务集之间不需要通信，不共享资源，也不需要运行在相同的线程或服务器上。这是 Kafka 提供的最有用的特性之一——减少管道各个部分之间的依赖，如图 14-13 所示。

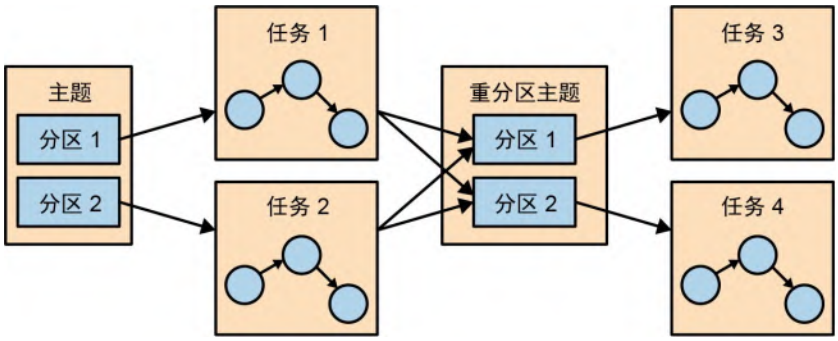

图 **14-13**：两组任务读取经过重分区的主题

14.5.5 在故障中存活下来

Streams 的扩展模型不仅允许扩展应用，还能让我们优雅地处理故障。首先，Kafka 是高可用的，所以保存在 Kafka 中的数据也是高可用的。如果应用程序发生故障需要重启，那么可以从 Kafka 中找到上一次处理的数据在流中的位置，并从这个位置继续处理。如果本地状态丢失（比如可能需要替换服务器），则应用程序可以从保存在 Kafka 中的变更日志重新创建本地状态。

Streams 还利用了消费者的协调机制来实现任务高可用性。如果一个任务失败，那么只要还有其他活跃的线程或应用程序实例，就可以用另一个线程来重启这个任务。这类似于消费者群组的故障处理：如果一个消费者失效，就把分区分配给其他活跃的消费者。Kafka 消费者群组协调协议的改进也让 Streams 锦上添花，比如固定群组成员关系和协作再平衡（参见第 4 章），以及对 Kafka 精确一次性语义的改进（参见第 8 章）。

虽然这里所说的高可用性方法在理论上是可行的，但在现实中存在一定的复杂性，其中一个比较重要的问题是恢复速度。当一个线程开始继续处理另一个故障线程留给它的任务时，它需要先恢复状态（比如当前的聚合窗口）。这通常是通过重新读取内部 Kafka 主题来实现的。在恢复状态期间，流式处理作业将暂停处理数据，从而导致可用性降低和数据过时。

因此，缩短恢复时间往往就变成了缩短恢复状态所需的时间。这里的关键在于要确保所有的 Streams 主题都是主动压实的——减小 `min.compaction.lag.ms` 的值，并将日志片段大小设置为 100 MB，而不是默认的 1 GB（需要注意的是，每个分区的最后一片段，也就是活跃片段，不会被压实）。

为了让恢复进行得更快一些，建议使用任务备用副本（`standby replica`）。这些任务是活跃的影子任务，会在其他服务器上保留当前状态。当发生故障转移时，它们已经拥有最新的状态，并准备好在几乎不停机的情况下继续处理数据。

有关 Streams 可伸缩性和高可用性的更多内容可以参考博文“Running Streams Applications”和 Kafka 峰会的一次演讲“Deploying Kafka Streams Applications with Docker and Kubernetes (Gwen Shapira + Matthias J. Sax, Confluent) Kafka Summit SF 2018”。

14.6 流式处理应用场景

本章学习了如何进行流式处理——从一般性的概念和模式说起，并列举了一些 Streams 的例子。现在是时候介绍流式处理都有哪些常见的应用场景了。本章开头说过，如果想快速处理事件，而不是为处理一个批次数据等上几小时，但又不要求毫秒级的响应，那么流式处理（或持续处理）是最好的选择。话是没错，但听起来仍然十分抽象。下面列出了一些流式处理的真实应用场景。

