

Apache Kafka 实战

胡夕 著



- 基于Apache Kafka 1.0.0版本进行介绍，Kafka Contributor执笔。
- 从Kafka基本概念与特性开始，详细介绍了Kafka的部署、开发、运营、监控、调试、优化以及重要组件的设计原理，并给出了翔实的案例。
- 本书既适合作为Kafka的入门书籍，也适合系统架构师和一线开发工程师参考阅读。



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

Apache Kafka 实战

胡夕 著



電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书是涵盖 Apache Kafka 各方面的具有实践指导意义的工具书和参考书。作者结合典型的使用场景,对 Kafka 整个技术体系进行了较为全面的讲解,以便读者能够举一反三,直接应用于实践。同时,本书还对 Kafka 的设计原理及其流式处理组件进行了较深入的探讨,并给出了翔实的案例。

本书共分为 10 章:第 1 章全面介绍消息引擎系统以及 Kafka 的基本概念与特性,快速带领读者走进 Kafka 的世界;第 2 章简要回顾了 Apache Kafka 的发展历史;第 3 章详细介绍了 Kafka 集群环境的搭建;第 4、5 章深入探讨了 Kafka 客户端的使用方法;第 6 章带领读者一览 Kafka 内部设计原理;第 7~9 章以实例的方式讲解了 Kafka 集群的管理、监控与调优;第 10 章介绍了 Kafka 新引入的流式处理组件。

本书适合所有对云计算、大数据处理感兴趣的技术人员阅读,尤其适合对消息引擎、流式处理技术及框架感兴趣的技术人员参考阅读。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有,侵权必究。

图书在版编目(CIP)数据

Apache Kafka 实战 / 胡夕著. —北京:电子工业出版社, 2018.5
ISBN 978-7-121-33776-5

I. ①A... II. ①胡... III. ①分布式操作系统 IV. ①TP316.4

中国版本图书馆 CIP 数据核字(2018)第 037942 号

责任编辑:付 睿

印 刷:三河市双峰印刷装订有限公司

装 订:三河市双峰印刷装订有限公司

出版发行:电子工业出版社

北京市海淀区万寿路 173 信箱

邮编:100036

开 本:787×980 1/16 印张:25

字数:557 千字

版 次:2018 年 5 月第 1 版

印 次:2018 年 9 月第 3 次印刷

定 价:89.00 元

凡所购买电子工业出版社图书有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系及邮购电话:(010)88254888,88258888。

质量投诉请发邮件至 zltz@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式:010-51260888-819, faq@phei.com.cn。

前言

2011 年年初，美国领英公司（LinkedIn）开源了一款基础架构软件，以奥地利作家弗兰兹·卡夫卡（Franz Kafka）的名字命名，之后 LinkedIn 将其贡献给 Apache 基金会，随后该软件于 2012 年 10 月成功完成孵化并顺利晋升为 Apache 顶级项目——这便是大名鼎鼎的 Apache Kafka。历经 7 年发展，2017 年 11 月，Apache Kafka 正式演进到 1.0 时代，本书就是基于 1.0.0 版本来展开介绍 Kafka 的设计原理与实战的。

背景

这是一个最好的大数据时代，这是一个最坏的大数据时代！

很抱歉，我使用了这句改编后的狄更斯名言作为开头，我想没有谁会质疑“当今是大数据时代”这个论点。今年（2018 年）两会上李克强总理所做的政府工作报告中多次提及大数据等关键词，这已然成为“大数据”第 5 次被写入政府工作报告了。具体到大数据行业内，各种各样的大数据产业方兴未艾，其中在实时流式处理领域涌现出大量的技术与框架，令技术人员们应接不暇。实时流式处理系统在克服了传统批处理系统延时方面的固有缺陷的同时，还摆脱了设计上的桎梏，实现了“梦寐以求”的正确性。可以说，对于流式处理从业人员来说，这正是摩拳擦掌、大展宏图的最好时代。

与此同时，我们也清醒地意识到当今大数据领域内的细分越来越精细化。不必说日渐火爆的人工智能和机器学习潮流引诱着我们改弦易辙，也不必说那些纷繁复杂的技术框架令人眼花缭乱，单是静下心来沉淀所学、思考方向的片刻时光于我们这些从业者来说都已显得弥足珍贵。我们仿佛在黑暗密林中徘徊，试图找出那条通往光明的“康庄大道”。每当发现了一条羊肠小路都好似救命稻草一般紧紧抓住。多年后我们回望那只不过是不断追逐热点罢了，在技术的海洋中我们迷失了前进的方向。从这个意义上说，这实在是一个糟糕的时代。

时光切回到 4 年前的某个下午，那时我正在做着 Kafka 的大数据项目。我突然发现与其盲目跟风各种技术趋势，何不精进手头的工作，把当前工作中用到的技术搞明白，于是我萌发了

研究 Kafka 的想法。直到今天，我都无比庆幸那个午后做出的冲动决定，正如 Adam Grant 在《离经叛道》一书中所说：最正确的决定都是在冲动之下做出的。诚不欺我！

想要深入学习 Kafka，不掌握 Scala 语言是不行的，毕竟 Kafka 就是使用 Scala 语言编写的。苦于当时没有合适的 Scala 中文书籍，我依稀记得找到了一本 600 多页的 Scala 原版书（*Programming Scala Edition 2*）进行学习。那段时间实在是难熬！不得不说，英文版书籍虽然内容翔实，但在表述上实在晦涩难懂，比如 `partially applied function` 和 `partial function` 两者的区别直至今天我都不是特别清晰，还是要不断地翻阅资料才能隐约记得它们之间的不同。庆幸的是，我没有半途而废，600 多页的英文文档硬是啃了下来。对于 Scala 的初步掌握也让我觉得研究 Kafka 的时机到了。有意思的是，在之后通读 Kafka 的源码时我不禁大呼上当，Kafka 的源码中只使用了最简单的函数式编程，我有些后悔自己花了那么多时间去学习 Scala 的函数式编程，当然这是后话。

既然是研究 Kafka，那么研读源码是必不可少的步骤。如果不分析源码，我们就无法定位问题发生的根本原因。实话实说，阅读别人源码的过程是痛苦的，因而在理解的过程中我走了不少弯路。为了记录阅读 Kafka 源码的心得，我努力为每个 Kafka 源码包撰写博客。现在翻看我之前的博客，大家还能看到那好似流水账一般的 Kafka 源码分析系列文章。

随着对源码的不断熟悉，我加入了 Apache Kafka 社区，希望贡献自己的微薄之力。时至今日，我依然记得当初发送邮件要求加入开发组时的惶恐，也记得第一次贡献代码时的惴惴不安；我记得为了研究某个 Kafka bug，自己曾忘记吃中饭的执着，也记得自己被标记为“Kafka contributor”时的喜悦。在混迹社区的日子里，我逐渐认识了一些 Kafka 的 committer 们，比如 Kafka PMC 成员王国璋，国璋兄对于网上 Kafka 问题的权威解答令我受教良多，同时我也很感激他于百忙之中为本书写推荐语。还有 Kafka 的三位原作者之一的饶军（Rao Jun），几次问题交流让我看到了他霸气的决断能力以及对于疑难问题原因的毒辣分析。当然还有非常敬业的 Ijuma，他是我见过的最勤劳的 Kafka committer，没有之一。在编写本书的过程中，我都或多或少地得到过他们的帮助，再次表示衷心感谢。

由于对 Kafka 研究的日益深入，我终于有了写书的冲动。我希望通过把学到的知识和原理集中整理并书写成文字来帮助那些尚未接触 Kafka 的广大读者快速上手，降低他们学习使用 Kafka 的成本，于是有了今天这本《Apache Kafka 实战》。借着写作本书的契机，我本人对 Kafka 的方方面面做了梳理，自觉收获良多。每当搞懂了一个以前未了解的机制时，心中的那种满足感和兴奋感至今都令人神往。在此，我深深地希望读者在阅读完本书后也能有这样的体会。

面向的读者

我衷心希望本书可以成为各行各业的大数据从业者使用消息队列甚至是进入流式处理领域

内的“敲门砖”，也希望各大公司能够充分利用 Kafka 来实现自己的业务目标。

在编写本书的过程中，我阅读了大量的英文资料和源代码，试图通过自己的理解将 Kafka 的使用实战技巧深入浅出地呈现给广大读者。没错，我希望这本书给人的感觉是通俗易懂、深入浅出，从而方便引领读者快速进入 Kafka 学习的大门。

我本人维护了一个微信公众号（名为“大数据 Kafka 技术分享”），希望在该公众号中我能和读者朋友们一起深入交流和探讨 Kafka 学习过程中碰到的各种问题，同时我也会及时分享和推送各种最新的 Kafka 使用心得。

致谢

非常感谢 Kafka PMC 成员、Kafka Committer 王国璋对本书的大力支持。自开始编写本书之日起，国璋兄就给予我很大的鼓励与帮助，这也让我坚定了传播 Kafka 实战心得的决心。

感谢腾讯 AI 平台助理总经理王迪先生和我的好友贾兴华，你们对本书的评价之高实在是过誉了，但也令本人倍感振奋。

感谢我的前同事、新浪微博技术专家付稳。付总对本书整体结构和具体知识点的建议发人深省，其独到的行业见解令人佩服。

非常感谢电子工业出版社的编辑付睿女士。她细致、专业、严谨的工作作风深深地感染了我，在本书编写过程中她总是能及时地就书中的内容给出合理的建议和指导。

另外，我还想感谢一下我的家人，特别是我的妻子刘丹女士。过去一年中正是你坚定的支持和默默的付出才成就我撰写本书。对于你偶尔在学术上给予的提点我既感到惊讶，同时也欣慰不已。这为我漫长枯燥的写书过程平添了很多温暖。

最后，非常感谢本书的每一位读者。本人已经在写作过程中收获良多，我衷心希望你们在阅读本书时也有大呼过瘾的感觉。另外，我在“知乎”（ID: huxihx）的 Kafka 专栏以及 StackOverflow 网站上也会尽力回答关于 Kafka 的各类问题，希望通过这些途径可以和读者进行更加深入的交流。

由于本人水平有限，书中难免有遗漏和疏忽，也恳请各位读者多多指正。

胡夕

2018 年 3 月 15 日于北京

个人博客：<https://www.cnblogs.com/huxi2b/>

微信公众号：大数据 Kafka 技术分享

电子邮箱：huxi_2b@hotmail.com

目录

第 1 章 认识 Apache Kafka.....	1
1.1 Kafka 快速入门.....	1
1.1.1 下载并解压缩 Kafka 二进制代码压缩包文件.....	2
1.1.2 启动服务器.....	3
1.1.3 创建 topic.....	3
1.1.4 发送消息.....	4
1.1.5 消费消息.....	4
1.2 消息引擎系统.....	5
1.2.1 消息设计.....	6
1.2.2 传输协议设计.....	6
1.2.3 消息引擎范型.....	6
1.2.4 Java 消息服务.....	8
1.3 Kafka 概要设计.....	8
1.3.1 吞吐量/延时.....	8
1.3.2 消息持久化.....	11
1.3.3 负载均衡和故障转移.....	12
1.3.4 伸缩性.....	13
1.4 Kafka 基本概念与术语.....	13
1.4.1 消息.....	14
1.4.2 topic 和 partition.....	16
1.4.3 offset.....	17
1.4.4 replica.....	18

1.4.5	leader 和 follower.....	18
1.4.6	ISR	19
1.5	Kafka 使用场景.....	20
1.5.1	消息传输.....	20
1.5.2	网站行为日志追踪.....	20
1.5.3	审计数据收集.....	20
1.5.4	日志收集.....	20
1.5.5	Event Sourcing.....	21
1.5.6	流式处理.....	21
1.6	本章小结	21
第 2 章	Kafka 发展历史.....	22
2.1	Kafka 的历史.....	22
2.1.1	背景.....	22
2.1.2	Kafka 横空出世	23
2.1.3	Kafka 开源	24
2.2	Kafka 版本变迁.....	25
2.2.1	Kafka 的版本演进	25
2.2.2	Kafka 的版本格式	26
2.2.3	新版本功能简介	26
2.2.4	旧版本功能简介	31
2.3	如何选择 Kafka 版本.....	35
2.3.1	根据功能场景.....	35
2.3.2	根据客户端使用场景.....	35
2.4	Kafka 与 Confluent.....	36
2.5	本章小结	37
第 3 章	Kafka 线上环境部署	38
3.1	集群环境规划	38
3.1.1	操作系统的选型.....	38
3.1.2	磁盘规划.....	40
3.1.3	磁盘容量规划.....	42

3.1.4	内存规划.....	43
3.1.5	CPU 规划.....	43
3.1.6	带宽规划.....	44
3.1.7	典型线上环境配置.....	45
3.2	伪分布式环境安装.....	45
3.2.1	安装 Java.....	46
3.2.2	安装 ZooKeeper.....	47
3.2.3	安装单节点 Kafka 集群.....	48
3.3	多节点环境安装.....	49
3.3.1	安装多节点 ZooKeeper 集群.....	50
3.3.2	安装多节点 Kafka.....	54
3.4	验证部署.....	55
3.4.1	测试 topic 创建与删除.....	55
3.4.2	测试消息发送与消费.....	57
3.4.3	生产者吞吐量测试.....	58
3.4.4	消费者吞吐量测试.....	58
3.5	参数设置.....	59
3.5.1	broker 端参数.....	59
3.5.2	topic 级别参数.....	62
3.5.3	GC 参数.....	63
3.5.4	JVM 参数.....	64
3.5.5	OS 参数.....	64
3.6	本章小结.....	65
第 4 章	producer 开发.....	66
4.1	producer 概览.....	66
4.2	构造 producer.....	69
4.2.1	producer 程序实例.....	69
4.2.2	producer 主要参数.....	75
4.3	消息分区机制.....	80
4.3.1	分区策略.....	80
4.3.2	自定义分区机制.....	80

4.4	消息序列化	83
4.4.1	默认序列化	83
4.4.2	自定义序列化	84
4.5	producer 拦截器	87
4.6	无消息丢失配置	90
4.6.1	producer 端配置	91
4.6.2	broker 端配置	92
4.7	消息压缩	92
4.7.1	Kafka 支持的压缩算法	93
4.7.2	算法性能比较与调优	93
4.8	多线程处理	95
4.9	旧版本 producer	96
4.10	本章小结	98
第 5 章	consumer 开发	99
5.1	consumer 概览	99
5.1.1	消费者 (consumer)	99
5.1.2	消费者组 (consumer group)	101
5.1.3	位移 (offset)	102
5.1.4	位移提交	103
5.1.5	__consumer_offsets	104
5.1.6	消费者组重平衡 (consumer group rebalance)	106
5.2	构建 consumer	106
5.2.1	consumer 程序实例	106
5.2.2	consumer 脚本命令	111
5.2.3	consumer 主要参数	112
5.3	订阅 topic	115
5.3.1	订阅 topic 列表	115
5.3.2	基于正则表达式订阅 topic	115
5.4	消息轮询	115
5.4.1	poll 内部原理	115
5.4.2	poll 使用方法	116

5.5	位移管理	118
5.5.1	consumer 位移	119
5.5.2	新版本 consumer 位移管理	120
5.5.3	自动提交与手动提交	121
5.5.4	旧版本 consumer 位移管理	123
5.6	重平衡 (rebalance)	123
5.6.1	rebalance 概览	123
5.6.2	rebalance 触发条件	124
5.6.3	rebalance 分区分配	124
5.6.4	rebalance generation	126
5.6.5	rebalance 协议	126
5.6.6	rebalance 流程	127
5.6.7	rebalance 监听器	128
5.7	解序列化	130
5.7.1	默认解序列化器	130
5.7.2	自定义解序列化器	131
5.8	多线程消费实例	132
5.8.1	每个线程维护一个 KafkaConsumer	133
5.8.2	单 KafkaConsumer 实例+多 worker 线程	135
5.8.3	两种方法对比	140
5.9	独立 consumer	141
5.10	旧版本 consumer	142
5.10.1	概览	142
5.10.2	high-level consumer	143
5.10.3	low-level consumer	147
5.11	本章小结	153
第 6 章	Kafka 设计原理	154
6.1	broker 端设计架构	154
6.1.1	消息设计	155
6.1.2	集群管理	166
6.1.3	副本与 ISR 设计	169

6.1.4	水印（watermark）和 leader epoch	174
6.1.5	日志存储设计	185
6.1.6	通信协议（wire protocol）	194
6.1.7	controller 设计	205
6.1.8	broker 请求处理	216
6.2	producer 端设计	219
6.2.1	producer 端基本数据结构	219
6.2.2	工作流程	220
6.3	consumer 端设计	223
6.3.1	consumer group 状态机	223
6.3.2	group 管理协议	226
6.3.3	rebalance 场景剖析	227
6.4	实现精确一次处理语义	230
6.4.1	消息交付语义	230
6.4.2	幂等性 producer（idempotent producer）	231
6.4.3	事务（transaction）	232
6.5	本章小结	234
第 7 章	管理 Kafka 集群	235
7.1	集群管理	235
7.1.1	启动 broker	235
7.1.2	关闭 broker	236
7.1.3	设置 JMX 端口	237
7.1.4	增加 broker	238
7.1.5	升级 broker 版本	238
7.2	topic 管理	241
7.2.1	创建 topic	241
7.2.2	删除 topic	243
7.2.3	查询 topic 列表	244
7.2.4	查询 topic 详情	244
7.2.5	修改 topic	245
7.3	topic 动态配置管理	246

7.3.1	增加 topic 配置.....	246
7.3.2	查看 topic 配置.....	247
7.3.3	删除 topic 配置.....	248
7.4	consumer 相关管理.....	248
7.4.1	查询消费者组.....	248
7.4.2	重设消费者组位移.....	251
7.4.3	删除消费者组.....	256
7.4.4	kafka-consumer-offset-checker.....	257
7.5	topic 分区管理.....	258
7.5.1	preferred leader 选举.....	258
7.5.2	分区重分配.....	260
7.5.3	增加副本因子.....	263
7.6	Kafka 常见脚本工具.....	264
7.6.1	kafka-console-producer 脚本.....	264
7.6.2	kafka-console-consumer 脚本.....	265
7.6.3	kafka-run-class 脚本.....	267
7.6.4	查看消息元数据.....	268
7.6.5	获取 topic 当前消息数.....	270
7.6.6	查询__consumer_offsets.....	271
7.7	API 方式管理集群.....	273
7.7.1	服务器端 API 管理 topic.....	273
7.7.2	服务器端 API 管理位移.....	275
7.7.3	客户端 API 管理 topic.....	276
7.7.4	客户端 API 查看位移.....	280
7.7.5	0.11.0.0 版本客户端 API.....	281
7.8	MirrorMaker.....	285
7.8.1	概要介绍.....	285
7.8.2	主要参数.....	286
7.8.3	使用实例.....	287
7.9	Kafka 安全.....	288
7.9.1	SASL+ACL.....	289
7.9.2	SSL 加密.....	297

7.10 常见问题	301
7.11 本章小结	304
第 8 章 监控 Kafka 集群	305
8.1 集群健康度检查	305
8.2 MBean 监控	306
8.2.1 监控指标	306
8.2.2 指标分类	308
8.2.3 定义和查询 JMX 端口	309
8.3 broker 端 JMX 监控	310
8.3.1 消息入站/出站速率	310
8.3.2 controller 存活 JMX 指标	311
8.3.3 备份不足的分区数	312
8.3.4 leader 分区数	312
8.3.5 ISR 变化速率	313
8.3.6 broker I/O 工作处理线程空闲率	313
8.3.7 broker 网络处理线程空闲率	314
8.3.8 单个 topic 总字节数	314
8.4 clients 端 JMX 监控	314
8.4.1 producer 端 JMX 监控	314
8.4.2 consumer 端 JMX 监控	316
8.5 JVM 监控	317
8.5.1 进程状态	318
8.5.2 GC 性能	318
8.6 OS 监控	318
8.7 主流监控框架	319
8.7.1 JmxTool	320
8.7.2 kafka-manager	320
8.7.3 Kafka Monitor	325
8.7.4 Kafka Offset Monitor	327
8.7.5 CruiseControl	329
8.8 本章小结	330

第 9 章 调优 Kafka 集群	331
9.1 引言	331
9.2 确定调优目标	333
9.3 集群基础调优	334
9.3.1 禁止 atime 更新	335
9.3.2 文件系统选择	335
9.3.3 设置 swapiness	336
9.3.4 JVM 设置	337
9.3.5 其他调优	337
9.4 调优吞吐量	338
9.5 调优延时	342
9.6 调优持久性	343
9.7 调优可用性	347
9.8 本章小结	349
第 10 章 Kafka Connect 与 Kafka Streams	350
10.1 引言	350
10.2 Kafka Connect	351
10.2.1 概要介绍	351
10.2.2 standalone Connect	353
10.2.3 distributed Connect	356
10.2.4 开发 connector	359
10.3 Kafka Streams	362
10.3.1 流处理	362
10.3.2 Kafka Streams 核心概念	364
10.3.3 Kafka Streams 与其他框架的异同	368
10.3.4 Word Count 实例	369
10.3.5 Kafka Streams 应用开发	372
10.3.6 Kafka Streams 状态查询	382
10.4 本章小结	386

第 1 章

认识 Apache Kafka

随着大数据时代的到来，数据中蕴含的价值日益得到展现，仿佛一座待人挖掘的金矿，引来无数的掘金者。但随着数据量越来越大，如何实时准确地收集并分析数据成为摆在所有从业人员面前的难题。

本章作为本书的第 1 章，将带领读者对 Apache Kafka 系统及其生态圈建立一个宏观的概念和认识。同时，本章将结合消息引擎系统的相关知识与设计理念，循序渐进地对 Kafka 系统的设计架构和相关概念进行展开，并给出简单示例以快速上手 Kafka。

学习本章，你将了解到以下内容。

- 消息引擎系统的定义与特点。
- Apache Kafka 系统概要设计及术语。
- Apache Kafka 快速入门。

1.1 Kafka 快速入门

Kafka 的核心功能是什么？一言以蔽之，高性能的消息发送与高性能的消息消费。接下来，本书打算先给出 Kafka 的快速入门教程，即 Kafka 的“Hello, world”示例。这么做与其他书不同，其目的就是可以让各位读者快速体验 Kafka 的核心功能——消息的发送与消费。这对于很多不太熟悉 Kafka 的初学者来说，可以让他们瞬间提升学习的快感，从而增强对于掌握与理解 Kafka 系统的信心。

作为消息引擎系统中的佼佼者，Kafka 诸多独特的设计理念及强大的功能特性非常值得学习，但现在我们首先跑通一个 Kafka 的简单示例，切身感受一下 Kafka 这个目前消息引擎领域

内的“王者”是什么样子的。

1.1.1 下载并解压缩 Kafka 二进制代码压缩包文件

Kafka 官网的下载地址是 <https://kafka.apache.org/downloads>，打开下载页面后我们可以看到不同版本的 Kafka 二进制代码压缩包下载链接，如图 1.1 所示。

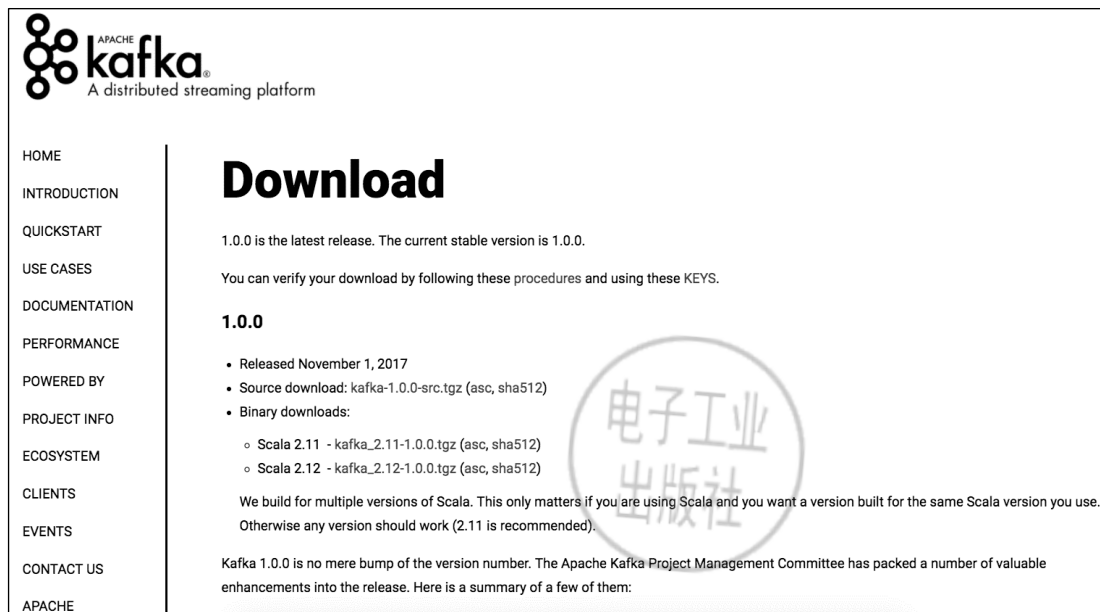


图 1.1 Kafka 下载页面

由图 1.1 可知，当前最新的 Kafka 版本是 1.0.0，提供了两个二进制压缩包可供下载。

- `kafka_2.11-1.0.0.tgz`
- `kafka_2.12-1.0.0.tgz`

上面两个文件中的 2.11/2.12 分别表示编译 Kafka 的 Scala 语言版本，后面的 1.0.0 是 Kafka 的版本。由于 Kafka 社区提供 Scala 2.12 编译包的时间还不长，目前还算不上十分稳定，所以我们这次选择 Scala 2.11 版本编译的 Kafka，即 `kafka_2.11-1.0.0.tgz`。

下载完毕之后，执行下列解压缩命令将解压缩后的目录放置到你的机器上的某个文件夹下：

```
tar -zxf kafka_2.11-1.0.0.tgz
cd kafka_2.11_1.0.0
```

1.1.2 启动服务器

解压缩成功后便应该启动 Kafka 服务器了。不过在此之前，我们需要首先启动 ZooKeeper 服务器（ZooKeeper 是为 Kafka 提供协调服务的工具，后续章节会讲述其具体设计原理）。Kafka 内置提供了一个 ZooKeeper 服务器以及一组相关的管理脚本，我们直接使用这个内置 ZooKeeper 即可，如下列命令：

```
bin/zookeeper-server-start.sh config/zookeeper.properties
[2017-12-19 21:13:08,280] INFO Reading configuration from: config/zookeeper.
properties (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
...
[2017-12-19 21:13:08,364] INFO binding to port 0.0.0.0/0.0.0.0:2181
(org.apache.zookeeper.server.NIOServerCnxnFactory)
```

在运行上面的命令前，必须提前安装好 Java 环境。对于运行 Kafka 而言，至少要求安装 Java 7 版本。从上面的输出中可以发现 0.0.0.0/0.0.0.0:2181 的字样，这通常表明 ZooKeeper 已经成功地在端口 2181 上启动了。接下来我们启动 Kafka 服务器：

```
bin/kafka-server-start.sh config/server.properties
...
[2017-04-19 21:15:16,432] INFO Kafka commitId : 576d93a8dc0cf421 (org.apache.
kafka.common.utils.AppInfoParser)
[2017-04-19 21:15:16,458] INFO [Kafka Server 0], started (kafka.server.
KafkaServer)
```

控制台输出结尾处的 “[Kafka Server 0], started” 标志 Kafka 服务器启动成功，默认的服务端口是 9092。

1.1.3 创建 topic

服务器启动后，我们需要创建一个主题（topic）用于消息的发送与接收。这一步将创建一个名为 test 的 topic，该 topic 只有一个分区（partition），且该 partition 也只有一个副本（replica）处理消息。我们将在 1.4 节中详细介绍 topic、partition 和 replica 的含义与它们彼此之间的关系。

为了创建 topic，我们要确保之前启动的 ZooKeeper 和 Kafka 的终端不被关闭，然后再打开一个新的终端，执行以下命令：

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

topic 创建成功了！接下来依然使用该命令来查看该 topic 的状态：

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

```
Topic:test PartitionCount:1 ReplicationFactor:1 Configs:  
Topic: test Partition: 0 Leader: 0 Replicas: 0 Isr: 0
```

这段命令的输出非常清晰地表明我们创建的 topic 名为 test，分区数（PartitionCount）是 1，副本数（ReplicationFactor）也是 1。

1.1.4 发送消息

Kafka 默认提供了脚本工具可以不断地接收标准输入并将它们发送到 Kafka 的某个 topic 上。用户在控制台终端下启动该命令，输入一行文本数据，然后该脚本将该行文本封装成一条 Kafka 消息发送给指定的 topic。

为了使用该脚本工具发送消息，用户需要再打开一个新的终端，执行下列命令：

```
bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test  
Hello, Kafka  
This is my first Kafka message.
```

打开这个控制台后用户可以不断地键入字符形成消息。每当按下回车键后该行文本即会被发送。另外，若一段时间内（默认是 30 秒）没有任何错误出现，那么这通常表明消息被成功发送了。按下 Ctrl + C 组合键则退出这个控制台。

1.1.5 消费消息

同样地，Kafka 也提供了一个对应的脚本用于消费某（些）topic 下的消息并打印到标准输出。这对于我们测试、调试消息消费非常方便。这一步中你需要再打开一个新的终端，然后执行下面的命令：

```
bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic  
test --from-beginning  
Hello, Kafka  
This is my first Kafka message.
```

如果之前的所有命令都成功运行，这一步你应该会看见上一步中发送的那两条消息，而且你还可以尝试在第一个消息发送终端中键入新的消息，之后马上切换到消息接收终端上看看是否可以看见这条新生产的消息。

值得注意的是，kafka-console-producer.sh 和 kafka-console-consumer.sh 脚本还有其他很多有用的参数选项，如果不加任何参数直接运行它们，则会打印它们各自的使用帮助文档。

另外，上面所有步骤中使用的都是 Linux/UNIX 下的脚本。如果使用的是 Windows 系统，那么改用 bin/windows/*.bat 脚本即可。它们的使用方法及命令调用的参数都是一致的。

好了，相信此时读者对 Kafka 已经有了一个大概的感性认识。下面我们将详细讨论 Kafka

的核心功能——消息引擎的理论背景并给出 Kafka 的概要设计。各位读者准备好了吗？让我们开始吧！

1.2 消息引擎系统

说到消息引擎，和它类似的术语有很多，比如消息队列、消息中间件等。笔者个人更愿意称它为消息引擎。毕竟“消息队列”这个名字给出了一个很不准确的暗示，仿佛它就是以队列的方式实现的，而“消息中间件”的提法似乎有过度夸张强调“中间件”之嫌，不免令人疑惑这东西真实的用途。

其实，这类系统的英文名是 **Messaging system**——国内很多文献简单地将其翻译成消息系统。笔者认为这样翻译并不是很准确，因为它只片面强调了消息主体的作用，而忽视了这类系统天然就具备的且很重要的传递属性（就像引擎一样，具备某种能量转换、传输的能力），因此笔者更倾向于将其翻译成消息引擎系统或消息传输系统。

根据维基百科的定义，企业消息引擎系统（EMS）是企业发布的一组规范。公司使用这组规范实现在不同系统之间传递语义准确的消息。在实际使用场景中，消息引擎系统通常以软件接口为主要形式，实现了松耦合的异步式数据传递语义。

既然是用于在不同应用间传输消息的系统，那么消息引擎系统中的消息自然是最关键的因素之一。其实，这里的消息可以是任何形式的数据，比如电子邮件、传真、即时消息，甚至是其他服务等，总之都是对企业有价值的信息，如图 1.2 所示。

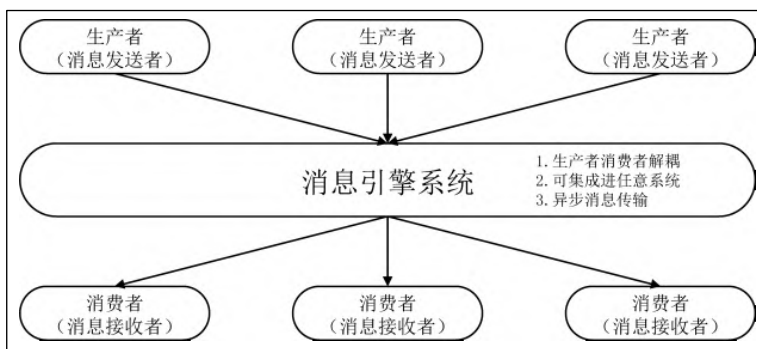


图 1.2 消息引擎架构图

如果我们仔细查看上述消息引擎的定义，就会发现其设计理念向我们传达了一个非常明确的信号，即在设计一个消息引擎系统时需要考虑的两个重要因素：

- 消息设计。
- 传输协议设计。

1.2.1 消息设计

消息引擎系统在设计消息时一定要考虑语义的清晰和格式上的通用性。一条消息要有能够完整清晰表达业务的能力，它不能是含糊不清、语义不明甚至无法处理的。同时，为了更好地表达语义以及最大限度地提高重用性，消息通常都采用结构化的方式进行设计。比如 SOAP 协议中的消息就采用了 XML 格式，而 Web Service 也支持 JSON 格式的消息。在后面的章节中我们会谈到 Kafka 的消息是用二进制方式来保存的，但依然是结构化的消息。

可以发现，不论是使用 XML、JSON、二进制表示，抑或是其他自定义的结构化类型，消息主体本身一般都是结构化的数据，这给后续消息引擎系统的处理带来了极大的便利。

1.2.2 传输协议设计

这是消息引擎系统中更为关键的部分——如何设计消息传输的协议。从狭义的角度来说，消息传输协议指定了消息在不同系统之间传输的方式。目前主流的协议包括 AMQP、Web Service + SOAP 以及微软的 MSMQ 等。从广义的角度来说，这类协议可能包括任何能够在不同系统间传输消息或是执行语义操作的协议或框架。比如现在主流的 RPC 及序列化框架，包括 Google 的 Protocol Buffers（简称 Google PB，值得注意的是，虽然 Google 并没有开源 PB 的 RPC 框架部分，但其依然是一款非常优秀的序列化框架）、阿里系的 Dubbo 等。

Kafka 自己设计了一套二进制的消息传输协议，而没有采用诸如 Google PB 这样的框架。在后面的章节中我们会详细阐述这其中的原因以及探讨 Kafka 消息传输协议及其背后的设计理念。

1.2.3 消息引擎范型

在 1.2.2 节中，我们提到了传输协议的设计。传输协议作为一个基础构建块（building block），其服务于消息引擎系统实现的消息引擎范型。根据维基百科的定义，一个消息引擎范型是一个基于网络的架构范型，描述了消息引擎系统的两个不同的子部分是如何互连且交互的。如果把消息引擎系统的这两个子系统比喻成两座城市，那么之前谈到的传输协议就是需要铺设的沥青公路，而引擎范型决定了来往穿梭于这两座城市的路线。

最常见的两种消息引擎范型是消息队列模型和发布/订阅模型。消息队列（message queue）模型是基于队列提供消息传输服务的，多用于进程间通信（inter-process communication, IPC）以及线程间通信。该模型定义了消息队列（queue）、发送者（sender）和接收者（receiver），提供了一种点对点（point-to-point, p2p）的消息传递方式，即发送者发送每条消息到队列的指

定位置，接收者从指定位置获取消息。一旦消息被消费（consumed），就会从队列中移除该消息。每条消息由一个发送者生产出来，且只被一个消费者（consumer）处理——发送者和消费者之间是一对一的关系。生活中接线生的工作就是一个典型的基于队列的消息引擎模型。每个打进来的电话都进入一个排队队列，然后只由一个接线生进行处理。同一个客户不会被第二个接线生处理。这就是典型的基于队列的消息引擎模型，如图 1.3 所示。

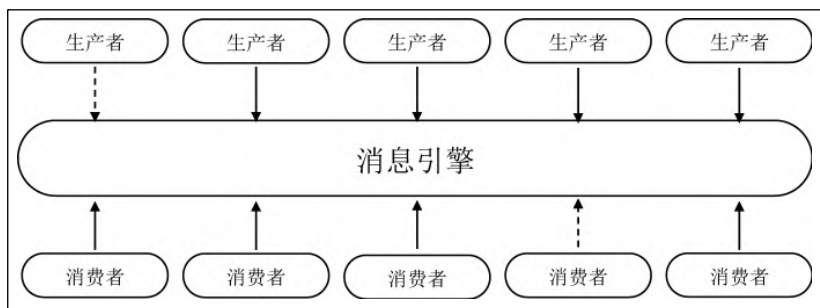


图 1.3 基于队列的消息引擎模型

而另一种模型就是发布/订阅模型（publish/subscribe，或简称为 pub/sub），与前一种模型不同，它有主题（topic）的概念：一个 topic 可以理解为由逻辑语义相近的消息的容器。这种模型也定义了类似于生产者/消费者这样的角色，即发布者（publisher）和订阅者（subscriber）。发布者将消息生产出来发送到指定的 topic 中，所有订阅了该 topic 的订阅者都可以接收到该 topic 下的所有消息。通常具有相同订阅 topic 的所有订阅者将接收到同样的消息，如图 1.4 所示。生活中报纸的订阅就是一种典型的发布/订阅模型：很多读者都会订阅同一个报社（类比于同一个 topic）出版的报纸，这样每当报纸更新（生产新的消息）时，这些读者都可以收到最新的报纸（接收最新的消息）。

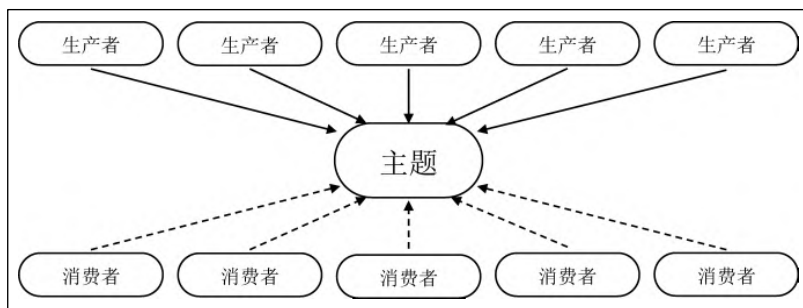


图 1.4 基于发布/订阅的消息引擎模型

显然，Kafka 必须要同时支持这两种消息引擎模型。在后续的章节中，我们将探讨 Kafka 如何引入消息组（consumer group）的概念来同时支持这两种模型。

1.2.4 Java 消息服务

Java 消息服务，即 Java Message Service（简称 JMS）。严格来说，它只是一套 API 规范，提供了很多接口用于实现分布式系统间的消息传递。JMS 同时支持上面两种消息引擎模型。实际上，当前很多主流的消息引擎系统都完全支持 JMS 规范，比如 ActiveMQ、RabbitMQ（通过 RabbitMQ JMS Client）、IBM WebSphere MQ 和 Kafka 等。当然 Kafka 并没有完全遵照 JMS 规范，它另辟蹊径，探索出了一条独有的道路。

1.3 Kafka 概要设计

前面简要介绍了消息引擎系统，而 Kafka 的设计初衷就是为了解决互联网公司超大量级数据的实时传输。为了实现这个目标，Kafka 在设计之初就需要考虑以下 4 方面的问题。

- 吞吐量/延时。
- 消息持久化。
- 负载均衡和故障转移。
- 伸缩性。

1.3.1 吞吐量/延时

对于任何一个消息引擎而言，吞吐量（throughput）都是至关重要的性能指标。那么何为吞吐量呢？通常来说，吞吐量是某种处理能力的最大值。而对于 Kafka 而言，它的吞吐量就是每秒能够处理的消息数或者每秒能够处理的字节数。很显然，我们自然希望消息引擎的吞吐量越大越好。

消息引擎系统还有一个名为延时的性能指标。它衡量的是一段间隔，可能是发出某个操作与接收到操作响应（response）之间的时间，或者是在系统中导致某些物理变更的起始时刻与变更正式生效时刻之间的间隔。对于 Kafka 而言，延时可以表示客户端发起请求与服务器处理请求并发送响应给客户端之间的这一段时间。显而易见，延时间隔越短越好。

在实际使用场景中，这两个指标通常是一对矛盾体，即调优其中一个指标通常会使另一个指标变差。当然它们之间的关系也不是等比例地此消彼长的关系，比如牺牲 20% 的延时就能换来 20% 的吞吐量的提升。其实关于这一点，很多读者可能会感到困惑：为什么需要同时研究这两个指标呢？如果确定了其中的一个指标，另一个指标也应该是确定的才对啊？好吧，按照这种假设，若 Kafka 处理一条消息需要花费 2 毫秒，那么计算得到的吞吐量不会超过 500 条消息/秒（ $1000/2=500$ ）。

但是若我们采用批处理（batching）的思想，还是使用上面的例子，情况就可以大不一样了。这一次我们不是一条一条地发送消息，而是一小批一小批（micro-batch）地发送。假设在发送前我们首先会等待一段时间（假设是 8 毫秒），那么此时消息发送的延时变成了 10 毫秒（2+8），即延时增加了 4 倍，但假设在这 8 毫秒中我们总共累积了 1000 条消息，那么系统整体的吞吐量就变成了 100000 条/秒（ $1000/0.01=100000$ ），吞吐量提升了近 200 倍！各位读者，看到 micro-batch 的威力了吧？这也就是目前诸如 Storm Trident 和 Spark Streaming 等消息处理平台所采用的处理语义思路。

好了，说了这么多吞吐量和延时，那么对于 Kafka 而言它是如何做到高吞吐量、低延时的呢？首先，Kafka 的写入操作是很快的，这主要得益于它对磁盘的使用方法的不同。虽然 Kafka 会持久化所有数据到磁盘，但本质上每次写入操作其实都只是把数据写入到操作系统的页缓存（page cache）中，然后由操作系统自行决定什么时候把页缓存中的数据写回磁盘上。这样的设计有 3 个主要优势。

- 操作系统页缓存是在内存中分配的，所以消息写入的速度非常快。
- Kafka 不必直接与底层的文件系统打交道。所有烦琐的 I/O 操作都交由操作系统来处理。
- Kafka 写入操作采用追加写入（append）的方式，避免了磁盘随机写操作。

各位读者请特别留意上面的第 3 点。对于普通的物理磁盘（非固态硬盘）而言，我们总是认为磁盘的读/写操作是很慢的。事实上普通 SAS 磁盘随机读/写的吞吐量的确是很慢的，但是磁盘的顺序读/写操作其实是非常快的，它的速度甚至可以匹敌内存的随机 I/O 速度，如图 1.5 所示。随机内存 I/O 的速度是 36.7MB/s（这里我们并不关心具体的值是什么，暂且假设单位是 MB/s。实际上只需要保证比较的双方具有相同的单位即可），而顺序磁盘 I/O 的速度甚至达到了 52.2MB/s，丝毫不逊于内存的 I/O 操作性能。

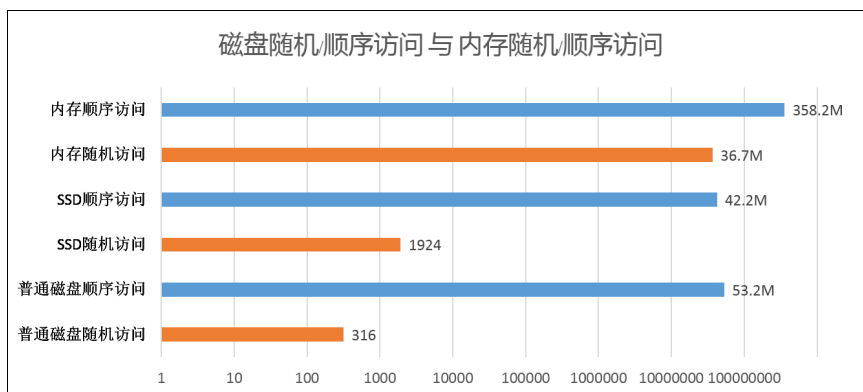


图 1.5 磁盘与内存的随机/顺序访问速度比较

鉴于这一事实，Kafka 在设计时采用了追加写入消息的方式，即只能在日志文件末尾追加写入新的消息，且不允许修改已写入的消息，因此它属于典型的磁盘顺序访问型操作，所以 Kafka 消息发送的吞吐量是很高的。在实际使用过程中可以很轻松地做到每秒写入几万甚至几十万条消息。

下面我们来看看 Kafka 的消费端是如何做到高吞吐量、低延时的。之前提到了 Kafka 是把消息写入操作系统的页缓存中的。那么同样地，Kafka 在读取消息时会首先尝试从 OS 的页缓存中读取，如果命中便把消息经页缓存直接发送到网络的 Socket 上。这个过程就是利用 Linux 平台的 sendfile 系统调用做到的，而这种技术就是大名鼎鼎的零拷贝（Zero Copy）技术。