客户服务

假设我们刚刚向一家大型连锁酒店预订了一个房间，并希望收到电子邮件确认和收据。但是，在预订了几分钟之后我们还没有收到确认邮件，于是打电话向客服确认。客服的回复是：“我在我们的系统中看不到订单，将数据从预订系统加载到客服系统的批处理作业每天只运行一次，所以请明天再打电话过来。你应该可以在 2~3 个工作日之后收到确认邮件。”这样的服务质量有点儿糟糕，而我们已经不止一次在一家大型连锁酒店遭遇过类似的问题。我们希望连锁酒店的每一个系统在预订之后的几秒或几分钟之内都能发出通知，包括客服中心、酒店、发送确认邮件的系统、网站等。我们还希望客服中心能够立即拉取到我们在这家连锁酒店的历史入住数据，知道我们是忠实顾客，从而为我们升级服务。如果用流式处理应用程序来构建这些系统，它们就可以几近实时地接收和处理事件，带来更好的用户体验。有了这样的系统，顾客就可以在几分钟之内收到确认邮件，并及时从信用卡中扣除费用，然后发送票据，服务台就可以马上回答有关房间预订情况的问题了。

物联网

物联网包含了很多东西，从可调节温度和订购洗衣剂的家居设备到制药行业的实时质量监控设备。流式处理在这方面最为常见的应用是预测何时该进行设备维护。这与应用程序监控有点儿相似，只是监控的对象是硬件，这在很多行业中很常见，包括制造业、通信（识别故障基站）、有线电视（在用户投诉之前识别出故障机顶盒）等。每一种场景都有自己的模式，但目标是一样的，即处理大量来自设备的事件，并识别出故障设备的模式，比如交换机丢包、制造过程中需要更大的力气来拧紧螺丝，或者用户频繁重启有线电视机顶盒。

欺诈检测

欺诈检测也叫**异常检测**，是一个非常广泛的领域，专注于捕获系统中的“作弊者”或不良分子。欺诈检测的应用包括信用卡欺诈检测、股票交易欺诈检测、视频游戏欺诈检测和网络安全风险。在这些欺诈行为造成大规模破坏之前，越早识别出它们越好。一个几近实时的可以快速对事件做出响应（比如停止一个还没有通过审核的交易）的系统比在 3 天之后才能检测出欺诈行为的批处理系统要好得多。这也是一个在大规模事件流中识别模式的问题。

在网络安全领域，有一个被称为发信标（beacon）的欺诈手法。黑客在组织内部植入恶意软件，恶意软件会时不时地连接到外部网络接收命令。由于恶意软件可以在任意时间以任意频率接收命令，因此很难被检测到。通常，网络可以抵挡来自外部的攻击，但难以阻止从内部到外部的突围。通过处理大量的网络连接事件流，识别出不正常的通信模式（例如，检测出主机访问了平常不经常访问的某些 IP 地址），我们可以在蒙受更大的损失之前向安全组织发出告警。

14.7 如何选择流式处理框架

在选择流式处理框架时，需要着重考虑应用程序的类型。不同类型的应用程序需要不同的流式处理解决方案。

数据摄取

数据摄取的目的是将数据从一个系统移动到另一个系统，并在传输过程中对数据做一些修改，使其更适用于目标系统。

低延迟处理

任何要求立即得到响应的应用程序。有些欺诈检测系统就属于这一类。

异步微服务

这些微服务负责执行大型业务流程中的一些简单的操作，比如更新库存信息。这些应用程序需要通过维护本地状态缓存来提升性能。

几近实时的数据分析

这些流式应用程序通过执行复杂的聚合和连接操作来对数据进行切分，并生成有趣的业务见解。

选择什么样的流式处理系统在很大程度上取决于你要解决什么问题。

- 如果你要解决数据摄取问题，那么就要考虑是需要流式处理系统还是更简单的专注于数据摄取的系统，比如 Connect。如果你确定需要流式处理系统，那么就要确保它和目标系统都有可用的连接器。
- 如果你要进行低延迟处理，那么就要考虑是否一定要使用流。一般来说，请求与响应模式更适合用来处理这种任务。如果你确定需要流式处理系统，那么就选择一种支持逐事件低延迟模型而不是微批次模型的流式处理系统。
- 如果你要构建异步微服务，那么就需要可以与消息总线（希望是 Kafka）集成的流式处理系统，它应该具备变更捕获能力，可以将上游的变更更新到微服务的本地缓存里，并且支持本地存储，可以作为微服务数据的缓存和物化视图。
- 如果你要构建复杂的数据分析引擎，那么就需要支持本地存储的流式处理系统，不过这次不是为了本地缓存和物化视图，而是为了支持高级聚合、时间窗口和连接，因为如果没有本地存储，就很难实现这些特性。流式处理系统的 API 需要支持自定义聚合、时间窗口操作和多种连接类型。

除了具体的应用场景，还需要从全局考虑如下事项。

系统的可操作性

它是否容易部署？是否容易监控和调试？是否容易扩展？是否能够与已有的基础设施集成？如果出现错误需要重新处理数据该怎么办？

API 的可用性和可调试性

用同一种框架的不同版本开发高质量的应用程序所耗费的时间可能千差万别。因为开发时间和发布时机太重要了，所以需要选择一个高效的系统。

化繁为简

大部分系统声称它们支持基于时间窗口的高级聚合和本地缓存，但问题是，它们够简单吗？它们是帮你处理了伸缩和故障恢复方面的问题，还是只提供了脆弱的抽象并让你自己处理剩下的脏活？系统提供的 API 越简洁，封装的细节越多，开发人员的效率就越高。

社区

大部分流式处理框架是开源的。对开源软件来说，一个充满生机的社区是不可替代的。好的社区意味着用户可以定期获得新的功能特性，而且质量相对较高（没有人会使用糟糕的软件），bug 可以很快地得到修复，用户的问题可以及时得到解答。如果你遇到一个奇怪的问题并在谷歌上搜索，那么可以搜索到相关的信息，因为其他人也在使用这个系统，并遇到过同样的问题。

14.8 小结

本章首先介绍了流式处理，给出了流式处理的形式化定义，展示了它的一些常见属性，并将它与其他编程范式进行了比较。然后列举了 3 个基于 Streams 的应用程序，以此来解释一些非常重要的流式处理概念。接下来给出了 Streams 的架构概览，并阐述了它的内部原理。最后列出了流式处理的一些应用场景，给出了选择流式处理框架的一些建议，并以此结束本书。

附录 A　在其他操作系统中安装 Kafka

Kafka 基本上就是一个 Java 应用程序，可以运行在任何安装了 JRE 的操作系统中。不过，它针对 Linux 做了优化，所以在 Linux 操作系统中可以获得最好的性能。如果让它运行在其他操作系统中，则有可能会出现与特定操作系统相关的问题。所以，在桌面操作系统中进行 Kafka 开发或测试时，最好能够让它运行在与生产环境配置相匹配的虚拟机里。

A.1 在 Windows 系统中安装 Kafka

截至 Windows 10，可以通过两种方式运行 Kafka。传统的方式是使用原生的 Java 安装包。Windows 10 用户可以使用 Windows 的 Linux 子系统。推荐使用第二种方式，因为它提供了与生产环境最为相似的配置。下面就从这种方式开始说起。

A.1.1 使用 Windows 的 Linux 子系统

如果你的操作系统是 Windows 10，那么可以通过 Windows 的 Linux 子系统（WSL）来安装原生的 Ubuntu。在本书（英文版）出版时，微软仍然只是把它作为一个实验特性。它有点儿像虚拟机，但又不像虚拟机那样需要那么多资源，它还提供了更丰富的系统集成。

可以按照微软开发者网络上的“What Is the Windows Subsystem for Linux?”页面所提供的说明来安装 WSL。在安装完 WSL 后，需要通过 `apt`（假设你已经安装了这个工具）来安装 JDK。

```
$ sudo apt install openjdk-16-jre-headless
[sudo] password for username:
Reading package lists... Done
Building dependency tree
Reading state information... Done
[...]
done.
$
```

在安装完 JDK 后，继续按照第 2 章的说明来安装 Kafka 即可。

A.1.2 使用原生 Java 包

如果你使用的是旧版本的 Windows，或者不想使用 WSL 环境，那么可以使用 Windows 的原生 Java 环境来运行 Kafka。需要注意的是，这可能会引入 Windows 系统所特有的问题。在 Kafka 开发社区中，开发者们可能不会像 Linux 系统那样关注这些问题。

在安装 ZooKeeper 和 Kafka 之前，必须要有一个 Java 环境。建议安装最新版本的 Oracle Java 16，可以在 Oracle Java SE 下载页面找到安装包。下载完整版的 JDK，这样就有了所有可用的 Java 开发工具，然后按照指示一步一步安装即可。

注意安装路径

在安装 Java 和 Kafka 时，建议把它们安装在不包含空格的目录中。Windows 允许路径里包含空格，但基于 UNIX 环境设计的应用程序不支持这样的路径，而且要指定这样的路径也很麻烦。在安装 Java 时要注意这一点。如果要安装 JDK 16.0.1，那么可以把它安装在 C:\Java\jdk-16.0.1 目录下。

在安装完 Java 后，接下来要设置环境变量。环境变量可以在 Windows 的控制面板里设置，根据操作系统版本的不同，它的位置可能不一样。在 Windows 10 中设置环境变量的步骤如下。

01. 选择“系统和安全”。
02. 选择“系统”。
03. 选择“高级系统设置”，打开系统属性窗口。
04. 在“高级”选项卡中单击“环境变量”按钮。

在这里添加一个名为 `JAVA_HOME` 的用户变量（参见图 A-1），并把它的值设为 Java 的安装路径。然后继续修改系统变量 `Path`，往里面添加一个新条目 **%JAVA_HOME%\bin**。保存配置，退出控制面板。