为了方便不太了解 sendfile 和零拷贝的读者理解，笔者在这里简单介绍一下它的特性。传统的 Linux 操作系统中的 I/O 接口是依托于数据拷贝来实现的，但在零拷贝技术出现之前，一个 I/O 操作会将同一份数据进行多次拷贝，如图 1.6 所示。数据传输过程中还涉及内核态与用户态的上下文切换，CPU 的开销非常大，因此极大地限制了 OS 高效进行数据传输的能力。零拷贝技术很好地改善了这个问题：首先在内核驱动程序处理 I/O 数据的时候，它不再需要进行上下文的切换，节省了内核缓冲区与用户态应用程序缓冲区之间的数据拷贝，同时它利用直接存储器访问技术（Direct Memory Access, DMA）执行 I/O 操作，因此也避免了 OS 内核缓冲区之间的数据拷贝，故而得名零拷贝，如图 1.7 所示。

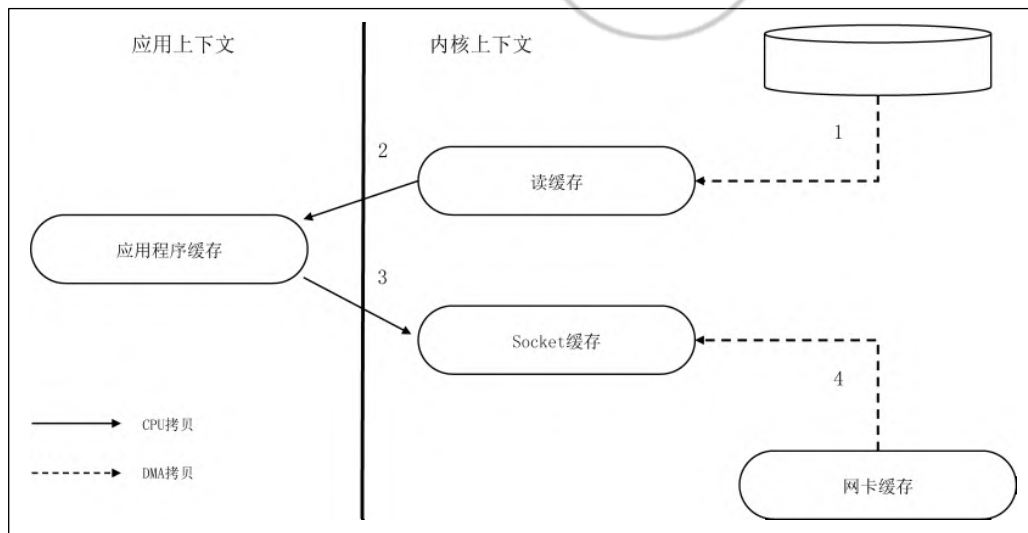


图 1.6 无零拷贝数据传输

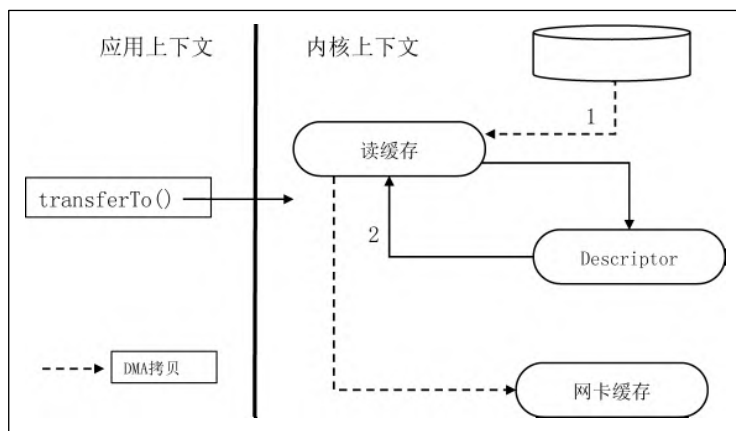


图 1.7 零拷贝数据传输

Linux 提供的 `sendfile` 系统调用实现了这种零拷贝技术，而 Kafka 的消息消费机制使用的就是 `sendfile`——严格来说是通过 Java 的 `FileChannel.transferTo` 方法实现的。

除了零拷贝技术，Kafka 由于大量使用页缓存，故读取消息时大部分消息很有可能依然保存在页缓存中，因此可以直接命中缓存，不用“穿透”到底层的物理磁盘上获取消息，从而极大地提升了消息读取的吞吐量。事实上，如果我们监控一个经过良好调优的 Kafka 生产集群便可以发现，即使是那些有负载的 Kafka 服务器，其磁盘的读操作也很少，这是因为大部分的消息读取操作会直接命中页缓存。

总结一下，Kafka 就是依靠下列 4 点达到了高吞吐量、低延时的设计目标的。

- 大量使用操作系统页缓存，内存操作速度快且命中率高。
- Kafka 不直接参与物理 I/O 操作，而是交由最擅长此事的操作系统来完成。
- 采用追加写入方式，摒弃了缓慢的磁盘随机读/写操作。
- 使用以 `sendfile` 为代表的零拷贝技术加强网络间的数据传输效率。

1.3.2 消息持久化

Kafka 是要持久化消息的，而且要把消息持久化到磁盘上。这样做的好处如下。

- **解耦消息发送与消息消费：**本质上来说，Kafka 最核心的功能就是提供了生产者-消费者模式的完整解决方案。通过将消息持久化使得生产者方不再需要直接和消费者方耦合，它只是简单地把消息生产出来并交由 Kafka 服务器保存即可，因此提升了整体的吞吐量。
- **实现灵活的消息处理：**很多 Kafka 的下游子系统（接收 Kafka 消息的系统）都有这样的需求——对于已经处理过的消息可能在未来的某个时间点重新处理一次，即所谓的

消息重演（message replay）。消息持久化便可以很方便地实现这样的需求。

另外，Kafka 实现持久化的设计也有新颖之处。普通的系统在实现持久化时可能会先尽量使用内存，当内存资源耗尽时，再一次性地把数据“刷盘”；而 Kafka 则反其道而行之，所有数据都会立即被写入文件系统的持久化日志中，之后 Kafka 服务器才会返回结果给客户端通知它们消息已被成功写入。这样做既实时保存了数据，又减少了 Kafka 程序对于内存的消耗，从而将节省出的内存留给页缓存使用，更进一步地提升了整体性能。

1.3.3 负载均衡和故障转移

作为一个功能完备的分布式系统，Kafka 如果只提供了最基本的消息引擎功能肯定不足以帮助它脱颖而出。一套完整的消息引擎解决方案中必然要提供负载均衡（load balancing）和故障转移（fail-over）功能。

我们先来讨论负载均衡的实现。何为负载均衡？顾名思义就是让系统的负载根据一定的规则均衡地分配在所有参与工作的服务器上，从而最大限度地提升系统整体的运行效率。具体到 Kafka 来说，默认情况下 Kafka 的每台服务器都有均等的机会为 Kafka 的客户提供服务，可以把负载分散到所有集群中的机器上，避免出现“耗尽某台服务器”的情况发生。

Kafka 实现负载均衡实际上是通过智能化的分区领导者选举（partition leader election）来实现的。我们会在 1.4 节中详细讨论 partition 以及 leader 的含义，但现在各位读者只需要知道 Kafka 默认提供了很智能的 leader 选举算法，可以在集群的所有机器上以均等机会分散各个 partition 的 leader，从而整体上实现了负载均衡。

除了负载均衡，完备的分布式系统还需要支持故障转移。所谓故障转移，是指当服务器意外中止时，整个集群可以快速地检测到该失效（failure），并立即将该服务器上的应用或服务自动转移到其他服务器上。故障转移通常是以“心跳”或“会话”的机制来实现的，即只要主服务器与备份服务器之间的心跳无法维持或主服务器注册到服务中心的会话超时过期了，那么就认为主服务器已无法正常运行，集群会自动启动某个备份服务器来替代主服务器的工作。

Kafka 服务器支持故障转移的方式就是使用会话机制。每台 Kafka 服务器启动后会以会话的形式把自己注册到 ZooKeeper 服务器上。一旦该服务器运转出现问题，与 ZooKeeper 的会话便不能维持从而超时失效，此时 Kafka 集群会选举出另一台服务器来完全代替这台服务器继续提供服务，如图 1.8 所示。

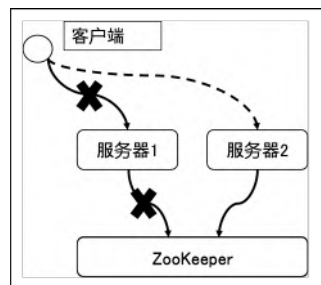


图 1.8 Kafka 故障转移

1.3.4 伸缩性

有了消息的持久化，Kafka 实现了高可靠性；有了负载均衡和使用文件系统的独特设计，Kafka 实现了高吞吐量；有了故障转移，Kafka 实现了高可用性。那么作为分布式系统中的高伸缩性，Kafka 又是如何做到的呢？

所谓伸缩性，英文名是 scalability。根据 Java 大神 Brian Goetz 在其经典著作 *Java Concurrency in Practice* 中的定义，伸缩性表示向分布式系统中增加额外的计算资源（比如 CPU、内存、存储或带宽）时吞吐量提升的能力。举一个例子来说，对于计算密集型（computation-intensive）的业务而言，CPU 的消耗一定是最大的，这类系统上的操作我们称之为 CPU-bound。那么如果一个 CPU 的运算能力是 U，我们自然希望两个 CPU 的运算能力是 2U，即可以线性地扩容计算能力，这种线性伸缩性是最理想的状态，但在实际中几乎不可能达到，毕竟分布式系统中有很多隐藏的“单点”瓶颈制约了这种线性的计算能力扩容。

阻碍线性扩容的一个很常见的因素就是状态的保存。我们知道，不论是哪类分布式系统，集群中的每台服务器一定会维护很多内部状态。如果由服务器自己来保存这些状态信息，则必须要处理一致性的问题。相反，如果服务器是无状态的，状态的保存和管理交于专门的协调服务来做（比如 ZooKeeper），那么整个集群的服务器之间就无须繁重的状态共享，这极大地降低了维护复杂度。倘若要扩容集群节点，只需简单地启动新的节点机器进行自动负载均衡就可以了。

Kafka 正是采用了这样的思想——每台 Kafka 服务器上的状态统一交由 ZooKeeper 保管。扩展 Kafka 集群也只需要一步：启动新的 Kafka 服务器即可。当然这里需要言明的是，在 Kafka 服务器上并不是所有状态都不保存，它只保存了很轻量级的内部状态，因此在整个集群间维护状态一致性的代价是很低的。

总结一下，上面主要讨论了 Kafka 在吞吐量/延时、消息持久化、负载均衡/故障转移和伸缩性这 4 个方面的设计理念。正是得益于这些设计特性，Kafka 才成为了一个完备的分布式消息引擎解决方案，赢得了广大用户的赞誉。

1.4 Kafka 基本概念与术语

Kafka 到底是什么？应该这样讲，目前 Kafka 的标准定位是分布式流式处理平台。嗯？等等！之前不是一直说 Kafka 是一个消息引擎系统吗？怎么这里又说它是流式处理平台了？各位读者不要着急，容笔者慢慢道来。

Kafka 自推出伊始的确是以消息引擎的身份出现的，其强大的消息传输效率和完备的分布

式解决方案，使它很快成为业界翘楚。随着 Kafka 的不断演进，Kafka 开发团队日益发现经 Kafka 交由下游数据处理平台做的事情 Kafka 自己也可以做，因此在 Kafka 0.10.0.0 版本正式推出了 Kafka Streams，即流式处理组件。自此 Kafka 正式成为了一个流式处理框架，而不仅仅是消息引擎了。

其实不管是消息引擎还是流式处理平台，Kafka 的处理流程从未发生质的变化。所谓万变不离其宗，概括起来如图 1.9 所示。

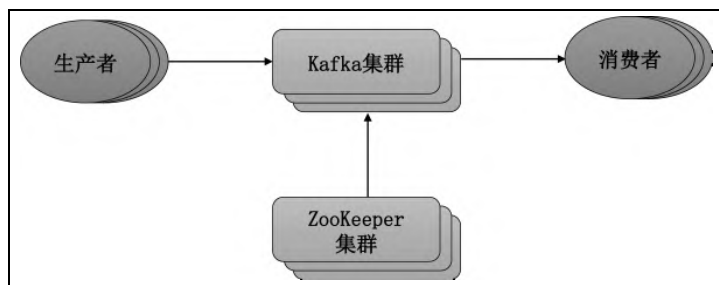


图 1.9 Kafka 架构图

由图 1.9 可见，不论 Kafka 如何变迁，其核心架构总是类似的，无非是生产一些消息然后再消费一些消息。如果总结起来那就是三句话：

- 生产者发送消息给 Kafka 服务器。
- 消费者从 Kafka 服务器读取消息。
- Kafka 服务器依托 ZooKeeper 集群进行服务的协调管理。

到目前为止，本书一直使用 Kafka 服务器这样的字眼。事实上，Kafka 服务器有一个官方名字：**broker**——鉴于中文并没有特别合适的名词对应，以后本书会统一使用 **broker** 来表示 Kafka 服务器。

值得注意的是，本书主要讨论 Kafka 消息引擎核心功能的使用。鉴于目前流式处理组件刚刚推出，所以只会对其做简要介绍。

Kafka 有一些基本术语需要掌握，这是后续学习 Kafka 的基础。首先，Kafka 是分布式的集群。一个集群可能由一台或多台机器组成。事实上，国外大型互联网公司的 Kafka 集群会有上千台之多，而在 Kafka 集群中保存的每条消息都归属于一个 **topic**，即之前提到的 **topic**。本节将分别从消息、**topic**、**partition** 和 **replica** 几个方面详细介绍 Kafka 的基本概念。

1.4.1 消息

既然 Kafka 的核心功能就是消息引擎，那么对于消息的设计自然是首当其冲的事情。

Kafka 没有令人失望，其对消息格式的设计与保存的确有很多创新之处。

首先，Kafka 中的消息格式由很多字段组成，其中的很多字段都是用于管理消息的元数据字段，对用户来说是完全透明的。Kafka 消息格式共经历过 3 次变迁，它们被分别称为 V0、V1 和 V2 版本。目前大部分用户使用的应该还是 V1 版本的消息格式。V1 版本消息的完整格式如图 1.10 所示。（我们会在第 6 章中详细讨论这 3 种版本。）

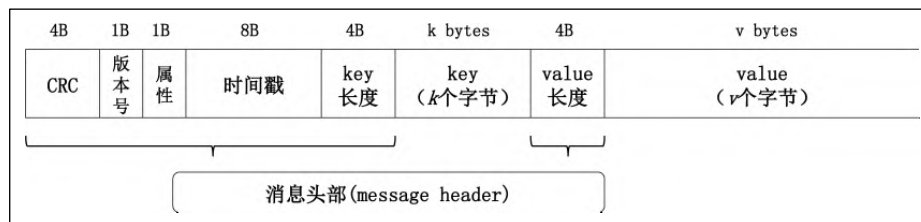


图 1.10 消息的完整格式

如图 1.10 所示，消息由消息头部、key 和 value 组成。消息头部包括消息的 CRC 码、消息版本号、属性、时间戳、键长度和消息体长度等信息。其实，对于普通用户来说，掌握以下 3 个字段的含义就足够一般的使用了。

- **Key:** 消息键，对消息做 partition 时使用，即决定消息被保存在某 topic 下的哪个 partition。
- **Value:** 消息体，保存实际的消息数据。
- **Timestamp:** 消息发送时间戳，用于流式处理及其他依赖时间的处理语义。如果不指定则取当前时间。

另外这里单独提一下消息的属性字段，Kafka 为该字段分配了 1 字节。目前只使用了最低的 3 位用于保存消息的压缩类型，其余 5 位尚未使用。当前只支持 4 种压缩类型：0（无压缩）、1（GZIP）、2（Snappy）和 3（LZ4）。

其次，Kafka 使用紧凑的二进制字节数组来保存上面这些字段，也就是说没有任何多余的比特位浪费。试想如果我们使用 Java 对象来保存上面的消息格式，假设我们简化定义 Message Java 类为：

```
public class Message {
    private CRC32 crc;
    private short version;
    private boolean codecEnabled;
    private short codecClassOrdinal;
    private String key;
    private String value;
}
```

可以看到上面 Java 类实现方式是非常朴素的，基本上就是由一个 CRC32 校验码、一个版本号、两个与压缩相关的字段以及消息键值和消息体组成的。在 Java 内存模型（Java memory model, JMM）中，对象保存的开销其实相当大，对于小对象而言，通常要花费 2 倍的空间来保存数据（甚至更糟）。另外，随着堆上数据量越来越大，GC 的性能会下降很多，从而整体上拖慢了系统的吞吐量。

尽管 JMM 会对上面的 Java 类进行优化——重排各个字段在内存的布局以减少内存使用量，但该 Message 类的一个实例依然需要消耗 40 个左右的字节，而其中有 7 字节只是为了补齐之用（padding）。更糟的是，运行 Java 的操作系统通常都默认开启了页缓存机制，也就是说堆上保存的对象很有可能在页缓存中还保留一份，这造成了极大的资源浪费。

因此 Kafka 在消息设计时特意避开了繁重的 Java 堆上内存分配，直接使用紧凑二进制字节数组 ByteBuffer 而不是独立的对象，因此我们至少能够访问多一倍的可用内存。按照 Kafka 官网的说法，在一台 32GB 内存的机器上，Kafka 几乎能用到 28~30GB 的物理内存，同时还不必担心 GC 的糟糕性能。如果使用 ByteBuffer 来保存同样的消息，只需要 24 字节，比起纯 Java 堆的实现减少了 40% 的空间占用，好处不言而喻。这种设计的好处还包括加入了扩展的可能性。

同时，大量使用页缓存而非堆内存还有一个好处——当出现 Kafka broker 进程崩溃时，堆内存上的数据也一并消失，但页缓存的数据依然存在。下次 Kafka broker 重启后可以继续提供服务，不需要再单独“热”缓存了。

1.4.2 topic 和 partition

在之前的 1.1 节中我们谈到了 Kafka 中的主题（topic）和分区（partition）。在本节中我们详细说说这两个 Kafka 核心概念。

从概念上来说，topic 只是一个逻辑概念，代表了一类消息，也可以认为是消息被发送到的地方。通常我们可以使用 topic 来区分实际业务，比如业务 A 使用一个 topic，业务 B 使用另外一个 topic。

Kafka 中的 topic 通常都会被多个消费者订阅，因此出于性能的考量，Kafka 并不是 topic-message 的两级结构，而是采用了 topic-partition-message 的三级结构来分散负载。从本质上说，每个 Kafka topic 都由若干个 partition 组成，如图 1.11 所示。

这张来自 Kafka 官网的 topic 和 partition 关系图非常清楚地表明了它们二者之间的关系：topic 是由多个 partition 组成的，而 Kafka 的 partition 是不可修改的有序消息序列，也可以说是有序的消息日志。每个 partition 有自己专属的 partition 号，通常是从 0 开始的。用户对 partition 唯一能做的操作就是在消息序列的尾部追加写入消息。partition 上的每条消息都会被

分配一个唯一的序列号——按照 Kafka 的术语来讲，该序列号被称为位移（offset）。该位移值是从 0 开始顺序递增的整数。位移信息可以唯一定位到某 partition 下的一条消息。

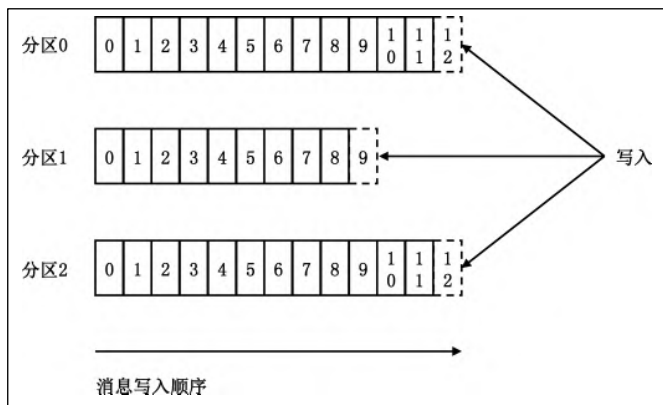


图 1.11 topic 和 partition

值得一提的是，Kafka 的 partition 实际上并没有太多的业务含义，它的引入就是单纯地为了提升系统的吞吐量，因此在创建 Kafka topic 的时候可以根据集群实际配置设置具体的 partition 数，实现整体性能的最大化。

1.4.3 offset

前面说过，topic partition 下的每条消息都被分配一个位移值。实际上，Kafka 消费者端也有位移（offset）的概念，但一定要注意这两个 offset 属于不同的概念，如图 1.12 所示。

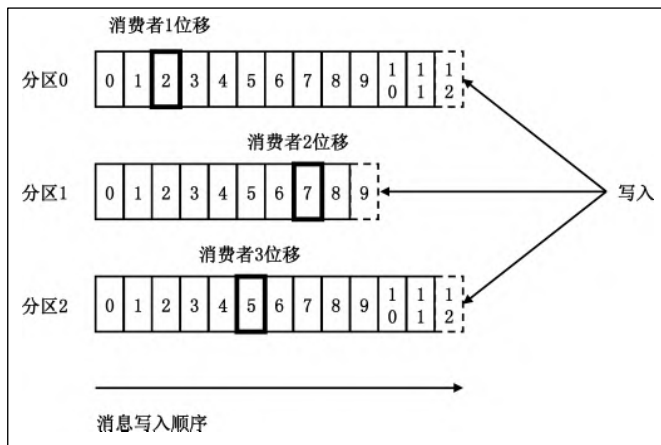


图 1.12 消息位移与消费者位移

显然，每条消息在某个 partition 的位移是固定的，但消费该 partition 的消费者的位移会随着消费进度不断前移，但终究不可能超过该分区最新一条消息的位移。因此以后在讨论位移的问题时一定要给出清晰的上下文环境，这样才能明确要讨论的是哪个位移。

综合之前说的 topic、partition 和 offset，我们可以断言 Kafka 中的一条消息其实就是一个 <topic,partition,offset>三元组（tuple），通过该元组值我们可以在 Kafka 集群中找到唯一对应的那条消息。

1.4.4 replica

既然我们已知 partition 是有序消息日志，那么一定不能只保存这一份日志，否则一旦保存 partition 的 Kafka 服务器挂掉了，其上保存的消息也就都丢失了。分布式系统必然要实现高可靠性，而目前实现的主要途径还是依靠冗余机制——简单地说，就是备份多份日志。这些备份日志在 Kafka 中被称为副本（replica），它们存在的唯一目的就是防止数据丢失，这一点一定要记住！

副本分为两类：领导者副本（leader replica）和追随者副本（follower replica）。follower replica 是不能提供服务给客户端的，也就是说不负责响应客户端发来的消息写入和消息消费请求。它只是被动地向领导者副本（leader replica）获取数据，而一旦 leader replica 所在的 broker 宕机，Kafka 会从剩余的 replica 中选举出新的 leader 继续提供服务。下面我们就来看看什么是 leader 和 follower。

1.4.5 leader 和 follower

如前所述，Kafka 的 replica 分为两个角色：领导者（leader）和追随者（follower）。如今这种角色设定几乎完全取代了过去的主备的提法（Master-Slave）。和传统主备系统（比如 MySQL）不同的是，在这类 leader-follower 系统中通常只有 leader 对外提供服务，follower 只是被动地追随 leader 的状态，保持与 leader 的同步。follower 存在的唯一价值就是充当 leader 的候补：一旦 leader 挂掉立即就会有一个追随者被选举成为新的 leader 接替它的工作。Kafka 就是这样的设计，如图 1.13 所示。

Kafka 保证同一个 partition 的多个 replica 一定不会分配在同一台 broker 上。毕竟如果同一个 broker 上有同一个 partition 的多个 replica，那么将无法实现备份冗余的效果。

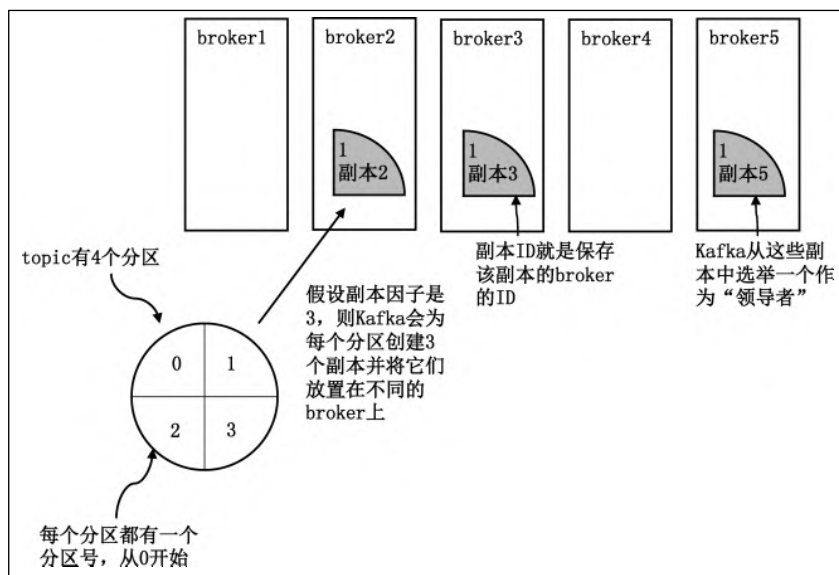


图 1.13 Kafka leader-follower 系统

1.4.6 ISR

ISR 的全称是 in-sync replica，翻译过来就是与 leader replica 保持同步的 replica 集合。这是一个特别重要的概念。前面讲了很多关于 Kafka 的副本机制，比如一个 partition 可以配置 N 个 replica，那么这是否就意味着该 partition 可以容忍 $N-1$ 个 replica 失效而不丢失数据呢？答案是“否”！

Kafka 为 partition 动态维护一个 replica 集合。该集合中的所有 replica 保存的消息日志都与 leader replica 保持同步状态。只有这个集合中的 replica 才能被选举为 leader，也只有该集合中所有 replica 都接收到了同一条消息，Kafka 才会将该消息置于“已提交”状态，即认为这条消息发送成功。回到刚才的问题，Kafka 承诺只要这个集合中至少存在一个 replica，那些“已提交”状态的消息就不会丢失——记住这句话的两个关键点：①ISR 中至少存在一个“活着的” replica；②“已提交”消息。有些 Kafka 用户经常抱怨：我向 Kafka 发送消息失败，然后造成数据丢失。其实这是混淆了 Kafka 的消息交付承诺（message delivery semantic）：Kafka 对于没有提交成功的消息不做任何交付保证，它只保证在 ISR 存活的情况下“已提交”的消息不会丢失。

正常情况下，partition 的所有 replica（含 leader replica）都应该与 leader replica 保持同步，即所有 replica 都在 ISR 中。因为各种各样的原因，一小部分 replica 开始落后于 leader replica 的进度。当滞后到一定程度时，Kafka 会将这些 replica “踢”出 ISR。相反地，当这

些 replica 重新“追上”了 leader 的进度时，那么 Kafka 会将它们加回到 ISR 中。这一切都是自动维护的，不需要用户进行人工干预，因而在保证了消息交付语义的同时还简化了用户的操作成本。

1.5 Kafka 使用场景

Kafka 以消息引擎闻名，因此它特别适合处理生产环境中的那些流式数据。以下就是 Kafka 在实际应用中的一些典型使用场景。

1.5.1 消息传输

Kafka 非常适合替代传统的消息总线（message bus）或消息代理（message broker）。传统的这类系统擅长于解耦生产者和消费者以及批量处理消息，而这些特点 Kafka 都具备。除此之外，Kafka 还具有更好的吞吐量特性，其内置的分区机制和副本机制既实现了高性能的消息传输，同时还达到了高可靠性和高容错性。因此 Kafka 特别适合用于实现一个超大量级消息处理应用。

1.5.2 网站行为日志追踪

Kafka 最早就是用于重建用户行为数据追踪系统的。很多网站上的用户操作都会以消息的形式发送到 Kafka 的某个对应的 topic 上。这些点击流蕴含了巨大的商业价值，事实上，目前就有很多创业公司使用机器学习或其他实时处理框架来帮助收集并分析用户的点击流数据。鉴于这种点击流数据量是很大的，Kafka 超强的吞吐量特性此时就有了用武之地。

1.5.3 审计数据收集

很多企业和组织都需要对关键的操作和运维进行监控和审计。这就需要从各个运维应用程序处实时汇总操作步骤信息进行集中式管理。在这种使用场景下，你会发现 Kafka 是非常适合的解决方案，它可以便捷地对多路消息进行实时收集，同时由于其持久化的特性，使得后续离线审计成为可能。

1.5.4 日志收集

这可能是 Kafka 最常见的使用方式了——日志收集汇总解决方案。每个企业都会产生大量的服务日志，这些日志分散在不同的机器上。我们可以使用 Kafka 对它们进行全量收集，并集中送往下游的分布式存储中（比如 HDFS 等）。比起其他主流的日志抽取框架（比如 Apache

Flume)，Kafka 有更好的性能，而且提供了完备的可靠性解决方案，同时还保持了低延时的特点。

1.5.5 Event Sourcing

Event Sourcing 实际上是领域驱动设计（Domain-Driven Design, DDD）的名词，它使用事件序列来表示状态变更，这种思想和 Kafka 的设计特性不谋而合。还记得吧，Kafka 也是用不可变更的消息序列来抽象化表示业务消息的，因此 Kafka 特别适合作为这种应用的后端存储。

1.5.6 流式处理

前面简要提到过，很多用户接触到 Kafka 都是因为它的消息引擎功能。自 0.10.0.0 版本开始，Kafka 社区推出了一个全新的流式处理组件 Kafka Streams。这标志着 Kafka 正式进入流式处理框架俱乐部。相比老牌流式处理框架 Apache Storm、Apache Samza，或是最近风头正劲的 Spark Streaming，抑或是 Apache Flink，Kafka Streams 的竞争力如何？让我们拭目以待。

1.6 本章小结

本章首先给出了 Kafka 的“Hello,world”实例带领读者快速入门，然后简要介绍了消息引擎系统的背景知识，包括消息引擎系统的定义、消息范型等。之后又分别从吞吐量/延时、持久化、负载均衡、故障转移和伸缩性几个方面概要介绍了 Kafka 的设计理念，同时讨论了掌握 Kafka 必备的基本概念和术语。本章最后给读者呈现了 6 个典型的 Kafka 使用场景。

通过本章的学习，希望读者可以迅速掌握 Kafka 是用来做什么的，最大限度地压缩初次了解 Kafka 的学习和使用成本。

第 2 章

Kafka 发展历史

学习本章，你将了解到以下内容。

- Kafka 的历史。
- Apache Kafka 版本更迭。
- Apache Kafka 新旧版本功能概要。



2.1 Kafka 的历史

想要全面掌握一个框架的使用方法，除了学习框架的基本用法与设计理念外，了解该框架产生的背景和历史将是大有裨益的。通常，它以一种全新的角度从一个侧面帮助我们理解该框架所解决的实际问题。本节我们就来了解一下 Apache Kafka 的“前世今生”。

2.1.1 背景

众所周知，Kafka 最早是由美国领英公司（下称 LinkedIn）的工程师们研发的，当时主要用于解决 LinkedIn 数据管道（data pipeline）的问题。

根据笔者能够查阅到的公开信息显示，LinkedIn 作为一家体量很大的互联网公司，必然有数据强实时处理方面的需求。当时在 LinkedIn 内部有诸多的子系统用于执行各种数据的收集与分析，最主要的两类包括：

- 业务系统和应用程序的性能监控指标数据。
- 用户操作行为数据。

对于第一类数据而言，虽然开源社区有一些可能的解决方案和框架，但 LinkedIn 内部还

是开发了一套自有系统来监控和追踪这些数据。比起已有的开源解决方案，这套公司内部研发的系统可以高度匹配公司数据业务需求，有很强的针对性。但在当时，这套系统也暴露出了一些明显的缺陷。

- 数据正确性不足——比如在收集这些监控指标数据时采用了基于轮询（polling）的方式，那么必然需要指定轮询的时间间隔——这个间隔可能就是一个启发式（heuristic）决策的结果。如果指定不当，必然造成数据的不准确。
- 系统需要高度定制化，人工维护成本很高——这套系统普适性较差，如果要对公司内部各个业务系统的数据收集，则需要做大量定制操作，从而引入大量的人工成本。

与此同时，LinkedIn 自行研发了另外一套系统用于收集第二类数据。这套系统可以将业务服务器中的数据定期地以 XML 消息格式发送到一个统一的地方用于离线批处理（offline batch processing）。既然是离线处理，必然无法做到强实时性，而对数据进行实时处理几乎已经成为当下所有互联网公司最迫切的基本需求。

Google 的 Tyler Akidau 大神曾在 *The world beyond batch: Streaming 101* 一文中指出，流式处理只要能够实现以下两个方面就能完全替代当前的离线批处理方式。

- **正确性**：一旦流式处理实现了正确性，它便足以匹敌批处理。
- **时间推导工具**：一旦流式处理提供了时间推导工具，它便完全超过了批处理。

由于本书并不是专门讨论流式处理的书籍，故这里就不做过多展开了。我们只需要了解批处理的固有缺陷是无法达成实时性就可以了。

总之，LinkedIn 实现了两套（当然，也有可能是多套）系统分别收集以上这两类数据。据公开资料显示，这两套系统无法实现交互、实时性差，而且维护成本很高。基于这些问题，LinkedIn 的工程师们尝试使用 ActiveMQ（另外一种很流行的消息引擎框架）来解决这一切，不过鉴于 ActiveMQ 扩展性不足且问题较多，最终还是放弃了。

上面所说的所有努力都预示着必将有一个“大一统”的系统出现从而取代原有系统。没错，这就是 Kafka！

2.1.2 Kafka 横空出世

为了解决以上问题，LinkedIn 的工程师团队设计出了一套分布式的高性能消息引擎服务，并将其命名为 Kafka。

值得一提的是，很多用户都好奇 Kafka 这个名字的由来。笔者也是在查阅了大量的资料之后发现了 Kafka 三位原作者之一 Jay Kreps 的这句话：

I thought that since Kafka was a system optimized for writing using a writer's name would make sense. I had taken a lot of lit classes in college and liked Franz Kafka. Plus the name sounded cool for an open source project.

翻译一下就是：因为 Kafka 系统的写操作性能特别强，所以找个作家的名字来命名似乎是一个好主意。我在大学时上了很多文学课，非常喜欢 Franz Kafka。另外为开源项目起 Kafka 这个名字听上去很酷。

另外再提一句，Jay Kreps 是流式处理方面大神级别的人物，也是公开资料中显示的 Kafka 三位原作者之一（另外两位分别是 Jun Rao 和 Neha Narkhede）。上面提到的 Google 大神 Tyler Akidau 也自称是 Jay 的拥趸。

Kafka 设计之初就旨在提供 3 个方面的功能特性。

- 为生产者和消费者提供一套简单 API。
- 降低网络传输和磁盘存储开销。
- 具有高伸缩性架构。

现在来看这 3 点 Kafka 都做到了，后续章节中我们会对这 3 点进行详细展开并讨论。

随着 Kafka 的不断完善，Kafka 日益成为一个通用的数据管道，同时兼具高性能和高伸缩性。在 LinkedIn，Kafka 既用于在线系统，也用于离线系统；既从上游系统接收数据，也会给下游系统输送数据；既提供消息的流转服务，也用于数据的持久化存储。因此，Kafka 逐渐发展成 LinkedIn 内部的基础设施，承接了大量的上下游子系统。

2.1.3 Kafka 开源

为了让 Kafka 惠及更多的人，也为了更好地理解用户的使用需求，LinkedIn 公司在 2010 年年底正式将 Kafka 开源，并将源码贡献给了 Apache 软件基金会。这款功能强大的消息引擎正式走入开源社区。

2011 年 7 月，Kafka 正式进入 Apache 进行孵化，并于 2012 年 10 月顺利毕业。自此之后，Kafka 日益发展成为一个非常活跃的社区。截至笔者写稿时，Kafka 已经成长为一个 Star 数 7400+、近 400 名代码贡献者的大社区了，如图 2.1 所示。

今天，Kafka 已经被越来越多的大公司应用到它们自己的数据管道系统中。同时它还在不断演进着，并支持越来越多的功能供全球用户使用，使之日益成为当下大数据时代数据管道技术的首选。

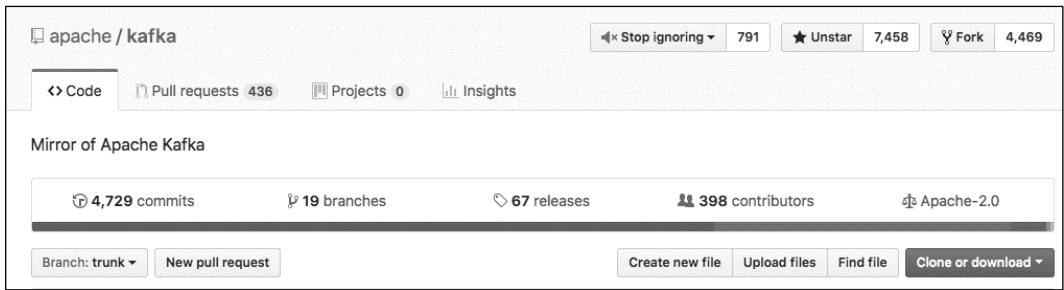


图 2.1 Kafka GitHub 首页

2.2 Kafka 版本变迁

Apache Kafka 是非常活跃的社区，而它的版本也在不断地演进着。截至笔者写稿时，Kafka 在 GitHub 上的 Star 数已达 7400+，迭代版本达 67 个之多。针对目前这么多版本，用户应该如何区分并有针对性地选取适合的版本呢？本节将给出答案。

2.2.1 Kafka 的版本演进

自 2011 年完成孵化到笔者写稿时，Kafka 经历了几次大的版本变迁。每次发布大的版本时都会加入新的功能，表 2.1 总结了 Kafka 几次大版本变迁的功能演变。

表 2.1 Kafka 不同版本功能表

版 本	功 能 变 化	说 明
0.7	提供了最基本的消息引擎服务	很多国内早期用户接触 Kafka 都是从这个版本开始的
0.8	增加了集群间的备份机制	该版本正式加入了备份机制，使得 Kafka 成为完备的分布式消息引擎解决方案
0.8.2.x	使用 Java 重写了 producer	该版本引入了 Java 版本的 producer，以替代原 Scala 版本的 producer
0.9.0.x	①增加了 Kafka 安全性设置； ②使用 Java 重写了 consumer； ③增加了 Kafka Connect 组件	该版本正式加入 Kafka Security，增加了使用 Kafka 集群的安全度；另外该版本还引入了 Java 版本的 consumer 以替代原 Scala 版本的 consumer；该版本引入了 Kafka Connect 组件，该组件实现了高扩展性、高可靠性的数据抽取服务
0.10.0.x	增加了 Kafka Streams 组件	自 0.10.0 版本开始，Kafka 正式转型成为分布式流式处理平台
0.11.0.x	增加了对事务的支持以及实现了精确一次处理语义	自 0.11.0.0 版本开始，Kafka 正式支持事务以及精确一次处理语义（Exactly-Once-Semantic）
1.0.0	优化了 Kafka Streams API 以及各种监控指标的完善	自 1.0.0 版本开始，Kafka 正式进入到 1.0 稳定版本

当然，表 2.1 并没有详尽地给出每次版本升级时所有的功能变更，只罗列了那些最重要的功能升级。截至笔者撰写本章时，Kafka 社区推出的最新版本是 1.0.0，进一步完善了各种 API 以及系统功能的稳定性。

2.2.2 Kafka 的版本格式

在 1.0.0 版本之前，Kafka 版本号的格式是 0.<major>.<minor>.<patch>，其中 major 表示主版本号，有重大的功能变化时才会变更；minor 表示次版本号，新增一般功能时变更；patch 表示对次版本的修订次数或补丁包数。以 0.10.2.1 版本为例，我们说主版本号是 10 或 0.10，次版本号是 2，该次版本的 patch 数是 1。可以看出，这个版本格式和 Linux 内核版本格式非常类似。了解了这一点，我们就可以知道很多用户宣称“他们使用的 Kafka 版本是 2.10 或 2.11”这一说法是错误的。实际上，这里的 2.10 和 2.11 是 Scala 语言环境的版本——Kafka 最开始是使用 Scala 语言编写的，就像普通的 Java 程序可以选择不同的 Java 版本编译一样，Kafka 也有 3 个对应的编译版本，分别是：

- kafka_2.10-<kafka 版本>
- kafka_2.11-<kafka 版本>
- kafka_2.12-<kafka 版本>

前面的 2.** 是 Scala 的版本号，而非 Kafka 的版本号。目前广泛使用的 Kafka 版本应该是 0.8.2.x、0.9.x、0.10.x 和 0.11.x。注意，Kafka 从 0.11.0.0 版本开始不再支持 Scala 2.10，故用户在下载 0.11.0.x 版本的 Kafka 时不会再看到 kafka_2.10-0.11.0.0 字样的下载包。

自 1.0.0 版本开始，Kafka 版本号正式升级为 3 位，格式是 <major>.<minor>.<patch>，它们的含义保持不变，只不过是去掉了开头的数字 0。毕竟 Kafka 已经演变到 1.0 时代，0 已经不再适合用来表征版本了。

2.2.3 新版本功能简介

在 2.2.1 节中，我们提到了在 Kafka 版本的不断演进过程中，社区分别推出了新版本的生产者（producer）和消费者（consumer）。下面就简单介绍一下这个新版本的客户端。在 Kafka 世界中，通常把 producer 和 consumer 通称为客户端（即 clients），这是与服务器端（即 broker）相对应的。

新版本 producer

在 Kafka 0.9.0.0 版本中，社区正式使用 Java 版本的 producer 替换了原 Scala 版本的 producer。新版本的 producer 的主要入口类是 org.apache.kafka.clients.producer.KafkaProducer，

而非原来的 `kafka.producer.Producer`。我们会在 2.2.4 节中讨论 `kafka.producer.Producer`。

新版本 `producer` 重写了之前服务器端代码提供的很多数据结构，摆脱了对服务器端代码库的依赖，同时新版本的 `producer` 也不再依赖于 `ZooKeeper`，甚至不需要和 `ZooKeeper` 集群进行直接交互，降低了系统的维护成本，也简化了部署 `producer` 应用的开销成本。一段典型的新版本 `producer` 代码如下：

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("acks", "all");
props.put("retries", 0);
props.put("batch.size", 16384);
props.put("linger.ms", 1);
props.put("buffer.memory", 33554432);
props.put("key.serializer", "org.apache.kafka.common.serialization.
StringSerializer");
props.put("value.serializer", "org.apache.kafka.common.serialization.
StringSerializer");

Producer<String, String> producer = new KafkaProducer<>(props);
for(int i = 0; i < 100; i++)
    producer.send(new ProducerRecord<String, String>("my-topic", Integer.
toString(i), Integer.toString(i)));

producer.close();
```

上面的代码中比较关键的是 `KafkaProducer.send` 方法，它是实现发送逻辑的主要入口方法。新版本 `producer` 整体工作流程图如图 2.2 所示。

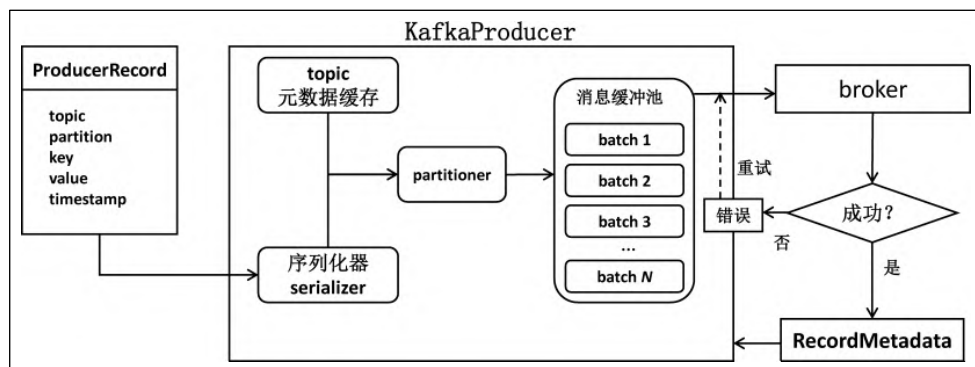


图 2.2 新版本 `producer` 整体工作流程图

我们会在 Kafka 内部原理相关章节中详细介绍 `producer` 的内部架构和工作原理，现在只需

要知道新版本的 `producer` 大致就是将用户待发送的消息封装成一个 `ProducerRecord` 对象，然后使用 `KafkaProducer.send` 方法进行发送。实际上，`KafkaProducer` 拿到消息后对其进行序列化，然后结合本地缓存的元数据信息确立目标分区，最后写入内存缓冲区。同时，`KafkaProducer` 中还有一个专门的 `Sender I/O` 线程负责将缓冲区中的消息分批次发送给 `Kafka broker`。

比起旧版本的 `producer`，新版本在设计理念上有以下几个特点（或者说是优势）。

- 发送过程被划分到两个不同的线程：用户主线程和 `Sender I/O` 线程，逻辑更容易把控。
- 完全是异步发送消息，并提供回调机制（`callback`）用于判断发送成功与否。
- 分批机制（`batching`），每个批次中包括多个发送请求，提升整体吞吐量。
- 更加合理的分区策略：对于没有指定 `key` 的消息而言，旧版本 `producer` 分区策略是默认在一段时间内将消息发送到固定分区，这容易造成数据倾斜（`skewed`）；新版本采用轮询方式，消息发送将更加均匀化。
- 底层统一使用基于 `Java Selector` 的网络客户端，结合 `Java` 的 `Future` 实现更加健壮和优雅的生命周期管理。

当然，新版本 `producer` 的设计优势还有很多，比如监控指标更加完善等。以上 5 点只罗列出了最重要的设计特性。

新版本 `producer` 的 API 设计得也足够简单易用，只需要记住几个常用的方法就可以了，如图 2.3 所示。

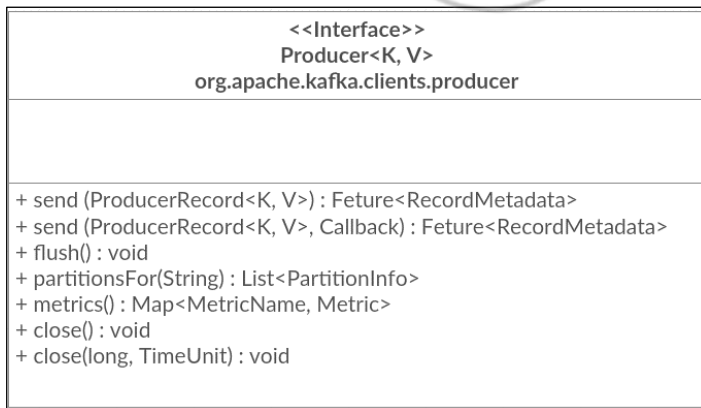


图 2.3 新版本 `producer` 的 API

图 2.3 中比较关键的方法如下。

- `send`: 实现消息发送的主逻辑方法。
- `close`: 关闭 `producer`。后面章节中会讲到正确关闭 `producer` 对于程序正确性来讲至关

重要。

- **metrics**: 获取 **producer** 的实时监控指标数据, 比如发送消息的速率等。

鉴于新版本在设计 and 实际使用上的诸多优势, 社区已于 0.9.0.0 版本正式废弃了旧版本 **producer**, 并且推荐所有依然使用旧版本 **producer** 的用户尽早升级到新版本。

新版本 consumer

Kafka 0.9.0.0 版本不仅废弃了旧版本 **producer**, 还提供了新版本的 **consumer**。同样地, 新版本 **consumer** 也是使用 Java 语言编写的, 也不再需要依赖 **ZooKeeper** 的帮助。新版本 **consumer** 的入口类是 `org.apache.kafka.clients.consumer.KafkaConsumer`。由此也可以看出, 新版本客户端的代码包都是 `org.apache.kafka.clients`, 这一点需要特别注意, 因为它是区分新旧客户端的一个重要特征。

在旧版本 **consumer** 中, 消费位移 (offset) 的保存与管理都是依托于 **ZooKeeper** 来完成的。当数据量很大且消费很频繁时, **ZooKeeper** 的读/写性能往往容易成为系统瓶颈。这是旧版本 **consumer** 为人诟病的缺陷之一。而在新版本 **consumer** 中, 位移的管理与保存不再依靠 **ZooKeeper** 了, 自然这个瓶颈就消失了。

一段典型的 **consumer** 代码如下:

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("group.id", "test");
props.put("enable.auto.commit", "true");
props.put("auto.commit.interval.ms", "1000");
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<>(props);
consumer.subscribe(Arrays.asList("foo", "bar"));
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records)
        System.out.printf("offset = %d, key = %s, value = %s\n",
record.offset(), record.key(), record.value());
}
```

同理, 上面代码中比较关键的是 `KafkaConsumer.poll` 方法。它是实现消息消费的主逻辑入口方法。新版本 **consumer** 在设计时摒弃了旧版本多线程消费不同分区的思想, 采用了类似于 Linux `epoll` 的轮询机制, 使得 **consumer** 只使用一个线程就可以管理连向不同 **broker** 的多个

Socket，既减少了线程间的开销成本，同时也简化了系统的设计。

比起旧版本 consumer，新版本在设计上的突出优势如下。

- 单线程设计——单个 consumer 线程可以管理多个分区的消费 Socket 连接，极大地简化了实现。虽然 0.10.1.0 版本额外引入了一个后台心跳线程（background heartbeat thread），不过双线程的设计依然比旧版本 consumer 鱼龙混杂的多线程设计要简单得多。
- 位移提交与保存交由 Kafka 来处理——位移不再保存在 ZooKeeper 中，而是单独保存在 Kafka 的一个内部 topic 中，这种设计既避免了 ZooKeeper 频繁读/写的性能瓶颈，同时也依托 Kafka 的备份机制天然地实现了位移的高可用管理。
- 消费者组的集中式管理——上面提到了 ZooKeeper 要管理位移，其实它还负责管理整个消费者组（consumer group）的成员。这进一步加重了对于 ZooKeeper 的依赖。新版本 consumer 改进了这种设计，实现了一个集中式协调者（coordinator）的角色。所有组成员的管理都交由该 coordinator 负责，因此对于 group 的管理将更加可控。

比起旧版本而言，新版本在 API 设计上提供了更加丰富的功能，具体 API 方法如图 2.4 所示。

<<Interface>> Consumer<K, V> org.apache.kafka.clients.consumer
<pre> + poll(long) : ConsumerRecords<K, V> + subscribe(Collection<String>, ConsumerRebalanceListener) : void + commitSync(Map<TopicPartition, OffsetAndMetadata>) : void + commitAsync(Map<TopicPartition, OffsetAndMetadata>, OffsetCommitCallback) : void + seek(TopicPartition, long) : void + assignment() : Set<TopicPartition> + subscription() : Set<String> + assign(Collection<TopicPartition>) : void + unsubscribe() : void + seekToBeginning(Collection<TopicPartition>) : void + seekToEnd(Collection<TopicPartition>) void + position(TopicPartition) : long + committed(TopicPartition) : OffsetAndMetadata + metrics() : Map<MetricName, Metric> + partitionsFor(String) : List<PartitionInfo> + listTopics() : Map<String, List<PartitionInfo>> + paused() : Set<TopicPartition> + pause(Collection<TopicPartition>) : void + resume(Collection<TopicPartition>) : void + offsetsForTimes(Map<TopicPartition, Long>) : Map<TopicPartition, OffsetAndTimestamp> + beginningOffsets(Collection<TopicPartition>) : Map<TopicPartition, Long> + endOffsets(Collection<TopicPartition>) : Map<TopicPartition, Long> + close(long, TimeUnit) : void + wakeup() : void </pre>

图 2.4 新版本 consumer 的 API

显然，图 2.4 中 API 提供的方法有很多，其中比较关键的方法如下。

- poll: 最重要的方法。它是实现读取消息的核心方法。
- subscribe: 订阅方法, 指定 consumer 要消费哪些 topic 的哪些分区。
- commitSync/commitAsync: 手动提交位移方法。新版本 consumer 允许用户手动提交位移, 并提供了同步/异步两种方式。
- seek/seekToBeginning/seekToEnd: 设置位移方法。除了提交位移, consumer 还可以直接消费特定位移处的消息。

和 producer 不同的是, 目前新旧 consumer 共存于最新版本的 Kafka 中。虽然社区曾计划投票决定在 0.11.0.0 这个大版本上正式放弃对于旧版本 consumer 的支持, 但是目前使用旧版本 consumer 的用户依然不在少数, 故即使在最新的 1.0.0 版本中旧版本依然没有被移除, 可以预见这种共存的局面还将维持一段时间。

2.2.4 旧版本功能简介

前面谈到的都是新版本客户端, 包括了 producer 和 consumer。新版本客户端必然是社区以后极力推荐使用的, 但不可否认的是, 旧版本依然有着广泛的用户基础, 特别是对于那些早期使用 Kafka 的公司来说, 他们大多数使用的是 Kafka 0.8.x 这个版本。就拿其中最广泛应用的 0.8.2.2 这个版本而言, 这个版本的 Kafka 刚刚推出 Java 版 producer, 而 Java consumer 甚至还没有开发。所以, 我们还是有必要简要了解一下旧版本客户端, 毕竟很多核心设计思想都是一脉相承的。

旧版本 producer

说起旧版本 producer, 真可谓是“历史悠久”了, 笔者当年为了研究它着实花了一番功夫, 想到其如今已经尘封入土不禁令人唏嘘。

这里频频提到的旧版本就是指由 Scala 语言编写的 producer, 在比较新的 Kafka 官网介绍中用户已经找不到对于它的介绍了, 但是对于依然使用 0.8.x 版本 Kafka 的用户而言, Scala producer 依然可能活跃在他们的线上系统中。

Scala producer 的入口类是 kafka.producer.Producer, 一段典型的代码如下:

```
Properties props = new Properties();
props.put("metadata.broker.list", "localhost:9092, localhost:9093,
localhost:9094");
props.put("serializer.class", "kafka.serializer.StringEncoder");
props.put("request.required.acks", "1");
ProducerConfig config = new ProducerConfig(props);
Producer<String, String> producer = new Producer<String, String> (config);
KeyedMessage<String, String> msg = new KeyedMessage<String, String>("my-
```

```
topic", "hello, world.");
Producer.send(msg);
```

很明显，上面代码中的主要逻辑是由 `Producer.send` 方法实现的。该方法默认是同步机制的，即每条消息要等待服务器端发送响应给客户端，明确告知消息发送结果之后，才能开始下一条消息的发送，因此旧版本 `producer` 的吞吐量性能是很差的。当然它也提供了一个参数用于实现异步的消息发送逻辑，但是凡事有利就有弊，旧版本 `producer` 异步发送会有丢失消息的可能性，对于那些对数据有较强持久化要求的用户来说，异步并不是一个可选项。

图 2.5 是旧版本 `producer` 的工作流程图。

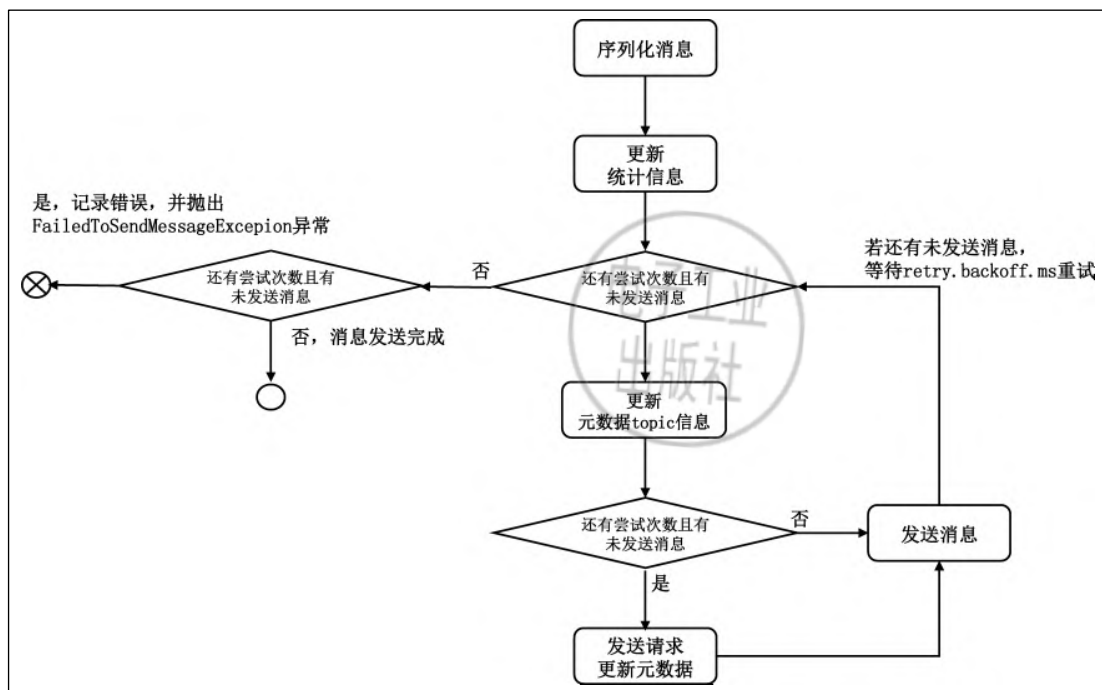


图 2.5 旧版本 `producer` 的工作流程图

如果读者仔细对比图 2.2 和图 2.5，就可以发现新旧版本 `producer` 的工作流程原理有很大的相似性，只是新版本 `producer` 在各个方面都要优于旧版本。因此社区才会在 Kafka 0.9.0.0 版本中正式将旧版本“下架”。

对于 API 而言，旧版本的 `producer` 提供的功能也非常有限，如图 2.6 所示。

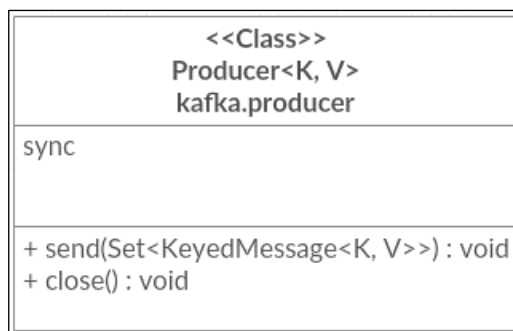


图 2.6 旧版本 producer API

由图 2.6 可知，旧版本只提供了 send 和 close 两个方法，另外还提供了一个 sync 参数用于控制该 producer 是同步发送消息还是异步发送。因此整套 API 的设计实际上是非常“简陋”的。

旧版本 consumer

不同于 producer，旧版本 consumer，即 Scala consumer，其实并没有那么“旧”，也没有那么弱。如前所述，依然有很多用户在生产环境中使用着旧版本 consumer。和新旧版本 producer 之间有巨大的性能差异不同，新旧版本 consumer 的性能差异似乎也没有那么大。换句话说，旧版本 consumer 没有那么“不堪”。这也是社区迟迟没有“下架”它的原因之一。

说起旧版本 consumer，就不能不提高阶消费者（high-level consumer）和低阶消费者（low-level consumer）。没错，它们是专属于旧版本而言的。切记新版本是没有 high-level 和 low-level 之分的！high-level consumer 其实就是指消费者组，而 low-level consumer 是指单个 consumer，即 standalone consumer。单个 consumer 是没有什么消费者组的概念的，每个 consumer 都单独进行自己的工作，与其他 consumer 不产生任何关联；而消费者组就是大家作为一个团队一起工作，彼此之间会“相互照应”。

我们先说 low-level consumer。low-level consumer 的底层实现就是 SimpleConsumer 类。一旦应用此类，Kafka 会认为用户有自行管理消费者的需求，从而不会为用户提供任何组管理方面的功能（包括负载均衡和故障转移等），而用户需要自己解决这方面的问题。因此使用 SimpleConsumer 可以说是既灵活又麻烦。鉴于这些特点，很多需要灵活定制实现的第三方框架往往会采用这种 low-level consumer，比如 Apache Storm 的 Kafka 插件 storm-kafka 就使用了 SimpleConsumer 来实现 KafkaSpout。

low-level consumer 的 API 设计得非常简单，具体类图如图 2.7 所示。

<<Class>> SimpleConsumer kafka.consumer
+ fetch(FetchRequest) : FetchResponse + send(GroupCoordinatorRequest) : GroupCoordinatorResponse + send(TopicMetadataRequest) : TopicMetadataResponse + getOffsetsBefore(OffsetRequest) : OffsetResponse + commitOffsets(OffsetCommitRequest) : OffsetCommitResponse + fetchOffsets(OffsetFetchRequest) : OffsetFetchResponse + earliestOrLatestOffset(TopicPartition, long, int) : long + close() : void

图 2.7 旧版本 low-level consumer API

如果我们对比新版本 API 就会发现，很多方法都是类似的，都有获取消息、提交位移等功能。当然在底层实现上两者差异明显，在后续章节中我们会详细展开讨论。

细心的读者可能会发现在 SimpleConsumer 的 API 设计中还有 send 方法，难道 consumer 还需要发送消息吗？其实，这里的 send 不是指发送消息，而是指发送具体的请求。事实上，尽管旧版本 consumer 已经不推荐用户使用了，但 Kafka 服务器底层依然有一部分代码在使用 SimpleConsumer 负责向其他 broker 发送特定类型的请求，即使用这里的 send 方法进行发送，所以读者不要把它和 producer 的 send 方法搞混淆了。

如果说 low-level consumer 既麻烦又灵活，那么 high-level consumer 便是既省事又死板。如图 2.8 所示，high-level consumer 主要的方法是 createMessageStreams——该方法负责创建一个或多个 KafkaStream，用于真正的消息消费。如果是在多台机器上，用户只需要简单地启动多个配置有相同组 ID (group.id) 的 consumer 进程；若是在同一台机器上，createMessageStreams 方法也允许用户直接指定线程数来创建多 consumer 实例。不管是哪种方法，这些 consumer 实例都会自动组成一个消费者组来共同承担消费任务，假如任意时刻有 consumer 进程或实例宕机，该消费者组都会帮用户自动处理，根本不需要人工干预。如此看来，使用 high-level consumer 是很省事的，但为什么说它是死板的呢？high-level consumer 不似 SimpleConsumer 那样灵活，可以从分区的任意位置开始消费。它只能从上次保存的位移处开始顺序读取消息，使用起来无法实现高度定制化的消费策略，故所以说它是死板的。

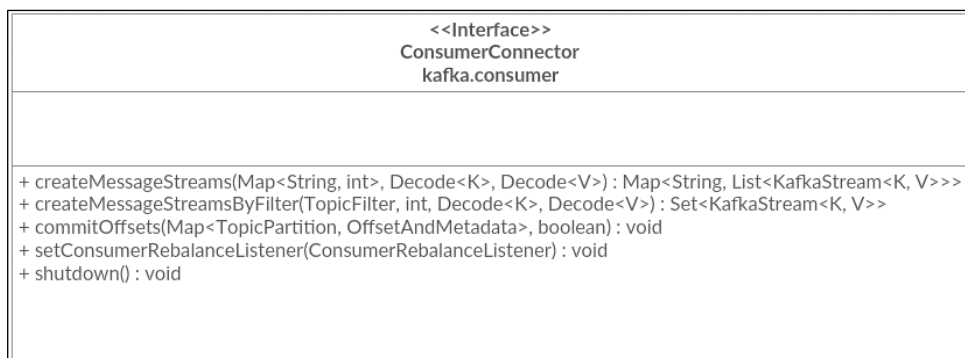


图 2.8 旧版本 high-level consumer API

2.3 如何选择 Kafka 版本

既然 Kafka 目前存在多个大版本以及多套客户端 API，广大用户在使用过程中应该如何选择合适的 Kafka 呢？

2.3.1 根据功能场景

如果用户要在生产环境中应用流式处理组件 Kafka Streams，那么就必须使用 Kafka 0.10.0.0（含）之后的版本。若是从零开始搭建或处于技术选型阶段，笔者推荐使用最新版本的 Kafka，即 1.0.0，毕竟这个版本中修复了很多关于 Streams 的 bug，并且完善了 Kafka Streams 的各种 API 接口。

如果要在生产环境中启用 Kafka Security，那么至少要使用 0.9.0.0 及以后的 Kafka 版本，但最好能使用 0.10.0.1 之后的版本。

当然，如果将 Kafka 用于传统的消息引擎服务，甚至是分布式存储之用，那么对于这种需求而言在版本的选择上并没有太多的限制，只需要 0.8.x 以后的版本就行，因为到 0.8 版本才加入了集群间的备份机制。没有备份的消息引擎系统不能算是一个完备的解决方案。

2.3.2 根据客户端使用场景

所谓客户端使用场景无非两种：自行研发客户端和第三方框架提供客户端。如果是自行研发客户端，笔者推荐使用新版本的客户端，至少要是 0.10.1.0 版本的 Kafka，因为自这个版本开始新版本 consumer 才算比较稳定。如果你之前的生产环境部署的是该版本之前的 Kafka，那么还是建议使用旧版本 consumer 为妙。

如果是第三方框架直接提供的客户端，比如 Storm 或 Spark，那么一定要注意这些框架官网上的说明。我们以 Storm 为例：Storm 目前提供了两套 Kafka 实际数据传输的插件 storm-kafka 和 storm-kafka-client，如图 2.9 所示。

storm-kafka-client	STORM-2306 - Messaging subsystem redesign. New Backpressure model.	8 days ago
storm-kafka-monitor	STORM-2706: Upgrade to Curator 4.0.0	4 months ago
storm-kafka	STORM-2306 - Messaging subsystem redesign. New Backpressure model.	8 days ago

图 2.9 Storm kafka 插件

其中，storm-kafka 使用了旧版本 consumer 进行开发，storm-kafka-client 则使用了新版本 consumer。因此你需要根据使用的具体插件来确定 Kafka 的版本。比如你的生产环境中部署了 Storm 且使用了 storm-kafka-client 中的 KafkaSpout 实现，那么你的 Kafka 版本就必须是 0.9.0.0 或更高的版本。

总之，笔者建议使用比较新的 Kafka 版本，例如 0.10.1.0 或更高版本。另外根据客户端的稳定程度，笔者总结了一份详细的主流版本-客户端版本推荐关系表，如表 2.2 所示。

表 2.2 Kafka 版本-客户端版本推荐表

版 本	推荐 producer	推荐 consumer	原 因
0.8.2.2	旧版	旧版	新 producer 尚不稳定
0.9.0.x	新版	旧版	新 producer 已经稳定，可以用于生产环境
0.10.0.x	新版	旧版	新 consumer 尚不稳定，不推荐使用
0.10.1.0	新版	新版	从这一版本开始新 consumer 日趋稳定，可以尝试部署线上环境
0.10.2.x 及以后	新版	新版	推荐使用新版客户端到线上环境

2.4 Kafka 与 Confluent

随着 Kafka 代码被贡献给 Apache 软件基金会，成功孵化成 Apache 顶级项目并顺利毕业，世界上有越来越多的公司和个人开始接触到 Kafka，并不断地向 LinkedIn 的 Kafka 创始团队寻求技术上的支持与帮助，这让 3 位 Kafka 创始人感到这是一个巨大的商业机会。

于是，在 2014 年 Jay Kreps、Jun Rao 和 Neha Narkhede 离开 LinkedIn 公司创办了 Confluent.io。这家公司从事商业化 Kafka 工具开发以及提供实时流式处理方面的产品。值得一提的是，Confluent 于 2017 年 3 月成功融资 C 轮 5000 万美元，可见资本市场对于基于 Kafka 的实时流式处理很看好。

该公司基于 Apache Kafka 提供了 Confluent Platform。如果我们把 Apache Kafka 比作 Linux 的内核，那么 Confluent 就类似于 Linux 的某个发行版（比如 CentOS 或 Ubuntu）。Confluent 提供了一些 Apache Kafka 没有的组件和功能，比如完善的跨数据中心数据备份以及集群监控解决方案等。

另外 Confluent 还分为开源版本和企业版本。在企业版本中它提供了对于底层 Kafka 集群完整的可视化监控解决方案，以及一些辅助系统帮助管理集群。Confluent 的开源版本与 Apache 社区的 Kafka 并无太大的区别，用户甚至完全可以使用 Confluent Open Source 来替代 Apache Kafka，它的下载地址是 <https://www.confluent.io/download/>。

2.5 本章小结

本章回顾了 Apache Kafka 的发展历史，包括了研发 Kafka 的现实背景，LinkedIn 公司捐献给社区的开源进程，以及 Kafka 版本的变迁历程。通过对这段历史的回顾，读者可以清楚地了解 Kafka 的“前世今生”。

之后本章结合不同的 Kafka 版本简要概述了新旧两个版本客户端的设计异同，并根据这种异同讨论了如何在生产环境中确定 Kafka 版本。

最后我们讨论了 Apache Kafka 与 Confluent，完整地解释了两者的关系以及具体选择上的依据。

第 3 章

Kafka 线上环境部署

学习本章，你将了解到以下内容。

- Kafka 多节点环境安装部署。
- Kafka 参数配置。
- Kafka 环境验证。



3.1 集群环境规划

第 1 章曾带领读者搭建了一个最简单的 Kafka 环境——一个只有单 broker 节点的集群（也被称为伪集群）。显然，对于正式的生产环境或线上环境而言，这是远远不够的。典型的生产环境至少需要部署多个节点共同组成一个分布式集群整体为我们提供服务。本章将会详细讨论生产环境中集群的安装、配置与验证。不过在此之前，我们还需要解决 3 方面的问题。它们分别是操作系统的选型、硬件规划和容量规划。

3.1.1 操作系统的选型

谈到操作系统，很多人可能会问：Kafka 不是 JVM 系的大数据框架吗？而 Java 又是跨平台的语言，那么使用什么操作系统有什么区别吗？当然有区别！

众所周知，Kafka 的服务器端代码是由 Scala 语言编写的，而新版本客户端代码是由 Java 语言编写的。和 Java 一样，Scala 编译器会把源程序 .scala 文件编译成 .class 文件，因此 Scala 也是 JVM 系的语言。这样来看的话，貌似只要是支持 Java 程序部署的平台都应该能够部署 Kafka。但既然本章讨论的是生产环境的部署，那么我们就需要仔细地选型各种操作系统并梳

理出它们对于 Kafka 的相适性。

目前部署 Kafka 最多的 3 类操作系统分别是 Linux、OS X 和 Windows，其中部署在 Linux 上的最多，而 Linux 也是推荐的操作系统。为什么呢？且不说当前的现状的确是 Linux 服务器数量最多，单论它与 Kafka 本身的相适性，Linux 也要比 Windows 等其他操作系统更加适合部署 Kafka。这里笔者罗列出自己能想到的两个主要原因：I/O 模型的使用和数据网络传输效率。

谈到 I/O 模型，就不能不说当前主流且耳熟能详的 5 种模型：阻塞 I/O、非阻塞 I/O、I/O 多路复用、信号驱动 I/O 和异步 I/O。每一种 I/O 模型都有典型的使用场景，比如 Socket 的阻塞模式和非阻塞模式就对应于前两种模型，而 Linux 中的 select 函数就属于 I/O 多路复用模型，至于第 5 种模型其实很少有 UNIX 和类 UNIX 系统支持，Windows 的 IOCP（I/O Completion Port，简称 IOCP）属于此模型。至于大名鼎鼎的 Linux epoll 模型，则可以看作兼具第 3 种和第 4 种模型的特性。

由于篇幅有限，我们不会针对每种 I/O 模型进行详细的展开，但通常情况下我们会认为 epoll 比 select 模型高级。毕竟 epoll 取消了轮询机制，取而代之的是回调机制（callback）。这样当底层连接 Socket 数较多时，可以避免很多无意义的 CPU 时间浪费。另外，Windows 的 IOCP 模型可以说是真正的异步 I/O 模型，但由于其母系统的局限性，IOCP 并没有广泛应用。

说了这么多，这些和 Kafka 又有什么关系呢？关键就在于 clients 底层网络库的设计。Kafka 新版本 clients 在设计底层网络库时采用了 Java 的 Selector 机制，而后者在 Linux 上的实现机制就是 epoll；但是在 Windows 平台上，Java NIO 的 Selector 底层是使用 select 模型而非 IOCP 实现的，只有 Java NIO2 才是使用 IOCP 实现的。因此在这一点上，在 Linux 上部署 Kafka 要比在 Windows 上部署能够得到更高效的 I/O 处理性能。

对于第二个方面，即数据网络传输效率而言，Linux 也更有优势。具体来说，Kafka 这种应用必然需要大量地通过网络与磁盘进行数据传输，而大部分这样的操作都是通过 Java 的 FileChannel.transferTo 方法实现的。在 Linux 平台上该方法底层会调用 sendfile 系统调用，即采用了 Linux 提供的零拷贝（Zero Copy）技术。

如前面章节所言，这种零拷贝技术可以有效地改善数据传输的性能。在内核驱动程序处理 I/O 数据的时候，它可以减少甚至完全规避不必要的 CPU 数据拷贝操作，避免数据在操作系统内核地址空间和用户应用程序地址空间的缓冲区区间进行重复拷贝，因而可以获得很好的性能。Linux 提供的诸如 mmap、sendfile 以及 splice 等系统调用即实现了这样的技术。

然而对于 Windows 平台而言，虽然它也提供了 TransmitFile 函数来支持零拷贝技术，但是直到 Java 8u60 版本 Windows 平台才正式让 FileChannel 的 transferTo 方法调用该函数。具体详见这个 JDK bug: http://bugs.java.com/view_bug.do?bug_id=8064407。

鉴于很多公司目前的生产环境中还没有正式上线 Java 8，因而在 Windows 平台上部署 Kafka 将很有可能无法享受到零拷贝技术带来的高效数据传输。

综合以上两点差异以及目前主流服务器通常在 Linux 上部署的事实，笔者强烈推荐 Kafka 的生产环境集群首选 Linux 操作系统。当然至于是 Linux 的哪个发行版，笔者倒没有特别的偏好，只要是读者熟悉的即可。

3.1.2 磁盘规划

前面主要讨论了该如何选择合适的操作系统平台来安装部署 Kafka。从现在开始，我们将分别从磁盘、内存、带宽和 CPU 等几个方面探讨部署 Kafka 集群所必要的关键规划因素。首先从磁盘开始说起。

如果问哪个因素对 Kafka 性能最重要？磁盘无疑是排名靠前的答案。众所周知，Kafka 是大量使用磁盘的。Kafka 的每条消息都必须被持久化到底层的存储中，并且只有被规定数量的 broker 成功接收后才能通知 clients 消息发送成功，因此消息越是被更快地保存在磁盘上，处理 clients 请求的延时越低，表现出来的用户体验也就越好。

在确定磁盘时，一个常见的问题就是选择普通的机械硬盘（HDD）还是固态硬盘（SSD）。机械硬盘成本低且容量大，而 SSD 通常有着极低的寻道时间（seek time）和存取时间（access time），性能上的优势很大，但同时也有着非常高的成本。因此在规划 Kafka 线上环境时，读者就需要根据公司自身的实际条件进行有针对性的选型。但以笔者使用 Kafka 的经验来看，Kafka 使用磁盘的方式在很大程度上抵消了 SSD 提供的那些突出优势。众所周知，SSD 强就强在它不是机械装置，而全部由电子芯片及电路板组成，因而可以极大地避免传统机械硬盘缓慢的磁头寻道时间。一般机械硬盘的寻道时间都是毫秒级的。若有大量的随机 I/O 操作，则整体的磁盘延时将是非常可观的，但 SSD 则不受这样的拖累。可是这点差异对于 Kafka 来说又显得不是那么重要。为什么？因为 Kafka 是顺序写磁盘的，而磁盘顺序 I/O 的性能，即使机械硬盘也是不弱的——顺序 I/O 不需要频繁地移动磁头，因而节省了耗时的寻道时间。所以从磁盘的使用这个方面来看，笔者并不认为两者有着巨大的性能差异。关于 Kafka 底层的持久化实现，我们会在第 6 章中详细讨论。因此对于预算有限且追求高性价比的公司而言，机械硬盘完全可以胜任 Kafka 存储的任务。

关于磁盘的选择，另一个比较热门的争论就在于，JBOD 与磁盘阵列（下称 RAID）之争。这里的 JBOD 全称是 Just Bunch Of Disks，翻译过来就是一堆普通磁盘的意思。在部署线上 Kafka 环境时，应当如何抉择呢？是使用一堆普通商用磁盘进行安装还是搭建专属的 RAID 呢？答案依然是具体问题具体分析。

首先分析一下 RAID 与 Kafka 的相适性。常见的 RAID 是 RAID 10，也被称为 RAID 1+0，它结合了磁盘镜像和磁盘条带化两种技术共同保护数据，既实现了不错的性能也提供了很高的可靠性。RAID 10 集合了 RAID 0 和 RAID 1 的优点，但在空间上使用了磁盘镜像，因此整体的磁盘使用率只有 50%，换句话说就是将一半的磁盘容量都用作提供冗余。自 Kafka 0.8.x 版本开始，用户就可以使用 RAID 作为存储来为 Kafka 提供服务了。事实上，根据公开的资料显示，LinkedIn 公司的 Kafka 集群就是使用 RAID 10 作为底层存储的。除了默认提供的数据冗余之外，RAID 10 还可以将数据自动地负载分布到多个磁盘上。

由此可见，RAID 作为 Kafka 的底层存储其实主要的优势有两个。

- 提供冗余的数据存储空间。
- 天然提供负载均衡。

以上两个优势对于任何系统而言都是非常好的特性。不过对于 Kafka 而言，就像在第 1 章中介绍的那样，Kafka 在框架层面其实已经提供了这两个特性：通过副本机制提供冗余和高可靠性，以及通过分散到各个节点的领导者选举机制来实现负载均衡，所以从这方面来看，RAID 的优势就显得不是那么明显了。当然笔者绝没有全盘否定 RAID 的意思，实际上，依然有很多公司和组织使用或者打算在 RAID 之上构建 Kafka 集群。不过既然是资源规划和硬件选型，我们不妨看下 LinkedIn 公司是怎么做的。

之前提到过，LinkedIn 公司目前的 Kafka 就搭建于 RAID 10 之上。他们在 Kafka 层面设定的副本数是 2，因此根据 RAID 10 的特性，这套集群实际上提供了 4 倍的数据冗余，且只能容忍一台 broker 宕机（因为副本数=2）。若 LinkedIn 公司把副本数提高到 3，那么就提供了 6 倍的数据冗余。这将是一笔很大的成本开销。但是，如果我们假设 LinkedIn 公司使用的是 JBOD 方案。虽然目前 JBOD 有诸多限制，但其低廉的价格和超高的性价比的确是非常大的优势。另外通过一些简单的设置，JBOD 方案可以达到和 RAID 方案一样的数据冗余效果。比如说，如果使用 JBOD 并且设置副本数为 4，那么 Kafka 集群依然提供 4 倍的数据冗余，但是这个方案中整个集群可以容忍最多 3 台 broker 宕机而不丢失数据。对比之前的 RAID 方案，JBOD 方案没有牺牲任何高可靠性或是增加硬件成本，同时还提升了整个集群的高可用性。

事实上，LinkedIn 公司目前正在计划将整个 Kafka 集群从 RAID 10 迁移到 JBOD 上，只不过在整个过程中 JBOD 方案需要解决当前 Kafka 一些固有缺陷，比如：

- 任意磁盘损坏都会导致 broker 宕机——普通磁盘损坏的概率是很大的，因此这个缺陷从某种程度上来说是致命的。不过社区正在改进这个问题，未来版本中只要为 broker 配置的多块磁盘中还有状态良好的磁盘，broker 就不会挂掉。
- JBOD 的管理需要更加细粒度化——目前 Kafka 没有提供脚本或其他工具用于在不同磁

盘间进行分区手动分配，但这是使用 JBOD 方案中必要的功能。

- JBOD 也应该提供类似于负载均衡的功能——目前只是简单地依赖轮询的方式为新副本数据选择磁盘，后续需要提供更加丰富的策略。

结合 JBOD 和 RAID 之间的优劣对比以及 LinkedIn 公司的实际案例，笔者认为：对于一般的公司或组织而言，选择 JBOD 方案的性价比更高。另外推荐用户为每个 broker 都配置多个日志路径，每个路径都独立挂载在不同的磁盘上，这使得多块物理磁盘磁头同时执行物理 I/O 写操作，可以极大地加速 Kafka 消息生产的速度。

最后关于磁盘的一个建议就是，尽量不要使用 NAS（Network Attached Storage）这样的网络存储设备。对比本地存储，人们总是以为 NAS 方案速度更快也更可靠，其实不然。NAS 一个很大的弊端在于，它们通常都运行在低端的硬件上，这就使得它们的性能很差，可能比一台笔记本电脑的硬盘强不了多少，表现为平均延时有很大的不稳定性，而几乎所有高端的 NAS 设备厂商都售卖专有的硬件设备，因此成本的开销也是一个需要考虑的因素。

综合以上所有的考量，笔者给硬盘规划的结论性总结如下。

- 追求性价比的公司可以考虑使用 JBOD。
- 使用机械硬盘完全可以满足 Kafka 集群的使用，SSD 更好。

3.1.3 磁盘容量规划

Kafka 集群到底需要多大的磁盘容量？这又是一个非常经典的规划问题。如前所述，Kafka 的每条消息都保存在实际的物理磁盘中，这些消息默认会被 broker 保存一段时间之后清除。这段时间是可以配置的，因此用户可以根据自身实际业务场景和存储需求来大致计算线上环境所需的磁盘容量。

让我们以一个实际的例子来看下应该如何思考这个问题。假设在你的业务场景中，clients 每天会产生 1 亿条消息，每条消息保存两份并保留一周的时间，平均一条消息的大小是 1KB，那么我们需要为 Kafka 规划多少磁盘空间呢？如果每天 1 亿条消息，那么每天产生的消息会占用 $1 \text{ 亿} \times 2 \times 1\text{KB} / 1000 / 1000 = 200\text{GB}$ 的磁盘空间。我们最好再额外预留 10% 的磁盘空间用于其他数据文件（比如索引文件等）的存储，因此在这种使用场景下每天新发送的消息将占用 210GB 左右的磁盘空间。因为还要保存一周的数据，所以整体的磁盘容量规划是 $210 \times 7 \approx 1.5\text{TB}$ 。当然，这是无压缩的情况。如果在 clients 启用了消息压缩，我们可以预估一个平均的压缩比（比如 0.5），那么整体的磁盘容量就是 0.75TB。

总之对于磁盘容量的规划和以下多个因素有关。

- 新增消息数。

- 消息留存时间。
- 平均消息大小。
- 副本数。
- 是否启用压缩。

3.1.4 内存规划

乍一看似乎关于内存规划的讨论没什么必要，毕竟用户能做的就只是分配一个合适大小的内存，其他也没有可以调整的地方了。其实不然！Kafka 对于内存的使用可称作其设计亮点之一。虽然在前面我们强调了 Kafka 大量依靠文件系统和磁盘来保存消息，但其实它还会对消息进行缓存，而这个消息缓存的地方就是内存，具体来说就是操作系统的页缓存（page cache）。

Kafka 虽然会持久化每条消息，但其实这个工作都是底层的文件系统来完成的，Kafka 仅仅将消息写入 page cache 而已，之后将消息“冲刷”到磁盘的任务完全交由操作系统来完成。另外 consumer 在读取消息时也会首先尝试从该区域中查找，如果直接命中则完全不用执行耗时的物理 I/O 操作，从而提升了 consumer 的整体性能。不论是缓冲已发送消息还是待读取消息，操作系统都要先开辟一块内存区域用于存放接收的 Kafka 消息，因此这块内存区域大小的设置对于 Kafka 的性能就显得尤为关键了。

有些令人惊讶的是，Kafka 对于 Java 堆内存的使用反而不是很多，因为 Kafka 中的消息通常都属于“朝生夕灭”的对象实例，可以很快地垃圾回收（GC）。一般情况下，broker 所需的堆内存都不会超过 6GB。所以对于一台 16GB 内存的机器而言，文件系统 page cache 的大小甚至可以达到 10~14GB！

除以上这些考量之外，用户还需要把 page cache 大小与实际线上环境中设置的日志段大小相比较（关于日志段的描述会在第 6 章中详细展开）。假设单个日志段文件大小设置为 10GB，那么你至少应该给予 page cache 10GB 以上的内存空间。这样，待消费的消息有很大概率会保存在页缓存中，故 consumer 能够直接命中页缓存而无须执行缓慢的磁盘 I/O 读操作。

总之对于内存规划的建议如下。

- 尽量分配更多的内存给操作系统的 page cache。
- 不要为 broker 设置过大的堆内存，最好不超过 6GB。
- page cache 大小至少要大于一个日志段的大小。

3.1.5 CPU 规划

比起磁盘和内存，CPU 于 Kafka 而言并没有那么重要——严格来说，Kafka 不属于计算密

集型（CPU-bound）的系统，因此对于 CPU 需要记住一点就可以了：追求多核而非高时钟频率。简单来说，Kafka 的机器有 16 个 CPU 核这件事情比该机器 CPU 时钟高达 4GHz 更加重要，因为 Kafka 可能无法充分利用这 4GHz 的频率，但几乎肯定会用满 16 个 CPU 核。Kafka broker 通常会创建几十个后台线程，再加上多个垃圾回收线程，多核系统显然是最佳的配置选择。

当然，凡事皆有例外。若 clients 端启用了消息压缩，那么除了要为 clients 机器分配足够的 CPU 资源外，broker 端也有可能需要大量的 CPU 资源——尽管 Kafka 0.10.0.0 改进了在 broker 端的消息处理，免了解压缩消息的负担以节省磁盘占用和网络带宽，但并非所有情况下都可以避免这种解压缩（比如 clients 端和 broker 端配置的消息版本号不匹配）。若出现这种情况，用户就需要为 broker 端的机器也配置充裕的 CPU 资源。

基于以上的判断依据，我们对 CPU 资源规划的建议如下。

- 使用多核系统，CPU 核数最好大于 8。
- 如果使用 Kafka 0.10.0.0 之前的版本或 clients 端与 broker 端消息版本不一致（若无显式配置，这种情况多半由 clients 和 broker 版本不一致造成），则考虑多配置一些资源以防止消息解压缩操作消耗过多 CPU。

3.1.6 带宽规划

对于 Kafka 这种在网络间传输大量数据的分布式数据管道而言，带宽资源至关重要，并且特别容易成为系统的瓶颈，因此一个快速且稳定的网络是 Kafka 集群搭建的前提条件。低延时的网络以及高带宽有助于实现 Kafka 集群的高吞吐量以及用户请求处理低延时。

当前主流的网络环境皆是使用以太网，带宽主要也有两种：1Gb/s 和 10Gb/s，即平时所说的千兆位网络和万兆位网络。无论是哪种带宽，对于大多数的 Kafka 集群来说都足矣了。

前面的 3.1.3 节中给出了一种用于评估磁盘容量的方法。使用该方法我们可以估算出 Kafka 集群所需的磁盘空间。结合每台 broker 自身的磁盘容量，就可以计算出整个集群需要的机器数。除了磁盘的因素，我们还必须根据实际业务网络环境来考量带宽的影响。

举一个实际的例子来说明如何规划带宽资源。假设用户网络环境中的带宽是 1Gb/s，用户的业务目标是每天用 1 小时处理 1TB 的业务消息，那么在这种情况下 Kafka 到底需要多少台机器呢？让我们来计算一下：网络带宽是 1Gb/s，即每秒传输 1Gb 的数据，假设分配的机器为 Kafka 专属使用（通常不建议与其他应用或是框架部署在同一台机器上）且为 Kafka 分配 70% 的带宽资源——考虑到机器上还有其他的进程使用网络且网卡通常不能用满，超过一定阈值可能出现网络丢包的情况，因此 70% 的设定实际上是很合理的——那么 Kafka 单台 broker 的带宽就是 $1\text{Gb/s} \times 0.7 \approx 710\text{Mb/s}$ ，但事实上这是 Kafka 所使用的最高带宽，用户不能奢望 Kafka 集

群平时就一直使用如此多的带宽，毕竟万一碰到突发流量，会极容易把网卡“打满”，因此在70%的基础上，一般再截取1/3，即 $710\text{Mb/s} / 3 \approx 240\text{Mb/s}$ 。这里的1/3是一个相对保守的数字，用户可以根据自身的业务特点酌情增加。好了，根据现有的网络情况，我们明确了单台broker的带宽是240Mb/s。如果要在1小时内处理1TB的业务消息，即每秒需要处理292MB左右的数据，也就是每秒2336Mb数据，那么至少需要 $2336/240 \approx 10$ 台broker机器。若副本数是2，那么这个数字还需要再翻1倍，即20台broker机器。根据万兆位网卡来评估broker机器的方法是类似的。

关于带宽资源方面的规划，用户还需要注意的是尽量避免使用跨机房的网络环境，特别是那些跨城市甚至是跨大洲的网络。因为这些网络条件下请求的延时将会非常高，不管是broker端还是clients端都需要额外做特定的配置才能适应。

综合上述内容，我们对带宽资源规划的建议如下。

- 尽量使用高速网络。
- 根据自身网络条件和带宽来评估Kafka集群机器数量。
- 避免使用跨机房网络。

3.1.7 典型线上环境配置

下面给出一份典型的线上环境配置，用户可以参考这份配置以及结合自己的实际情况进行二次调整。

- CPU 24核。
- 内存 32GB。
- 磁盘 1TB 7200转 SAS 盘两块。
- 带宽 1Gb/s。
- `ulimit -n 1000000`。
- Socket Buffer 至少 64KB——适用于跨机房网络传输。

3.2 伪分布式环境安装

和Hadoop、HBase这样的大多数分布式框架类似，Kafka集群搭建也分为单节点的伪分布式集群和多节点的分布式集群两种方式，只是搭建方法的区别不似其他框架那么明显。本节我们讨论Kafka单节点环境的安装方法。