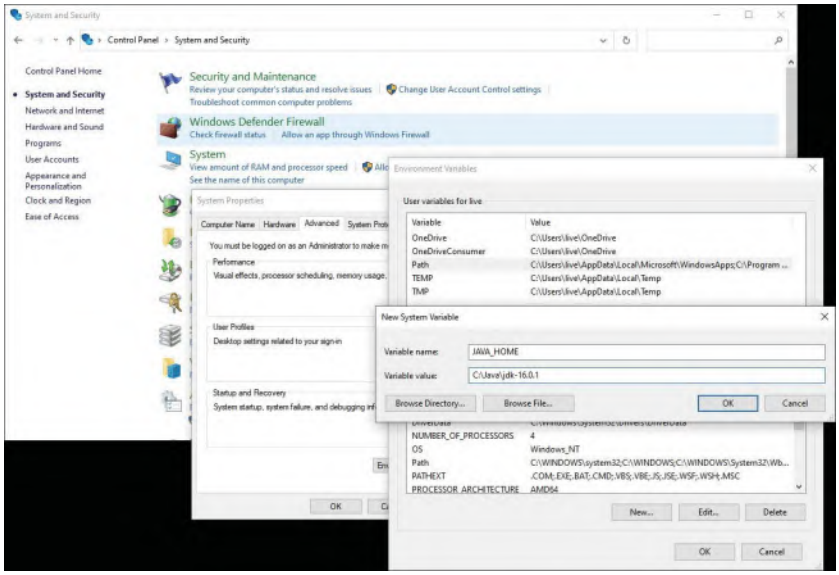

图 A-1：添加 JAVA_HOME 变量

接下来就可以安装 Kafka 了。Kafka 的安装包里已经包含了 ZooKeeper，所以不需要再单独安装 ZooKeeper。当前版本的 Kafka 可以从 Kafka 官方网站下载。在本书（英文版）出版时，Kafka 的版本是 2.8.0，相应的 Scala 版本是 2.13.0。下载的文件经过 GZip 压缩，并用 tar 来打包，所以需要用 Windows 压缩工具（如 8Zip）来解压。与在 Linux 系统中安装 Kafka 类似，必须为 Kafka 选择一个解压目录。这里假设 Kafka 会被解压到 C:\ kafka_2.13-2.8.0。

与在 Linux 系统中运行 ZooKeeper 和 Kafka 有些不同，在 Windows 系统中运行 ZooKeeper 和 Kafka 必须使用 Windows 的批处理文件而不是 shell 脚本。批处理文件不支持在后端运行应用程序，所以运行每一个应用程序都需要单独的命令行窗口。下面先启动 ZooKeeper。

```
PS C:\> cd kafka_2.13-2.8.0
PS C:\kafka_2.13-2.8.0> bin\windows\zookeeper-server-start.bat C:
\kafka_2.13-2.8.0\config\zookeeper.properties
[2021-07-18 17:37:12,917] INFO Reading configuration from: C:
\kafka_2.13-2.8.0\config\zookeeper.properties (org.apache.zookeeper.server.quo
rum.QuorumPeerConfig)
[...]
[2021-07-18 17:37:13,135] INFO PrepRequestProcessor (sid:0) started, reconfigEna
bled=false (org.apache.zookeeper.server.PrepRequestProcessor)
[2021-07-18 17:37:13,144] INFO Using checkIntervalMs=60000 maxPerMinute=10000
(org.apache.zookeeper.server.ContainerManager)
```

在 ZooKeeper 开始运行之后，打开另一个命令行窗口来启动 Kafka。

```
PS C:\> cd kafka_2.13-2.8.0
PS C:\kafka_2.13-2.8.0> .\bin\windows\kafka-server-start.bat C:
\kafka_2.13-2.8.0\config\server.properties
[2021-07-18 17:39:46,098] INFO Registered kafka:type=kafka.Log4jController
MBean (kafka.utils.Log4jControllerRegistration$)
[...]
[2021-07-18 17:39:47,918] INFO [KafkaServer id=0] started (kafka.server.Kafka
Server)
[2021-07-18 17:39:48,009] INFO [broker-0-to-controller-send-thread]: Recorded
new controller, from now on will use broker 192.168.0.2:9092 (id: 0 rack: null)
(kafka.server.BrokerToControllerRequestThread)
```