单节点伪分布式环境是指集群由一台ZooKeeper服务器和一台Kafka broker服务器组成，

如图 3.1 所示。

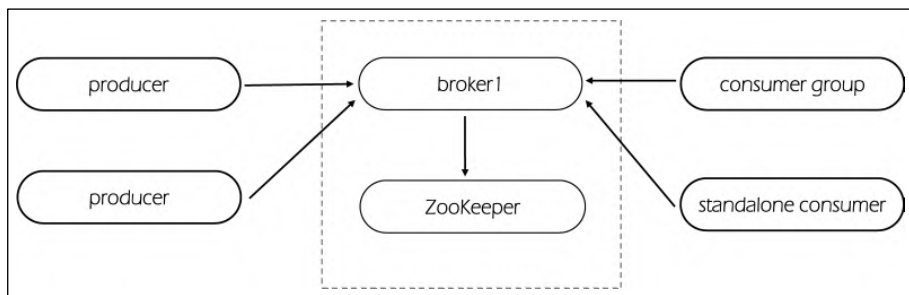


图 3.1 伪分布式集群

为了搭建图 3.1 中的单节点 Kafka 集群，我们必须依次执行以下操作。

- 安装 Java。
- 安装 ZooKeeper。
- 安装 Apache Kafka。

3.2.1 安装 Java

不论是 ZooKeeper 还是 Kafka 都需要提前安装好 Java 并且正确配置好 Java 环境。鉴于 Java 7 自 2015 年 8 月便不再更新，笔者强烈建议 Java 虚拟机版本使用 Java 8，并且使用比较成熟的 Oracle 公司的 HotSpot 虚拟机。为节省篇幅，本章将以 Linux CentOS 6 64 位作为安装演示的操作系统，其他 Linux 发行版的安装方法是类似的。Windows 和 Mac OS 平台上的安装步骤参考官网。

Java 安装可以使用 `java -version` 命令来进行验证。默认情况下，该命令输出显示安装过的 Java 版本。若没有安装 Java，该命令会提示“无法找到 Java 虚拟机”。另外笔者推荐使用 Oracle 版本的虚拟机，而非 OpenJDK 版本的虚拟机。

好了，下面就开始在 CentOS 6 测试环境中安装 Java 8。首先到 Oracle 官网 <http://www.oracle.com/technetwork/java/javase/downloads/index.html> 下载最新的 Java 版本。截至笔者写稿时最新的 Java 8 版本是 JDK 8u162，如图 3.2 所示。

之后单击图 3.2 箭头所指的下载按钮（Download），进入 JDK 8 下载页面，勾选“同意许可证协议”之后单击 `jdk-8u131-linux-x64.tar.gz`，即可下载 Java 的安装文件到本地。若用户使用的是 Linux 终端而非图形化界面，则可以直接运行下面的命令下载 RPM 格式的 Java 安装文件：

```
wget --no-check-certificate --no-cookie --header "Cookie: oraclelicense=accept-securebackup-cookie;" http://download.oracle.com/otn-pub/java/jdk/8u162-b12/0da788060d494f5095bf862473_5fa2f1/jdk-8u162-linux-x64.rpm
```

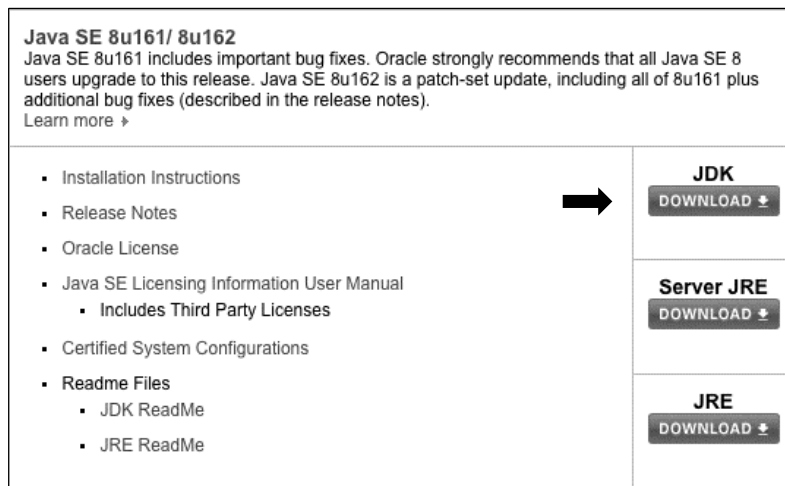


图 3.2 Java 8 下载页面

下载完毕之后执行 `sudo rpm -ivh jdk-8u162-linux-x64.rpm` 命令进行安装，默认的安装路径是 `/usr/java/jdk1.8.0_162`，因此设置 `JAVA_HOME` 环境变量指向该路径并把该路径下的 `bin` 路径加入 `$PATH` 中，然后 `source` 它们就可以了，命令如下：

```
export JAVA_HOME=/usr/java/jdk1.8.0_162
export JRE_HOME=$JAVA_HOME/jre
export PATH=$PATH:$JRE_HOME/bin:$JAVA_HOME/bin
```

最后使用 `java -version` 命令验证 Java 是否安装成功，如图 3.3 所示。

```
[work@10.10.10.10 ~]$ java -version
java version "1.8.0_162"
Java(TM) SE Runtime Environment (build 1.8.0_162-b12)
Java HotSpot(TM) 64-Bit Server VM (build 25.162-b12, mixed mode)
```

图 3.3 验证 Java 是否安装成功

3.2.2 安装 ZooKeeper

ZooKeeper 是安装 Kafka 集群必要的组件，并且 Kafka 大量地使用 ZooKeeper 来保存集群的元数据信息以及 consumer 位移信息（老版本）。虽然在伪分布式集群中直接使用 Kafka 自带的 ZooKeeper 可能更方便，但其实单独安装一个外部的 ZooKeeper 服务器同样很简单。当前最新版本的 Kafka 集成了 3.4.10 版本的 ZooKeeper，不过 ZooKeeper 社区当前最新的稳定版本

是 3.4.11，本节中我们依然使用 3.4.10 版本的 ZooKeeper。

与 Java 类似，首先从 ZooKeeper 下载地址中 <https://www-us.apache.org/dist/zookeeper/stable/> 下载 ZooKeeper 二进制代码压缩包 `zookeeper-3.4.10.tar.gz`，然后依次执行下面的命令进行安装：

```
tar -zxvf zookeeper-3.4.10.tar.gz
mv zookeeper-3.4.10 zookeeper
mkdir -p /home/user/zk
cd zookeeper
```

之后编辑 `zookeeper/conf` 目录下的 `zoo.cfg` 文件，写入下列内容，然后保存 `zoo.cfg` 文件即可：

```
tickTime=2000
dataDir=/home/user/zk
clientPort=2181
```

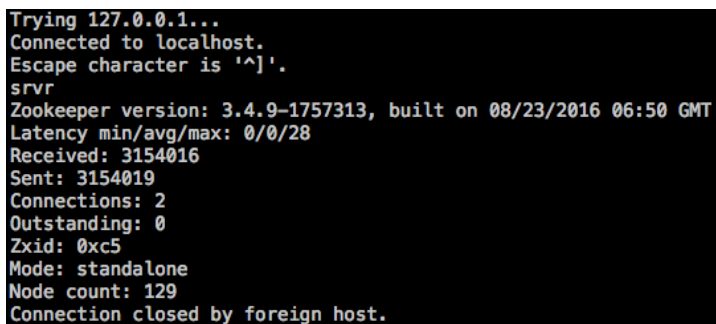
下面可以启动 ZooKeeper 服务器了，假设你已经处于 `zookeeper` 目录下，则执行如下命令：

```
bin/zkServer.sh start
```

如果一切顺利，你应该可以看到下列输出：

```
JMX enabled by default
Using config: /usr/local/zookeeper/bin/.../conf/zoo.cfg
Starting zookeeper ... STARTED
```

以上输出表明 ZooKeeper 服务器启动成功，现在我们可以使用 Telnet 连入 2181 端口所在的 ZooKeeper 服务，并执行 `srvr` 命令获取 ZooKeeper 服务器的状态信息，如图 3.4 所示。



```
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
srvr
Zookeeper version: 3.4.9-1757313, built on 08/23/2016 06:50 GMT
Latency min/avg/max: 0/0/28
Received: 3154016
Sent: 3154019
Connections: 2
Outstanding: 0
Zxid: 0xc5
Mode: standalone
Node count: 129
Connection closed by foreign host.
```

图 3.4 获取 ZooKeeper 服务器的状态信息

3.2.3 安装单节点 Kafka 集群

成功安装 Java 和 ZooKeeper 之后，下一步就是安装 Apache Kafka 了。在单节点的伪分布式集群中我们只会安装并启动一个 Kafka broker。后面 3.3 节中我们会给出多节点分布式 Kafka

集群的安装方法。

第一步我们下载 Apache Kafka，官网地址是 <http://kafka.apache.org/downloads.html>。截至笔者写稿时，当前最新的版本是 1.0.0，因此我们需要下载的文件是 `kafka_2.11-1.0.0.tgz`。笔者在这里选用由 Scala 2.11 编译的版本，因为社区刚刚加入对 Scala 2.12 的支持且 Scala 2.12 不支持 Java 7，某些用户可能无法使用该版本编译的 Kafka。

文件下载完成之后执行解压缩操作并创建保存 Kafka 数据的文件目录：

```
tar -zxvf kafka_2.11-1.0.0.tgz
mv kafka_2.11-1.0.0 kafka
mkdir -p /home/work/kafka/data-logs
cd kafka
```

之后打开 `config` 目录下的 `server.properties` 文件，修改下列配置：

```
log.dirs=/home/work/kafka/data-logs
```

然后保存修改并退出。最后通过下列命令启动 Kafka broker：

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

如果你想要在后台运行 Kafka broker，只需要在启动命令中加入 `-daemon`：

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

看到以下输出证明启动成功：

```
INFO [Kafka Server 0], started (kafka.server.KafkaServer)
```

3.3 多节点环境安装

单节点的 Kafka 伪集群应付日常的应用开发或是功能验证绰绰有余，但若在生产环境中直接使用则无法充分利用 Kafka 提供的分布式特性，比如负载均衡和故障转移等。此外，Kafka 还具有优秀的准线性扩容的能力，因此用户可以很容易地扩展 Kafka 节点数量以应对不断增长的消息处理需求。同时由于 Kafka 提供了完备的备份机制，多节点集群天然地为用户提供高可用保障，极大地降低了人工维护的成本。

从本质上来说，多节点 Kafka 集群由一套多节点 ZooKeeper 集群和一套多节点 Kafka 集群组成，如图 3.5 所示。

为了搭建图 3.5 中的多节点 Kafka 集群，我们必须依次执行以下操作。

- 安装多节点 ZooKeeper 集群。

- 安装多节点 Kafka 集群。

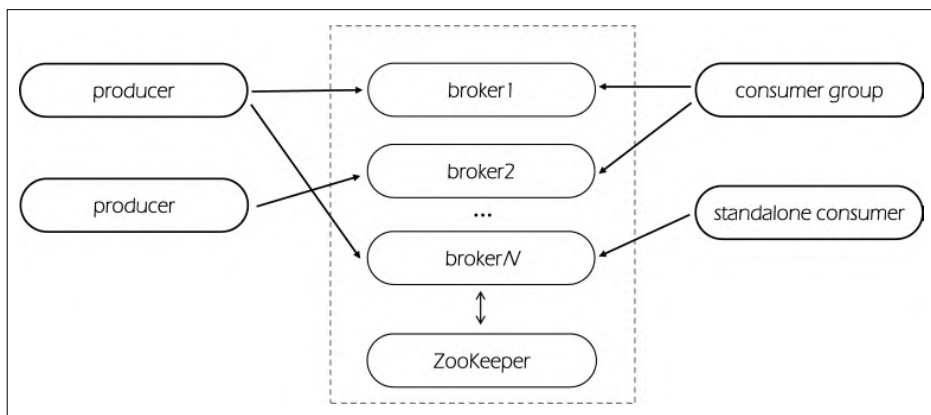


图 3.5 多节点 Kafka 集群

3.3.1 安装多节点 ZooKeeper 集群

目前来说 Kafka 可以说是强依赖 ZooKeeper 的，因此生产环境中一个高可用、高可靠的 ZooKeeper 集群也是必不可少的。ZooKeeper 集群通常被称为一个 ensemble。只要这个 ensemble 中的大多数节点存活，那么 ZooKeeper 集群就能正常提供服务。显然，既然是大多数，那么最好使用奇数个服务器，即 $2n + 1$ 个服务器，这样整个 ZooKeeper 集群最多可以容忍 n 台服务器宕机而保证依然提供服务。如果使用偶数个服务器则通常会浪费一台服务器的资源。

下面举一个例子来说明：假设我们使用 5 台 ZooKeeper 服务器构建集群，倘若 2 台服务器宕机，剩下的 3 台服务器占了半数以上，故而 ZooKeeper 服务正常工作；但假如我们使用了 4 台服务器，若 2 台服务器宕机，剩下的 2 台服务器不满足“半数以上服务器存活”的条件，因此此时 ZooKeeper 集群将停止服务——由此可见，虽然使用了 4 台服务器，但我们依然只能容忍 1 台服务器崩溃，这就是为什么 ZooKeeper 集群节点数量通常是奇数的原因。

基于上面的规则，一个生产环境中最少的 ZooKeeper 集群节点数量是 3，这样 1 个节点“挂掉”了不会影响整个集群的运作。在实际使用场景中，5 台服务器构成的 ZooKeeper 集群也是十分常见的，而再多数量的集群则不常见。当然具体数量的确定要依据用户对高可靠性的需求。通常我们都需要在高可靠性与成本间取得适当的平衡。

在安装 ZooKeeper 集群之前，我们假定 Java 已经被正确地安装和配置到了生产环境中，这里不再赘述。本例将安装一个有 3 个节点的 ZooKeeper 集群，并假设 3 个节点机器的主机名分别是 zk1、zk2 和 zk3。多节点模式中所用到的配置文件与我们在单节点 ZooKeeper 安装相关

章节中的配置文件大部分相同，只是有些微小的差别。下面的例子给出了一份典型的多节点环境配置文件 zoo.cfg:

```
tickTime=2000
dataDir=/usr/zookeeper/data_dir
clientPort=2181
initLimit=5
syncLimit=2
server.1=zk1:2888:3888
server.2=zk2:2888:3888
server.3=zk3:2888:3888
```

比较关键的参数含义如下。

- **tickTime**: ZooKeeper 最小的时间单位，用于丈量心跳时间和超时时间等。通常设置成默认值 2 秒即可。
- **dataDir**: 非常重要的参数！ZooKeeper 会在内存中保存系统快照，并定期写入该路径指定的文件夹中。生产环境中需要注意该文件夹的磁盘占用情况。
- **clientPort**: ZooKeeper 监听客户端连接的端口，一般设置成默认值 2181 即可。
- **initLimit**: 指定 follower 节点初始时连接 leader 节点的最大 tick 次数。假设是 5，表示 follower 必须要在 $5 \times \text{tickTime}$ 时间内（默认是 10 秒）连接上 leader，否则将被视为超时。
- **syncLimit**: 设定了 follower 节点与 leader 节点进行同步的最大时间。与 initLimit 类似，它也是以 tickTime 为单位进行指定的。
- **server.X=host:port:port**: 配置文件中的最后 3 行都是这种形式的。这里的 X 必须是一个全局唯一的数字，且需要与 myid 文件中的数字相对应（关于 myid 文件的设置稍后会做详细讨论）。一般设置 X 值为 1~255 之间的整数。这行的后面还配置了两个端口，通常是 2888 和 3888。第一个端口用于使 follower 节点连接 leader 节点，而第二个端口则用于 leader 选举。

设置好配置文件，下面就该创建上面提到的 myid 文件。众所周知，每个 ZooKeeper 服务器都有一个唯一的 ID。这个 ID 主要用在两个地方：一个就是刚刚我们配置的 zoo.cfg 文件，另一个则是 myid 文件。myid 文件位于 zoo.cfg 中 dataDir 配置的目录下，其内容也很简单，仅是一个数字，即 ID。

下面就以一台机器为例展示一下如何安装 3 个节点的 ZooKeeper 集群。值得一提的是，在一台机器上安装多节点 ZooKeeper 集群和在多台机器上安装的方法是相同的，只是配置文件有一些微小的调整，我们会显式地解释具体的调整项差别。

首先，在 ZooKeeper 的 conf 目录下创建 3 个配置文件 zoo1.cfg、zoo2.cfg 和 zoo3.cfg，分别如图 3.6、图 3.7 和图 3.8 所示。注意，如果是在多台服务器上安装 ZooKeeper 集群，可以选择相同的配置文件名字。

```
tickTime=2000
dataDir=/mnt/disk/huxitest/data_logs/zookeeper1
clientPort=2181
initLimit=5
syncLimit=2
server.1=zk1:2888:3888
server.2=zk2:2889:3889
server.3=zk3:2890:3890
```

图 3.6 zoo1.cfg 配置文件

```
tickTime=2000
dataDir=/mnt/disk/huxitest/data_logs/zookeeper2
clientPort=2182
initLimit=5
syncLimit=2
server.1=zk1:2888:3888
server.2=zk2:2889:3889
server.3=zk3:2890:3890
```

图 3.7 zoo2.cfg 配置文件

```
tickTime=2000
dataDir=/mnt/disk/huxitest/data_logs/zookeeper3
clientPort=2183
initLimit=5
syncLimit=2
server.1=zk1:2888:3888
server.2=zk2:2889:3889
server.3=zk3:2890:3890
```

图 3.8 zoo3.cfg 配置文件

上面的配置文件中我们分别选取了 2181、2182 和 2183 这 3 个端口。如果是多机器安装方案，指定相同的端口号也是可以的，只要确保没有端口冲突就行，这也包括配置文件中的所有其他端口。

创建好 ZooKeeper 配置文件，下一步就是创建 myid 文件了。myid 文件必须位于配置文件中的 dataDir 中，即/mnt/disk/huxitest/data_logs/zookeeper1,2,3 下，具体命令如下：

```
echo "1" > /mnt/disk/huxitest/data_logs/zookeeper1/myid
echo "2" > /mnt/disk/huxitest/data_logs/zookeeper2/myid
echo "3" > /mnt/disk/huxitest/data_logs/zookeeper3/myid
```

下一步就是启动 3 个控制台终端分别在 ZooKeeper 的安装目录下执行以下命令启动 ZooKeeper 服务器：

```
java -cp zookeeper-3.4.10.jar:lib/slf4j-api-1.6.1.jar:lib/slf4j-log4j12-1.6.1.jar:lib/log4j-1.2.16.jar:conf org.apache.zookeeper.server.quorum.QuorumPeerMain conf/zoo1.cfg
java -cp zookeeper-3.4.10.jar:lib/slf4j-api-1.6.1.jar:lib/slf4j-log4j12-1.6.1.jar:lib/log4j-1.2.16.jar:conf org.apache.zookeeper.server.quorum.QuorumPeerMain conf/zoo2.cfg
java -cp zookeeper-3.4.10.jar:lib/slf4j-api-1.6.1.jar:lib/slf4j-log4j12-1.6.1.jar:lib/log4j-1.2.16.jar:conf org.apache.zookeeper.server.quorum.QuorumPeerMain conf/zoo3.cfg
```

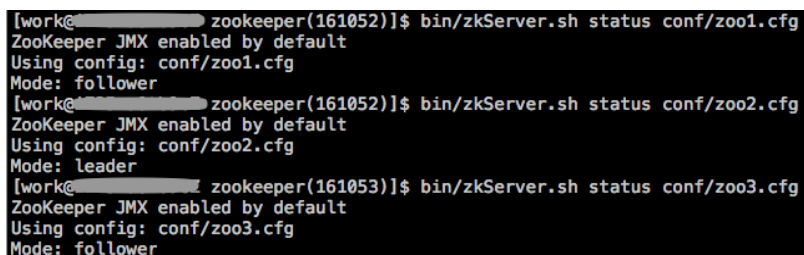
如果是多节点安装方案，既可以使用上面的命令启动 ZooKeeper，也可以直接运行 zkServer 脚本启动 ZooKeeper，比如：

```
bin/zkServer.sh(bat) start conf/zoo.cfg
```

当所有 ZooKeeper 服务器启动成功后，我们还需要检查一下整个集群的状态，分别执行以下命令：

```
cd /mnt/disk/huxitest/zookeeper
bin/zkServer.sh status conf/zoo1.cfg
bin/zkServer.sh status conf/zoo2.cfg
bin/zkServer.sh status conf/zoo3.cfg
```

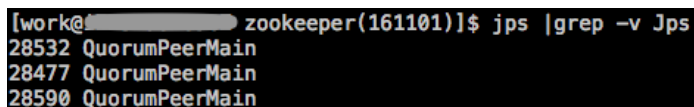
其输出结果如图 3.9 所示。



```
[work@zookeeper(161052)]$ bin/zkServer.sh status conf/zoo1.cfg
ZooKeeper JMX enabled by default
Using config: conf/zoo1.cfg
Mode: follower
[work@zookeeper(161052)]$ bin/zkServer.sh status conf/zoo2.cfg
ZooKeeper JMX enabled by default
Using config: conf/zoo2.cfg
Mode: leader
[work@zookeeper(161053)]$ bin/zkServer.sh status conf/zoo3.cfg
ZooKeeper JMX enabled by default
Using config: conf/zoo3.cfg
Mode: follower
```

图 3.9 ZooKeeper 集群状态

由图 3.9 可以看出，当前服务器 2 是整个集群的 leader，其他两个服务器则负责 follower 的角色。另外还可以执行 JDK 的 jps 命令来查看 ZooKeeper 进程以查看集群状态，如图 3.10 所示。



```
[work@zookeeper(161101)]$ jps |grep -v Jps
28532 QuorumPeerMain
28477 QuorumPeerMain
28590 QuorumPeerMain
```

图 3.10 jps 命令输出

ZooKeeper 的主进程名是 QuorumPeerMain，图 3.10 的 jps 命令输出也可以证明已经成功地

启动了 3 个 ZooKeeper 进程。值得一提的是，如果是在多台机器上搭建 ZooKeeper 集群，那么每台机器上都至少应该有一个这样的进程被启动。

好了，我们已经成功地搭建起 ZooKeeper 集群，下面需要搭建 Kafka 集群。

3.3.2 安装多节点 Kafka

安装多节点的 Kafka 比安装多节点的 ZooKeeper 要简单得多，我们只需要创建多份配置文件，然后指定它们启动 Kafka 服务即可。本例中依然使用一台机器来模拟一个 3 节点 Kafka 集群的搭建。同理，使用一台机器搭建 Kafka 集群和使用多台机器搭建是类似的，我们会显式地解释其中的不同之处。

搭建 Kafka 集群的第一步是要创建多份配置文件，如图 3.11、图 3.12 和图 3.13 所示。

```
[work@kafka(161125)]$ cat config/server1.properties
broker.id=0
delete.topic.enable=true
listeners=PLAINTEXT://kafka1:9092
log.dirs=/mnt/disk/huxitest/data_logs/kafka1
zookeeper.connect=zk1:8001,zk2:8002,zk3:8003
unclean.leader.election.enable=false
zookeeper.connection.timeout.ms=6000
```

图 3.11 kafka 配置文件 1

```
[work@kafka(161125)]$ cat config/server2.properties
broker.id=1
delete.topic.enable=true
listeners=PLAINTEXT://kafka2:9093
log.dirs=/mnt/disk/huxitest/data_logs/kafka2
zookeeper.connect=zk1:8001,zk2:8002,zk3:8003
unclean.leader.election.enable=false
zookeeper.connection.timeout.ms=6000
```

图 3.12 kafka 配置文件 2

```
[work@kafka(161126)]$ cat config/server3.properties
broker.id=2
delete.topic.enable=true
listeners=PLAINTEXT://kafka3:9094
log.dirs=/mnt/disk/huxitest/data_logs/kafka3
zookeeper.connect=zk1:8001,zk2:8002,zk3:8003
unclean.leader.election.enable=false
zookeeper.connection.timeout.ms=6000
```

图 3.13 kafka 配置文件 3

在上面 3 个配置文件中我们需要每台 Kafka 服务器指定不同的 broker ID。该 ID 在整个集群中必须是唯一的。而配置 listeners 时最好使用节点的 FQDN（Fully Qualified Domain Name），即全称域名，尽量不要使用 IP 地址。另外配置文件中还有一个特别重要的参数，即 zookeeper.connect。鉴于我们之前已经搭建了一个 3 节点 ZooKeeper 集群，此时配置 zookeeper.connect 必须同时指定所有的 ZooKeeper 节点。注意本例中使用的端口分别是 8001、8002 和 8003。

创建 Kafka 配置文件之后，剩下的工作就很简单了，只需要执行下列命令启动 Kafka broker 服务器：

```
bin/kafka-server-start.sh -daemon config/server1.properties
bin/kafka-server-start.sh -daemon config/server2.properties
bin/kafka-server-start.sh -daemon config/server3.properties
```

查看位于 Kafka logs 目录下的 server.log，确认 Kafka broker 已经启动成功，如图 3.14 所示。

```
[work@ ~]$ logs(161139) tail -f server.log
[2017-05-16 11:38:43,594] INFO [ExpirationReaper-2], Starting (kafka.server.DelayedOperationPurgatory$ExpiredOperationReaper)
[2017-05-16 11:38:43,618] INFO [GroupCoordinator 2]: Starting up. (kafka.coordinator.GroupCoordinator)
[2017-05-16 11:38:43,619] INFO [GroupCoordinator 2]: Startup complete. (kafka.coordinator.GroupCoordinator)
[2017-05-16 11:38:43,621] INFO [Group Metadata Manager on Broker 2]: Removed 0 expired offsets in 2 milliseconds. (kafka.coordinator.GroupMetadataManager)
[2017-05-16 11:38:43,645] INFO Will not load MX4J, mx4j-tools.jar is not in the classpath (kafka.utils.Mx4JLoader$)
[2017-05-16 11:38:43,679] INFO Creating /brokers/ids/2 (is it secure? false) (kafka.utils.ZKCheckedEphemeral)
[2017-05-16 11:38:43,709] INFO Result of znode creation is: OK (kafka.utils.ZKCheckedEphemeral)
[2017-05-16 11:38:43,714] INFO Registered broker 2 at path /brokers/ids/2 with addresses: PLAINTEXT -> EndPoint(kafka3,9094,PLAINTEXT) (kafka.utils.ZKUtils)
[2017-05-16 11:38:43,715] WARN No meta.properties file under dir /mnt/disk/huxitest/data_logs/kafka3/meta.properties (kafka.server.BrokerMetadataCheckpoint)
[2017-05-16 11:38:43,800] INFO [Kafka Server 2], started (kafka.server.KafkaServer)
```

图 3.14 broker 启动日志

另外，也可以使用 jps 命令来确认 broker 进程启动成功，如图 3.15 所示。

```
[work@ ~]$ logs(161142) jps |grep Kafka
31488 Kafka
31763 Kafka
31212 Kafka
```

图 3.15 查看 broker 进程状态

至此，一个 3 节点的 Kafka 分布式集群就搭建成功了。如果需要为集群增加更多的 Kafka broker 节点，只需要配置一份类似的配置文件，然后利用该文件直接运行启动命令即可。应该说，Kafka 对于集群扩展操作是很友好的，不需要用户承担太多的维护和管理成本。

3.4 验证部署

成功搭建起多节点的 Kafka 集群还不够，我们还需要验证线上环境是没有错误且可以使用的。下面将从以下几个方面分别来验证 Kafka 集群部署的正确性。

- 测试 topic 创建与删除。
- 测试消息的生产与发送。
- 生产者吞吐量测试。
- 消费者吞吐量测试。

3.4.1 测试 topic 创建与删除

topic 能够正确地创建与删除才能说明 Kafka 集群在正常工作，因为通常它都表明了 Kafka

的控制器（controller）已经被成功地选举出来并开始履行自身的职责。很多用户喜欢在集群搭建后就立即开始运行 producer 和 consumer，甚至完全依靠 Kafka 的自动 topic 创建功能，而不去手动创建并做验证。笔者其实并不推荐这种做法，以笔者多年使用 Kafka 的经验，建议读者在开始使用 Kafka 集群前最好提前把所需的 topic 创建出来，并执行对应的命令做验证。这样既可以测试整个集群的运行状况，也保证了 producer 和 consumer 运行时不会因为 topic 分区 leader 的各种问题导致短暂停顿现象。

接下来，我们首先创建一个测试 topic，名为 test-topic。为了充分利用搭建的 3 台服务器，我们创建 3 个分区，每个分区都分配 3 个副本，执行如下命令：

```
bin/kafka-topics.sh --zookeeper zk1:8001,zk2:8002,zk3:8003 --create --
topic test-topic --partitions 3 --replication-factor 3
Created topic "test-topic".
```

上面的输出表明 topic 已经被成功创建，但我们还需要运行一些命令来做详细验证，如下：

```
bin/kafka-topics.sh --zookeeper zk1:8001,zk2:8002,zk3:8003 -list
test-topic
bin/kafka-topics.sh --zookeeper zk1:8001,zk2:8002,zk3:8003 --describe --
topic test-topic
```

topic 详细分区信息如图 3.16 所示。

Topic:test-topic	PartitionCount:3	ReplicationFactor:3	Configs:
Topic: test-topic	Partition: 0	Leader: 0	Replicas: 0,1,2 Isr: 0,1,2
Topic: test-topic	Partition: 1	Leader: 1	Replicas: 1,2,0 Isr: 1,2,0
Topic: test-topic	Partition: 2	Leader: 2	Replicas: 2,0,1 Isr: 2,0,1

图 3.16 topic 详细分区信息

显而易见，该 topic 下有 3 个分区，每个分区有 3 个副本。每个分区的 leader 分别是 0、1、2，表明 Kafka 将该 topic 的这 3 个分区均匀地在 3 台 broker 上进行了分配。

下面来测试删除 topic，执行如下命令：

```
>bin/kafka-topics.sh --zookeeper zk1:8001,zk2:8002,zk3:8003 --delete --
topic test-topic
Topic test-topic is marked for deletion.
Note: This will have no impact if delete.topic.enable is not set to true.
```

上面的输出仅仅表示该 topic 被成功地标记为“待删除”，至于 topic 是否会被真正删除取决于 broker 端参数 delete.topic.enable。该参数在当前 Kafka 1.0.0 版本中被默认设置为 true，即表明 Kafka 默认允许删除 topic。事实上，该参数在旧版本中默认值一直是 false，故若用户显式设置该参数为 false，或使用了 1.0.0 之前版本的默认值，那么即使运行了上面的命令，Kafka 也不会删除该 topic。

由于在之前搭建 Kafka 集群时我们配置了该参数为 true，因此 Kafka 会将所有与该 topic 相

关的数据全部删除，不过这是一个异步过程，具体的删除操作对命令执行者来说是完全透明的。我们可以查询底层的文件系统来验证 topic 分区的日志目录是否被删除干净了。

另外，也可以再次执行 kafka-topics 脚本来列出当前的 topic 列表，如果 test-topic 不在该列表中，则表明该 topic 被删除成功，执行如下命令：

```
>bin/kafka-topics.sh --zookeeper zk1:8001,zk2:8002,zk3:8003 --list
```

本例中该命令输出为空，表明 test-topic 被成功删除。有些时候，用户可能发现 topic 没有被成功删除，这可能是 topic 分区数过多或数据过多的原因。Kafka 当前只能一个分区一个分区地删除数据，无法做到同时删除，因此用户可以查询底层的文件系统来判断删除操作是否在正常执行。如果发现删除操作停滞了，这通常表明该 Kafka 集群有问题，此时便需要进一步地详查才能确定无法删除的真正原因，比如是否有分区正处于被分配过程中等。

3.4.2 测试消息发送与消费

成功测试了 topic 之后接下来我们测试一下 Kafka 集群是否可以正常地发送消息和读取消息。为了实现这一目的，我们需要用到 Kafka 默认提供的 kafka-console-producer 和 kafka-console-consumer 脚本。这对脚本可以很方便地用来测试消息的发送和读取。发送消息时，用户从键盘输入消息，按回车键后即表示发送该条消息。下面我们打开两个终端，一个用于发送消息，另一个用于接收消息，如图 3.17 和图 3.18 所示。

```
[work@kafka(161638)]$ bin/kafka-console-producer.sh --broker-list kafka1:9092,kafka2:9093,kafka3:9094 --topic test-topic
this is a test message
hello, Kafka
```

图 3.17 console-producer 执行界面

```
[work@kafka(161641)]$ bin/kafka-console-consumer.sh --bootstrap-server kafka1:9092,kafka2:9093,kafka3:9094 --topic test-topic --from-beginning
this is a test message
hello, Kafka
```

图 3.18 console-consumer 执行界面

图 3.17 中使用 kafka-console-producer 脚本生产了两条消息，在另一个终端中开启 kafka-console-consumer 之后可以马上观测到刚才生产的两条消息已经输出在图 3.17 的终端上了。

现在在 console-consumer 终端上按 Ctrl+C 组合键停掉当前的 consumer 程序，然后执行下列命令开启一个新 consumer，如图 3.19 所示。

```
[work@kafka(161650)]$ bin/kafka-console-consumer.sh --zookeeper zk1:8001,zk2:8002,zk3:8003 --topic test-topic --from-beginning
Using the ConsoleConsumer with old consumer is deprecated and will be removed in a future major release. Consider using the new consumer by passing [bootstrap-server] instead of [zookeeper].
hello, Kafka
this is a test message
```

图 3.19 旧版本 console-consumer 执行界面

细心的读者不难发现，这次执行 `kafka-console-consumer` 脚本指定了不同的参数。图 3.17 指定的是 `bootstrap-server`，而图 3.19 指定的是 `zookeeper`。这两个 `consumer` 分别代表了第 2 章中提到的新版本 `consumer` 和旧版本 `consumer`。如图 3.19 所示，旧版本 `consumer` 也能正确地读取之前生产的两条消息。

在后续 Kafka 版本中可能会完全移除旧版本的 `consumer` 代码，但目前来说我们还要同时验证新版本和旧版本 `consumer`，这样才能百分百地保证集群是可以正常工作的。

3.4.3 生产者吞吐量测试

除了基本的 `console-producer` 和 `console-consumer` 脚本可以用于测试简单的消息发送与接收，Kafka 还提供了性能吞吐量测试脚本，它们分别是 `kafka-producer-perf-test` 脚本和 `kafka-consumer-perf-test` 脚本。本节中我们首先讨论 `kafka-producer-perf-test` 脚本。

`kafka-producer-perf-test` 脚本是 Kafka 提供的用于测试 `producer` 性能的脚本，该脚本可以很方便地计算出 `producer` 在一段时间内的吞吐量和平均延时，具体使用说明如图 3.20 所示。

```
[work@ kafka(161734)]$ bin/kafka-producer-perf-test.sh --topic test-topic --num-records 500000 --record-size 200 --throughput -1 --producer-props bootstrap.servers=kafka1:9092,kafka2:9093,kafka3:9094 acks=-1
145012 records sent, 28996.6 records/sec (5.53 MB/sec), 1092.3 ms avg latency, 2362.0 max latency.
260120 records sent, 51703.4 records/sec (9.86 MB/sec), 2902.9 ms avg latency, 3513.0 max latency.
500000 records sent, 41963.911037 records/sec (8.00 MB/sec), 2362.85 ms avg latency, 3513.00 ms max latency, 2792 ms
50th, 3144 ms 95th, 3364 ms 99th, 3503 ms 99.9th.
```

图 3.20 `kafka-producer-perf-test` 脚本执行界面

图 3.20 中该脚本的输出结果表明在这台测试机上运行一个 Kafka `producer` 的平均吞吐量是 8MB/s，即占用 64Mb/s 左右的带宽，平均每秒能发送 41963 条消息，平均延时是 2.36 秒，最大延时是 3.51 秒，平均有 50% 的消息发送需要花费 2.79 秒，95% 的消息发送需要花费 3.14 秒，99% 的消息发送需要花费 3.36 秒，而 99.9% 的消息发送需要花费 3.5 秒。本例测试出来的 `producer` 占用的带宽是 64Mb/s ($8 \times 8.0.0$ MB/s)，和千兆位网卡相比，这点带宽远远没有达到网卡的上限，因此这也说明 `producer` 还有很大的优化空间。

在 `kafka-producer-perf-test` 脚本的实际使用过程中，最好让该脚本长时间稳定地运行一段时间，这样测试出来的结果才能准确。毕竟脚本运行之初会执行很多初始化工作，运行之处的吞吐量不能真实反映系统的实际情况。关于这个脚本各个参数的用法，我们会在稍后的章节中详细展开讨论。

3.4.4 消费者吞吐量测试

和 `kafka-producer-perf-test` 脚本类似，Kafka 为 `consumer` 也提供了方便、便捷的性能测试脚本，即 `kafka-consumer-perf-test` 脚本。我们首先用它在刚刚搭建的 Kafka 集群环境中测试一

下新版本 consumer 的吞吐量，如图 3.21 所示。

```

$ bin/kafka-consumer-perf-test.sh --broker-list kafka1:9092,kafka2:9093,kafka3:9094
--fetch-size 2000 --messages 500000 --topic test-topic
start.time, end.time, data.consumed.in.MB, MB.sec, data.consumed.in.nMsg, nMsg.sec, rebalance.time.ms, fetch.time.ms, fetc
h.MB.sec, fetch.nMsg.sec
2018-08-14 09:45:33:410, 2018-08-14 09:45:58:678, 488.2813, 19.3241, 500000, 19787.8740, 16, 25252, 19.3363, 19800.4118

```

图 3.21 kafka-consumer-perf-test 脚本测试新版本 consumer 吞吐量

图 3.21 的例子中我们测试消费 50 万条消息的 consumer 的吞吐量。结果表明，在该环境中 consumer 在 1 秒多的时间内总共消费了 95MB 的消息，因此吞吐量大约是 92MB/s，即 736Mb/s。这样我们就对本机 consumer 的性能有了一个大致的了解，为后续评估整体 consumer 性能目标提供了一个很有力的基础。

图 3.21 测试的是新版本 consumer 的吞吐量，下面依然使用该脚本测试一下旧版本 consumer，如图 3.22 所示。

```

[work@i1000000000 kafka(161744)]$ bin/kafka-consumer-perf-test.sh --zookeeper zk1:8001,zk2:8002,zk3:
8003 --message-size 200 --messages 500000 --topic test-topic --threads 3
start.time, end.time, data.consumed.in.MB, MB.sec, data.consumed.in.nMsg, nMsg.sec
2017-05-16 17:44:37:147, 2017-05-16 17:44:38:444, 190.7349, 147.0585, 1000002, 771011.5652

```

图 3.22 kafka-consumer-perf-test 脚本测试旧版本 consumer 吞吐量

图 3.22 中运行 consumer 指定的是 ZooKeeper 的连接信息，说明我们使用的是旧版本 consumer。结果表明本次测试共消费 190MB 左右的消息，吞吐量大约是 147MB/s。

3.5 参数设置

前面我们通过 Kafka 提供的各种脚本工具验证了 Kafka 集群能够正常工作。接下来，需要讨论 Kafka 集群涉及的各方面参数，主要包括以下几种参数。

- broker 端参数。
- topic 级别参数。
- GC 配置参数。
- JVM 参数。
- OS 参数。

3.5.1 broker 端参数

Kafka broker 端提供了很多参数用于调优系统的各个方面，有一些参数是所有 Kafka 环境都需要考虑和配置的，不论是单机环境还是分布式环境。这些参数都是 Kafka broker 的基础配

置，一定要明确它们的含义。

broker 端参数需要在 Kafka 目录下的 `config/server.properties` 文件中进行设置。当前对于绝大多数的 broker 端参数而言，Kafka 尚不支持动态修改——这就是说，如果要新增、修改，抑或是删除某些 broker 参数的话，需要重启对应的 broker 服务器。下面就让我们来看看主要的参数配置。

- **broker.id**——Kafka 使用唯一的一个整数来标识每个 broker，这就是 **broker.id**。该参数默认是 -1。如果不指定，Kafka 会自动生成一个唯一值。总之，不管用户指定什么都必须保证该值在 Kafka 集群中是唯一的，不能与其他 broker 冲突。在实际使用中，推荐使用从 0 开始的数字序列，如 0、1、2……
- **log.dirs**——非常重要的参数！该参数指定了 Kafka 持久化消息的目录。若待保存的消息数量非常多，那么最好确保该文件夹下有充足的磁盘空间。该参数可以设置多个目录，以逗号分隔，比如 `/home/kafka1,/home/kafka2`。在实际使用过程中，指定多个目录的做法通常是被推荐的，因为这样 Kafka 可以把负载均匀地分配到多个目录下。若用户机器上有 N 块物理硬盘（并且假设这台机器完全给 Kafka 使用），那么设置 N 个目录（须挂载在不同磁盘上的目录）是一个很好的选择。 N 个磁头可以同时执行写操作，极大地提升了吞吐量。注意，这里的“均匀”是根据目录下的分区数进行比较的，而不是根据实际的磁盘空间。值得一提的是，若不设置该参数，Kafka 默认使用 `/tmp/kafka-logs` 作为消息保存的目录。把消息保存在 `/tmp` 目录下，在实际的生产环境中是极其不可取的。
- **zookeeper.connect**——同样是非常重要的参数。如果说前两个参数还有默认值可以使用的话（虽然极其不推荐将默认值应用到线上环境），那么此参数则完全没有默认值，是必须要设置的。该参数也可以是一个 CSV（comma-separated values）列表，比如在前面的例子中设置的那样：`zk1:2181,zk2:2181,zk3:2181`。如果要使用一套 ZooKeeper 环境管理多套 Kafka 集群，那么设置该参数的时候就必须指定 ZooKeeper 的 `chroot`，比如 `zk1:2181,zk2:2181,zk3:2181/kafka_cluster1`。结尾的 `/kafka_cluster1` 就是 `chroot`，它是可选的配置，如果不指定则默认使用 ZooKeeper 的根路径。在实际使用过程中，配置 `chroot` 可以起到很好的隔离效果。这样管理 Kafka 集群将变得更加容易。
- **listeners**——broker 监听器的 CSV 列表，格式是 `[协议]:/[主机名]:[端口],[[协议]]:/[主机名]:[端口]`。该参数主要用于客户端连接 broker 使用，可以认为是 broker 端开放给 clients 的监听端口。如果不指定主机名，则表示绑定默认网卡；如果主机名是 `0.0.0.0`，则表示绑定所有网卡。Kafka 当前支持的协议类型包括 PLAINTEXT、SSL 及 SASL_SSL 等。对于新版本的 Kafka，笔者推荐只设置 `listeners` 一个参数就够了，对于已经过时的两个参数 `host.name` 和 `port`，就不用再配置了。对于未启用安全的 Kafka 集群，使用 PLAINTEXT

协议足矣。如果启用了安全认证，可以考虑使用 SSL 或 SASL_SSL 协议。

- `advertised.listeners`——和 `listeners` 类似，该参数也是用于发布给 `clients` 的监听器，不过该参数主要用于 IaaS 环境，比如云上的机器通常都配有多块网卡（私网网卡和公网网卡）。对于这种机器，用户可以设置该参数绑定公网 IP 供外部 `clients` 使用，然后配置上面的 `listeners` 来绑定私网 IP 供 `broker` 间通信使用。当然不设置该参数也是可以的，只是云上的机器很容易出现 `clients` 无法获取数据的问题，原因就是 `listeners` 绑定的是默认网卡，而默认网卡通常都是绑定私网 IP 的。在实际使用场景中，对于配有多块网卡的机器而言，这个参数通常都是需要配置的。
- `unclean.leader.election.enable`——是否开启 `unclean leader` 选举。何为 `unclean leader` 选举？在第 1 章中我们提到了 ISR 的概念。ISR 中的所有副本都有资格随时成为新的 `leader`，但若 ISR 变空而此时 `leader` 宕机了，Kafka 应该如何选举新的 `leader` 呢？为了不影响 Kafka 服务，该参数默认值是 `false`，即表明如果发生这种情况，Kafka 不允许从剩下存活的非 ISR 副本中选择一个当 `leader`。因为如果允许，这样做固然可以让 Kafka 继续提供服务给 `clients`，但会造成消息数据的丢失，而在一般的用户使用场景中，数据不丢失是基本的业务需求，因此设置此参数为 `false` 显得很有必要。事实上，Kafka 社区在 1.0.0 版本才正式将该参数默认值调整为 `false`，这表明社区在高可用性与数据完整性之间选择了后者。
- `delete.topic.enable`——是否允许 Kafka 删除 `topic`。默认情况下，Kafka 集群允许用户删除 `topic` 及其数据。这样当用户发起删除 `topic` 操作时，`broker` 端会执行 `topic` 删除逻辑。在实际生产环境中我们发现允许 Kafka 删除 `topic` 其实是一个很方便的功能，再加上自 Kafka 0.9.0.0 新增的 ACL 权限特性，以往对于误操作和恶意操作的担心完全消失了，因此设置该参数为 `true` 是推荐的做法。
- `log.retention.{hours|minutes|ms}`——这组参数控制了消息数据的留存时间，它们是“三兄弟”。如果同时设置，优先选取 `ms` 的设置，`minutes` 次之，`hours` 最后。有了这 3 个参数，用户可以很方便地在 3 个时间维度上设置日志的留存时间。默认的留存时间是 7 天，即 Kafka 只会保存最近 7 天的数据，并自动删除 7 天前的数据。当前较新版本的 Kafka 会根据消息的时间戳信息进行留存与否的判断。对于没有时间戳的老版本消息格式，Kafka 会根据日志文件的最近修改时间（`last modified time`）进行判断。可以说，这组参数定义的是时间维度上的留存策略。实际线上环境中，需要根据用户的业务需求进行设置。保存消息很长时间的业务通常都需要设置一个较大的值。
- `log.retention.bytes`——如果说上面那组参数定义了时间维度上的留存策略，那么这个参数便定义了空间维度上的留存策略，即它控制着 Kafka 集群需要为每个消息日志保存多大的数据。对于大小超过该参数的分区日志而言，Kafka 会自动清理该分区的过期

日志段文件。该参数默认值是-1，表示 Kafka 永远不会根据消息日志文件总大小来删除日志。和上面的参数一样，生产环境中需要根据实际业务场景设置该参数的值。

- **min.insync.replicas**——该参数其实是与 producer 端的 **acks** 参数配合使用的。关于 **acks** 含义的介绍，我们留到第 4 章中详细展开。这里只需要了解 **acks=-1** 表示 producer 端寻求最高等级的持久化保证，而 **min.insync.replicas** 也只有在 **acks=-1** 时才有意义。它指定了 broker 端必须成功响应 clients 消息发送的最少副本数。假如 broker 端无法满足该条件，则 clients 的消息发送并不会被视为成功。它与 **acks** 配合使用可以令 Kafka 集群达成最高等级的消息持久化。在实际使用中如果用户非常在意被发送的消息是否真的成功写入了所有副本，那么推荐将参数设置为副本数-1。举一个例子，假设某个 topic 的每个分区的副本数是 3，那么推荐设置该参数为 2，这样我们就能够容忍一台 broker 宕机而不影响服务；若设置参数为 3，那么只要任何一台 broker 宕机，整个 Kafka 集群将无法继续提供服务。因此用户需要在高可用和数据一致性之间取得平衡。
- **num.network.threads**——一个非常重要的参数。它控制了一个 broker 在后台用于处理网络请求的线程数，默认是 3。通常情况下，broker 启动时会创建多个线程处理来自其他 broker 和 clients 发送过来的各种请求。注意，这里的“处理”其实只是负责转发请求，它会将接收到的请求转发到后面的处理线程中。在真实的环境中，用户需要不断地监控 **NetworkProcessorAvgIdlePercent** JMX 指标。如果该指标持续低于 0.3，笔者建议适当增加该参数的值。在第 8 章中我们将会详细探讨各种 JMX 监控。
- **num.io.threads**——这个参数就是控制 broker 端实际处理网络请求的线程数，默认值是 8，即 Kafka broker 默认创建 8 个线程以轮询方式不停地监听转发过来的网络请求并进行实时处理。Kafka 同样也为请求处理提供了一个 JMX 监控指标 **RequestHandlerAvgIdlePercent**。如果发现该指标持续低于 0.3，则可以考虑适当增加该参数的值。
- **message.max.bytes**——Kafka broker 能够接收的最大消息大小，默认是 977KB，还不到 1MB，可见是非常小的。在实际使用场景中，突破 1MB 大小的消息十分常见，因此用户有必要综合考虑 Kafka 集群可能处理的最大消息尺寸并设置该参数值。

上面这些参数是笔者认为最重要的 broker 端参数，但不可否认的是，Kafka broker 端的参数远远不止这些。笔者粗略数了一下，以 Kafka 1.0.0 版本为例，broker 端的参数有 167 个之多。显然我们不可能涵盖所有的参数，其他的参数含义以及使用方法详见 Kafka 官网 <https://kafka.apache.org/documentation/#brokerconfigs>。

3.5.2 topic 级别参数

除 broker 端参数之外，Kafka 还提供了一些 topic 级别的参数供用户使用。所谓的 topic 级

别，是指覆盖 broker 端全局参数。每个不同的 topic 都可以设置自己的参数值。举一个例子来说，上面提到的日志留存时间。显然，在实际使用中，在全局设置一个通用的留存时间并不方便，因为每个业务的 topic 可能有不同的留存策略。如果只能设置全局参数，那么势必要取所有业务留存时间的最大值作为全局参数值，这样必然会造成空间的浪费。因此 Kafka 提供了很多 topic 级别的参数，常见的包括如下几个。

- `delete.retention.ms`——每个 topic 可以设置自己的日志留存时间以覆盖全局默认值。
- `max.message.bytes`——覆盖全局的 `message.max.bytes`，即为每个 topic 指定不同的最大消息尺寸。
- `retention.bytes`——覆盖全局的 `log.retention.bytes`，每个 topic 设置不同的日志留存尺寸。

关于如何设置 topic 级别的参数，我们会在第 7 章中详细介绍。

3.5.3 GC 参数

Kafka broker 端代码虽然是用 Scala 语言编写的，但终归要编译为.class 文件在 JVM 上运行。既然是 JVM 上面的应用，垃圾回收（garbage collection，GC）参数的设置就显得非常重要。

对于 Java 版本而言，笔者这边极其不推荐使用 Java 6 及以前的版本作为 Kafka 的 JVM 运行环境。Java 6 版本太过陈旧，如果有条件的话请马上升级到 Java 7 版本或以上。事实上就连 Java 7 也已经快 2 年不更新了，甚至 Kafka 社区目前在讨论未来版本中移除对于 Java 7 的支持。

不过若用户使用 Java 7，那么在选择 GC 收集器时可以根据以下法则进行确认。

- 如果用户机器上的 CPU 资源非常充裕，那么推荐使用 CMS 收集器。这样可以充分利用多 CPU 执行并发垃圾收集。启用方法为 `-XX:+UseCurrentMarkSweepGC`。
- 相反地，则使用吞吐量收集器，即所谓的 `throughput collector`。这样不会挤占紧张的 CPU 资源，使 Kafka broker 达到最大的吞吐量。启用方法为 `-XX:+UseParallelGC`。

若用户使用的是 Java 8——这是推荐的版本。实际上如果用户在 Kafka 官网上下载使用 Scala 2.12 编译的 Kafka 二进制压缩包，那么就必须安装并使用 Java 8——推荐使用 G1 垃圾收集器。根据笔者的实际使用经验，在没有任何调优的情况下，G1 收集器本身会比 CMS 表现出更好的性能，主要体现在 Full GC 的次数更少、需要微调的参数更少等方面。因此推荐用户始终使用 G1 收集器，不论是在 broker 端还是在 clients 端。

除此之外，我们还需要打开 GC 日志的监控，并实时确保不会出现“G1HR #StartFullGC”。至于 G1 的其他参数，可以根据实际使用情况酌情考虑做微小调整。

3.5.4 JVM 参数

之前说过，Kafka 推荐用户使用最新版本的 JDK——当前最新的 Oracle JDK 版本是 1.8.0_162。另外鉴于 Kafka broker 主要使用的是堆外内存，即大量使用操作系统的页缓存，因此其实并不需要为 JVM 分配太多的内存。在实际使用中，通常为 broker 设置不超过 6GB 的堆空间。以下就是一份典型的生产环境中的 JVM 参数列表：

```
-Xmx6g -Xms6g -XX:MetaspaceSize=96m -XX:+UseG1GC -XX:MaxGCPauseMillis=20  
-XX:InitiatingHeapOccupancyPercent=35 -XX:G1HeapRegionSize=16M -XX:MinMe-  
taspaceFreeRatio=50 -XX:MaxMetaspaceFreeRatio=80
```

3.5.5 OS 参数

前面提到过 Kafka 支持很多平台，但到目前为止被广泛使用并已被证明表现良好的平台，依然是 Linux 平台。目前 Kafka 社区在 Windows 平台上已经发现了一些特有的问题，而且在 Windows 平台上的工具支持也不像 Linux 上面那样丰富，因此笔者推荐应该将生产环境部署在 Linux 平台上。

通常情况下，Kafka 并不需要太多的 OS 级别的参数调优，但依然有一些 OS 参数是必须要调整的。

- **文件描述符限制：**Kafka 会频繁地创建并修改文件系统中的文件，这包括消息的日志文件、索引文件及各种元数据管理文件等。因此如果一个 broker 上面有很多 topic 的分区，那么这个 broker 势必就需要打开很多个文件——大致数量约等于分区数 $\times$ （分区总大小/日志段大小） $\times$ 3。举一个例子，假设 broker 上保存了 50 个分区，每个分区平均尺寸是 10GB，每个日志段大小是 1GB，那么这个 broker 需要维护 1500 个左右的文件描述符。因此在实际使用场景中最好首先增大进程能够打开的最大文件描述符上限，比如设置一个很大的值，如 100000。具体设置方法为 `ulimit -n 100000`。
- **Socket 缓冲区大小：**这里指的是 OS 级别的 Socket 缓冲区大小，而非 Kafka 自己提供的 Socket 缓冲区参数。事实上，Kafka 自己的参数将其设置为 64KB，这对于普通的内网环境而言通常是足够的，因为内网环境下往返时间（round-trip time, RTT）一般都很低，不会产生过多的数据堆积在 Socket 缓冲区中，但对于那些跨地区的数据传输而言，仅仅增加 Kafka 参数就不够了，因为前者也受限于 OS 级别的设置。因此如果是做远距离的数据传输，那么建议将 OS 级别的 Socket 缓冲区调大，比如增加到 128KB，甚至更大。
- **最好使用 Ext4 或 XFS 文件系统：**其实 Kafka 操作的都是普通文件，并没有依赖于特定的文件系统，但依然推荐使用 Ext4 或 XFS 文件系统，特别是 XFS 通常都有着更好

的性能。这种性能的提升主要影响的是 Kafka 的写入性能。根据官网的测试报告，使用 XFS 的写入时间大约是 160 毫秒，而使用 Ext4 大约是 250 毫秒。因此生产环境中最好使用 XFS 文件系统。

- **关闭 swap**：其实这是很多使用磁盘的应用程序的常规调优手段，具体命令为 `sysctl vm.swappiness = <一个较小的数>`，即大幅度降低对 swap 空间的使用，以免极大地拉低性能。后面的章节中会详细讨论为何不显式设置该值为 0。
- **设置更长的 flush 时间**：我们知道 Kafka 依赖 OS 页缓存的“刷盘”功能实现消息真正写入物理磁盘，默认的刷盘间隔是 5 秒。通常情况下，这个间隔太短了，适当增加该值可以在很大程度上提升 OS 物理写入操作的性能。LinkedIn 公司自己就将该值设置为 2 分钟以增加整体的物理写入吞吐量。

3.6 本章小结

本章我们着重讲述了如何搭建 Kafka 生产环境以及设置 Kafka 集群需要考虑的各方面参数，并且针对如何评估 Kafka 线上集群分别从各个方面进行了探讨。

相信各位读者在阅读完本章之后已经可以独立地根据自身业务的需要搭建起 Kafka 生产集群。

第 4 章

producer 开发

不论 Kafka 如何演变，万变不离其宗的是，它一定有一些外部的生产者（producer）应用程序给自己发送消息，然后还有一些外部的消费者（consumer）应用程序读取 producer 发送的消息。

本章着重讨论 Kafka 的 producer 设计以及基于 Java 版本 producer 的开发与使用。

学习本章，你将了解到以下内容。

- Kafka producer 概要。
- Kafka producer 简要设计。
- Java 版本 producer 及 API。

4.1 producer 概览

简单来说，Kafka producer 就是负责向 Kafka 写入数据的应用程序。在第 1 章中我们谈到过 Kafka 有多种使用场景，但不管是哪种使用场景，producer 都是必要的组件，否则没有给 Kafka 生产消息的程序，后面的一切也都无从谈起。

自 0.9.0.0 版本起，Apache Kafka 发布了 Java 版本的 producer 供用户使用，但作为一个比较完善的生态系统，Kafka 必然要支持多种语言的 producer，这其中比较著名的当属 C/C++ 平台上的 producer 库 librdkafka（准确来说，这些库也同时包含了对 consumer 的支持，统称它们是 clients 似乎更加合理一些），而像 Python、Go 或 .NET 这种主流的编程语言也有对应的 producer 库。当前 Apache Kafka 支持的第三方 clients 库的完整列表如下。

- C/C++
- Python
- Golang
- Erlang
- .NET
- Clojure
- Ruby
- Node.js
- Proxy (HTTP REST 等)
- Perl
- stdin/stdout
- PHP
- Rust
- Alternative Java
- Storm
- Scala DSL
-

值得注意的是，上面这些第三方库基本上都是由非 Apache Kafka 社区的人维护的，其中一些比较大的库已经由 Confluent 公司的人参与研发和维护（前面提到过的 Confluent 公司发布的企业级产品包含了 librdkafka 库支持 C++等平台），但如果用户下载的是 Apache Kafka，默认是不包含这些库的，需要额外单独下载对应的库。关于这些第三方库的详细信息以及下载地址，请访问 <https://cwiki.apache.org/confluence/display/KAFKA/Clients>。

另外，Apache Kafka 还封装了一套二进制通信协议，用于对外提供各种各样的服务。对于 producer 而言，用户几乎可以直接使用任意编程语言按照该协议的格式进行编程，从而实现向 Kafka 发送消息。实际上内置的 Java 版本 producer 和上面列出的所有第三方库在底层都是相同的实现原理，只是在易用性和性能方面有所差别而已。这组协议本质上为不同的协议类型分别定义了专属的紧凑二进制字节数组格式，然后通过 Socket 发送给合适的 broker，之后等待 broker 处理完成后返还响应（response）给 producer。这样设计的好处在于具有良好的统一性——即所有的协议类型都是统一格式的，并且由于是自定义的二进制格式，这套协议并不依赖任何外部序列化框架，从而显得非常轻量级，而且也有很好的扩展性。

第6章将详细讨论这套二进制通信协议的设计与使用。在这里只需要知道 producer 底层是由它们实现的即可。

Kafka producer 在设计上要比 consumer 简单一些，因为它不涉及复杂的组管理操作，即每个 producer 都是独立进行工作的，与其他 producer 实例之间没有关联，因此它受到的牵绊自然也要少得多，实现起来也要简单得多。目前 producer 的首要功能就是向某个 topic 的某个分区发送一条消息，所以它首先需要确认到底要向 topic 的哪个分区写入消息——这就是分区器（partitioner）要做的事情。Kafka producer 提供了一个默认的分区器。对于每条待发送的消息而言，如果该消息指定了 key，那么该 partitioner 会根据 key 的哈希值来选择目标分区；若这条消息没有指定 key，则 partitioner 使用轮询的方式确认目标分区——这样可以最大限度地确保消息在所有分区上的均匀性。当然 producer 的 API 赋予了用户自行指定目标分区的权力，即用户可以在消息发送时跳过 partitioner 直接指定要发送到的分区。

另外，producer 也允许用户实现自定义的分区策略而非使用默认的 partitioner，这样用户可以很灵活地根据自身的业务需求确定不同的分区策略。后面章节中会详细讨论如何自定义分区策略。

有了 partitioner 的帮助，我们就可以确信具有相同 key 的所有消息都会被路由到相同的分区中。这有助于实现一些特定的业务需求，比如可以利用局部性原理，将某些 producer 发送的消息固定地发送到相同机架上的分区从而减少网络传输的开销等。当然了，如前所述，如果没有指定 key，那么所有消息会被均匀地发送到所有分区，而这通常也是最合理的分区策略。

在确认了目标分区后，producer 要做的第二件事情就是要寻找这个分区对应的 leader，也就是该分区 leader 副本所在的 Kafka broker。前面章节中提到了每个 topic 分区都由若干个副本组成，其中的一个副本充当 leader 的角色，也只有 leader 才能够响应 clients 发送过来的请求，而剩下的副本中有一部分副本会与 leader 副本保持同步，即所谓的 ISR。因此在发送消息时，producer 也就有了多种选择来实现消息发送。比如不等待任何副本的响应便返回成功，或者只是等待 leader 副本响应写入操作之后再返回成功等。不同的选择也有着不同的优缺点，我们会在后续章节中讨论如何选择不同的策略。

鉴于目前 Java 版本 producer 使用最为广泛且是 Kafka 社区官方提供的，本章将结合 Java 版本 producer 来探讨 Kafka producer 的设计与使用。从本质上说，Java 版本 producer 的工作原理如图 4.1 所示。

在图 4.1 中，producer 首先使用一个线程（用户主线程，也就是用户启动 producer 的线程）将待发送的消息封装进一个 ProducerRecord 类实例，然后将其序列化之后发送给 partitioner，再由后者确定了目标分区后一同发送到位于 producer 程序中一块内存缓冲区中。而 producer 的另一个工作线程（I/O 发送线程，也称 Sender 线程）则负责实时地从该缓冲区中提取出准备就绪的消息封装进一个批次（batch），统一发送给对应的 broker。整个 producer 的工作流程大概就是这样的。

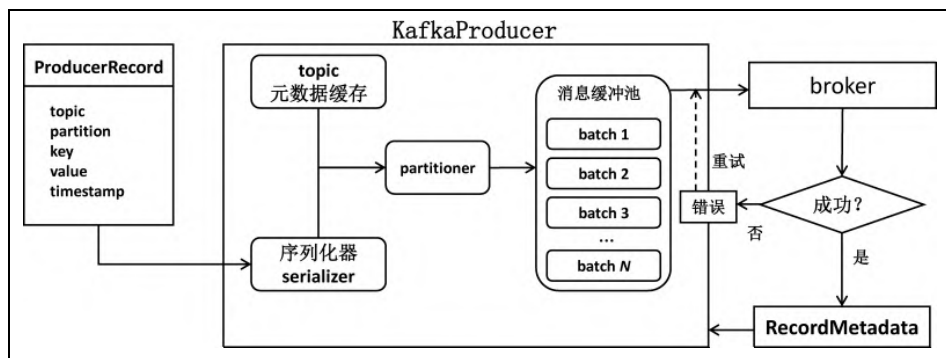


图 4.1 Java 版本 producer 工作流程

4.2 构造 producer

在初步了解了 Kafka producer（特别是 Java 版本 producer）之后，下面来看看如何使用程序 API 来构造一个 producer。我们仍然以 Java 版本 producer 的 API 为例。其他第三方库 API 的使用说明请参考对应库的官方文档。

4.2.1 producer 程序实例

首先，下面给出了一份可运行的 producer 程序代码清单。这份代码实现了最简单的功能：构造一条消息，然后发送给 Kafka。在运行这个 producer 程序之前，要保证启动一个最小规模的 Kafka 单机或集群环境。Kafka 运行环境搭建方法请参考第 3 章。

```

package com.huxi.kafkaapi;

import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerRecord;

import java.util.Properties;

public class ProducerTest {
    public static void main(String[] args) throws Exception {
        Properties props = new Properties();
        props.put("bootstrap.servers", "localhost:9092"); // 必须指定
        props.put("key.serializer", "org.apache.kafka.common.serialization.
StringSerializer"); // 必须指定
        props.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer"); // 必须指定
    }
}
  
```

```
        props.put("acks", "-1");
        props.put("retries", 3);
        props.put("batch.size", 323840);
        props.put("linger.ms", 10);
        props.put("buffer.memory", 33554432);
        props.put("max.block.ms", 3000);

        Producer<String, String> producer = new KafkaProducer<>(props);
        for(int i = 0; i < 100; i++)
            producer.send(new ProducerRecord<>("my-topic",
Integer.toString(i), Integer.toString(i)));

        producer.close();
    }
}
```

构造一个 `producer` 实例大致需要以下 5 个步骤。

(1) 构造一个 `java.util.Properties` 对象，然后至少指定 `bootstrap.servers`、`key.serializer` 和 `value.serializer` 这 3 个属性。在上面的代码清单中这 3 个属性后面都追加了注释，表明这是必须要指定的参数，它们没有默认值。

(2) 使用上一步中创建的 `Properties` 实例构造 `KafkaProducer` 对象。

(3) 构造待发送的消息对象 `ProducerRecord`，指定消息要被发送到的 `topic`、分区以及对应的 `key` 和 `value`。注意，分区和 `key` 信息可以不用指定，由 `Kafka` 自行确定目标分区。

(4) 调用 `KafkaProducer` 的 `send` 方法发送消息。

(5) 关闭 `KafkaProducer`。

1. 构造 `Properties` 对象

下面将详细展开每一步要做的事情。首先要构造一个 `Properties` 对象，在这一步中有 3 个参数或属性是必须要指定的。如果我们翻开 `Kafka` 官网中 `producer` 的参数列表（详见 <https://kafka.apache.org/documentation/#producerconfigs>）会发现这 3 个参数是没有默认值的。它们分别如下。

`bootstrap.servers`

该参数指定了一组 `host:port` 对，用于创建向 `Kafka broker` 服务器的连接，比如 `k1:9092`，`k2:9092`，`k3:9092`。上面的代码清单中指定了 `localhost:9092`，`producer` 使用时需要替换成实际的 `broker` 列表。如果 `Kafka` 集群中机器数很多，那么只需要指定部分 `broker` 即可，不需要列出所有的机器。因为不管指定几台机器，`producer` 都会通过该参数找到并发现集群中所有的 `broker`。

为该参数指定多台机器只是为了故障转移使用。这样即使某一台 broker 挂掉了，producer 重启后依然可以通过该参数指定的其他 broker 连入 Kafka 集群。

另外，如果 broker 端没有显式配置 listeners 使用 IP 地址，那么最好将该参数也配置成主机名，而不是 IP 地址。因为 Kafka 内部使用的就是 FQDN（Fully Qualified Domain Name）。

key.serializer

被发送到 broker 端的任何消息的格式都必须是字节数组，因此消息的各个组件必须首先做序列化，然后才能发送到 broker。该参数就是为消息的 key 做序列化之用的。这个参数指定的是实现了 `org.apache.kafka.common.serialization.Serializer` 接口的类的全限定名称。Kafka 为大部分的初始类型（primitive type）默认提供了现成的序列化器。上面的代码清单中使用了 `org.apache.kafka.common.serialization.StringSerializer`，该类会将一个字符串类型转换成字节数组。这个参数也揭示了一个事实，那就是用户可以自定义序列化器，只要实现 `Serializer` 接口即可。

需要注意的是，即使 producer 程序在发送消息时不指定 key，这个参数也是必须要设置的，否则程序会抛出 `ConfigException` 异常，提示“key.serializer”参数无默认值，必须要配置。

value.serializer

和 key.serializer 类似，只是它被用来对消息体（即消息 value）部分做序列化，将消息 value 部分转换成字节数组。上面的代码清单中该参数指定了与 key.serializer 相同的值，即都使用 `StringSerializer`。当然了，value.serializer 也可以设置成与 key.serializer 不同的值。

一定要注意的是，这两个参数都必须是全限定类名。只使用单独的类名是不行的，比如只是指定 `StringSerializer` 是不够的，必须是 `org.apache.kafka.common.serialization.Serializer` 这样的形式。这一规定对于自定义序列化也是适用的。

2. 构造 KafkaProducer 对象

设置了这 3 个属性之后，下面就要构造 `KafkaProducer` 对象了。`KafkaProducer` 是 producer 的主入口，所有的功能基本上都是由 `KafkaProducer` 来提供的。创建 `KafkaProducer` 实例很简单，只需要下面一句命令即可：

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

创建 producer 时也可以同时指定 key 和 value 的序列化类，比如这样：

```
Serializer<String> keySerializer = new StringSerializer();
Serializer<String> valueSerializer = new StringSerializer();
Producer<String, String> producer = new KafkaProducer<>(props, keySerializer,
valueSerializer);
```

如果采用这样的方式创建 producer，那么就不需要显式地在 Properties 中指定 key 和 value 序列化类了。

3. 构造 ProducerRecord 对象

构造好 KafkaProducer 实例后，下一步就是构造消息实例。Java 版本 producer 使用 ProducerRecord 类来表示每条消息。创建 ProducerRecord 也很简单，最简单的形式就是指定 topic 和 value，如下：

```
new ProducerRecord<>("my-topic", Integer.toString(i))
```

当然 ProducerRecord 还支持指定更多的消息信息，比如可以控制该消息直接被发往的分区以及消息的时间戳。具体的 API 格式请参见 <https://kafka.apache.org/10/javadoc/index.html?org/apache/kafka/clients/producer/KafkaProducer.html>。

不过这里要注意的是，一定要谨慎指定时间戳，因为在目前的 Kafka 设计中，时间戳索引文件中的索引项都是严格按照时间戳顺序排列的，所以如果在 producer 端随意指定时间戳，会导致该消息的时间序列混乱，这样在使用根据时间戳查询位移的功能时则不会找到这条消息。，同时 Kafka 的消息留存策略也会受到影响，因此最好还是让 Kafka 自行来指定时间戳比较稳妥。

4. 发送消息

Kafka producer 发送消息的主方法是 send 方法。虽然 send 方法只有两个简单的方法签名，但其实 producer 在底层完全地实现了异步化发送，并且通过 Java 提供的 Future 同时实现了同步发送和异步发送+回调（Callback）两种发送方式。实际上，本节开始的代码清单中的调用方式实现了第 3 种发送方式——这种方式有一个专有名词 fire and forget，即发送之后便不再理会发送结果。这种方式在实际中是不被推荐使用的，因为对于发送结果 producer 程序完全不知，所以在真实使用场景中，同步和异步的发送方式还是最常见的两种方式。

异步发送

实际上所有的写入操作默认都是异步的。Java 版本 producer 的 send 方法会返回一个 Java Future 对象供用户稍后获取发送结果，这就是所谓的回调机制。send 方法提供了回调类参数来实现异步发送以及对发送结果的响应，具体代码如下：

```
producer.send(record, new Callback() {  
    @Override  
    public void onCompletion(RecordMetadata metadata, Exception exception) {  
        if (exception == null) {  
            // 消息发送成功  
        } else {  
            // 执行错误处理逻辑  
        }  
    }  
})
```

```

    }
}
});

```

上面代码中的 `Callback` 就是发送消息后的回调类，实现方法是 `onCompletion`。该方法的两个输入参数 `metadata` 和 `exception` 不会同时非空，也就是说至少有一个是 `null`。当消息发送成功时，`exception` 是 `null`；反之，若消息发送失败，`metadata` 就是 `null`。因此在写 `producer` 程序时，最好写 `if` 语句进行判断。

另外，上面的 `Callback` 实际上是一个 `Java` 接口，用户可以创建自定义的 `Callback` 实现类来处理消息发送后的逻辑，只要该具体类实现 `org.apache.kafka.clients.producer.Callback` 接口即可。

同步发送

同步发送和异步发送其实就是通过 `Java` 的 `Future` 来区分的，调用 `Future.get()` 无限等待结果返回，即实现同步发送的效果，具体代码如下：

```

ProducerRecord<String, String> record = new ProducerRecord<>("test", Integer.
toString(i));
producer.send(record).get();

```

使用 `Future.get` 会一直等待下去直至 `Kafka broker` 将发送结果返回给 `producer` 程序。当结果从 `broker` 处返回时 `get` 方法要么返回发送结果要么抛出异常交由 `producer` 自行处理。如果没有错误，`get` 将返回对应的 `RecordMetadata` 实例（包含了已发送消息的所有元数据信息），包括消息发送的 `topic`、分区以及该消息在对应分区的位移信息。

不管是同步发送还是异步发送，发送都有可能失败，导致返回异常错误。当前 `Kafka` 的错误类型包含了两类：可重试异常和不可重试异常。常见的可重试异常如下。

- **LeaderNotAvailableException**：分区的 `leader` 副本不可用，这通常出现在 `leader` 换届选举期间，因此通常是瞬时的异常，重试之后可以自行恢复。
- **NotControllerException**：`controller` 当前不可用（`controller` 是 `Kafka` 集群中非常重要的角色，我们会在第 6 章中详细讨论 `controller` 的设计与实现）。这通常表明 `controller` 在经历新一轮的选举，这也是可以通过重试机制自行恢复的。
- **NetworkException**：网络瞬时故障导致的异常，可重试。

对于这种可重试的异常，如果在 `producer` 程序中配置了重试次数，那么只要在规定重试次数内自行恢复了，便不会出现在 `onCompletion` 的 `exception` 中。不过若超过了重试次数仍没有成功，则仍然会被封装进 `exception` 中。此时就需要 `producer` 程序自行处理这种异常。

所有可重试异常都继承自 `org.apache.kafka.common.errors.RetriableException` 抽象类。理论

上讲所有未继承自 `RetriableException` 类的其他异常都属于不可重试异常，这类异常通常都表明了一些非常严重或 Kafka 无法处理的问题，与 `producer` 相关的如下。

- `RecordTooLargeException`：发送的消息尺寸过大，超过了规定的大小上限。显然这种异常无论如何重试都是无法成功的。
- `SerializationException`：序列化失败异常，这也是无法恢复的。
- `KafkaException`：其他类型的异常。

所有这些不可重试异常一旦被捕获都会被封装进 `Future` 的计算结果并返回给 `producer` 程序，用户需要自行处理这些异常。由于不可重试异常和可重试异常在 `producer` 程序端可能有不同的处理逻辑，因此可以使用下面的代码进行区分：

```
producer.send(record, new Callback() {  
    @Override  
    public void onCompletion(RecordMetadata metadata, Exception exception) {  
        if (exception == null) {  
            // 消息发送成功  
        } else {  
            if (exception instanceof RetriableException) {  
                // 处理可重试瞬时异常  
            } else {  
                // 处理不可重试异常  
            }  
        }  
    }  
});
```

5. 关闭 producer

`producer` 程序结束时一定要关闭 `producer`！这一点怎么强调都不为过。毕竟 `producer` 程序运行时占用了系统资源（比如创建了额外的线程、申请了很多内存以及创建了多个 `Socket` 连接等），因此必须要显式地调用 `KafkaProducer.close` 方法关闭 `producer`。不管发送是成功还是失败，只要 `producer` 程序完成了既定的工作，就应该被关闭。

如果是调用普通的无参数 `close` 方法，`producer` 会被允许先处理完之前的发送请求后再关闭，即所谓的“优雅”关闭退出（`graceful shutdown`）；同时，`KafkaProducer` 还提供了一个带超时参数的 `close` 方法 `close(timeout)`。如果调用此方法，`producer` 会等待 `timeout` 时间来完成所有处理中的请求，然后强行退出。这就是说，若 `timeout` 超时，则 `producer` 会强制结束，并立即丢弃所有未发送以及未应答的发送请求。在某种程度上，这会给用户一种错觉：仿佛 `producer` 端的程序丢失了要发送的消息。因此在实际场景中一定要谨慎使用带超时的 `close` 方法。

4.2.2 producer 主要参数

除了前面的 3 个参数：bootstrap.servers、key.serializer 和 value.serializer 之外，Java 版本 producer 还提供了很多其他很重要的参数。本章将选取其中一些重要的参数进行详细讨论。详细的参数列表以及含义和默认值可以访问 <https://kafka.apache.org/documentation/#producer-configs>。

acks

acks 参数用于控制 producer 生产消息的持久性（durability）。对于 producer 而言，Kafka 在乎的是“已提交”消息的持久性。一旦消息被成功提交，那么只要有任何一个保存了该消息的副本“存活”，这条消息就会被视为“不会丢失的”。经常碰到有用户抱怨 Kafka 的 producer 会丢消息，其实这里混淆了一个概念，即那些所谓的“已丢失”的消息其实并没有被成功写入 Kafka。换句话说，它们并没有被成功提交，因此 Kafka 对这些消息的持久性不做任何保障——当然，producer API 确实提供了回调机制供用户处理发送失败的情况。

具体来说，当 producer 发送一条消息给 Kafka 集群时，这条消息会被发送到指定 topic 分区 leader 所在的 broker 上，producer 等待从该 leader broker 返回消息的写入结果（当然并不是无限等待，是有超时时间的）以确定消息被成功提交。这一切完成后 producer 可以继续发送新的消息。Kafka 能够保证的是 consumer 永远不会读取到尚未提交完成的消息——这和关系型数据库很类似，即在大部分情况下，某个事务的 SQL 查询都不会看到另一个事务中尚未提交的数据。

显然，leader broker 何时发送写入结果返还给 producer 就是一个需要仔细考虑的问题了，它也会直接影响消息的持久性甚至是 producer 端的吞吐量：producer 端越快地接收到 leader broker 响应，它就能越快地发送下一条消息，即吞吐量也就越大。producer 端的 acks 参数就是用来控制做这件事情的。acks 指定了在给 producer 发送响应前，leader broker 必须要确保已成功写入该消息的副本数。当前 acks 有 3 个取值：0、1 和 all。

- acks = 0：设置成 0 表示 producer 完全不理睬 leader broker 端的处理结果。此时，producer 发送消息后立即开启下一条消息的发送，根本不等待 leader broker 端返回结果。由于不接收发送结果，因此在这种情况下 producer.send 的回调也就完全失去了作用，即用户无法通过回调机制感知任何发送过程中的失败，所以 acks=0 时 producer 并不保证消息会被成功发送。但凡事有利就有弊，由于不需要等待响应结果，通常这种设置下 producer 的吞吐量是最高的
- acks = all 或者-1：表示当发送消息时，leader broker 不仅会将消息写入本地日志，同时还会等待 ISR 中所有其他副本都成功写入它们各自的本地日志后，才发送响应结果给

producer。显然当设置 `acks=all` 时，只要 ISR 中至少有一个副本是处于“存活”状态的，那么这条消息就肯定不会丢失，因而可以达到最高的消息持久性，但通常这种设置下 producer 的吞吐量也是最低的。

- `acks = 1`：是 0 和 all 折中的方案，也是默认的参数值。producer 发送消息后 leader broker 仅将该消息写入本地日志，然后便发送响应结果给 producer，而无须等待 ISR 中其他副本写入该消息。那么此时只要该 leader broker 一直存活，Kafka 就能够保证这条消息不丢失。这实际上是一种折中方案，既可以达到适当的消息持久性，同时也保证了 producer 端的吞吐量。

总结一下，acks 参数控制 producer 实现不同程度的消息持久性，它有 3 个取值，对应的优缺点以使用场景如表 4.1 所示。

表 4.1 acks 参数取值说明

acks	producer 吞吐量	消息持久性	使用场景
0	最高	最差	① 完全不关心消息是否发送成功 ② 允许消息丢失（比如统计服务器日志等）
1	适中	适中	一般场景即可
all 或 -1	最差	最高	不能容忍消息丢失

在 producer 程序中设置 acks 非常简单，只需要在构造 `KafkaProducer` 的 `Properties` 对象中增加“acks”属性即可：

```

props.put("acks", "1");
// 或者
props.put(ProducerConfig.ACKS_CONFIG, "1");
    
```

值得注意的是，该参数的类型是字符串，因此必须要写成“1”而不是 1，否则程序会报错，提示你没有指定正确的参数类型。

buffer.memory

该参数指定了 producer 端用于缓存消息的缓冲区大小，单位是字节，默认值是 33554432，即 32MB。如前所述，由于采用了异步发送消息的设计架构，Java 版本 producer 启动时会首先创建一块内存缓冲区用于保存待发送的消息，然后由另一个专属线程负责从缓冲区中读取消息执行真正的发送。这部分内存空间的大小即是由 `buffer.memory` 参数指定的。若 producer 向缓冲区写消息的速度超过了专属 I/O 线程发送消息的速度，那么必然造成该缓冲区空间的不断增大。此时 producer 会停止手头的工作等待 I/O 线程追上来，若一段时间之后 I/O 线程还是无法追上 producer 的进度，那么 producer 就会抛出异常并期望用户介入进行处理。

虽说 producer 在工作过程中会用到很多部分的内存，但我们几乎可以认为该参数指定的内

存大小就是 producer 程序使用的内存大小。若 producer 程序要给很多分区发送消息，那么就需要仔细地设置这个参数以防止过小的内存缓冲区降低了 producer 程序整体的吞吐量。

和 acks 设置方法类似，可以使用下面的代码来设置 buffer.memory：

```
props.put("buffer.memory", 33554432);
// 或者
props.put(ProducerConfig.BUFFER_MEMORY_CONFIG, 33554432);
```

compression.type

compression.type 参数设置 producer 端是否压缩消息，默认值是 none，即不压缩消息。和任何系统相同的是，Kafka 的 producer 端引入压缩后可以显著地降低网络 I/O 传输开销从而提升整体吞吐量，但也会增加 producer 端机器的 CPU 开销。另外，如果 broker 端的压缩参数设置得与 producer 不同，broker 端在写入消息时也会额外使用 CPU 资源对消息进行对应的解压-重压缩操作。

目前 Kafka 支持 3 种压缩算法：GZIP、Snappy 和 LZ4。由于篇幅有限，我们在这里不对这 3 种压缩算法的优劣进行比较，不过根据实际使用经验来看 producer 结合 LZ4 的性能是最好的。由于 Kafka 源代码中某个关键设置的硬编码使得 Snappy 的表现远不如 LZ4，因此至少对于当前最新版本的 Kafka（1.0.0 版本）而言，若要使用压缩，compression.type 最好设置为 LZ4。令人感到振奋的是，Facebook 公司于 2016 年 8 月底正式开源了新一代的压缩算法 Zstandard。该算法有着超高的压缩率（根据其官网测试结果，Zstandard 的压缩率可达 2.8，Snappy 和 LZ4 分别是 2.091 和 2.101）以及优良的压缩/解压缩速度。Kafka 社区已经计划将在未来的版本中支持该压缩算法，相信在 Kafka 的后续版本中就可以使用 Zstandard 来压缩 producer 发送的消息了。

类似地，设置 compression.type 的方法如下：

```
props.put("compression.type", "lz4");
// 或者
props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "lz4");
```

retries

Kafka broker 在处理写入请求时可能因为瞬时的故障（比如瞬时的 leader 选举或者网络抖动）导致消息发送失败。这种故障通常都是可以自行恢复的，如果把这些错误封装进回调函数的异常中返还给 producer，producer 程序也并没有太多可以做的，只能简单地在回调函数中重新尝试发送消息。与其这样，还不如 producer 内部自动实现重试。因此 Java 版本 producer 在内部自动实现了重试，当然前提就是要设置 retries 参数。

该参数表示进行重试的次数，默认值是 0，表示不进行重试。在实际使用过程中，设置重

试可以很好地应对那些瞬时错误，因此推荐用户设置该参数为一个大于 0 的值。只不过在考虑 `retries` 的设置时，有两点需要着重注意。

- 重试可能造成消息的重复发送——比如由于瞬时的网络抖动使得 `broker` 端已成功写入消息但没有成功发送响应给 `producer`，因此 `producer` 会认为消息发送失败，从而开启重试机制。为了应对这一风险，Kafka 要求用户在 `consumer` 端必须执行去重处理。令人欣喜的是，社区已于 0.11.0.0 版本开始支持“精确一次”处理语义，从设计上避免了类似的问题。
- 重试可能造成消息的乱序——当前 `producer` 会将多个消息发送请求（默认是 5 个）缓存在内存中，如果由于某种原因发生了消息发送的重试，就可能造成消息流的乱序。为了避免乱序发生，Java 版本 `producer` 提供了 `max.in.flight.requests.per.connection` 参数。一旦用户将此参数设置成 1，`producer` 将确保某一时刻只能发送一个请求。

另外，`producer` 两次重试之间会停顿一段时间，以防止频繁地重试对系统带来冲击。这段时间是可以配置的，由参数 `retry.backoff.ms` 指定，默认是 100 毫秒。由于 `leader` “换届选举”是最常见的瞬时错误，笔者推荐用户通过测试来计算平均 `leader` 选举时间并根据该时间来设定 `retries` 和 `retry.backoff.ms` 的值。

同样地，设置 `retries` 的代码如下：

```
props.put("retries", 100);  
// 或者  
props.put(ProducerConfig.RETRIES_CONFIG, 100);
```

`batch.size`

`batch.size` 是 `producer` 最重要的参数之一！它对于调优 `producer` 吞吐量和延时性能指标都有着非常重要的作用。前面提到过，`producer` 会将发往同一分区的多条消息封装进一个 `batch` 中。当 `batch` 满了的时候，`producer` 会发送 `batch` 中的所有消息。不过，`producer` 并不总是等待 `batch` 满了才发送消息，很有可能当 `batch` 还有很多空闲空间时 `producer` 就发送该 `batch`。

显然，`batch` 的大小就显得非常重要。通常来说，一个小的 `batch` 中包含的消息数很少，因而一次发送请求能够写入的消息数也很少，所以 `producer` 的吞吐量会很低；但若一个 `batch` 非常之巨大，那么会给内存使用带来极大的压力，因为不管是否能够填满，`producer` 都会为该 `batch` 分配固定大小的内存。因此 `batch.size` 参数的设置其实是一种时间与空间权衡的体现。

`batch.size` 参数默认值是 16384，即 16KB。这其实是一个非常保守的数字。在实际使用过程中合理地增加该参数值，通常都会发现 `producer` 的吞吐量得到了相应的增加。

设置 `batch.size` 的方式和其他参数类似，如下面的代码：

```
props.put("batch.size", 1048576);
// 或者
props.put(ProducerConfig.BATCH_SIZE_CONFIG, 1048576);
```

linger.ms

上面说到 `batch.size` 时，我们提到了消息没有填满 `batch` 也可以被发送的情况。这是为什么呢？难道不是等 `batch` 满了再发送比较好吗？实际上这也是一种权衡，即吞吐量与延时之间的权衡。`linger.ms` 参数就是控制消息发送延时行为的。该参数默认值是 0，表示消息需要被立即发送，无须关心 `batch` 是否已被填满，大多数情况下这是合理的，毕竟我们总是希望消息被尽可能快地发送。不过这样做会拉低 `producer` 吞吐量，毕竟 `producer` 发送的每次请求中包含的消息数越多，`producer` 就越能将发送请求的开销摊薄到更多的消息上，从而提升吞吐量。

设置 `linger.ms` 的方式和其他参数类似，如下面的代码：

```
props.put("linger.ms", 100);
// 或者
props.put(ProducerConfig.LINGER_MS_CONFIG, 100);
```

max.request.size

官网中给出的解释是，该参数用于控制 `producer` 发送请求的大小。实际上该参数控制的是 `producer` 端能够发送的最大消息大小。由于请求有一些头部数据结构，因此包含一条消息的请求的大小要比消息本身大。不过姑且把它当作请求的最大尺寸是安全的。如果 `producer` 要发送尺寸很大的消息，那么这个参数就是要被设置的。默认的 1048576 字节太小了，通常无法满足企业级消息的大小要求。

设置 `max.request.size` 的方式和其他参数类似，如下面的代码：

```
props.put("max.request.size", 10485760);
// 或者
props.put(ProducerConfig.MAX_REQUEST_SIZE_CONFIG, 10485760);
```

request.timeout.ms

当 `producer` 发送请求给 `broker` 后，`broker` 需要在规定的时间范围内将处理结果返还给 `producer`。这段时间便是由该参数控制的，默认是 30 秒。这就是说，如果 `broker` 在 30 秒内都没有给 `producer` 发送响应，那么 `producer` 就会认为该请求超时了，并在回调函数中显式地抛出 `TimeoutException` 异常交由用户处理。

默认的 30 秒对于一般的情况而言是足够的，但如果 `producer` 发送的负载很大，超时的情况就很容易碰到，此时就应该适当调整该参数值。

设置 `request.timeout.ms` 的方式和其他参数类似，如下面的代码：

```
props.put("request.timeout.ms", 60000);
// 或者
props.put(ProducerConfig.REQUEST_TIMEOUT_MS_CONFIG, 60000);
```

4.3 消息分区机制

4.3.1 分区策略

Kafka producer 发送过程中一个很重要的步骤就是要确定将消息发送到指定的 topic 的哪个分区中。producer 提供了分区策略以及对应的分区器 (partitioner) 供用户使用。随 Kafka 发布的默认 partitioner 会尽力确保具有相同 key 的所有消息都会被发送到相同的分区上；若没有为消息指定 key，则该 partitioner 会选择轮询的方式来确保消息在 topic 的所有分区上均匀分配——注意，对于旧版本的 producer 而言，partitioner 不是这样实现的，它无法做到这种均匀分配。我们会在本章 4.9 节中讨论旧版本的 producer。

4.3.2 自定义分区机制

对于有 key 的消息而言，Java 版本 producer 自带的 partitioner 会根据 murmur2 算法计算消息 key 的哈希值，然后对总分区数求模得到消息要被发送到的目标分区号。但是有的时候用户可能想实现自己的分区策略，而这又是默认 partitioner 无法提供的，那么此时用户就可以使用 producer 提供的自定义分区策略了。

若要使用自定义分区机制，用户需要完成两件事情。

(1) 在 producer 程序中创建一个类，实现 org.apache.kafka.clients.producer.Partitioner 接口。主要分区逻辑在 Partitioner.partition 中实现。

(2) 在用于构造 KafkaProducer 的 Properties 对象中设置 partitioner.class 参数。

下面结合一个实例来说明如何指定自定义的分区策略。按照上面的步骤，第一步需要先创建一个自定义分区类并实现 Partitioner 接口。首先，看看这个接口的定义：

```
public interface Partitioner extends Configurable, Closeable {

    /**
     * 计算给定消息要被发送到哪个分区
     *
     * @param topic topic 名称
     * @param key 消息键值或 null
     * @param keyBytes 消息键值序列化字节数组或 null
     */
}
```

```

        * @param value 消息体或 null
        * @param valueBytes 消息体序列化字节数组或 null
        * @param cluster 集群元数据
        */
        public int partition(String topic, Object key, byte[] keyBytes,
Object value, byte[] valueBytes, Cluster cluster);

        /**
        * 关闭 partitioner
        */
        public void close();
    }