A.2 在 macOS 系统中安装 Kafka

苹果的 macOS 运行在 Darwin 中，而 Darwin 是一个基于 FreeBSD 的 UNIX 操作系统。也就是说，在 macOS 系统中运行应用程序与在 UNIX 系统中差不多，所以在 macOS 系统中安装为 UNIX 而设计的应用程序（如 Kafka）也不会有太大难度。既可以通过包管理器（如 Homebrew）来安装 Java 和 Kafka，也可以手动安装它们（这样可以自由选择版本）。

A.2.1 使用 Homebrew

如果已经在 macOS 系统中安装了 Homebrew，就可以用它来安装 Kafka。它会确保先安装 Java，然后再安装 Kafka 2.8.0（在本书英文版出版时）。

如果还未安装 Homebrew，则可以先按照安装页面上提供的步骤安装 Homebrew，然后再用它来安装 Kafka。Homebrew 会确保先安装所有的依赖项，包括 Java。

```
$ brew install kafka
==> Installing dependencies for kafka: openjdk, openssl@1.1 and zookeeper
==> Installing kafka dependency: openjdk
==> Pouring openjdk--16.0.1.big_sur.bottle.tar.gz
[...]
==> Summary
/usr/local/Cellar/kafka/2.8.0: 200 files, 68.2MB
$
```

Homebrew 会将 Kafka 安装到 /usr/local/Cellar 目录下，不过文件会被链接到其他目录。

- 二进制文件和脚本文件在 /usr/local/bin 目录下。
- Kafka 配置文件在 /usr/local/etc/kafka 目录下。
- ZooKeeper 配置文件在 /usr/local/etc/zookeeper 目录下。
- log.dirs（Kafka 的数据目录）会被设置为 /usr/local/var/lib/kafka-logs。

在安装完成后，可以启动 ZooKeeper 和 Kafka（这里是在前台运行 Kafka）。

```
$ /usr/local/bin/zkServer start
ZooKeeper JMX enabled by default
Using config: /usr/local/etc/zookeeper/zoo.cfg
Starting zookeeper ... STARTED
$ /usr/local/bin/kafka-server-start /usr/local/etc/kafka/server.properties
[2021-07-18 17:52:15,688] INFO Registered kafka:type=kafka.Log4jController
MBean (kafka.utils.Log4jControllerRegistration$)
[...]
[2021-07-18 17:52:18,187] INFO [KafkaServer id=0] started (kafka.server.Kafka
Server)
[2021-07-18 17:52:18,232] INFO [broker-0-to-controller-send-thread]: Recorded
new controller, from now on will use broker 192.168.0.2:9092 (id: 0 rack: null)
(kafka.server.BrokerToControllerRequestThread)
```

A.2.2 手动安装

与 Windows 系统的手动安装类似，在 macOS 系统中手动安装 Kafka 需要先安装 JDK。可以从 Oracle Java SE 下载页面下载 macOS 对应的 JDK 版本，然后再下载 Kafka。假设下载的 Kafka 会被解压到 /usr/local/kafka_2.13-2.8.0 目录。

与在 Windows 系统中启动 ZooKeeper 和 Kafka 类似，在 macOS 系统中启动 ZooKeeper 和 Kafka 需要确保设置了正确的 JAVA_HOME 变量。

```
$ export JAVA_HOME=`/usr/libexec/java_home -v 16.0.1`
$ echo $JAVA_HOME
/Library/Java/JavaVirtualMachines/jdk-16.0.1.jdk/Contents/Home

$ /usr/local/kafka_2.13-2.8.0/bin/zookeeper-server-start.sh -daemon /usr/local/
```

```
kafka_2.13-2.8.0/config/zookeeper.properties
$ /usr/local/kafka_2.13-2.8.0/bin/kafka-server-start.sh /usr/local/
kafka_2.13-2.8.0/config/server.properties
[2021-07-18 18:02:34,724] INFO Registered kafka:type=kafka.Log4jController
MBean (kafka.utils.Log4jControllerRegistration$)
[...]
[2021-07-18 18:02:36,873] INFO [KafkaServer id=0] started (kafka.server.Kafka
Server)
[2021-07-18 18:02:36,915] INFO [broker-0-to-controller-send-thread]: Recorded
new controller, from now on will use broker 192.168.0.2:9092 (id: 0 rack: null)
(kafka.server.BrokerToControllerRequestThread)((("macOS, installing Kafka on",
startref="ix_macOS")))((("operating systems", "other than Linux, installing
Kafka on", startref="ix_OSinstall")))
```

附录 B　其他 Kafka 工具

Kafka 社区创建了一个强大的工具和平台生态系统，让运行和使用 Kafka 变得更加容易。虽然这里提供的并不是一个详尽的清单，但确实列出了几种比较流行的可以帮助你入门的工具。