```

`partitioner` 接口的主要方法是 `partition` 方法，该方法接收消息所属的 `topic`、`key` 和 `value`，还有集群的元数据信息，一起来确定目标分区；而 `close` 方法是用于关闭 `partitioner` 的，主要是为了关闭那些创建 `partitioner` 时初始化的系统资源等。

现在举一个实际的例子来看看如何实现一个自定义的 `partitioner`。假设我们的消息中有一些消息是用于审计功能的，这类消息的 `key` 会被固定地分配一个字符串“`audit`”。我们想要让这类消息发送到 `topic` 的最后一个分区上，便于后续统一处理，而对于相同 `topic` 下的其他消息则采用随机发送的策略发送到其他分区上。那么现在就可以这样来实现自定义的分区策略，如下列代码所示：

```

public class AuditPartitioner implements Partitioner {

    private Random random;

    @Override
    public void configure(Map<String, ?> configs) {
        // 该方法实现必要资源的初始化工作
        random = new Random();
    }

    @Override
    public int partition(String topic, Object keyObj, byte[] keyBytes,
Object value, byte[] valueBytes, Cluster cluster) {
        String key = (String) keyObj;
        List<PartitionInfo> partitionInfoList = cluster.availablePartitionsForTopic
(topic);

        int partitionCount = partitionInfoList.size();
        int auditPartition = partitionCount - 1;
        return key == null || key.isEmpty() || !key.contains("audit")

```



```

        ? random.nextInt(partitionCount - 1) : auditPartition;
    }

    @Override
    public void close() {
        // 该方法实现必要资源的清理工作
    }
}

```

如上面的代码所示，在 `partition` 方法中首先对消息的 `key` 做了判断，判断它是否为空或者它是否是普通消息，如果是则随机发送到 `topic` 中除最后一个分区以外的其他分区中；若该消息属于 `audit` 消息，那么就固定地发送到 `topic` 的最后一个分区。值得注意的是，这个方法会调用 `Cluster.availablePartitionsForTopic` 从集群元数据中把属于该 `topic` 的所有分区信息都读取出来以供分区策略使用。

怎么样？实现一个自定义的分区策略还是很简单的吧。用户不仅可以根据 `key` 来指定策略，还可以根据方法传入的 `value` 信息做一些定制化分区策略。具体实现方法和上面的代码类似，这里就不赘述了。

好了，我们已经创建了一个自定义分区策略类，现在应该如何使用它呢？按照本章节开始时的步骤，需要执行第 2 步，指定 `partitioner.class`。具体方法是在构建 `KafkaProducer` 之前为 `Properties` 增加该属性，详见下面的代码：

```

props.put("partitioner.class", "com.xxx.yyy.producer.AuditPartitioner");
// 或者
props.put(ProducerConfig.PARTITIONER_CLASS_CONFIG,
"com.xxx.yyy.producer.AuditPartitioner");

```

`AuditPartitioner` 类就是我们之前创建的自定义分区类，但是在指定属性的时候必须要给出完整的类路径，不能只是单纯的类名。

下面测试一下这个自定义分区是否可用。首先在构建 `producer` 时指定自定义分区类的全限定名，然后构造 3 类消息分别是无 `key` 消息、非 `audit` 消息和 `audit` 消息，最后使用 `producer` 发送它们。部分代码如下：

```

String topic = "test-topic";
Producer<String, String> producer = new KafkaProducer<>(props);
ProducerRecord nonKeyRecord = new ProducerRecord(topic, "non-key record");
ProducerRecord auditRecord = new ProducerRecord(topic, "audit", "audit record");
ProducerRecord nonAuditRecord = new ProducerRecord(topic, "other", "non-audit record");
producer.send(nonKeyRecord).get();
producer.send(nonAuditRecord).get();

```

```
producer.send(auditRecord).get();
producer.send(nonKeyRecord).get();
producer.send(nonAuditRecord).get();
```

如上面的代码所示，我们发送一次 `audit` 消息，其他消息分别发送两次。现在使用 Kafka 提供的 `GetOffsetShell` 工具类帮助我们查询 `test-topic` 每个分区的信息数。查询结果如下：

```
> bin/kafka-run-class.sh kafka.tools.GetOffsetShell --broker-list
localhost:9092 --topic test-topic
test-topic:2:1
test-topic:1:2
test-topic:0:2
```

输出结果表明 `test-topic` 的分区 2 有 1 条消息，而分区 1 和分区 0 分别有 2 条消息。这与预期是一致的，即 `audit` 消息会被固定地发送到最后一个分区（分区 2）上，其他消息会以轮询的方式交替保存在分区 0 和 1 上。`AuditPartitioner` 设定的自定义分区策略生效了。

好了，至此我们就完整地实现了一个自定义的消息分区策略并成功地应用在了 Java 版本 producer 上。

4.4 消息序列化

4.4.1 默认序列化

在网络中发送数据都是以字节的方式，Kafka 也不例外。Apache Kafka 支持用户给 broker 发送各种类型的消息。它可以是一个字符串、一个整数、一个数组或是其他任意的对象类型。序列化器（serializer）负责在 producer 发送前将消息转换成字节数组；而与之相反，解序列化器（deserializer）则用于将 consumer 接收到的字节数组转换成相应的对象。`serializer` 和 `deserializer` 的关系如图 4.2 所示。本节中只讨论 `serializer`，`deserializer` 将在第 5 章中做详细讨论。

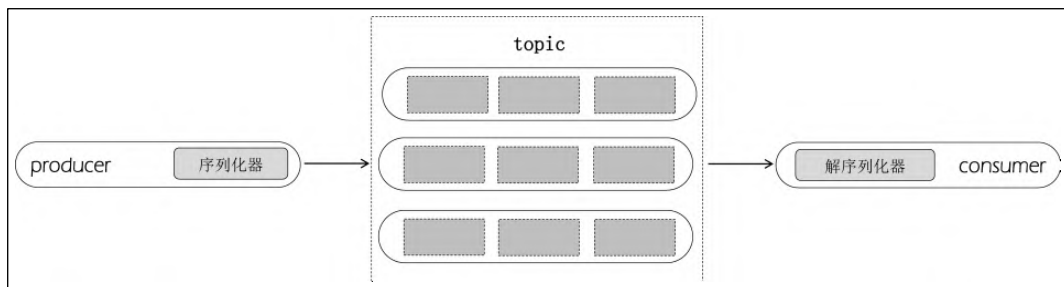


图 4.2 producer serializer 和 deserializer

Kafka 1.0.0 默认提供了十几种序列化器，其中常用的 serializer 如下。

- `ByteArraySerializer`: 本质上什么都不用做，因为已经是字节数组了。
- `ByteBufferSerializer`: 序列化 `ByteBuffer`。
- `BytesSerializer`: 序列化 Kafka 自定义的 `Bytes` 类。
- `DoubleSerializer`: 序列化 `Double` 类型。
- `IntegerSerializer`: 序列化 `Integer` 类型。
- `LongSerializer`: 序列化 `Long` 类型。
- `StringSerializer`: 序列化 `String` 类型。

由此可见，producer 已经初步建立了序列化机制来应对简单的消息类型，但若涉及复杂的类型（比如 Avro 或其他序列化框架），那么就需要用户自行定义 serializer。

producer 的序列化机制使用起来非常简单，只需要在构造 producer 时同时指定参数 `key.serializer` 和 `value.serializer` 的值即可，如下面的代码：

```
props.put("key.serializer", "org.apache.kafka.common.serialization.
IntegerSerializer");
props.put("value.serializer", "org.apache.kafka.common.serialization.
StringSerializer");
// 或者
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, "org.apache.kafka.
common.serialization.IntegerSerializer");
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, "org.apache.kafka.
common.serialization.StringSerializer");
```

由此可见，用户可以为消息的 `key` 和 `value` 指定不同类型的 `serializer`，只要与解序列类型分别保持一致就可以。

4.4.2 自定义序列化

和分区策略类似，Kafka 支持用户自定义消息序列化。若要编写一个自定义的 `serializer`，需要完成以下 3 件事情。

(1) 定义数据对象格式。

(2) 创建自定义序列化类，实现 `org.apache.kafka.common.serialization.Serializer` 接口，在 `serialize` 方法中实现序列化逻辑。

(3) 在用于构造 `KafkaProducer` 的 `Properties` 对象中设置 `key.serializer` 或 `value.serializer`——取决于为消息 `key` 还是 `value` 做自定义序列化。

下面结合一个实例来说明如何创建自定义的 `serializer`。首先定义待序列化的数据对象。本例中使用一个简单的 Java POJO 对象，如下面的代码所示：

```
public class User {

    private String firstName;
    private String lastName;
    private int age;
    private String address;

    public User(String firstName, String lastName, int age, String address) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
        this.address = address;
    }

    @Override
    public String toString() {
        return "User{" +
            "firstName='" + firstName + '\'' +
            ", lastName='" + lastName + '\'' +
            ", age=" + age +
            ", address='" + address + '\'' +
            '}';
    }
}
```

接下来创建 `serializer`。本例中使用了 `jackson-mapper-asl` 包的 `ObjectMapper` 帮助我们直接把对象转成字节数组。为了使用该类，你需要在 `producer` 工程中增加依赖：

Maven

```
<dependency>
  <groupId>org.codehaus.jackson</groupId>
  <artifactId>jackson-mapper-asl</artifactId>
  <version>1.9.13</version>
</dependency>
```

Gradle

```
compile group: 'org.codehaus.jackson', name: 'jackson-mapper-asl', version:
'1.9.13'
```

`UserSerializer` 代码如下：

```
public class UserSerializer implements Serializer {
```

```
private ObjectMapper objectMapper;

@Override
public void configure(Map configs, boolean isKey) {
    objectMapper = new ObjectMapper();
}

@Override
public byte[] serialize(String topic, Object data) {
    byte[] ret = null;
    try {
        ret = objectMapper.writeValueAsString(data).getBytes("utf-8");
    } catch (Exception e) {
        logger.warn("failed to serialize the object: {}", data, e);
    }
    return ret;
}

@Override
public void close() {
}
```

最后一步就是指定 `serializer`，然后构建消息进行发送，关键代码如下：

```
Properties props = new Properties();
props.put(...)
props.put("key.serializer", "org.apache.kafka.common.serialization.
StringSerializer");
props.put("value.serializer", "huxi.test.producer.UserSerializer");

String topic = "test-topic";
Producer<String, User> producer = new KafkaProducer<>(props);

// 构建 User 实例
User user = new User("XI", "HU", 33, "Beijing, China");
// 构建 ProducerRecord 实例，注意范型格式是<String, User>
ProducerRecord<String, User> record = new ProducerRecord<>(topic, user);
producer.send(record).get();
producer.close();
```

实际上，以上所有的代码都只实现了图 4.2 的左半部分，即自定义序列化发送消息。图 4.2 中的右半部分我们会留到第 5 章中实现——自定义解序列化将还原 `User` 对象供 `consumer` 端消费。

4.5 producer 拦截器

producer 拦截器（interceptor）是一个相当新的功能，它和 consumer 端 interceptor（将在第 5 章中介绍）是在 Kafka 0.10.0.0 版本中被引入的，主要用于实现 clients 端的定制化控制逻辑。

对于 producer 而言，interceptor 使得用户在消息发送前以及 producer 回调逻辑前有机会对消息做一些定制化需求，比如修改消息等。同时，producer 允许用户指定多个 interceptor 按序作用于同一条消息从而形成一个拦截链（interceptor chain）。intercetpor 的实现接口是 `org.apache.kafka.clients.producer.ProducerInterceptor`，其定义的方法如下。

- `onSend(ProducerRecord)`：该方法封装进 `KafkaProducer.send` 方法中，即它运行在用户主线程中。producer 确保在消息被序列化以计算分区前调用该方法。用户可以在该方法中对消息做任何操作，但最好保证不要修改消息所属的 topic 和分区，否则会影响目标分区的计算。
- `onAcknowledgement(RecordMetadata, Exception)`：该方法会在消息被应答之前或消息发送失败时调用，并且通常都是在 producer 回调逻辑触发之前。`onAcknowledgement` 运行在 producer 的 I/O 线程中，因此不要在该方法中放入很“重”的逻辑，否则会拖慢 producer 的消息发送效率。
- `close`：关闭 interceptor，主要用于执行一些资源清理工作。

如前所述，interceptor 可能运行在多个线程中，因此在具体实现时用户需要自行确保线程安全。另外，若指定了多个 interceptor，则 producer 将按照指定顺序调用它们，同时把每个 interceptor 中捕获的异常记录到错误日志中而不是向上传递。这在使用过程中要特别留意。

为了更好地说明 interceptor 的使用，下面实现一个简单的双 interceptor 组成的拦截链。第一个 interceptor 会在消息发送前将时间戳信息加到消息 value 的最前部；第二个 interceptor 会在消息发送后更新成功发送消息数或失败发送消息数。这两个 interceptor 实现的逻辑其实都很简单，特别是第一个 interceptor 做的事情实际上并无太大的实际意义。这里只是为了演示如何使用 interceptor 以及连接链。

首先，创建第一个 interceptor——`TimeStampPrependerInterceptor`，关键代码如下：

```
public class TimeStampPrependerInterceptor implements ProducerInterceptor<String,
String> {
    @Override
    public void configure(Map<String, ?> configs) {

    }
}
```

```
@Override
public ProducerRecord onSend(ProducerRecord record) {
    return new ProducerRecord(
        record.topic(), record.partition(), record.timestamp(),
        record.key(), System.currentTimeMillis() + "," + record.value().toString());
}

@Override
public void onAcknowledgement(RecordMetadata metadata, Exception
exception) {

}

@Override
public void close() {
}
}
```

上面代码的关键部分在于，在 `onSend` 方法中会创建一个新的 `record`，把时间戳写入消息体的最前部。下面定义第二个 `interceptor`——`CounterInterceptor`，该 `interceptor` 会在消息发送后更新“发送成功消息数”和“发送失败消息数”两个计数器，并在 `producer` 关闭时打印这两个计数器。完整代码如下：

```
public class CounterInterceptor implements ProducerInterceptor<String,
String> {

    private int errorCounter = 0;
    private int successCounter = 0;

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

    @Override
    public ProducerRecord<String, String> onSend(ProducerRecord<String,
String> record) {
        return record;
    }

    @Override
    public void onAcknowledgement(RecordMetadata metadata, Exception
exception) {
        if (exception == null) {
            successCounter++;
        } else {

```

```

        errorCounter++;
    }
}

@Override
public void close() {
    // 保存结果
    System.out.println("Successful sent: " + successCounter);
    System.out.println("Failed sent: " + errorCounter);
}
}

```

定义好 `interceptor` 之后，需要在 `producer` 主程序中指定它们，代码如下：

```

Properties props = new Properties();
props.put(...);
// 构建拦截链
List<String> interceptors = new ArrayList<>();
interceptors.add("huxi.test.producer.TimestampPrependerInterceptor");
// interceptor 1
interceptors.add("huxi.test.producer.CounterInterceptor"); // interceptor 2
props.put(ProducerConfig.INTERCEPTOR_CLASSES_CONFIG, interceptors);
...

String topic = "test-topic";
Producer<String, String> producer = new KafkaProducer<>(props);
for (int i = 0; i < 10; i++) {
    ProducerRecord<String, String> record = new ProducerRecord<>(topic,
"message" + i);
    producer.send(record).get();
}

// 一定要关闭 producer，这样才会调用 interceptor 的 close 方法
producer.close();

```

上面的代码发送 10 条消息到 `Kafka`，我们利用 `kafka-console-consumer` 工具来测试两个 `interceptor` 是否工作正常，输出如图 4.3 所示。

```

bogon:kafka_0.10.2.1 huxi$ bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic test-topic
1497237162939,message0
1497237163137,message1
1497237163139,message2
1497237163141,message3
1497237163146,message4
1497237163149,message5
1497237163154,message6
1497237163158,message7
1497237163162,message8
1497237163167,message9

```

图 4.3 `interceptor` 测试输出

如图 4.3 所示，所有消息的 `value` 前面都被加上了发送时的时间戳，这就说明 `TimeStampPrependerInterceptor` 是正常工作的。而运行 `producer` 之后的输出如下：

```
Successful sent: 10
Failed sent: 0
```

此输出说明 `CounterInterceptor` 也是能够正常工作的。双 `interceptor` 构成的连接链生效了。

4.6 无消息丢失配置

如前所述，Java 版本 `producer` 用户采用异步发送机制。`KafkaProducer.send` 方法仅仅把消息放入缓冲区中，由一个专属 I/O 线程负责从缓冲区中提取消息并封装进消息 `batch` 中，然后发送出去。显然，这个过程中存在着数据丢失的窗口：若 I/O 线程发送之前 `producer` 崩溃，则存储缓冲区中的消息全部丢失了。这是 `producer` 需要处理的很重要的问题。

`producer` 的另一个问题就是消息的乱序。假设客户端依次执行下面的语句发送两条消息到相同的分区：

```
producer.send(record1);
producer.send(record2);
```

若此时由于某些原因（比如瞬时的网络抖动）导致 `record1` 未发送成功，同时 `Kafka` 又配置了重试机制以及 `max.in.flight.requests.per.connection` 大于 1（默认值是 5），那么 `producer` 重试 `record1` 成功后，`record1` 在日志中的位置反而位于 `record2` 之后，这样造成了消息的乱序。要知道很多实际使用场景中都有事件强顺序保证的要求。

鉴于 `producer` 的这两个问题，应该如何规避呢？首先，对于消息丢失的问题，很容易想到的一个方案就是：既然异步发送可能丢失数据，改成同步发送似乎是一个不错的主意。比如这样：

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

采用同步发送当然是可以的，但是性能会很差，并不推荐在实际场景中使用。因此最好能有一份配置，既使用异步方式还能有效地避免数据丢失，即使出现 `producer` 崩溃的情况也不会有问题。

本节首先给出 `producer` 端“无消息丢失配置”，然后再分别解释每个参数配置的含义。具体配置参数列表如下。

- `block.on.buffer.full = true`
- `acks = all or -1`

- `retries = Integer.MAX_VALUE`
- `max.in.flight.requests.per.connection = 1`
- 使用带回调机制的 `send` 发送消息，即 `KafkaProducer.send(record, callback)`
- Callback 逻辑中显式地立即关闭 producer，使用 `close(0)`
- `unclean.leader.election.enable = false`
- `replication.factor = 3`
- `min.insync.replicas = 2`
- `replication.factor > min.insync.replicas`
- `enable.auto.commit = false`

下面分别从 producer 端和 broker 端分别讨论一下上述参数这样设置的含义。

4.6.1 producer 端配置

`block.on.buffer.full = true`

实际上这个参数在 Kafka 0.9.0.0 版本已经被标记为“deprecated”，并使用 `max.block.ms` 参数替代，但这里还是推荐用户显式地设置它为 `true`，使得内存缓冲区被填满时 producer 处于阻塞状态并停止接收新的消息而不是抛出异常；否则 producer 生产速度过快会耗尽缓冲区。新版本 Kafka（0.10.0.0 之后）可以不用理会这个参数，转而设置 `max.block.ms` 即可。

`acks = all`

设置 `acks` 为 `all` 很容易理解，即必须要等到所有 follower 都响应了发送消息才能认为提交成功，这是 producer 端最强程度的持久化保证。

`retries = Integer.MAX_VALUE`

设置成 `MAX_VALUE` 纵然有些极端，但其实想表达的是 producer 要开启无限重试。用户不必担心 producer 会重试那些肯定无法恢复的错误，当前 producer 只会重试那些可恢复的异常情况，所以放心地设置一个比较大的值通常能很好地保证消息不丢失。

`max.in.flight.requests.per.connection = 1`

设置该参数为 1 主要是为了防止 topic 同分区下的消息乱序问题。这个参数的实际效果其实限制了 producer 在单个 broker 连接上能够发送的未响应请求的数量。因此，如果设置成 1，则 producer 在某个 broker 发送响应之前将无法再给该 broker 发送 PRODUCE 请求。

使用带有回调机制的 `send`

不要使用 `KafkaProducer` 中单参数的 `send` 方法，因为该 `send` 调用仅仅是把消息发出而不会

理会消息发送的结果。如果消息发送失败，该方法不会得到任何通知，故可能造成数据的丢失。实际环境中一定要使用带回调机制的 `send` 版本，即 `KafkaProducer.send(record, callback)`。

Callback 逻辑中显式立即关闭 producer

在 Callback 的失败处理逻辑中显式调用 `KafkaProducer.close(0)`。这样做的目的是为了处理消息的乱序问题。若不使用 `close(0)`，默认情况下 producer 会被允许将未完成的消息发送出去，这样就有可能造成消息乱序。

4.6.2 broker 端配置

`unclean.leader.election.enable = false`

关闭 unclean leader 选举，即不允许非 ISR 中的副本被选举为 leader，从而避免 broker 端因日志水位截断而造成的消息丢失。

`replication.factor >= 3`

设置成 3 主要是参考了 Hadoop 及业界通用的三备份原则，其实这里想强调的是一定要使用多个副本来保存分区的信息。

`min.insync.replicas > 1`

用于控制某条消息至少被写入到 ISR 中的多少个副本才算成功，设置成大于 1 是为了提升 producer 端发送语义的持久性。记住只有在 producer 端 `acks` 被设置成 `all` 或 `-1` 时，这个参数才有意义。在实际使用时，不要使用默认值。

确保 `replication.factor > min.insync.replicas`

若两者相等，那么只要有一个副本挂掉，分区就无法正常工作，虽然有很高的持久性但可用性被极大地降低了。推荐配置成 `replication.factor = min.insync.replicas + 1`。

4.7 消息压缩

众所周知，数据压缩显著地降低了磁盘占用或带宽占用，从而有效地提升了 I/O 密集型应用的性能。不过引入压缩同时会消耗额外的 CPU 时钟周期，因此压缩是 I/O 性能和 CPU 资源的平衡（trade-off）。

Kafka 自 0.7.x 版本便开始支持压缩特性——producer 端能够将一批消息压缩成一条消息发送，而 broker 端将这条压缩消息写入本地日志文件。当 consumer 获取到这条压缩消息时，它

会自动地对消息进行解压缩，还原成初始的消息集合返还给用户。如果使用一句话来总结 Kafka 压缩特性的话，那么就是——producer 端压缩，broker 端保持，consumer 端解压缩。所谓的 broker 端保持是指 broker 端在通常情况下不会进行解压缩操作，它只是原样保存消息而已。这里的“通常情况下”表示要满足一定的条件。如果有些前置条件不满足（比如需要进行消息格式的转换等），那么 broker 端就需要对消息进行解压缩然后再重新压缩。

4.7.1 Kafka 支持的压缩算法

当前 Kafka 支持 3 种压缩算法：GZIP、Snappy 和 LZ4。虽然默认情况下 Kafka 是不压缩消息的，但用户可以通过设定 producer 端参数 `compression.type` 来开启消息压缩，即构造 `KafkaProducer` 的属性对象时进行设置。假定要设置使用 Snappy 压缩算法，则设置方法如下：

```
props.put("compression.type", "snappy");  
// 或者  
props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "snappy");
```

值得一提的是，Facebook 公司于 2016 年 8 月底开源了新的压缩算法 Zstandard。据 Facebook 工程师 Yann Collet 和 Chip Turner 介绍，Zstandard 是少有的能够同时在效率和性能方面超过当前业界“翘楚”Zlib 的压缩算法之一。当前，Kafka 社区已经计划在后续 Kafka 版本中增加对 Zstandard 算法的支持。感兴趣的读者可以访问 [https://cwiki.apache.org/confluence/display/KAFKA/KIP-110%3A+Add+Codec+for+Zstandard+Compression/](https://cwiki.apache.org/confluence/display/KAFKA/KIP-110%3A+Add+Codec+for+Zstandard+Compression) 查询该功能的研发进度。

4.7.2 算法性能比较与调优

对 Kafka 源代码熟悉的读者会发现 `KafkaProducer.send` 方法逻辑的主要耗时都在消息压缩操作上，因此妥善地调优压缩算法至关重要。不过在此之前，首先比较一下目前 Kafka 支持的压缩算法的性能。

大家可能觉得 GZIP、Snappy 和 LZ4 在实际环境中的表现各有千秋，但对 Kafka 而言，性能测试的结果出奇地一致，即 `LZ4 >> Snappy > GZIP`。本来 Snappy 和 LZ4 无论在压缩速率还是压缩比率上都差不多，但由于 Kafka 源代码中对 Snappy 的某个关键参数进行了硬编码，使得 Snappy 与 Kafka 的结合表现并不优秀，至少比 LZ4 要差很多。图 4.4 和图 4.5 分别表示 Snappy 和 LZ4 的常规对比以及应用在 Kafka 上的 producer 性能对比。

如图 4.5 所示，目前 Kafka 对 LZ4 压缩算法的支持是最好的。启用 LZ4 进行消息压缩的 producer 的吞吐量是最高的。

下面说说如何调优 producer 的压缩性能。首先判断是否启用压缩的依据是 I/O 资源消耗与 CPU 资源消耗的对比。如果生产环境中的 I/O 资源非常紧张，比如 producer 程序消耗了大量的

网络带宽或 broker 端的磁盘占用率非常高，而 producer 端的 CPU 资源非常富裕，那么就可以考虑为 producer 开启消息压缩。反之则不需要设置消息压缩以节省宝贵的 CPU 时钟周期。

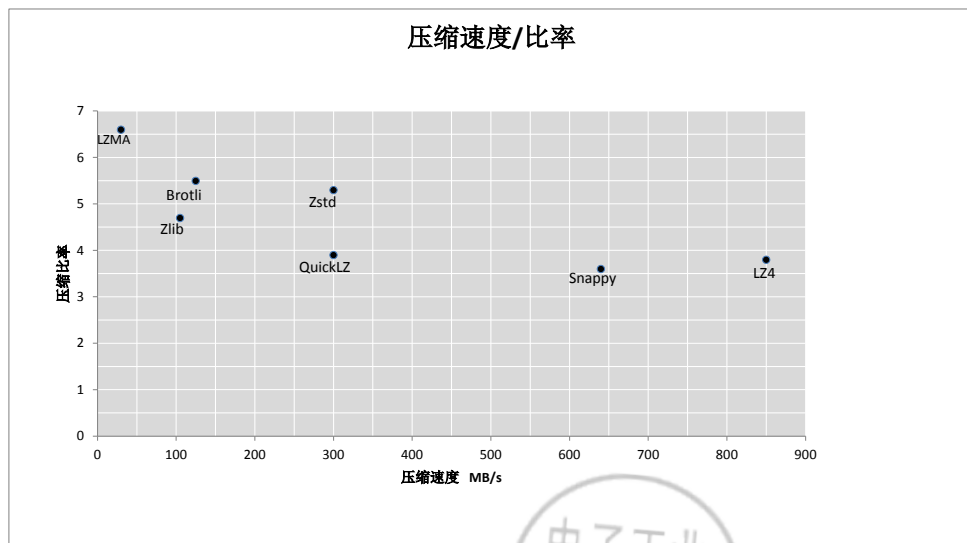


图 4.4 Snappy 与 LZ4 性能比较
(图片来源 <https://i.stack.imgur.com/LPCSe.png>)

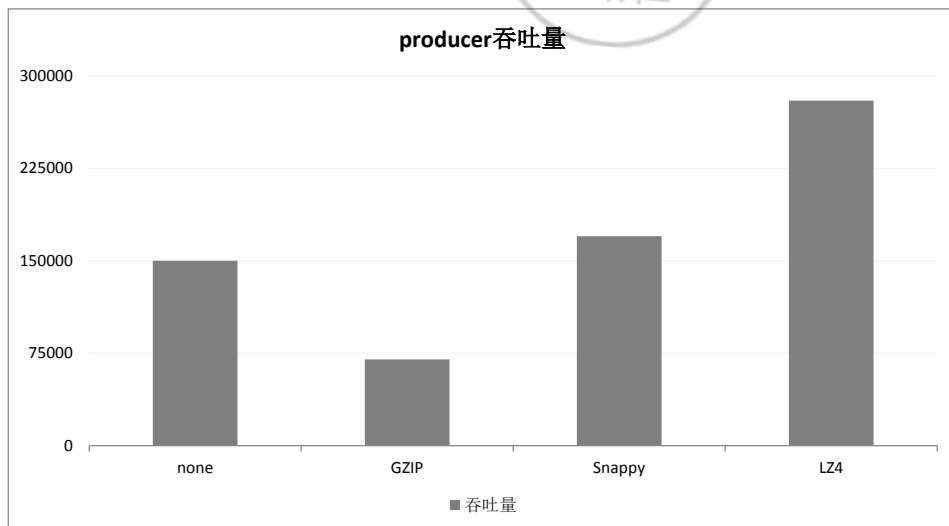


图 4.5 不同压缩算法的 producer 性能比较
(图片来源 http://blog.yaorenjie.com/images/kafka010/producer_throughput.png)

其次，压缩的性能与 producer 端的 batch 大小息息相关。通常情况下我们可以认为 batch 越大需要压缩的时间就越长。针对不同压缩算法和 batch 大小的一组实验数据如图 4.6 所示。

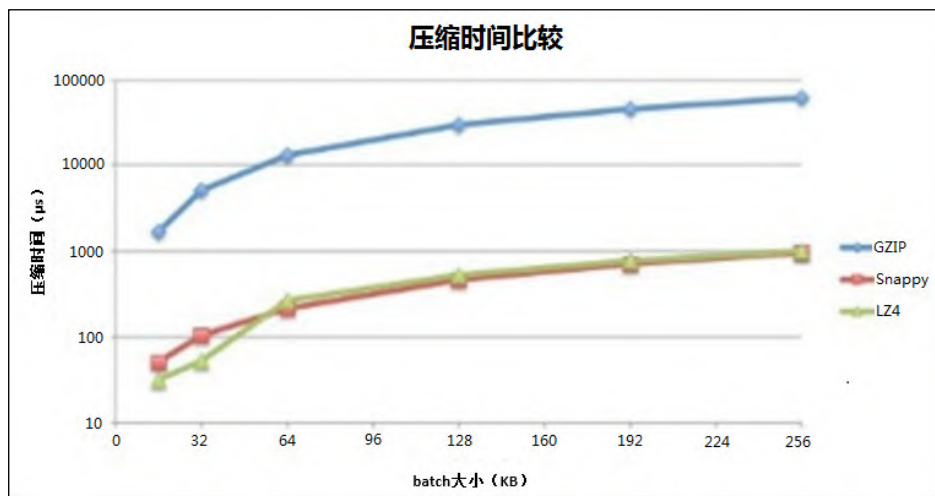


图 4.6 不同 batch 大小的压缩时间

(图片来源 <https://www.slideshare.net/JiangjieQin/producer-performance-tuning-for-apache-kafka-63147600>)

如图 4.6 所示，batch 大小越大，压缩时间就越长，不过时间的增长不是线性的，而是越来越平缓的。如果发现压缩很慢，说明系统的瓶颈在用户主线程而不是 I/O 发送线程，因此可以考虑增加多个用户线程同时发送消息，这样通常能显著地提升 producer 吞吐量。

4.8 多线程处理

实际环境中只使用一个用户主线程通常无法满足所需的吞吐量目标，因此需要构造多个线程或多个进程来同时给 Kafka 集群发送消息。这样在使用过程中就存在着两种基本的使用方法。

- 多线程单 KafkaProducer 实例。
- 多线程多 KafkaProducer 实例。

多线程单 KafkaProducer 实例

顾名思义，这种方法就是在全局构造一个 KafkaProducer 实例，然后在多个线程中共享使用。由于 KafkaProducer 是线程安全的，所以这种使用方式也是线程安全的。

多线程多 KafkaProducer 实例

除了上面的用法，还可以在每个 producer 主线程中都构造一个 KafkaProducer 实例，并且保证此实例在该线程中封闭（thread confinement，线程封闭是实现线程安全的重要手段之一）。显然，这两种方式各有优劣，如表 4.2 所示。

表 4.2 两种 KafkaProducer 使用方式比较

	说 明	优 势	劣 势
单 KafkaProducer 实例	所有线程共享一个 KafkaProducer 实例	实现简单，性能好	①所有线程共享一个内存缓冲区，可能需要较多内存； ②一旦 producer 某个线程崩溃导致 KafkaProducer 实例被“破坏”，则所有用户线程都无法工作
多 KafkaProducer 实例	每个线程维护自己专属的 KafkaProducer 实例	①每个用户线程拥有专属的 KafkaProducer 实例、缓冲区空间及一组对应的配置参数，可以进行细粒度的调优； ②单个 KafkaProducer 崩溃不会影响其他 producer 线程工作	需要较大的内存分配开销

如果是对分区数不多的 Kafka 集群而言，笔者比较推荐使用第一种方法，即在多个 producer 用户线程中共享一个 KafkaProducer 实例。若是对那些拥有超多分区的集群而言，采用第二种方法具有较高的可控性，方便 producer 的后续管理。

4.9 旧版本 producer

旧版本 producer 或“old producer”是 Kafka 最开始提供的 producer API。由于完全使用 Scala 语言编写，也被称为 Scala producer。该 producer 已经在 Kafka 0.9.0.0 版本被正式废弃，未来可能会完全从 Kafka 代码中移除。

新旧两版本 producer 的主要区别如下。

- 旧版本 producer 的主要入口是 kafka.producer.Producer，新版本 producer 的主要入口是 org.apache.kafka.clients.producer.KafkaProducer。
- 旧版本 producer 默认同步发送，新版本默认异步发送。
- 引入 kafka-core 依赖（比如 kafka_2.11）就可以直接使用旧版本 producer，新版本 producer 则需要引入 kafka-clients 依赖。
- 新旧两个版本所用参数列表几乎完全不同。典型的参数就是 metadata.broker.list（旧版

本)与 bootstrap.servers (新版本)。

- 旧版本直接与 ZooKeeper 通信来发送数据, 新版本彻底摆脱对 ZooKeeper 的依赖。

鉴于目前仍然有很多用户在使用 Kafka 0.8.x 和旧版本 producer, 我们在这里简要介绍一下旧版本 producer 的工作原理。旧版本 producer 的工作原理如图 4.7 所示。

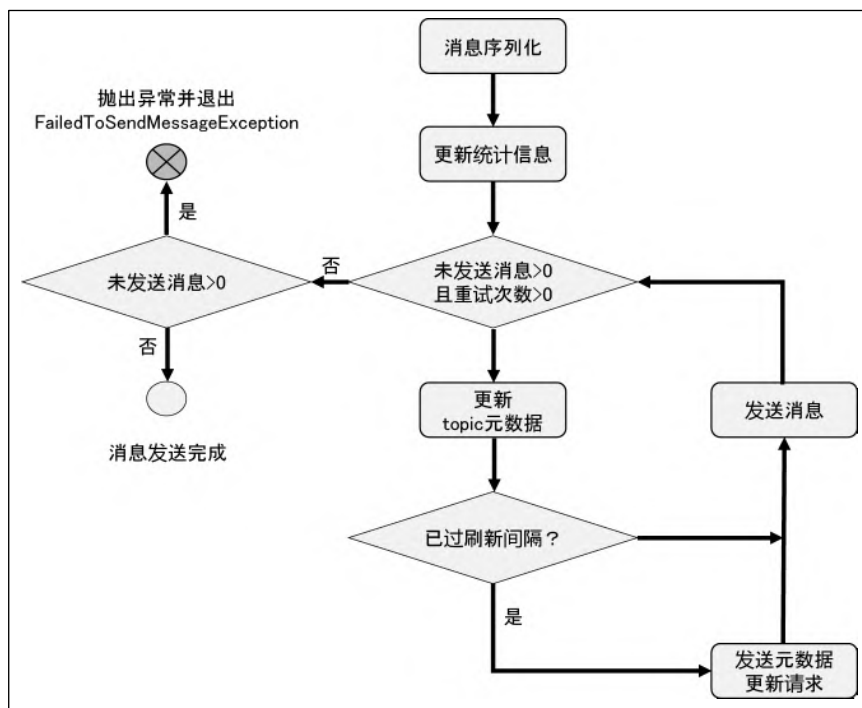


图 4.7 旧版本 producer 工作流程图

细心的读者会发现大致的流程和新版本 producer 类似, 比如依然都有序列化、请求元数据和发送等主要逻辑。只是在底层的实现上和 Java 版本 producer 有着巨大的不同。

旧版本 producer API 有两个主要的 producer: SyncProducer 和 AsyncProducer, 分别负责同步发送和异步发送。由于目前旧版本 producer 已经不被社区推荐使用了, 故 Kafka 官方强烈推荐用户升级 Kafka producer 版本并始终使用新版本的 producer API。如果读者一定要使用旧版本 producer, 则访问 <https://kafka.apache.org/081/documentation.html#producerapi>/查看具体的 API 用法。

4.10 本章小结

本章详细探讨了 Kafka 新版本 producer 的各个方面，包括 producer 的概要设计、producer 程序的开发、参数设置以及 producer 的各个子功能。

各位读者在阅读本章之后应该可以结合自身业务目标的需求来开发实际的可供线上部署的 Kafka producer 程序。

在第 5 章中，会重点讨论 Kafka consumer 的设计与使用。



第 5 章

consumer 开发

不论 Kafka 如何演变，万变不离其宗的是它一定有一些外部的生产者（producer）应用程序给 Kafka 发送消息，然后还有一些外部的消费者（consumer）应用程序读取 producer 发送的消息。

本章着重讨论 Kafka 的 consumer 设计以及基于 Java 版本 consumer 的开发与使用。

学习本章，你将了解到以下内容。

- Kafka consumer 概要。
- Kafka consumer 简要设计。
- Java 版本 consumer 及 API。

5.1 consumer 概览

Kafka 消费者（consumer）是从 Kafka 读取数据的应用。若干个 consumer 订阅 Kafka 集群中的若干个 topic 并从 Kafka 接收属于这些 topic 的消息。在开始学习 consumer 程序开发之前，我们有必要梳理一下 consumer 基本概念以及一些关键的技术要点。

5.1.1 消费者（consumer）

顾名思义，consumer 是读取 Kafka 集群中某些 topic 消息的应用程序。在当前 Kafka 生态中，consumer 可以由多种编程语言实现，比如常见的 Java、C/C++、Go 等。我们在本章中主要讨论使用 Java 语言开发 consumer 程序。除了 consumer 的定义之外，我们还需要明确

consumer 的版本和分类。

先说版本吧，Kafka 最初开源时自带由 Scala 语言编写的 consumer 客户端——我们称之为 Scala consumer 或 Old consumer，即旧版本 consumer。随着时间的推移，Kafka 社区日益发现该版本 consumer 存在着诸多设计缺陷，例如在旧版本 consumer 中当用户不使用消费者组（consumer group，有关 consumer group 的内容将在 5.1.2 节中讨论），而直接使用 low-level consumer（旧版本 consumer 分 high-level 和 low-level 两种 API。我们将在后续章节中详细探讨这两种 API 的区别）时，用户必须自行实现错误处理和故障转移（failover）等功能。因此社区在 0.9.0.0 版本正式推出了新版本的 consumer 客户端。由于它是用 Java 语言编写的，因此我们称之为 Java consumer 或 new consumer，即新版本 consumer。

事实上，这两个版本在设计上差异极大，很多用户甚至都无法区分自己到底使用的是哪个版本的 consumer，特别是新版本 consumer 颠覆了旧版本 consumer 管理和保存位移（5.1.3 节将讨论 consumer 的位移）的机制。这也是为什么当用户切换到新版本 consumer 时会抱怨 Kafka-manager 监控框架（Yahoo 开源的一款 Kafka 监控软件，详见 <https://github.com/yahoo/kafka-manager>）无法监控 consumer 位移信息的原因。表 5.1 总结了新旧两版本 consumer 的主要区别以及区分方法。

表 5.1 新旧版本 consumer 对比

	编程语言	API 包名	主要使用类
新版本	Java	org.apache.kafka.clients.consumer.*	KafkaConsumer
旧版本	Scala	kafka.consumer.*	ZookeeperConsumerConnector SimpleConsumer

查看表 5.1 读者能够很容易地分辨清楚程序中使用的到底是新 consumer 还是旧 consumer 了。

除了要正确区分 consumer 版本，我们还需要了解 consumer 的分类。严格来说，所谓的 consumer 分类并不是 Kafka 社区已有的概念。实际上 Kafka 官方文档中也找不到这样的提法。不过在笔者看来，consumer 的分类至关重要，因为只有理解了它们的含义以及区别，我们才能更好地在实际使用场景中选择最合理的 consumer 分类帮助我们完成业务目标。好了，言归正传，笔者把 consumer 分为如下两类。

- 消费者组（consumer group）。
- 独立消费者（standalone consumer）。

本章后续内容会陆续对这两类 consumer 做出详细分析。在这里，只需要了解 consumer

group 是由多个消费者实例（consumer instance）构成一个整体进行消费的，而 standalone consumer 则单独执行消费操作。

综上所述，当我们在讨论 consumer 或开发 consumer 程序的时候，必须给出明确的消费者上下文信息（consumer context），即所使用 consumer 的版本以及 consumer 的分类。本章剩余部分的内容也都会以这部分上下文信息为基础而展开。

5.1.2 消费者组（consumer group）

笔者曾试图用自己的话来定义 consumer group，也见过他人描述，但都觉得不如 Kafka 官网的一句话来得言简意赅，只用一句话就清晰无歧义地给出了 consumer group 的含义同时还明确点出了它的特性。在这一点上 Kafka 官网可谓无出其右。说了这么多，我们就来看看官网关于 consumer group 的定义：

Consumers label themselves with a consumer group name, and each record published to a topic is delivered to one consumer instance within each subscribing consumer group.

翻译一下就是：消费者使用一个消费者组名（即 group.id）来标记自己，topic 的每条消息都只会被发送到每个订阅它的消费者组的一个消费者实例上。

这句话给出了 3 个非常重要的信息：①一个 consumer group 可能有若干个 consumer 实例（一个 group 只有一个实例也是允许的）；②对于同一个 group 而言，topic 的每条消息只能被发送到 group 下的一个 consumer 实例上；③topic 消息可以被发送到多个 group 中。

有印象的读者一定还记得，我们在第 1 章中提到过 Kafka 同时支持基于队列和基于发布/订阅的两种消息引擎模型。事实上 Kafka 就是通过 consumer group 实现的对这两种模型的支持。

- 所有 consumer 实例都属于相同 group——实现基于队列的模型。每条消息只会被一个 consumer 实例处理。
- consumer 实例都属于不同 group——实现基于发布/订阅的模型。极端的情况是每个 consumer 实例都设置完全不同的 group，这样 Kafka 消息就会被广播到所有 consumer 实例上。

图 5.1 清晰地展示了两个 consumer group 订阅相同 topic 的场景，图中 topic 有 P0、P1、P2 和 P3 这 4 个分区，分别保存在 Server1 和 Server2 两个 broker 上。具体的 consumer group 分配情况如图 5.1 所示。

Group A 只有两个 consumer 实例，每个实例消费两个分区的数据；而 Group B 中有 4 个 consumer 实例，每个实例分别消费一个分区的数据。图 5.1 同时也很好地说明了 Kafka 在为 consumer group 成员分配分区时可以做到公平的分配。

看到这里可能很多读者会问：为什么需要 consumer group？我把多个 consumer 实例放入一个组下有什么好处吗？实际上，consumer group 是用于实现高伸缩性、高容错性的 consumer 机制。组内多个 consumer 实例可以同时读取 Kafka 消息，而且一旦有某个 consumer “挂”了，consumer group 会立即将已崩溃 consumer 负责的分区转交给其他 consumer 来负责，从而保证整个 group 可以继续工作，不会丢失数据——这个过程被称为重平衡（rebalance）。我们会在 5.6 节中讨论 consumer group 的 rebalance 机制。

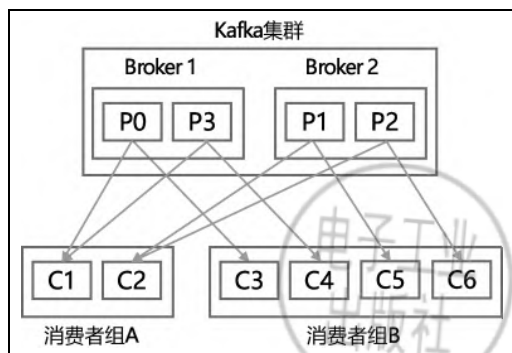


图 5.1 consumer group 订阅 topic

另外由于 Kafka 目前只提供单个分区内的消息顺序，而不会维护全局的消息顺序，因此如果用户要实现 topic 全局的消息读取顺序，就只能通过让每个 consumer group 下只包含一个 consumer 实例的方式来间接实现。

下面总结一下 consumer group 的含义和特点。

- consumer group 下可以有一个或多个 consumer 实例。一个 consumer 实例可以是一个线程，也可以是运行在其他机器上的进程。
- group.id 唯一标识一个 consumer group。
- 对某个 group 而言，订阅 topic 的每个分区只能分配给该 group 下的一个 consumer 实例（当然该分区还可以被分配给其他订阅该 topic 的消费者组）。

5.1.3 位移（offset）

需要明确指出的是，这里的 offset 指代的是 consumer 端的 offset，与分区日志中的 offset

是不同的含义。每个 consumer 实例都会为它消费的分区分维护属于自己的位置信息来记录当前消费了多少条消息。这在 Kafka 中有一个特有的术语：位移（offset）。很多消息引擎都把消费端的 offset 保存在服务器端（broker）。这样做的好处当然是实现简单，但会有以下 3 个方面的问题。

- broker 从此变成了有状态的，增加了同步成本，影响伸缩性。
- 需要引入应答机制（acknowledgement）来确认消费成功。
- 由于要保存许多 consumer 的 offset，故必然引入复杂的数据结构，从而造成不必要的资源浪费。

Kafka 则选择了不同的方式：让 consumer group 保存 offset，那么只需要简单地保存一个长整型数据就可以了，同时 Kafka consumer 还引入了检查点机制（checkpointing）定期对 offset 进行持久化，从而简化了应答机制的实现。图 5.2 展示了 consumer 端 offset 的保存方式。从图 5.2 中我们可以看到当前 Kafka consumer 在内部使用一个 map 来保存其订阅 topic 所属分区的 offset。

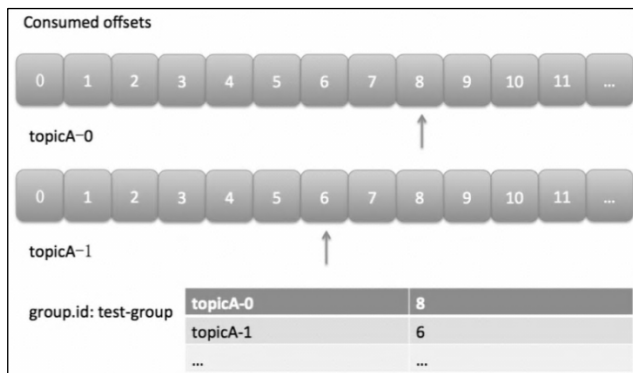


图 5.2 consumer 位移保存

5.1.4 位移提交

consumer 客户端需要定期地向 Kafka 集群汇报自己消费数据的进度，这一过程被称为位移提交（offset commit）。位移提交这件事情对于 consumer 而言非常重要，它不仅表征了 consumer 端的消费进度，同时也直接决定了 consumer 端的消费语义保证。我们会在本章后续部分中详细展开消费语义保证的话题以及位移提交是如何影响这种语义保证的。

新版本和旧版本 consumer 提交位移的方式截然不同：旧版本 consumer 会定期将位移信息提交到 ZooKeeper 下的固定节点上，具体路径如图 5.3 所示。

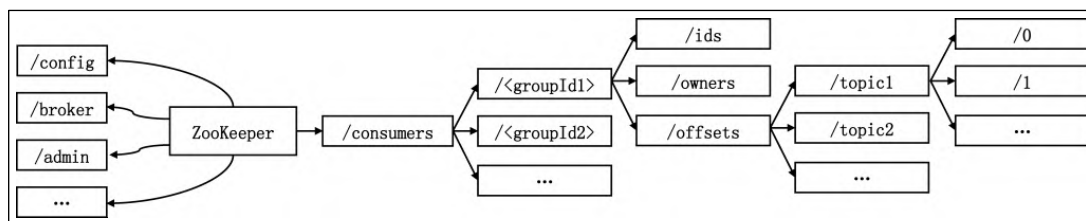


图 5.3 ZooKeeper 位移提交路径

该路径是 `/consumers/<group.id>/offsets/<topic>/<partitionId>`，其中 `group.id`、`topic` 和 `partitionId` 是变化的值，因用户的不同环境而不同。

随着 Kafka consumer 在实际场景的不断应用，社区发现旧版本 consumer 把位移提交到 ZooKeeper 的做法并不合适。ZooKeeper 本质上只是一个协调服务组件，它并不适合作为位移信息的存储组件，毕竟频繁高并发的读/写操作并不是 ZooKeeper 擅长的事情。因此在 0.9.0.0 版本 Kafka 推出的新版本 consumer 中，位移提交的方式被彻底颠覆——新版本 consumer 把位移提交到 Kafka 的一个内部 topic (`__consumer_offsets`) 上。注意，这个 topic 名字的前面有两个下画线！

这个内部 topic 很神秘，官网上也仅仅出现了一次。对于普通用户来说，你只需要了解该 topic 是一个内部 topic，通常不能直接操作该 topic 就可以了，特别是注意不要擅自删除或搬移该 topic 的日志文件。由于新版本 consumer 提交位移到这个 topic，因此 consumer 不再依赖 ZooKeeper（主要是指保存 offset 这件事情），这就是为什么我们开发新版本 consumer 应用时不再需要连接 ZooKeeper 的原因——旧版本 consumer 程序开发要求用户必须指定 ZooKeeper 地址。

5.1.5 `__consumer_offsets`

对于想深入了解 `__consumer_offsets` 的读者而言，我们在这里简要地介绍一下这个神秘的 topic。

首先，要明确的是，这个 topic 通常情况下都是给新版本 consumer 使用的。为什么说是通常情况？因为旧版本 consumer 确实也提供了一个特定的参数让用户在使用旧版本 consumer 时把位移提交到这个 topic 上。那就是设置旧版本 consumer 的参数 `offsets.storage=kafka`。不过笔者很少看到有这样的用法，因此我们可以安全地认为这个内部 topic 就是为新版本 consumer 保存位移的。

其次，`__consumer_offsets` 是 Kafka 自行创建的，因此用户不可擅自删除该 topic 的所有信息。很多 Kafka 初学者在搭建起 Kafka 集群并执行了一些消费操作后，会发现在 Kafka 的日志

目录下出现了很多诸如图 5.4 这样的文件夹，于是询问是否可以手动删除它们。

__consumer_offsets-0	__consumer_offsets-18	__consumer_offsets-27	__consumer_offsets-36	__consumer_offsets-45
__consumer_offsets-1	__consumer_offsets-19	__consumer_offsets-28	__consumer_offsets-37	__consumer_offsets-46
__consumer_offsets-10	__consumer_offsets-2	__consumer_offsets-29	__consumer_offsets-38	__consumer_offsets-47
__consumer_offsets-11	__consumer_offsets-20	__consumer_offsets-3	__consumer_offsets-39	__consumer_offsets-48
__consumer_offsets-12	__consumer_offsets-21	__consumer_offsets-30	__consumer_offsets-4	__consumer_offsets-49
__consumer_offsets-13	__consumer_offsets-22	__consumer_offsets-31	__consumer_offsets-40	__consumer_offsets-5
__consumer_offsets-14	__consumer_offsets-23	__consumer_offsets-32	__consumer_offsets-41	__consumer_offsets-6
__consumer_offsets-15	__consumer_offsets-24	__consumer_offsets-33	__consumer_offsets-42	__consumer_offsets-7
__consumer_offsets-16	__consumer_offsets-25	__consumer_offsets-34	__consumer_offsets-43	__consumer_offsets-8
__consumer_offsets-17	__consumer_offsets-26	__consumer_offsets-35	__consumer_offsets-44	__consumer_offsets-9

图 5.4 __consumer_offsets 日志

现在我们知道了这些文件夹是属于__consumer_offsets 的，所以不可以删除它们。通常情况下，这样的文件夹应该有 50 个，编号从 0 到 49。

如果打开图 5.4 中的任意一个文件夹，会发现它就是一个正常的 Kafka topic 日志文件目录，里面至少有一个日志文件（.log）和两个索引文件（.index 和.timeindex）。只不过该日志中保存的消息都是 Kafka 集群上 consumer（特别是 consumer group）的位移信息罢了。__consumer_offsets 的每条消息格式大致如图 5.5 所示。

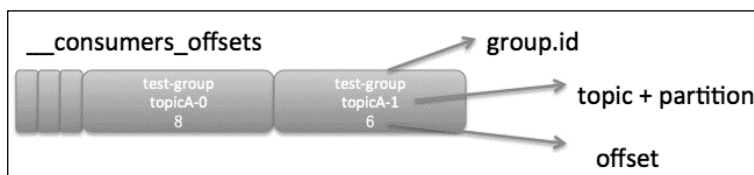


图 5.5 __consumer_offsets 的消息格式

你可以把它想象成一个 KV 格式的消息，key 就是一个三元组：group.id + topic + 分区号，而 value 就是 offset 的值。每当更新同一个 key 的最新 offset 值时，该 topic 就会写入一条含有最新 offset 的消息，同时 Kafka 会定期地对该 topic 执行压实操作（compact），即为每个消息 key 只保存含有最新 offset 的消息。这样既避免了对分区日志消息的修改，也控制住了__consumer_offsets topic 总体的日志容量，同时还能实时反映最新的消费进度。

考虑到一个 Kafka 生产环境中可能有很多 consumer 或 consumer group，如果这些 consumer 同时提交位移，则必将加重__consumer_offsets 的写入负载，因此社区特意为该 topic 创建了 50 个分区，并且对每个 group.id 做哈希求模运算，从而将负载分散到不同的__consumer_offsets 分区上。这就是说，每个 consumer group 保存的 offset 都有极大的概率分别出现在该 topic 的不同分区上。

总之，__consumer_offsets 是系统内部的 topic，用户应尽量避免执行该 topic 的任何操作。我们将会在监控相关章节中介绍如何利用__consumer_offsets 定位并读取 consumer group 的 offset 信息。

5.1.6 消费者组重平衡 (consumer group rebalance)

标题中特意强调了 consumer group。如果用户使用的是 standalone consumer，则压根就没有 rebalance 的概念，即 rebalance 只对 consumer group 有效。

何为 rebalance？它本质上是一种协议，规定了一个 consumer group 下所有 consumer 如何达成一致来分配订阅 topic 的所有分区。举个例子，假设我们有一个 consumer group，它有 20 个 consumer 实例。该 group 订阅了一个具有 100 个分区的 topic。那么正常情况下，consumer group 平均会为每个 consumer 分配 5 个分区，即每个 consumer 负责读取 5 个分区的数据。这个分配过程就被称作 rebalance。

新旧两个版本的 consumer 都有 rebalance 的过程，只不过它们在设计上有些不同罢了。5.6 节将以新版本 rebalance 的设计为主要研究目标进行详细讨论。

5.2 构建 consumer

初步了解了 Kafka consumer（特别是 new consumer）之后，本节我们来看看如何使用程序 API 构造一个 consumer。本节仍然以 Java 版本 consumer 的 API 为例。其他第三方库 API 使用方法请参照对应的官方文档。

5.2.1 consumer 程序实例

和讲解 producer 一样，我们首先给出一份可运行的 producer 程序代码清单。下面的代码实现了最基本的 consumer 功能：构造一个 consumer group 从指定 Kafka topic 消费消息。运行下列代码前，我们至少需要一个最小规模的 Kafka 单机或集群环境，并且创建了 topic 且成功向该 topic 发送了消息。Kafka 运行环境搭建方法请参考第 3 章。代码如下：

```
package huxi.test.consumer;

import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;

import java.util.Arrays;
import java.util.Properties;

public class ConsumerTest {

    public static void main(String[] args) {
        String topicName = "test-topic";
```

```

        String groupId = "test-group";

        Properties props = new Properties();
        props.put("bootstrap.servers", "localhost:9092"); //必须指定
        props.put("group.id", groupId); //必须指定
        props.put("enable.auto.commit", "true");
        props.put("auto.commit.interval.ms", "1000");
        props.put("auto.offset.reset", "earliest");//从最早的消息开始读取
        props.put("key.deserializer", "org.apache.kafka.common.serialization.
StringDeserializer"); //必须指定
        props.put("value.deserializer", "org.apache.kafka.common.serialization.
StringDeserializer"); //必须指定
        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
        //创建 consumer 实例
        consumer.subscribe(Arrays.asList(topicName)); //订阅 topic
        try {
            while (true) {
                ConsumerRecords<String, String> records = consumer.poll(1000);
                for (ConsumerRecord<String, String> record : records)
                    System.out.printf("offset = %d, key = %s, value = %s\n",
record.offset(), record.key(), record.value());
            }
        } finally {
            consumer.close();
        }
    }
}

```

构造 consumer 需要下面 6 个步骤：

- (1) 构造一个 `java.util.Properties` 对象，至少指定 `bootstrap.servers`、`key.deserializer`、`value.deserializer` 和 `group.id` 的值，即上面代码中显式指明必须指定带注释的 4 个参数。
- (2) 使用上一步创建的 `Properties` 实例构造 `KafkaConsumer` 对象。
- (3) 调用 `KafkaConsumer.subscribe` 方法订阅 consumer group 感兴趣的 topic 列表。
- (4) 循环调用 `KafkaConsumer.poll` 方法获取封装在 `ConsumerRecord` 的 topic 消息。
- (5) 处理获取到的 `ConsumerRecord` 对象。
- (6) 关闭 `KafkaConsumer`。

1. 构造 Properties 对象

这是构建 consumer 的第一步。在创建的 `Properties` 对象中，必须指定的参数有 4 个。如果

翻开 Kafka 官网中 new consumer 参数列表（详见 <https://kafka.apache.org/documentation/#newconsumerconfigs>），我们会发现以下 4 个参数都是没有默认值的，分别介绍如下。

bootstrap.servers

和 Java 版本 producer 相同，这是必须要指定的参数。该参数指定了一组 host:port 对，用于创建与 Kafka broker 服务器的 Socket 连接。可以指定多组，使用逗号分隔，如 kafka1:9092,kafka2:9092,kafka3:9092。由于笔者本机环境中是单节点的 Kafka 环境，因而上面代码中使用了 localhost:9092。在实际生产环境中需要替换成线上 broker 列表。另外，若 Kafka 集群中 broker 机器数很多，我们只需要指定部分 broker 即可，不需要列出完整的 broker 列表。这是因为不管指定了几台机器，consumer 启动后都能通过这些机器找到完整的 broker 列表，因此为该参数指定多台机器通常只是为了常规的 failover 使用。这样即使某一台 broker 挂掉了，consumer 重启后依然能够通过该参数指定的其他 broker 连接 Kafka 集群。

需要注意的是，如果 broker 端没有显式配置 listeners（或 advertised.listeners）使用 IP 地址的话，那么最好将 bootstrap.servers 配置成主机名而不要使用 IP 地址，因为 Kafka 内部使用的是全称域名（FQDN，Fully Qualified Domain Name）。倘若不统一，会出现无法获取元数据的异常。

group.id

该参数指定的是 consumer group 的名字。它能够唯一标识一个 consumer group。细心的读者可能会发现官网上该参数是有默认值的，即一个空字符串。但在开发 consumer 程序时我们依然要显式指定 group.id，否则 consumer 端会抛出 InvalidGroupIdException 异常。通常为 group.id 设置一个有业务意义的名字就可以了。

key.deserializer

consumer 代码从 broker 端获取的任何消息都是字节数组的格式，因此消息的每个组件都要执行相应的解序列化操作才能“还原”成原来的对象格式。这个参数就是为消息的 key 做解序列化的。该参数值必须是实现 org.apache.kafka.common.serialization.Deserializer 接口的 Java 类的全限定名称。Kafka 默认为绝大部分的初始类型（primitive type）提供了现成的解序列化器。上面代码中使用了 org.apache.kafka.common.serialization.StringDeserializer 类。该类会将接收到的字节数组转换成 UTF-8 编码的字符串。consumer 支持用户自定义 deserializer，这通常都与 producer 端自定义 serializer “遥相呼应”。毕竟“原汤化原食”，使用了什么类型的 serializer 就必须使用对应类型的 deserializer 将其还原。

值得注意的是，不论 consumer 消费的消息是否指定了 key，consumer 都必须要设置这个参数，否则程序会抛出 ConfigException，提示“key.deserializer”没有默认值。

value.deserializer

与 `value.deserializer` 类似，该参数被用来对消息体（即消息 `value`）进行解序列化，从而把消息“还原”回原来的对象类型。上面示例代码中使用了与 `key.deserializer` 相同的值 `StringDeserializer`，即把消息体转换成字符串类型。当然，`value.deserializer` 可以设置成与 `key.deserializer` 不同的值，前提是 `key.serializer` 与 `value.serializer` 设置了不同的值。

在使用过程中，我们一定要谨记 `key.deserializer` 和 `value.deserializer` 指定的是类的全限定名，单独指定类名是行不通的。比如 `value.deserializer = StringDeserializer` 是不行的，必须是 `value.deserializer = org.apache.kafka.common.serialization.StringDeserializer` 这样的形式。这一规定对自定义 `deserializer` 也适用。

2. 构造 KafkaConsumer 对象

设置好上述 4 个参数后，我们就可以开始构造 `KafkaConsumer` 对象了。`KafkaConsumer` 是 `consumer` 的主入口，所有的功能基本上都是由 `KafkaConsumer` 类提供的。创建 `KafkaConsumer` 实例很简单，只需要下面一句代码即可：

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

创建 `KafkaConsumer` 也可同时指定 `key` 和 `value` 的 `deserializer`：

```
KafkaConsumer consumer = new KafkaConsumer(props, new StringDeserializer(),
new StringDeserializer());
```

若采用这样的方式创建 `consumer`，则不需要显式地在 `Properties` 中指定 `key.deserializer` 和 `value.deserializer`。

3. 订阅 topic 列表

这一步使用 `KafkaConsumer.subscribe` 方法订阅 `consumer group` 要消费的 `topic` 列表。在上面的示例代码中我们只订阅了一个 `topic`——`test-topic`。如果要订阅多个 `topic`，可以使用下面类似的代码：

```
consumer.subscribe(Arrays.asList("topic1", "topic2", "topic3"));
```

该方法还支持正则表达式。假设 `consumer group` 要消费所有以 `kafka` 开头的 `topic`，则可以如此订阅：

```
consumer.subscribe(Pattern.compile("kafka.*"), new
NoOpConsumerRebalanceListener());
```

`subscribe` 方法的第二个参数是一个实现了 `org.apache.kafka.clients.consumer.ConsumerRebalanceListener` 接口的实现类。该接口主要用于 `rebalance`，我们会在本章后续内容

中探讨该类的使用方法。这里的 `NoOpConsumerRebalanceListener` 也实现了该接口，但却什么都不做。

需要特别注意的是，`subscribe` 方法不是增量式的，这意味着后续的 `subscribe` 调用会完全覆盖之前的订阅语句。比如：

```
consumer.subscribe(Arrays.asList("topic1", "topic2", "topic3"));
consumer.subscribe(Arrays.asList("topic4", "topic5", "topic6"));
```

连续执行这两个 `subscribe` 语句会令 `consumer group` 的订阅 `topic` 列表被设置为 `topic4`、`topic5` 和 `topic6`。

4. 获取消息

这是 `consumer` 的关键方法。`consumer` 使用 `KafkaConsumer.poll` 方法从订阅 `topic` 中并行地获取多个分区的信息。为了实现这一点，新版本 `consumer` 的 `poll` 方法使用了类似于 Linux 的 `select I/O` 机制——所有相关的事件（包括 `rebalance`、获取消息等）都发生在一个事件循环（`event loop`）中。这样 `consumer` 端只使用一个线程就能够完成所有类型的 I/O 操作。

一个常见的 `event loop` 写法如下：

```
try {
    while (running) {
        ConsumerRecords<String, String> records = consumer.poll(1000);
        // 执行具体的消费逻辑
    }
} finally {
    consumer.close();
}
```

上面代码中最关键的调用方法当属 `consumer.poll(1000)`。这里的 `1000` 是一个超时设定（`timeout`）。通常情况下如果 `consumer` 拿到了足够多的可用数据，那么它可以立即从该方法返回；但若当前没有足够多的数据可供返回，`consumer` 会处于阻塞状态。这个超时参数即控制阻塞的最大时间。这里的 `1000` 表示即使没有那么多数据，`consumer` 最多也不要等待超过 1 秒的时间。

这个超时设定给予了用户能够在 `consumer` 消费间隔之余做一些其他事情的能力。若用户有定时方面的需求，那么根据需求设定 `timeout` 是一个不错的选择。否则，设定一个比较大的值甚至 `Integer.MAX_VALUE`，是不错的建议。

5. 处理 `ConsumerRecord` 对象

上一步的 `poll` 调用返回以 `ConsumerRecord` 类封装的 `Kafka` 消息。拿到这些消息后

consumer 通常都包含处理这些消息的逻辑，毕竟 consumer 的目的不仅是要从 Kafka 处读取消息，还要对获取到的消息进行有意义的业务级处理。

令人有些意外的是，这里面可能会隐藏着某些误区：到底哪一步算作 consumer 端的消费？是调用 poll 方法这一步？还是处理 ConsumerRecord 对象这一步？抑或是两者加起来？有很多 Kafka 初学者经常会抱怨：我的 consumer 处理速度太慢了或我的 consumer 卡住了。那么显然，针对这样的问题我们首先必须要搞清楚 consumer 的消费具体指的是哪个环节。

从 Kafka consumer 的角度而言，poll 方法返回即认为 consumer 成功消费了消息，这可能和我们用户的观点有些不太一致，毕竟我们通常认为执行完真正的业务级处理之后才能算消费完毕。因此要回答上面的问题就必须明确 consumer 的瓶颈在哪里并根据结果有针对性地进行改进。如果你发现 poll 返回消息的速度过慢，那么可以调节相应的参数来提升 poll 方法的效率；若消息的业务级处理逻辑过慢，则应该考虑简化处理逻辑或者把处理逻辑放入单独的线程执行。

本节的示例代码中仅仅打印了消息的各种信息，包括消息的位移、key 和 value。实际使用中，用户需要替换成自己的业务处理逻辑。

6. 关闭 consumer

consumer 程序结束后一定要显式关闭 consumer 以释放 KafkaConsumer 运行过程中占用的各种系统资源（比如线程资源、内存、Socket 连接等）。关闭方法有两种，通常来说使用任何一种方法都是可行的。

- `KafkaConsumer.close()`：关闭 consumer 并最多等待 30 秒。
- `KafkaConsumer.close(timeout)`：关闭 consumer 并最多等待给定的 timeout 秒。

5.2.2 consumer 脚本命令

除了自己写程序构建 consumer 之外，Kafka 还自带了方便使用的控制台 consumer 脚本用于日常的调试验证。该脚本的名字叫 `kafka-console-consumer`，在 Linux 平台下它位于 `<kafka 目录>/bin` 下，在 Windows 平台下它位于 `<kafka 目录>/bin/windows` 下。

`kafka-console-consumer` 脚本常见的使用参数如下。

- `--bootstrap-servers`：指定 Kafka broker 列表，多台 broker 则以逗号分隔。这和前面 Java API 中的 `bootstrap.servers` 参数有相同的含义。讲到这里，不得不说说 Kafka 社区对于所有命令行脚本的参数命名统一的问题。目前来说，Kafka 所有命令行脚本表示相同含义的参数都不是统一的名字，使用起来非常不方便。比如这个 consumer 脚本中的名字是 `bootstrap-server`，到了 producer 脚本中变成了 `broker-list`，而其他脚本中可能又重

新使用了 `bootstrap.servers`，实在是令人眼花缭乱。读者实际使用过程中一定要仔细分辨。

- `--topic`: 指定要消费的 topic 名。
- `--from-beginning`: 是否指定从头消费。指定该参数与在 Java API 中设置 `auto.offset.reset=earliest` 是一样的效果。

当然，`kafka-console-consumer` 脚本的参数还有很多，但上面 3 个是最基本的参数设置，它们足以应付大多数的需求。各位读者可以不加参数地运行 `kafka-console-consumer` 脚本以获取完整的参数列表及其使用方法。一个典型的 `kafka-console-consumer` 脚本调用命令如下：

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

5.2.3 consumer 主要参数

我们在 5.2.1 节中提到了构建 consumer 所必需的 4 个参数 `bootstrap.servers`、`group.id`、`key.deserializer` 和 `value.deserializer`。Java 版本 consumer 还提供了很多其他非常重要的参数。本节将选取其中一些有代表性的参数进行详细讨论。完整的参数列表及其含义和默认值可以参见 <https://kafka.apache.org/documentation/#newconsumerconfigs>

`session.timeout.ms`

非常重要的参数之一！很多 Kafka 初学者搞不清楚到底这个参数是做什么用的，下面就来详细探讨一下。简单来说，`session.timeout.ms` 是 consumer group 检测组内成员发送崩溃的时间。假设你设置该参数为 5 分钟，那么当某个 group 成员突然崩溃了（比如被 `kill -9` 或宕机），管理 group 的 Kafka 组件（即消费者组协调者，也称 group coordinator，第 6 章中会详细讨论 coordinator）有可能需要 5 分钟才能感知到这个崩溃。显然我们想要缩短这个时间，让 coordinator 能够更快地检测到 consumer 失败。遗憾的是，这个参数还有另外一重含义：consumer 消息处理逻辑的最大时间——倘若 consumer 两次 poll 之间的间隔超过了该参数所设置的阈值，那么 coordinator 就会认为这个 consumer 已经追不上组内其他成员的消费进度了，因此会将该 consumer 实例“踢出”组，该 consumer 负责的分区也会被分配给其他 consumer。在最好的情况下，这会导致不必要的 rebalance，因为 consumer 需要重新加入 group。更糟的是，对于那些在被踢出 group 后处理的消息，consumer 都无法提交位移——这就意味着这些消息在 rebalance 之后会被重新消费一遍。如果一条消息或一组消息总是需要花费很长的时间处理，那么 consumer 甚至无法执行任何消费，除非用户重新调整参数。

这似乎给用户出了一个很大的难题。鉴于以上的“窘境”，Kafka 社区于 0.10.1.0 版本对

该参数的含义进行了拆分。在该版本及以后的版本中，`session.timeout.ms` 参数被明确为“coordinator 检测失败的时间”。因此在实际使用中，用户可以为该参数设置一个比较小的值让 coordinator 能够更快地检测 consumer 崩溃的情况，从而更快地开启 rebalance，避免造成更大的消费滞后（consumer lag）。目前该参数的默认值是 10 秒。

`max.poll.interval.ms`

如前所述，`session.timeout.ms` 中“consumer 处理逻辑最大时间”的含义被剥离出来了，Kafka 为这部分含义单独开放了一个参数——`max.poll.interval.ms`。在一个典型的 consumer 使用场景中，用户对于消息的处理可能需要花费很长时间。这个参数就是用于设置消息处理逻辑的最大时间的。假设用户的业务场景中消息处理逻辑是把消息“落地”到远程数据库中，且这个过程平均处理时间是 2 分钟，那么用户仅需要将 `max.poll.interval.ms` 设置为稍稍大于 2 分钟的值即可，而不必为 `session.timeout.ms` 也设置这么大的值。

通过将该参数设置成实际的逻辑处理时间再结合较低的 `session.timeout.ms` 参数值，consumer group 既实现了快速的 consumer 崩溃检测，也保证了复杂的事件处理逻辑不会造成不必要的 rebalance。

`auto.offset.reset`

指定了无位移信息或位移越界（即 consumer 要消费的消息的位移不在当前消息日志的合理区间范围）时 Kafka 的应对策略。特别要注意这里的无位移信息或位移越界，只有满足这两个条件中的任何一个时该参数才有效果。关于这一点，我们举一个实际的例子来说明。假设你首次运行一个 consumer group 并且指定从头消费。显然该 group 会从头消费所有数据，因为此时该 group 还没有任何位移信息。一旦该 group 成功提交位移后，你重启了 group，依然指定从头消费。此时你会发现该 group 并不会真的从头消费——因为 Kafka 已经保存了该 group 的位移信息，因此它会无视 `auto.offset.reset` 的设置。

目前该参数有如下 3 个可能的取值。

- `earliest`: 指定从最早的位移开始消费。注意这里最早的位移不一定就是 0。
- `latest`: 指定从最新处位移开始消费。
- `none`: 指定如果未发现位移信息或位移越界，则抛出异常。笔者在实际使用过程中几乎从未见过将该参数设置为 `none` 的用法，因此该值在真实业务场景中使用甚少。

`enable.auto.commit`

该参数指定 consumer 是否自动提交位移。若设置为 `true`，则 consumer 在后台自动提交位移；否则，用户需要手动提交位移。对于有较强“精确处理一次”语义需求的用户来说，最好

将该参数设置为 `false`，由用户自行处理位移提交问题。

`fetch.max.bytes`

一个经常被忽略的参数。它指定了 `consumer` 端单次获取数据的最大字节数。若实际业务消息很大，则必须要设置该参数为一个较大的值，否则 `consumer` 将无法消费这些消息。

`max.poll.records`

该参数控制单次 `poll` 调用返回的最大消息数。比较极端的做法是设置该参数为 1，那么每次 `poll` 只会返回 1 条消息。如果用户发现 `consumer` 端的瓶颈在 `poll` 速度太慢，可以适当增加该参数的值。如果用户的消息处理逻辑很轻量，默认的 500 条消息通常不能满足实际的消息处理速度。

`heartbeat.interval.ms`

该参数和 `request.timeout.ms`、`max.poll.interval.ms` 参数是最难理解的 `consumer` 参数。前面已经讨论了后两个参数的含义，这里解析一下 `heartbeat.interval.ms` 的含义及用法。

从表面上看，该参数似乎是心跳的间隔时间，但既然已经有了上面的 `session.timeout.ms` 用于设置超时，为何还要引入这个参数呢？这里的关键在于要搞清楚 `consumer group` 的其他成员如何得知要开启新一轮 `rebalance`——当 `coordinator` 决定开启新一轮 `rebalance` 时，它会将这个决定以 `REBALANCE_IN_PROGRESS` 异常的形式“塞进”`consumer` 心跳请求的 `response` 中，这样其他成员拿到 `response` 后才能知道它需要重新加入 `group`。显然这个过程越快越好，而 `heartbeat.interval.ms` 就是用来做这件事情的。

比较推荐的做法是设置一个比较低的值，让 `group` 下的其他 `consumer` 成员能够更快地感知新一轮 `rebalance` 开启了。注意，该值必须小于 `session.timeout.ms`！这很容易理解，毕竟如果 `consumer` 在 `session.timeout.ms` 这段时间内都不发送心跳，`coordinator` 就会认为它已经 `dead`，因此也就没有必要让它知晓 `coordinator` 的决定了。

`connections.max.idle.ms`

这又是一个容易忽略的参数！经常有用户抱怨在生产环境下周期性地观测到请求平均处理时间在飙升，这很有可能是因为 `Kafka` 会定期地关闭空闲 `Socket` 连接导致下次 `consumer` 处理请求时需要重新创建连向 `broker` 的 `Socket` 连接。当前默认值是 9 分钟，如果用户实际环境中不在乎这些 `Socket` 资源开销，比较推荐设置该参数值为 -1，即不要关闭这些空闲连接。

5.3 订阅 topic

5.3.1 订阅 topic 列表

新版本 consumer 中 consumer group 订阅 topic 列表非常简单，使用下面的语句即可实现：

```
consumer.subscribe(Arrays.asList("topic1", "topic2", "topic3"));
```

如果是使用独立 consumer（standalone consumer），则可以使用下面的语句实现手动订阅：

```
TopicPartition tp1 = new TopicPartition("topic-name", 0);
TopicPartition tp2 = new TopicPartition("topic-name", 1);
consumer.assign(Arrays.asList(tp1, tp2));
```

不管是哪种方法，consumer 订阅是延迟生效的，即订阅信息只有在下次 poll 调用时才会正式生效。如果在 poll 之前打印订阅信息，用户会发现它的订阅信息是空的，表明尚未生效。

5.3.2 基于正则表达式订阅 topic

除了 topic 列表订阅方式，新版本 consumer 还支持基于正则表达式的订阅方式。利用正则表达式可以达到某种程度上的动态订阅效果。一个实际的例子如下：

```
consumer.subscribe(Pattern.compile("kafka-.*"), new
ConsumerRebalanceListener()...);
```

使用基于正则表达式的订阅就必须指定 ConsumerRebalanceListener。该类是一个回调接口，用户需要通过实现这个接口来实现 consumer 分区分配方案发生变更时的逻辑。如果用户使用的是自动提交（即设置 enable.auto.commit=true），则通常不用理会这个类，使用下列的实现类就可以了：

```
consumer.subscribe(Pattern.compile("kafka-.*"), new
NoOpConsumerRebalanceListener());
```

但是，如果用户是手动提交位移的，则至少要在 ConsumerRebalanceListener 实现类的 onPartitionsRevoked 方法中处理分区分配方案变更时的位移提交。

5.4 消息轮询

5.4.1 poll 内部原理

归根结底，Kafka 的 consumer 是用来读取消息的，而且要能够同时读取多个 topic 的多个分区的消息。若要实现并行的消息读取，一种方法是使用多线程的方式，为每个要读取的分区

都创建一个专有的线程去消费（这其实就是旧版本 `consumer` 采用的方式，本章后面会专门讨论旧版本 `consumer` 的设计与使用）；另一种方法是采用类似于 Linux I/O 模型的 `poll` 或 `select` 等，使用一个线程来同时管理多个 `Socket` 连接，即同时与多个 `broker` 通信实现消息的并行读取——这就是新版本 `consumer` 最重要的设计改变。

一旦 `consumer` 订阅了 `topic`，所有的消费逻辑包括 `coordinator` 的协调、消费者组的 `rebalance` 以及数据的获取都会在主逻辑 `poll` 方法的一次调用中被执行。这样用户很容易使用一个线程来管理所有的 `consumer` I/O 操作。

有很多 Kafka 初学者对新版本 `consumer` 到底是否是多线程程序感到困惑。这里明确给出确定的答案：截止到目前，对最新版本的 Kafka（1.0.0）而言，新版本 Java `consumer` 是一个多线程或者说是一个双线程的 Java 进程——创建 `KafkaConsumer` 的线程被称为用户主线程，同时 `consumer` 在后台会创建一个心跳线程，该线程被称为后台心跳线程。`KafkaConsumer` 的 `poll` 方法在用户主线程中运行。这也同时表明：消费者组执行 `rebalance`、消息获取、`coordinator` 管理、异步任务结果的处理甚至位移提交等操作都是运行在用户主线程中的。因此仔细调优这个 `poll` 方法相关的各种处理超时时间参数至关重要。读者可以复习前面“`consumer` 主要参数”一节的内容以了解这些参数的使用方法。

5.4.2 poll 使用方法

`consumer` 订阅 `topic` 之后通常以事件循环的方式来获取订阅方案并开启消息读取。听上去似乎有些复杂，但其实用户要做的仅仅是写一个循环，然后重复性地调用 `poll` 方法。剩下所有的工作都交给 `poll` 方法帮用户完成。每次 `poll` 方法返回的都是订阅分区上的一组消息。当然如果某些分区没有准备好，某次 `poll` 返回的就是空的消息集合。下面的一段代码展示了常见的 `poll` 调用方式：

```
try {
    while (isRunning) {
        ConsumerRecords<String, String> records = consumer.poll(1000);
        for (ConsumerRecord<String, String> record : records)
            System.out.printf("topic = %s, partition=%d, offset=%d",
record.topic(), record.partition(), record.offset());
    }
} finally {
    consumer.close();
}
```

`poll` 方法根据当前 `consumer` 的消费位移返回消息集合。当 `poll` 首次被调用时，新的消费者组会被创建并根据对应的位移重设策略（`auto.offset.reset`）来设定消费者组的位移。一旦

consumer 开始提交位移，每个后续的 `rebalance` 完成后都会将位置设置为上次已提交的位移。传递给 `poll` 方法的超时设定参数用于控制 consumer 等待消息的最大阻塞时间。由于某些原因，broker 端有时候无法立即满足 consumer 端的获取请求（比如 consumer 要求至少一次获取 1MB 的数据，但 broker 端无法立即全部给出），那么此时 consumer 端将会阻塞以等待数据不断累积并最终满足 consumer 需求。如果用户不想让 consumer 一直处于阻塞状态，则需要给定一个超时时间。因此 `poll` 方法返回满足以下任意一个条件即可返回。

- 要么获取了足够多的可用数据。
- 要么等待时间超过了指定的超时设置。

前面我们谈到了 consumer 是单线程的设计理念（这里暂不考虑后台心跳线程，因为它只是一个辅助线程，并没有承担过重的消费逻辑），因此 consumer 就应该运行在它专属的线程中。新版本 Java consumer 不是线程安全的！如果没有显式地同步锁保护机制，Kafka 会抛出 `KafkaConsumer is not safe for multi-threaded access` 异常。如果用户在调用 `poll` 方法时看到了这样的报错，通常说明用户将同一个 `KafkaConsumer` 实例用在了多个线程中。至少对于目前的 Kafka 设计而言，这是不被允许的，用户最好不要这样使用。

另外在上面的代码中，我们在 `while` 的条件语句中指定了一个布尔变量值 `isRunning` 来标识是否要退出 consumer 消费循环并结束 consumer 应用。具体的做法是，将 `isRunning` 标识为 `volatile` 型，然后在其他线程中设置 `isRunning = false` 来控制 consumer 的结束。最后千万不要忘记关闭 consumer。这不仅可以清除 consumer 创建的各种 `Socket` 资源，还会通知消费者组 coordinator 主动离组从而更快地开启新一轮 `rebalance`。比较推荐的做法是，在 `finally` 代码块中显式调用 `consumer.close()`，从而保证 consumer 总是能够被关闭的。

`KafkaConsumer` 的 `poll` 方法为什么会有一个超时的参数？实际使用中应该根据什么规则来设定此值？要回答这些问题，我们需要思考一下引入这个参数的初衷。诚然，它是超时的设定，但 Kafka 社区引入这个参数的目的其实是想让 consumer 程序有机会定期“醒来”去做一些其他的事情。假设用户的 consumer 程序除了消费之外还需要定期地执行其他的常规任务（比如每隔 10 秒需要将消费情况记录到日志中），那么用户就可以使用 `consumer.poll(10000)` 来让 consumer 有机会在等待 Kafka 消息的同时还能够定期执行其他任务。这就是使用超时设定的最大意义。

另一方面，若用 consumer 程序除了消费消息之外没有其他的定时任务需要执行，即 consumer 程序唯一的目的是从 Kafka 获取消息然后进行处理，那么用户可采用与上面代码完全不同的 `poll` 调用方法，如下面的代码所示：

```
try {
    while (true) {
        ConsumerRecords<String, String> records = consumer.poll(Long.MAX_VALUE);
```

```
        for (ConsumerRecord<String, String> record : records)
            System.out.printf("topic = %s, partition=%d, offset=%d", record.
topic(), record.partition(), record.offset());
        }
    } catch (WakeupException e) {
        // 此处忽略此异常的处理
    } finally {
        consumer.close();
    }
}
```

请注意上面这段代码与前一段代码的区别：`consumer.poll(Long.MAX_VALUE)`；。在这段代码中我们让 `consumer` 程序在未获取到足够多数据时无限等待，然后通过捕获 `WakeupException` 异常来判断 `consumer` 是否结束。显然，这是与第一种调用方法完全不同的使用思想。

如果使用这种方式调用 `poll`，那么需要在另一个线程中调用 `consumer.wakeup()` 方法来触发 `consumer` 的关闭。前面我们说过，`KafkaConsumer` 不是线程安全的，但是有一个例外：用户可以安全地在另一个线程中调用 `consumer.wakeup()`。注意，只有 `wakeup` 方法是特例，其他 `KafkaConsumer` 方法都不能同时在多线程中使用。

`WakeupException` 异常是在 `poll` 方法中被抛出的，因此如果当前事件循环代码正在执行 `poll` 之后的消息处理逻辑，则它并不会马上响应 `wakeup`，只会等待下次 `poll` 调用时才进行响应。举一个实际的例子，假设每次 `poll` 返回消息后，用户 `consumer` 程序都需要为这些消息执行很繁重的计算工作，那么在计算过程中该 `consumer` 是不会响应另一个线程调用的 `wakeup` 的，它只能在下次 `poll` 时才响应。所以程序表现为不能立即退出，会有一段延迟时间。这也是为什么不推荐用户将很繁重的消息处理逻辑放入 `poll` 主线程执行的原因。

说了这么多，我们简单总结一下 `poll` 的使用方法。

- `consumer` 需要定期执行其他子任务：推荐 `poll`（较小超时时间）+ 运行标识布尔变量的方式。
- `consumer` 不需要定期执行子任务：推荐 `poll(MAX_VALUE)` + 捕获 `WakeupException` 的方式。

5.5 位移管理

本章已然零散地提到过位移或 `offset` 了，本节着重讨论 `consumer` 的位移管理，特别是 `consumer group` 对于位移的管理。

5.5.1 consumer 位移

consumer 端需要为每个它要读取的分区保存消费进度，即分区中当前最新消费消息的位置。该位置就被称为位移（offset）。consumer 需要定期地向 Kafka 提交自己的位置信息，实际上，这里的位移值通常是下一条待消费的消息的位置。假设 consumer 已经读取了某个分区中的第 N 条消息，那么它应该提交位移值为 N ，因为位移是从 0 开始的，位移为 N 的消息是第 $N+1$ 条消息。这样下次 consumer 重启时会从第 $N+1$ 条消息开始消费。总而言之，offset 就是 consumer 端维护的位置信息。

offset 对于 consumer 非常重要，因为它是实现消息交付语义保证（message delivery semantic）的基石。常见的 3 种消息交付语义保证如下。

- 最多一次（at most once）处理语义：消息可能丢失，但不会被重复处理。
- 最少一次（at least once）处理语义：消息不会丢失，但可能被处理多次。
- 精确一次（exactly once）处理语义：消息一定会被处理且只会被处理一次。

显然，若 consumer 在消息消费之前就提交位移，那么便可以实现 at most once——因为若 consumer 在提交位移与消息消费之间崩溃，则 consumer 重启后会从新的 offset 位置开始消费，前面的那条消息就丢失了。相反地，若提交位移在消息消费之后，则可实现 at least once 语义。由于 Kafka 没有办法保证这两步操作可以在同一个事务中完成，因此 Kafka 默认提供的就是 at least once 的处理语义。好消息是 Kafka 社区已于 0.11.0.0 版本正式支持事务以及精确一次处理语义。

既然 offset 本质上就是一个位置信息，那么就需要和其他一些位置信息区别开来。图 5.6 给出了与 consumer 相关的多个位置信息。



图 5.6 consumer 各种位置信息

下面分别来介绍它们的含义。

- 上次提交位移（last committed offset）：consumer 最近一次提交的 offset 值。
- 当前位置（current position）：consumer 已读取但尚未提交时的位置。
- 水位（watermark）：也被称为高水位（high watermark），严格来说它不属于

consumer 管理的范围，而是属于分区日志的概念。对于处于水位之下（或者说图 5.6 中位于水位左边）的所有消息，consumer 都是可以读取的，consumer 无法读取水位以上（图 5.6 中位于水位右边）的消息。

- 日志终端位移（Log End Offset, LEO）：也被称为日志最新位移。同样不属于 consumer 范畴，而是属于分区日志管辖。它表示了某个分区副本当前保存消息对应的最大的位移值。值得注意的是，正常情况下 LEO 不会比水位值小。事实上，只有分区所有副本都保存了某条消息，该分区的 leader 副本才会向上移动水位值。

由于本节讨论 consumer 的位移，我们只需要关心图 5.6 中 offset 和当前位置的含义以及关系就可以了。再次强调一下，consumer 最多只能读取到水位值标记的消息，而不能读取尚未完全被“写入成功”的消息，即位于水位值之上的消息。

5.5.2 新版本 consumer 位移管理

consumer 会在 Kafka 集群的所有 broker 中选择一个 broker 作为 consumer group 的 coordinator，用于实现组成员管理、消费分配方案制定以及提交位移等。为每个组选择对应 coordinator 的依据就是，5.1.6 节中介绍的内部 topic(`__consumer_offsets`)。和普通的 Kafka topic 相同，该 topic 配置有多个分区，每个分区有多个副本。它存在的唯一目的就是保存 consumer 提交的位移。

当消费者组首次启动时，由于没有初始的位移信息，coordinator 必须为其确定初始位移值，这就是 consumer 参数 `auto.offset.reset` 的作用。通常情况下，consumer 要么从最早的位移开始读取，要么从最新的位移开始读取。

当 consumer 运行了一段时间之后，它必须要提交自己的位移值。如果 consumer 崩溃或被关闭，它负责的分区就会被分配给其他 consumer，因此一定要在其他 consumer 读取这些分区前就做好位移提交工作，否则会出现消息的重复消费。

consumer 提交位移的主要机制是通过向所属的 coordinator 发送位移提交请求来实现的。每个位移提交请求都会往 `__consumer_offsets` 对应分区上追加写入一条消息。消息的 key 是 group.id、topic 和分区的元组，而 value 就是位移值。如果 consumer 为同一个 group 的同一个 topic 分区提交了多次位移，那么 `__consumer_offsets` 对应的分区上就会有若干条 key 相同但 value 不同的消息，但显然我们只关心最新一次提交的那条消息。从某种程度来说，只有最新提交的位移值是有效的，其他消息包含的位移值其实都已经过期了。Kafka 通过压实（compact）策略来处理这种消息使用模式。我们会在第 6 章中详细讨论这种策略以及如何应用这种策略到内部 topic 和位移管理上。

5.5.3 自动提交与手动提交

如前所述，位移提交策略对于提供消息交付语义至关重要。默认情况下，`consumer` 是自动提交位移的，自动提交间隔是 5 秒。这就是说若不做特定的设置，`consumer` 程序在后台自动提交位移。通过设置 `auto.commit.interval.ms` 参数可以控制自动提交的间隔。