读者须知

虽然本书合著者隶属于其中的一些公司或参与了其中一些项目的开发，但他们和 O'Reilly 出版社并不认为其中的一些工具一定胜过其他工具。所以，请读者务必自行调研这些平台和工具，看看它们是否适合用来完成你们的工作。

B.1 综合性平台

有几家公司提供了可以与 Kafka 完全集成的平台，包括所有组件的托管部署，以便用户专注于如何使用 Kafka，而不是如何运行它。如果你没有可用于学习如何运行 Kafka 和相关基础设施的资源（或者可能不想专门这么做），那么这些平台就是完美的解决方案。一些平台还提供了工具，比如模式管理、REST 接口，在某些情况下甚至提供了客户端库支持，以保证组件之间互操作的正确性。

名称：Confluent Cloud

描述：Confluent Cloud 是由 Kafka 部分原始作者成立的 Confluent 公司提供的 Kafka 托管解决方案，其包含了一些必备的工具，比如模式管理、客户端、REST 接口和监控等。它在三大云平台（AWS、微软 Azure 和谷歌云平台）上都可用，并由 Confluent 的核心 Kafka 贡献者提供支持。这个平台提供的多个组件（比如模式注册表和 REST 代理）可以作为独立工具使用，只是需要遵循 Confluent 社区许可，该许可限制了一些应用场景。

名称：Aiven

描述：Aiven 为很多数据平台提供托管解决方案，包括 Kafka。它开发了 Karapace，这是一个模式注册表和 REST 代理，与 Confluent 的组件兼容，但需要遵循 Apache 2.0 许可。除了三大云供应商，Aiven 还支持 DigitalOcean 和 UpCloud。

名称：CloudKarafka

描述：CloudKarafka 专注于为流行的基础设施服务（比如 DataDog 或 Splunk）提供 Kafka 托管解决方案。它支持在其平台上使用 Confluent 的模式注册表和 REST 代理，但只支持 Confluent 修改许可之前的 5.0 版本。CloudKarafka 在 AWS 和谷歌云平台上提供服务。

名称：Amazon Managed Streaming for Apache Kafka（Amazon MSK）

描述：亚马逊也提供了自己的 Kafka 托管平台，但只支持 AWS。它通过与 AWS Glue 集成来支持模式，但不直接支持 REST 代理。亚马逊提倡用户使用社区工具（比如 Cruise Control、Burrow 和 Confluent 的 REST 代理 ），但不直接为它们提供支持服务。因此，MSK 的集成度不如其他产品，但仍然可以提供核心的 Kafka 集群。

名称：Azure HDInsight

描述：微软 HDInsight 也提供了一个 Kafka 托管平台，其支持 Hadoop、Spark 和其他大数据组件。与 MSK 类似，HDInsight 只专注于提供核心 Kafka 集群，用户需要自己提供其他组件（包括模式注册表和 REST 代理）。一些第三方供应商提供了相关的部署模板，但微软不为这些模板提供支持服务。

名称：Cloudera

描述：Cloudera 从早期开始就是 Kafka 社区的一个知名组件，它将托管的 Kafka 作为其客户数据平台 Customer Data Platform（CDP）的流式数据组件。当然，CDP 不只专注于 Kafka，它还在公有云环境中运行，同时提供私有服务。

B.2 集群部署和管理

如果是在托管平台之外运行 Kafka，那么你就需要一些工具来帮你更好地运行 Kafka 集群，包括配置和部署、均衡数据和可视化集群。

名称：Strimzi

描述：Strimzi 提供了用于部署 Kafka 集群的 Kubernetes Operator，可以方便地在 Kubernetes 环境中运行 Kafka。它不提供托管服务，但可以让你轻松地在云端（无论是公共云还是私有云）启动和运行 Kafka。它还提供了 Strimzi Kafka Bridge，这是一个基于 Apache 2.0 许可的 REST 代理实现。目前，出于许可方面的考虑，Strimzi 还不支持模式注册表。

名称：AKHQ

描述：AKHQ 是用于管理 Kafka 集群并与之交互的 GUI 工具。它支持配置管理（包括用户和 ACL 配置），并提供了一些组件（比如模式注册表和 Kafka Connect）。它还提供了集群数据操作工具，可作为 Kafka 控制台工具的替代品。

名称：JulieOps

描述：JulieOps（之前的 Kafka 拓扑构建器）能够基于 GitOps 模型提供主题和 ACL 的自动化管理。除了可以查看当前配置的状态，JulieOps 还提供了一种用于声明式配置和修改主题、模式、ACL 的方法。

名称：Cruise Control

描述：Cruise Control 是 LinkedIn 提供的用于管理大规模集群（数百个集群，包含数千个 broker）的工具。它一开始是一种自动集群数据再均衡解决方案，后来也支持异常检测和操作管理（比如添加和删除 broker）。它是除测试集群之外的其他集群所必备的运维工具。

名称：Conduktor

描述：虽然不是开源的，但 Conductor 仍然是一个流行的用于管理 Kafka 集群并与之交互的桌面工具。它支持多个托管平台（包括 Confluent、Aiven 和 MSK）和不同的组件（比如 Connect、kSQL 和 Streams)。你还可以用它操作集群中的数据。它提供了一个免费许可，可用于操作单个集群。

B.3 监控和查看数据

运行 Kafka 的关键是确保集群和客户端的可用性。与其他很多应用程序一样，Kafka 提供了很多指标，但要充分利用好它们并不容易。很多大型的监控平台（如 Prometheus）可以很容易地获取来自 Kafka broker 和客户端的指标。除此之外，还有很多工具可用于分析这些指标。

名称：Xinfra Monitor

描述：Xinfra Monitor（之前的 Kafka Monitor）是 LinkedIn 开发的用于监控 Kafka 集群和 broker 可用性的工具。它通过向一些主题生成合成数据，让这些数据流经集群，来检测集群的延迟、可用性和完整性。它是一个很有用的工具，可以在不需要直接与客户端发生交互的情况下检测 Kafka 的运行状况。

名称：Burrow

描述：Burrow 是另一个最初由 LinkedIn 开发的工具，用于检测 Kafka 消费者的滞后情况。它可以在不直接与消费者发生交互的情况下检测消费者的健康状况。Burrow 得到了社区的积极支持，并拥有自己的工具生态系统，可以与其他组件集成。

名称：Kafka Dashboard

描述：当你在使用 DataDog 进行监控时，它提供了一个优秀的 Kafka Dashboard，让你在集成 Kafka 集群和监控技术栈时如虎添翼。它的目标是为 Kafka 集群提供集中式视图，以简化指标的显示。

名称：Streams Explorer

描述：Streams Explorer 是用于可视化 Kubernetes 集群中流经应用程序和连接器的数据流的工具。虽然它严重依赖于使用 Streams 或 Faust 构建部署，但也为这些应用程序及其指标提供易于理解的视图。

名称：kcat

描述：kcat（之前的 kafkacat）是控制台生产者和消费者的替代品。它小巧、速度快，使用 C 语言开发，无须安装 JVM。它可以输出集群元数据，为我们提供有限的集群状态视图。

B.4 客户端开发库

Kafka 项目为 Java 应用程序提供了客户端开发库，但只支持一种编程语言是远远不够的。Kafka 客户端有很多其他编程语言实现，比如 Python、Go 和 Ruby 等。此外，REST 代理（比如 Confluent、Strimzi 或 Karapace 提供的代理）可以覆盖各种应用场景。下面是一些经受了时间考验的客户端实现。

名称：librdkafka

描述：librdkafka 是 Kafka 客户端的 C 语言实现，被认为是性能最好的客户端开发库之一。因为太好了，所以 Confluent 支持的 Go 语言、Python 和 .Net 客户端都对其进行了包装。它采用了双条款 BSD 许可，可以很方便地用在任何一个应用程序中。

名称：Sarama

描述：Sarama 是 Shopify 开发的原生 Go 语言实现，采用了 MIT 许可。

名称：kafka-python

描述：kafka-python 是一个原生 Python 客户端实现，采用了 Apache 2.0 许可。

B.5 流式处理

尽管 Kafka 项目提供了用于构建应用程序的 Streams，但它并不是流式处理的唯一选择。

名称：Samza

描述：Apache Samza 是专门为 Kafka 设计的流式处理框架。它早于 Streams 问世，是由同一批人开发的，因此它们共享了很多概念。但与 Streams 不同，Samza 运行在 Yarn 上，并为应用程序提供了一个完整的运行框架。

名称：Spark

描述：Spark 是另一个面向数据批处理的 Apache 项目。它通过微批次来处理数据流，所以延迟会高一些，但它可以通过重新处理批次来实现容错，并且可以很容易地实现 Lambda 架构。广泛的社区支持也是它的优势之一。

名称：Flink

描述：Apache Flink 专门面向流式处理，具有非常低的延迟。与 Samza 一样，它支持 Yarn，但也可以运行在 Mesos、Kubernetes 或独立集群中。它的高级 API 支持 Python 和 R 语言。

名称：Beam

描述：Apache Beam 并不直接提供流式处理，它是一种集批处理和流式处理于一身的统一编程模型。它将 Samza、Spark 和 Flink 作为管道组件的运行器。

关于作者

格温 • 沙皮拉（**Gwen Shapira**）是 Confluent 的工程主管，在 Confluent 领导云原生 Kafka 团队，该团队致力于为 Confluent Cloud 打造更具弹性和可伸缩性的 Kafka 云服务。她拥有 15 年构建可伸缩数据架构的经验。格温经常在行业大会上演讲，是 Apache Kafka 项目的 Committer 和 PMC 成员。

托德 • 帕利诺（**Todd Palino**）是 LinkedIn 的首席站点可靠性工程师，负责解决整个平台在容量和效率管理方面遇到的挑战。此前，他在 LinkedIn 负责 Kafka 和 ZooKeeper 的架构、日常运维和工具开发工作，构建了高级的监控和通知系统。托德是开源项目 Burrow（一种 Kafka 消费者监控工具）的开发者，经常在行业大会和技术讲座上分享他在 SRE 方面的经验。托德在技术行业工作了 20 多年，包括在 Verisign 担任系统工程师，实现 DNS、网络和硬件的服务管理自动化，以及管理整个公司的硬件和软件标准。

拉吉尼 • 西瓦拉姆（**Rajini Sivaram**）是 Confluent 的首席工程师，为 Kafka 设计和开发跨集群复制功能，并为 Confluent Platform 和 Confluent Cloud 设计和开发安全功能。拉吉尼是 Apache Kafka 项目的 Committer 和 PMC 成员。在加入 Confluent 之前，在 Pivotal 工作，负责基于 Reactor 构建高性能的反应式 Kafka API。早些时候，拉吉尼在 IBM 为 Bluemix 平台开发 Kafka 即服务产品。她的技术经验横跨多个领域，从并行和分布式系统到 Java 虚拟机和消息传递系统。

克里特 • 佩蒂（**Krit Petty**）是 LinkedIn 的 Kafka 站点可靠性工程经理。在成为经理之前，他在团队中担任 SRE，负责管理 Kafka，应对前所未有的挑战，包括首次将 LinkedIn 的大规模 Kafka 集群迁移到微软的 Azure 云。克里拥有计算机科学硕士学位，之前做过 Linux 系统管理员，并作为软件工程师为石油和天然气行业的高性能计算项目开发软件。

关于封面

本书封面上的动物是一只蓝翅笑翠鸟（Dacelo leachii）。笑翠鸟属于翠鸟科，生活在新几内亚南部和澳大利亚北部不那么干旱的地区。它们是河翠鸟。

雄性笑翠鸟拥有五颜六色的羽毛，翅膀和尾部的羽毛是蓝色的；雌性笑翠鸟的尾部则是红棕色，带有黑色的条纹。雄性笑翠鸟和雌性笑翠鸟都有奶油色的腹部，伴有棕色的条纹，它们的虹膜是白色的。成年笑翠鸟的体形要比其他翠鸟小一些，身长 38~43 厘米，体重 260~330 克。

蓝翅笑翠鸟偏爱肉食，捕食对象随季节稍有不同。在夏季，它们通常以蜥蜴、昆虫和青蛙为食；在干燥的季节，它们则多以小龙虾、鱼、啮齿类动物和小型的鸟类为食。不过，在以鸟类为食的食物链里，红鹰和棕鹰鸮喜欢以蓝翅笑翠鸟为食。

每年的 9 月~12 月是蓝翅笑翠鸟的繁殖季节，它们把巢穴筑在树上，通过社区协作的方式养育下一代，一对笑翠鸟至少会得到另外一只笑翠鸟的帮助。它们一般会花大概 26 天的时间来孵蛋，每次孵 3~4 只蛋。雏鸟如果能够正常破壳而出，那么大概 36 天后就会长出羽毛。早出生的雏鸟会在第一周内尝试杀死更小的雏鸟。成年笑翠鸟会花 6~10 周的时间来训练那些存活下来的小鸟捕食。

O'Reilly 图书封面上的许多动物濒临灭绝，它们对世界很重要。如果想了解如何保护动物，请访问 animals.oreilly.com。

封面插图来自 Karen Montgomery 的作品，该插图基于 *EnglishCyclopaedia* 中的黑白雕刻。

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们：

- 微博 @图灵教育 : 好书、活动每日播报
- 微博 @图灵社区 : 电子书和好文章的消息
- 微博 @图灵新知 : 图灵教育的科普小组
- 微信 图灵访谈 : ituring_interview，讲述码农精彩人生
- 微信 图灵教育 : turingbooks