自动位移提交的优势是降低了用户的开发成本使得用户不必亲自处理位移提交；劣势是用户不能细粒度地处理位移的提交，特别是在有较强精确一次处理语义时。在这种情况下，用户可以使用手动位移提交。

所谓的手动位移提交就是用户自行确定消息何时被真正处理完并可以提交位移。在一个典型的 `consumer` 应用场景中，用户需要对 `poll` 方法返回的消息集合中的消息执行业务级的处理。用户想要确保只有消息被真正处理完成后再提交位移。如果使用自动位移提交则无法保证这种时序性，因此在这种情况下必须使用手动提交位移。设置使用手动提交位移非常简单，仅仅需要在构建 `KafkaConsumer` 时设置 `enable.auto.commit=false`，然后调用 `commitSync` 或 `commitAsync` 方法即可。一段典型的手动提交代码如下：

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("group.id", "test-group");
props.put("enable.auto.commit", "false");
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<>(props);
consumer.subscribe(Arrays.asList("test-topic"));
final int minBatchSize = 500;
List<ConsumerRecord<String, String>> buffer = new ArrayList<>();
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(1000);
    for (ConsumerRecord<String, String> record : records) {
        buffer.add(record);
    }
    if (buffer.size() >= minBatchSize) {
        insertIntoDb(buffer);
        consumer.commitSync();
        buffer.clear();
    }
}
```

上面的代码中 `consumer` 持续消费一批消息并把它们加入一个缓冲区中。当积累了足够多

的消息（本例为 500 条）便统一插入到数据库中。只有被成功插入到数据库之后，这些消息才算是真正被处理完。此时调用 `KafkaConsumer.commitSync` 方法进行手动位移提交，然后清空缓冲区以备缓存下一批消息。若在成功插入数据库之后但提交位移语句执行之前 `consumer` 程序崩溃，由于未成功提交位移，`consumer` 重启后会重新处理之前的一批消息并将它们再次插入到数据库中，从而造成消息多次被消费。

表 5.2 总结了 `consumer` 自动提交与手动提交之间的特点比较。

表 5.2 自动提交与手动提交的比较

	使用方法	优势	劣势	交付语义保证	使用场景
自动提交	默认不用配置或显式设置 <code>enable.auto.commit=true</code>	开发成本低，简单易用	无法实现精确控制，位移提交失败后不易处理	可能造成消息丢失，最多实现“最少一次”处理语义	对消息交付语义无需求，容忍一定的消息丢失
手动提交	设置 <code>enable.auto.commit=false</code> ；手动调用 <code>commitSync</code> 或 <code>commitAsync</code> 提交位移	可精确控制位移提交行为	额外的开发成本，须自行处理位移提交	易实现“最少一次”处理语义，依赖外部状态可实现“精确一次”处理语义	消息处理逻辑重，不允许消息丢失，至少要求“最少一次”处理语义

手动提交位移 API 进一步细分为同步手动提交和异步手动提交，即 `commitSync` 和 `commitAsync` 方法。如果调用的是 `commitSync`，用户程序会等待位移提交结束才执行下一条语句命令。相反地，若是调用 `commitAsync`，则是一个异步非阻塞调用。`consumer` 在后续 `poll` 调用时轮询该位移提交的结果。特别注意的是，这里的异步提交位移不是指 `consumer` 使用单独的线程进行位移提交。实际上 `consumer` 依然会在用户主线程的 `poll` 方法中不断轮询这次异步提交的结果。只是该提交发起时此方法是不会阻塞的，因而被称为异步提交。

目前为止我们谈到的 `commitSync` 或 `commitAsync`，都是指它的无参版本。当用户调用 `consumer.commitSync()`或 `consumer.commitAsync()`时，`consumer` 会为所有它订阅的分区提交位移。`commitSync` 和 `commitAsync` 方法还有另外带参数的重载方法。用户调用这个方法时需要指定一个 `Map` 显式地告诉 `Kafka` 为哪些分区提交位移。实际使用过程中笔者更加推荐这个版本，因为 `consumer` 只对它所拥有的分区做提交是更合理的行为，而且 `consumer` 通常都有更加细粒度化的位移提交策略。一段典型的手动提交部分分区位移的代码如下：

```
try {
    while(running) {
        ConsumerRecords<String, String> records = consumer.poll(1000);
        for (TopicPartition partition : records.partitions()) {
            List<ConsumerRecord<String, String>> partitionRecords =
records.records(partition);
            for (ConsumerRecord<String, String> record : partitionRecords) {
                System.out.println(record.offset() + ": " + record.value());
            }
        }
    }
}
```

```

        }
        long lastOffset = partitionRecords.get(partitionRecords.
size() - 1).offset();
        consumer.commitSync(Collections.singletonMap(partition, new
OffsetAndMetadata(lastOffset + 1)));
    }
}
} finally {
    consumer.close();
}
}

```

上面的代码按照分区级别进行位移提交。它首先对 `poll` 方法返回的消息集合按照分区进行分组。然后每个分区下的消息待处理完成后构造一个 `Map` 对象统一提交位移，从而实现了细粒度控制位移提交。这里需要特别注意的是，提交的位移一定是 `consumer` 下一条待读取消息的位移。这也是为什么上面的代码在构造 `OffsetMetadata` 元数据时使用 `offset + 1` 的原因。

5.5.4 旧版本 consumer 位移管理

旧版本 `consumer` 的位移默认保存在 `ZooKeeper` 节点中，与 `__consumer_offsets` 完全没有关系。读者可以复习 5.1.4 节的内容了解具体保存位移的 `ZooKeeper` 节点位置。

旧版本 `consumer` 也区分自动提交位移和手动提交位移，只不过区分它们的参数名叫 `auto.commit.enable`，而不是新版本的 `enable.auto.commit`。各位读者切记不要混淆。另外旧版本 `consumer` 默认的提交间隔是 60 秒，而不是新版本的 5 秒。该间隔由参数 `auto.commit.interval.ms` 控制。

当设置成手动提交位移时，用户需要在 `consumer` 程序中显式调用 `ConsumerConnector.commitOffsets` 方法来提交位移。和新版本 `consumer` 类似，如果直接调用 `commitOffsets()`，则会为该 `consumer` 订阅的所有分区都提交位移；若是调用 `commitOffsets(Map)` 版本，则可以实现细粒度化的位移提交。本章 5.10 节“旧版本 `consumer`”中将给出该方法调用的具体示例。

5.6 重平衡 (rebalance)

5.6.1 rebalance 概览

`consumer group` 的 `rebalance` 本质上是一组协议，它规定了一个 `consumer group` 是如何达成一致来分配订阅 `topic` 的所有分区的。假设某个组下有 20 个 `consumer` 实例，该组订阅了一个有着 100 个分区的 `topic`。正常情况下，`Kafka` 会为每个 `consumer` 平均分配 5 个分区。这个分配过程就被称为 `rebalance`。当 `consumer` 成功地执行 `rebalance` 后，组订阅 `topic` 的每个分区只

会分配给组内的一个 consumer 实例。

和旧版本 consumer 依托于 ZooKeeper 进行 rebalance 不同，新版本 consumer 使用了 Kafka 内置的一个全新的组协调协议（group coordination protocol）。对于每个组而言，Kafka 的某个 broker 会被选举为组协调者（group coordinator）。coordinator 负责对组的状态进行管理，它的主要职责就是当新成员到达时促成组内所有成员达成新的分区分配方案，即 coordinator 负责对组执行 rebalance 操作。

5.6.2 rebalance 触发条件

组 rebalance 触发的条件有以下 3 个。

- 组成员发生变更，比如新 consumer 加入组，或已有 consumer 主动离开组，再或是已有 consumer 崩溃时则触发 rebalance。
- 组订阅 topic 数发生变更，比如使用基于正则表达式的订阅，当匹配正则表达式的新 topic 被创建时则会触发 rebalance。
- 组订阅 topic 的分区数发生变更，比如使用命令行脚本增加了订阅 topic 的分区数。

真实应用场景中引发 rebalance 最常见的原因就是违背了第一个条件，特别是 consumer 崩溃的情况。这里的崩溃不一定就是指 consumer 进程“挂掉”或 consumer 进程所在的机器宕机。当 consumer 无法在指定的时间内完成消息的处理，那么 coordinator 就认为该 consumer 已经崩溃，从而引发新一轮 rebalance。举一个真实的案例，笔者曾经碰到过一个 Kafka 线上环境，发现该环境中的 consumer group 频繁地进行 rebalance，但组内所有 consumer 程序都未出现崩溃的情况，另外消费者组的订阅情况也从未发生过变更。经过一番详细的分析，最后笔者定位了原因：该 group 下的 consumer 处理消息的逻辑过重，而且事件处理时间波动很大，非常不稳定，从而导致 coordinator 会经常性地认为某个 consumer 已经挂掉，引发 rebalance。而 consumer 程序中包含了错误重试的代码，使得落后过多的 consumer 会不断地申请重新加入组，最后表现为 coordinator 不停地对 group 执行 rebalance，极大地降低了 consumer 端的吞吐量。鉴于目前一次 rebalance 操作的开销很大，生产环境中用户一定要结合自身业务特点仔细调优 consumer 参数 request.timeout.ms、max.poll.records 和 max.poll.interval.ms，以避免不必要的 rebalance 出现。

5.6.3 rebalance 分区分配

之前提到过在 rebalance 时 group 下所有的 consumer 都会协调在一起共同参与分区分配，这是如何完成的呢？Kafka 新版本 consumer 默认提供了 3 种分配策略，分别是 range 策略、

round-robin 策略和 sticky 策略。

所谓的分配策略决定了订阅 topic 的每个分区会被分配给哪个 consumer。range 策略主要是基于范围的思想。它将单个 topic 的所有分区按照顺序排列，然后把这些分区划分成固定大小的分区段并依次分配给每个 consumer；round-robin 策略则会把所有 topic 的所有分区顺序摆开，然后轮询式地分配给各个 consumer。最新发布的 sticky 策略有效地避免了上述两种策略完全无视历史分配方案的缺陷，采用了“有黏性”的策略对所有 consumer 实例进行分配，可以规避极端情况下的数据倾斜并且在两次 rebalance 间最大限度地维持了之前的分配方案。

通常意义上认为，如果 group 下所有 consumer 实例的订阅是相同，那么使用 round-robin 会带来更公平的分配方案，否则使用 range 策略的效果更好。此外，sticky 策略在 0.11.0.0 版本才被引入，故目前使用的用户并不多。新版本 consumer 默认的分配策略是 range。用户根据 consumer 参数 `partition.assignment.strategy` 来进行设置。另外 Kafka 支持自定义的分配策略，用户可以创建自己的 consumer 分配器（assignor）。

针对 rebalance 过程中的分区分配，下面举一个简单的例子，加以说明。假设目前某个 consumer group 下有两个 consumer：A 和 B。当第 3 个成员 C 加入时，满足了前面谈到的第一个触发条件，因此 coordinator 会执行 rebalance，并根据 range 分配策略重新为 A、B 和 C 分配分区，如图 5.7 所示。

由此可见，原先 A 和 B 分别处理 3 个分区的数据，rebalance 之后 A、B 和 C 各自承担 2 个分区的消费，可以说这个分配方案非常公平，每个 consumer 上的负载是相同的。

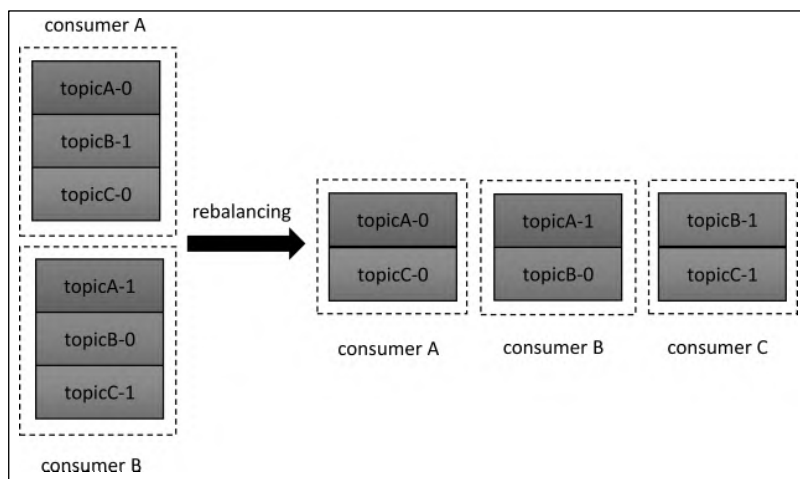


图 5.7 rebalance 执行分配

5.6.4 rebalance generation

某个 consumer group 可以执行任意次 rebalance。为了更好地隔离每次 rebalance 上的数据，新版本 consumer 设计了 rebalance generation 用于标识某次 rebalance。generation 这个词类似于 JVM 分代垃圾收集器中“分代”（严格来说，JVM GC 使用的是 generational）的概念。笔者把它翻译成“届”，表示 rebalance 之后的一届成员，在 consumer 中它是一个整数，通常从 0 开始。Kafka 引入 consumer generation 主要是为了保护 consumer group 的，特别是防止无效 offset 提交。比如上一届的 consumer 成员由于某些原因延迟提交了 offset，但 rebalance 之后该 group 产生了新一届的 group 成员，而这次延迟的 offset 提交携带的是旧的 generation 信息，因此这次提交会被 consumer group 拒绝。

很多 Kafka 用户在使用 consumer 时经常碰到的 ILLEGAL_GENERATION 异常就是这个原因导致的。事实上，每个 group 进行 rebalance 之后，generation 号都会加 1，表示 group 进入了一个新的版本。如图 5.8 所示，Generation 1 时 group 有 3 个成员，随后成员 2 退出组，coordinator 触发 rebalance，consumer group 进入到 Generation 2 时代，之后成员 4 加入，再次触发 rebalance，group 进入到 Generation 3 时代。

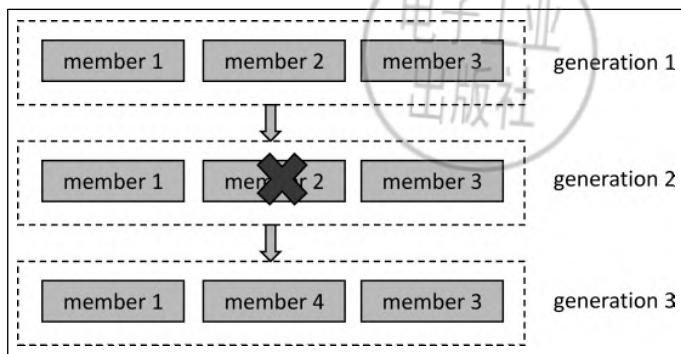


图 5.8 group generation 变更

5.6.5 rebalance 协议

前面提到过 rebalance 本质上是一组协议。group 与 coordinator 共同使用这组协议完成 group 的 rebalance。最新版本 Kafka 中提供了下面 5 个协议来处理 rebalance 相关事宜。

- JoinGroup 请求：consumer 请求加入组。
- SyncGroup 请求：group leader 把分配方案同步更新到组内所有成员中。
- Heartbeat 请求：consumer 定期向 coordinator 汇报心跳表明自己依然存活。

- **LeaveGroup 请求**: consumer 主动通知 coordinator 该 consumer 即将离组。
- **DescribeGroup 请求**: 查看组的所有信息, 包括成员信息、协议信息、分配方案以及订阅信息等。该请求类型主要供管理员使用。coordinator 不使用该请求执行 rebalance。

在 rebalance 过程中, coordinator 主要处理 consumer 发过来的 JoinGroup 和 SyncGroup 请求。当 consumer 主动离组时会发送 LeaveGroup 请求给 coordinator。

在成功 rebalance 之后, 组内所有 consumer 都需要定期地向 coordinator 发送 Heartbeat 请求。而每个 consumer 也是根据 Heartbeat 请求的响应中是否包含 REBALANCE_IN_PROGRESS 来判断当前 group 是否开启了新一轮 rebalance。

5.6.6 rebalance 流程

consumer group 在执行 rebalance 之前必须首先确定 coordinator 所在的 broker, 并创建与该 broker 相互通信的 Socket 连接。确定 coordinator 的算法与确定 offset 被提交到 __consumer_offsets 目标分区的算法是相同的。算法如下。

- 计算 $\text{Math.abs}(\text{groupID.hashCode}) \% \text{offsets.topic.num.partitions}$ 参数值 (默认是 50), 假设是 10。
- 寻找 __consumer_offsets 分区 10 的 leader 副本所在的 broker, 该 broker 即为这个 group 的 coordinator。

成功连接 coordinator 之后便可以执行 rebalance 操作。目前 rebalance 主要分为两步: 加入组和同步更新分配方案。

- **加入组**: 这一步中组内所有 consumer (即 group.id 相同的所有 consumer 实例) 向 coordinator 发送 JoinGroup 请求。当收集全 JoinGroup 请求后, coordinator 从中选择一个 consumer 担任 group 的 leader, 并把所有成员信息以及它们的订阅信息发送给 leader。特别需要注意的是, group 的 leader 和 coordinator 不是一个概念。leader 是某个 consumer 实例, coordinator 通常是 Kafka 集群中的一个 broker。另外 leader 而非 coordinator 负责为整个 group 的所有成员制定分配方案。
- **同步更新分配方案**: 这一步中 leader 开始制定分配方案, 即根据前面提到的分配策略决定每个 consumer 都负责哪些 topic 的哪些分区。一旦分配完成, leader 会把这个分配方案封装进 SyncGroup 请求并发送给 coordinator。比较有意思的是, 组内所有成员都会发送 SyncGroup 请求, 不过只有 leader 发送的 SyncGroup 请求中包含了分配方案。coordinator 接收到分配方案后把属于每个 consumer 的方案单独抽取出来作为 SyncGroup 请求的 response 返还给各自的 consumer。

图 5.9 和图 5.10 分别描述了加入组和同步分配方案的流程。

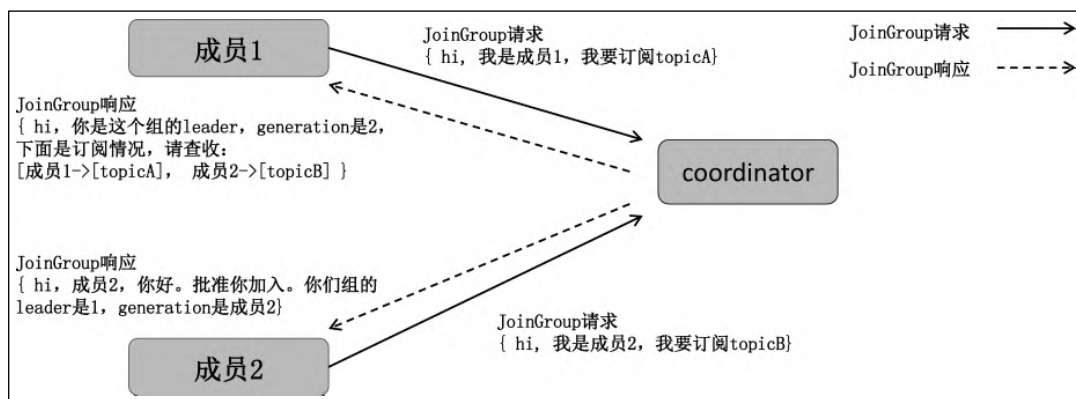


图 5.9 rebalance 加入组流程

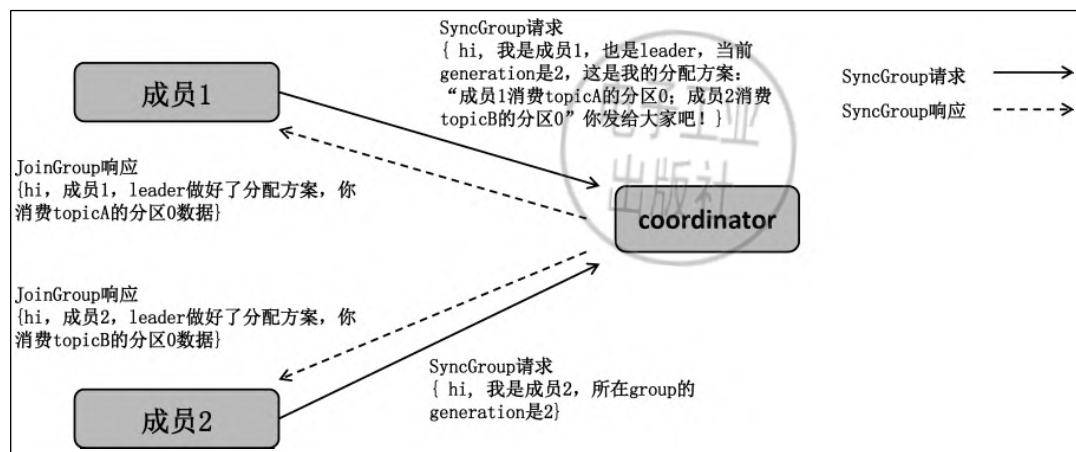


图 5.10 rebalance 同步分配方案流程

consumer group 分配方案是在 consumer 端执行的。Kafka 将这个权力下放给客户端主要是因为这样做可以有更好的灵活性。比如在这种机制下用户可以自行实现类似于 Hadoop 那样的机架感知（rack-aware）分配方案。同一个机架上的分区数据被分配给相同机架上的 consumer，减少网络传输的开销。而且，即使以后分区策略发生了变更，也只需要重启 consumer 应用即可，不必重启 Kafka 服务器。

5.6.7 rebalance 监听器

在位移提交章节中，提到过新版本 consumer 默认把位移提交到__consumer_offsets 中。其

实，Kafka 也支持用户把位移提交到外部存储中，比如数据库中。若要实现这个功能，用户就必须使用 `rebalance` 监听器。使用 `rebalance` 监听器的前提是用户使用 `consumer group`。如果使用的是独立 `consumer` 或是直接手动分配分区，那么 `rebalance` 监听器是无效的。

`rebalance` 监听器有一个主要的接口回调类 `ConsumerRebalanceListener`，里面就两个方法 `onPartitionsRevoked` 和 `onPartitionAssigned`。在 `coordinator` 开启新一轮 `rebalance` 前 `onPartitionsRevoked` 方法会被调用，而 `rebalance` 完成后会调用 `onPartitionsAssigned` 方法。

`rebalance` 监听器最常见的用法就是手动提交位移到第三方存储以及在 `rebalance` 前后执行一些必要的审计操作。下面笔者使用一个例子来演示如何使用 `rebalance` 监听器来向数据库提交位移并且统计 `group` 总的 `rebalance` 时间。代码如下：

```
public static void main(String[] args) {
    Properties props = new Properties();
    props.put("bootstrap.servers", "localhost:9092"); //必须指定
    props.put("group.id", "test-group"); //必须指定
    props.put("enable.auto.commit", "false");
    props.put("auto.offset.reset", "earliest");
    final KafkaConsumer consumer = new KafkaConsumer(props, new
StringDeserializer(), new StringDeserializer());

    final AtomicLong totalRebalanceTimeMs = new AtomicLong(0L);
    //总 rebalance 时长
    final AtomicLong joinStart = new AtomicLong(0L);
    consumer.subscribe(Arrays.asList("test-topic"), new
ConsumerRebalanceListener() {
        @Override
        public void onPartitionsRevoked(Collection<TopicPartition>
partitions) {
            for (TopicPartition tp : partitions) {
                saveOffsetInExternalStore(consumer.position(tp));
                //将该分区 offset 保存到外部存储
            }
            joinStart.set(System.currentTimeMillis());
        }

        @Override
        public void onPartitionsAssigned(Collection<TopicPartition>
partitions) {
            totalRebalanceTimeMs.addAndGet(System.currentTimeMillis()
- joinStart.get()); //更新总 rebalance 时长
            for (TopicPartition tp : partitions) {
                consumer.seek(tp, readOffsetFromExternalStore(tp));
```



```

        //从外部存储中读取该分区 offset
    }
}
});
try {
    while (true) {
        ConsumerRecords<String, String> records = consumer.poll(
(Long.MAX_VALUE);
        for (ConsumerRecord<String, String> record : records)
            System.out.printf("topic = %s, partition=%d, offset=%d",
record.topic(), record.partition(), record.offset());
        }
    } catch (WakeupException e) {
        //此处忽略此异常的处理
    } finally {
        System.out.println("Total rebalance time(ms): " +
totalRebalanceTimeMs.get());
        consumer.close();
    }
}
}

```

上面的代码中使用了一个 `totalRebalanceTimeMs` 变量来统计该 group 总的 rebalance 时间，具体方法是每次调用 `onPartitionsAssigned` 时更新该变量的值。另外代码调用了 `saveOffsetInExternalStore` 和 `readOffsetFromExternalStore` 方法，分别在 rebalance 前提交位移和 rebalance 后读取位移。`consumer` 的 `seek` 方法会将 `consumer` 当前位移指定到读取的位移处并从该位移处开始读取消息。

在这个例子中我们使用手动提交位移，而不是自动提交位移。很多用户可能会问，如果使用自动提交位移还需要在 rebalance 监听器中再提交位移吗？答案是不需要。`consumer` 每次 rebalance 时会检查用户是否启用了自动提交位移，如果是，它会帮用户执行提交，因此不需要用户在 rebalance 监听器中显式提交一遍。

鉴于 `consumer` 通常都要求 rebalance 在很短的时间内完成，用户千万不要在 rebalance 监听器的两个方法中放入执行时间很长的逻辑，特别是一些阻塞方法，如各种阻塞队列的 `take` 或 `poll` 等。

5.7 解序列化

5.7.1 默认解序列化器

解序列化（`deserializer`）或称反序列化与第 4 章中的序列化（`serializer`）是互逆的操作。

Kafka consumer 从 broker 端获取消息的格式是字节数组，consumer 需要把它还原回指定的对象类型，而这个对象类型通常都是与序列化对象类型一致的。比如 serializer 把一个字符串序列化成字节数组，consumer 使用对应的 deserializer 把字节数组还原回字符串。

Kafka 1.0.0 版本默认提供了多达十几种的 deserializer，常见的 deserializer 如下。

- ByteArrayDeserializer: 本质上什么都不做，因为已经是字节数组。
- ByteBufferDeserializer: 解序列化成 ByteBuffer。
- BytesDeserializer: 解序列化 Kafka 自定义的 Bytes 类。
- DoubleDeserializer: 解序列化 Double 类型。
- IntegerDeserializer: 解序列化 Integer 类型。
- LongDeserializer: 解序列化 Long 类型。
- StringDeserializer: 解序列化 String 类型。

由此可见，consumer 已经初步建立了解序列化机制来应对简单的消息类型，但若涉及复杂的类型（比如 Avro 或其他序列化框架），那么就需要用户自行定义 deserializer。

consumer 的序列化机制使用起来非常简单，只需要在构造 consumer 时同时指定参数 key.deserializer 和 value.deserializer 的值即可，如下列代码所示：

```
props.put("key.deserializer", "org.apache.kafka.common.serialization.
IntegerSerializer");
props.put("value.deserializer", "org.apache.kafka.common.serialization.
StringSerializer");
// 或者
props.put(ProducerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
"org.apache.kafka.common.serialization.IntegerDeserializer");
props.put(ProducerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
"org.apache.kafka.common.serialization.StringDeserializer");
```

5.7.2 自定义解序列化器

Kafka 支持用户自定义消息的 deserializer。成功编写一个自定义的 deserializer 需要完成以下 3 件事情。

- 定义或复用 serializer 的数据对象格式。
- 创建自定义 deserializer 类，令其实现 org.apache.kafka.common.serialization.Deserializer 接口。在 deserializer 方法中实现 deserialize 逻辑。

- 在构造 `KafkaConsumer` 的 `Properties` 对象中设置 `key.deserializer` 和（或）`value.deserializer` 为上一步的实现类。

我们依然使用第 4 章序列化相关章节中的 `User` 例子来实现自定义的 `deserializer`。代码如下。

`UserDeserializer` 类：

```
public class UserDeserializer implements Deserializer {

    private ObjectMapper objectMapper;

    @Override
    public void configure(Map configs, boolean isKey) {
        objectMapper = new ObjectMapper();
    }

    @Override
    public User deserialize(String topic, byte[] data) {
        User user = null;
        try {
            user = objectMapper.readValue(data, User.class);
        } finally {
            return user;
        }
    }

    @Override
    public void close() {}
}
```

然后构建 `KafkaConsumer` 时指定 `value.deserializer` 参数：

```
props.put("value.deserializer", "huxi.test.consumer.UserDeserializer");
```

5.8 多线程消费实例

如前所述，`KafkaConsumer` 是非线程安全的。它和 `KafkaProducer` 不同，后者是线程安全的，因此用户可以在多个线程中放心地使用同一个 `KafkaProducer` 实例，事实上这也是社区推荐的 `producer` 使用方法，因为通常它比每个线程维护一个 `KafkaProducer` 实例效率要高。

但是对于 `consumer` 而言，用户无法直接在多个线程中共享一个 `KafkaConsumer` 实例。那么应该如何实现多线程 `consumer` 消费呢？本节给出两种多线程消费的方法以及各自的实例。

5.8.1 每个线程维护一个 KafkaConsumer

在这个方法中，用户创建多个线程来消费 topic 数据。每个线程都会创建专属于该线程的 KafkaConsumer 实例，如图 5.11 所示。

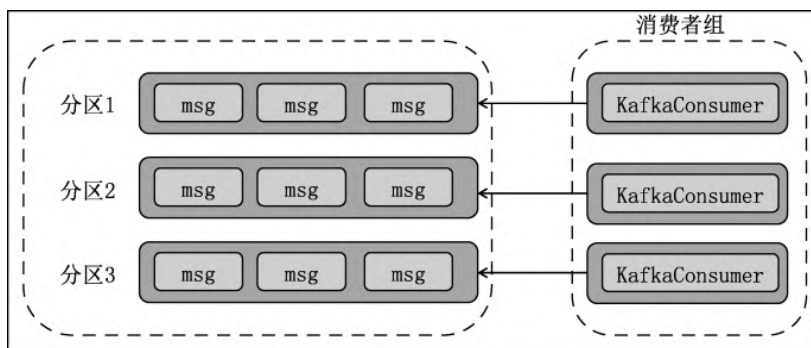


图 5.11 多线程维护专属 KafkaConsumer

由图 5.11 可知，consumer group 由多个线程的 KafkaConsumer 组成，每个线程负责消费固定数目的分区。下面给出一个完整的样例，该样例中包含 3 个类。

- ConsumerRunnable 类：消费线程类，执行真正的消费任务。
- ConsumerGroup 类：消费线程管理类，创建多个线程类执行消费任务。
- ConsumerMain 类：测试主方法类。

ConsumerRunnable.java

```
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;

import java.util.Arrays;
import java.util.Properties;

public class ConsumerRunnable implements Runnable {

    // 每个线程维护私有的 KafkaConsumer 实例
    private final KafkaConsumer<String, String> consumer;

    public ConsumerRunnable(String brokerList, String groupId, String topic) {
        Properties props = new Properties();
        props.put("bootstrap.servers", brokerList);
        props.put("group.id", groupId);
```

```
        props.put("enable.auto.commit", "true");//本例使用自动提交位移
        props.put("auto.commit.interval.ms", "1000");
        props.put("session.timeout.ms", "30000");
        props.put("key.deserializer", "org.apache.kafka.common.serialization.
StringDeserializer");
        props.put("value.deserializer", "org.apache.kafka.common.serialization.
StringDeserializer");
        this.consumer = new KafkaConsumer<>(props);
        consumer.subscribe(Arrays.asList(topic));//本例使用分区副本自动分配策略
    }

    @Override
    public void run() {
        while (true) {
            ConsumerRecords<String, String> records = consumer.poll(200);
            //本例使用 200 毫秒作为获取的超时时间
            for (ConsumerRecord<String, String> record : records) {
                //这里面写处理消息的逻辑，本例中只是简单地打印消息
                System.out.println(Thread.currentThread().getName() + "
consumed " + record.partition() +
                    "th message with offset:" + record.offset());
            }
        }
    }
}
```

ConsumerGroup.java

```
import java.util.ArrayList;
import java.util.List;

public class ConsumerGroup {

    private List<ConsumerRunnable> consumers;

    public ConsumerGroup(int consumerNum, String groupId, String topic,
String brokerList) {
        consumers = new ArrayList<>(consumerNum);
        for (int i = 0; i < consumerNum; ++i) {
            ConsumerRunnable consumerThread = new ConsumerRunnable
(brokerList, groupId, topic);
            consumers.add(consumerThread);
        }
    }

    public void execute() {
        for (ConsumerRunnable task : consumers) {
```

```

        new Thread(task).start();
    }
}

```

ConsumerMain.java

```

public class ConsumerMain {

    public static void main(String[] args) {
        String brokerList = "localhost:9092";
        String groupId = "testGroup1";
        String topic = "test-topic";
        int consumerNum = 3;

        ConsumerGroup consumerGroup = new ConsumerGroup(consumerNum,
groupId, topic, brokerList);
        consumerGroup.execute();
    }
}

```

5.8.2 单 KafkaConsumer 实例+多 worker 线程

此方法与第一种方法的区别在于，我们将消息的获取与消息的处理解耦，把后者放入单独的工作者线程中，即所谓的 worker 线程中。同时在全局维护一个或若干个 consumer 实例执行消息获取任务，如图 5.12 所示。

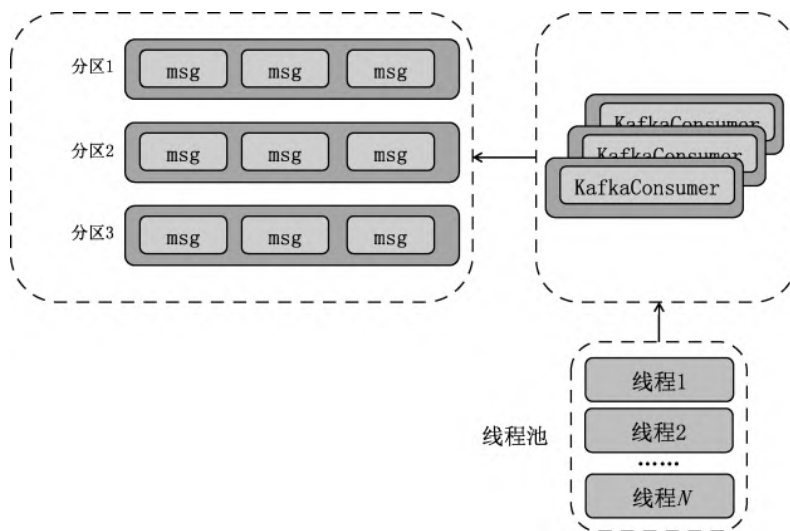


图 5.12 全局 consumer+多 worker 线程

本例使用全局的 `KafkaConsumer` 实例执行消息获取，然后把获取到的消息集合交给线程池中的 `worker` 线程执行工作。之后 `worker` 线程完成处理后上报位移状态，由全局 `consumer` 提交位移。代码中共有如下 3 个类。

- `ConsumerThreadHandler` 类：`consumer` 多线程管理类，用于创建线程池以及为每个线程分配消息集合。另外 `consumer` 位移提交也在该类中完成。
- `ConsumerWorker` 类：本质上是一个 `Runnable`，执行真正的消费逻辑并上报位移信息给 `ConsumerThreadHandler`。
- `Main` 类：测试主方法类。

`ConsumerThreadHandler.java`

```
package huxi.test.consumer.multithreaded;

import org.apache.kafka.clients.consumer.ConsumerRebalanceListener;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.clients.consumer.OffsetAndMetadata;
import org.apache.kafka.common.TopicPartition;
import org.apache.kafka.common.errors.WakeupException;

import java.util.Arrays;
import java.util.Collection;
import java.util.Collections;
import java.util.HashMap;
import java.util.Map;
import java.util.Properties;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class ConsumerThreadHandler<K, V> {

    private final KafkaConsumer<K, V> consumer;
    private ExecutorService executors;
    private final Map<TopicPartition, OffsetAndMetadata> offsets = new
HashMap<>();

    public ConsumerThreadHandler(String brokerList, String groupId,
String topic) {
        Properties props = new Properties();
```

```

        props.put("bootstrap.servers", brokerList); //必须指定
        props.put("group.id", groupId); //必须指定
        props.put("enable.auto.commit", "false");
        props.put("auto.offset.reset", "earliest");
        props.put("key.deserializer", "org.apache.kafka.common.serialization.
ByteArrayDeserializer");
        props.put("value.deserializer", "org.apache.kafka.common.serialization.
ByteArrayDeserializer");
        consumer = new KafkaConsumer<>(props);
        consumer.subscribe(Arrays.asList(topic), new
ConsumerRebalanceListener() {
            @Override
            public void onPartitionsRevoked(Collection<TopicPartition>
partitions) {
                consumer.commitSync(offsets); //提交位移
            }

            @Override
            public void onPartitionsAssigned(Collection<TopicPartition>
partitions) {
                offsets.clear();
            }
        });
    }

    /**
     * 消费主方法
     * @param threadNumber 线程池中的线程数
     */
    public void consume(int threadNumber) {
        executors = new ThreadPoolExecutor(
            threadNumber,
            threadNumber,
            0L,
            TimeUnit.MILLISECONDS,
            new ArrayBlockingQueue<Runnable>(1000),
            new ThreadPoolExecutor.CallerRunsPolicy());
        try {
            while (true) {
                ConsumerRecords<K, V> records = consumer.poll(1000L);
                if (!records.isEmpty()) {
                    executors.submit(new ConsumerWorker<>(records, offsets));
                }
            }
            commitOffsets();
        }
    }

```



```
        }
    } catch (WakeupException e) {
        //此处忽略此异常的处理
    } finally {
        commitOffsets();
        consumer.close();
    }
}

private void commitOffsets() {
    // 尽量降低 synchronized 块对 offsets 锁定的时间
    Map<TopicPartition, OffsetAndMetadata> unmodifiedMap;
    synchronized (offsets) {
        if (offsets.isEmpty()) {
            return;
        }
        unmodifiedMap = Collections.unmodifiableMap(new HashMap<>(offsets));
        offsets.clear();
    }
    consumer.commitSync(unmodifiedMap);
}

public void close() {
    consumer.wakeup();
    executors.shutdown();
}
}
```

ConsumerWorker.java

```
package huxi.test.consumer.multithreaded;

import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.OffsetAndMetadata;
import org.apache.kafka.common.TopicPartition;

import java.util.List;
import java.util.Map;

public class ConsumerWorker<K, V> implements Runnable {

    private final ConsumerRecords<K, V> records;
    private final Map<TopicPartition, OffsetAndMetadata> offsets;
```

```

        public ConsumerWorker(ConsumerRecords<K, V> record, Map<TopicPartition,
OffsetAndMetadata> offsets) {
            this.records = record;
            this.offsets = offsets;
        }

        @Override
        public void run() {
            for (TopicPartition partition : records.partitions()) {
                List<ConsumerRecord<K, V>> partitionRecords = records.records
(partition);
                for (ConsumerRecord<K, V> record : partitionRecords) {
                    // 插入消息处理逻辑, 本例只是打印消息
                    System.out.println(String.format("topic=%s, partition=%d,
offset=%d",
                                                    record.topic(), record.partition(), record.offset()));
                }

                // 上报位移信息
                long lastOffset = partitionRecords.get(partitionRecords.size()
- 1).offset();
                synchronized (offsets) {
                    if (!offsets.containsKey(partition)) {
                        offsets.put(partition, new OffsetAndMetadata(lastOffset
+ 1));
                    } else {
                        long curr = offsets.get(partition).offset();
                        if (curr <= lastOffset + 1) {
                            offsets.put(partition, new OffsetAndMetadata
(lastOffset + 1));
                        }
                    }
                }
            }
        }
    }
}

```

Main.java

```
package huxi.test.consumer.multithreaded;
```

```
public class Main {

    public static void main(String[] args) {
        String brokerList = "localhost:9092";
        String topic = "test-topic";
        String groupID = "test-group";
        final ConsumerThreadHandler<byte[], byte[]> handler = new
ConsumerThreadHandler<>(brokerList, groupID, topic);
        final int cpuCount = Runtime.getRuntime().availableProcessors();

        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                handler.consume(cpuCount);
            }
        };
        new Thread(runnable).start();

        try {
            // 20 秒后自动停止该测试程序
            Thread.sleep(20000L);
        } catch (InterruptedException e) {
            // 此处忽略此异常的处理
        }
        System.out.println("Starting to close the consumer...");
        handler.close();
    }

}
```

5.8.3 两种方法对比

两种方法各有优劣，用户需要根据实际业务需求选择不同的多线程 consumer 实现。表 5.3 概括总结了两种方法的优缺点。

表 5.3 自动提交与手动提交的比较

	优 点	缺 点
方法 1（每个线程维护专属 KafkaConsumer）	实现简单；速度较快，因为无线程间交互开销；方便位移管理；易于维护分区间的消息消费顺序	Socket 连接开销大；consumer 数受限于 topic 分区数，扩展性差；broker 端处理负载高（因为发往 broker 的请求数多）；rebalance 可能性增大
方法 2（全局 consumer+多 worker 线程）	消息获取与处理解耦；可独立扩展 consumer 数和 worker 数，伸缩性好	实现负载；难于维护分区内的消息顺序；处理链路变长，导致位移管理困难；worker 线程异常可能导致消费数据丢失

5.9 独立 consumer

目前为止我们讨论的 consumer 都是以 consumer group 的形式出现的。group 自动帮用户执行分区分配和 rebalance。对于需要多个 consumer 共同读取某个 topic 的需求来说，使用 group 是非常方便的。但有的时候用户依然有精确控制消费的需求，比如严格控制某个 consumer 固定地消费哪些分区。比如：

- 如果进程自己维护分区的状态，那么它就可以固定消费某些分区而不用担心消费状态丢失的问题。
- 如果进程本身已经是高可用且能够自动重启恢复错误（比如使用 YARN 和 Mesos 等容器调度框架），那么它就不需要让 Kafka 来帮它完成错误检测和状态恢复。

以上两种情况中 consumer group 都是无用武之地的，取而代之的是被称为独立 consumer（standalone consumer）的角色。standalone consumer 间彼此独立工作互不干扰。任何一个 consumer 崩溃都不影响其他 standalone consumer 的工作。

使用 standalone consumer 的方法就是调用 `KafkaConsumer.assign` 方法。还记得吧，之前订阅 topic 我们使用的是 `KafkaConsumer.subscribe` 方法，而 `assign` 方法则接收一个分区列表，直接赋予该 consumer 访问这些分区的权力。下面的代码演示了如何使用 `assign` 方法直接给 consumer 分配分区：

```
public static void main(String[] args) {
    Properties props = new Properties();
    props.put("bootstrap.servers", "localhost:9092"); //必须指定
    props.put("auto.offset.reset", "earliest");
    props.put("enable.auto.commit", "false");
    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<>
(props);

    List<TopicPartition> partitions = new ArrayList<>();
    List<PartitionInfo> allPartitions = consumer.partitionsFor
("test-topic");
    if (allPartitions != null && !allPartitions.isEmpty()) {
        for (PartitionInfo partitionInfo : allPartitions) {
            partitions.add(new TopicPartition(partitionInfo.topic(),
partitionInfo.partition()));
        }
    }
    consumer.assign(partitions);
}
```

```
        }
        consumer.assign(partitions);
    }

    try {
        while (true) {
            ConsumerRecords<String, String> records = consumer.poll(
                Long.MAX_VALUE);
            for (ConsumerRecord<String, String> record : records) {
                System.out.println(String.format("topic=%s, partition=%d,
                    offset=%d", record.topic(), record.partition(), record.offset()));
            }
            consumer.commitSync();
        }
    } catch (WakeupException e) {
        //此处忽略此异常的处理
    } finally {
        consumer.commitSync();
        consumer.close();
    }
}
```

上面的代码中使用 `assign` 固定地为 `consumer` 指定要消费的分区。如果发生多次 `assign` 调用，最后一次 `assign` 调用的分配生效，之前的都会被覆盖掉。还有一个值得注意的是，`assign` 和 `subscribe` 一定不要混用，即不能在一个 `consumer` 应用中同时使用 `consumer group` 和独立 `consumer`。

5.10 旧版本 consumer

5.10.1 概览

到目前为止，本章讨论的都是 Kafka 新版本 Java consumer，即 `org.apache.kafka.clients` 包中的 `KafkaConsumer`。在推出新版本之前，Kafka 提供了用 Scala 语言编写的 consumer。该 consumer 的叫法有很多——旧版本 consumer、Scala consumer 或 ZooKeeper consumer。总之它们指代的都是这个位于 Kafka 核心包中的 `kafka.consumer.Consumer` 以及实际接口类 `ConsumerConnector`。Kafka 社区于 0.11.0.0 版本正式将旧版本 consumer API 标记 `Deprecated`，明确告知用户该版本 consumer 已经不再推荐使用了。

不过鉴于目前有很多用户的生产环境中依然在使用旧版本 consumer，本节我们重点讨论一下旧版本 consumer 的使用。

旧版本 consumer 依然有 consumer group 和独立 consumer 的概念，只不过它们有特定的名字：high-level consumer 和 low-level consumer。不管它们是被翻译成“高（低）阶消费者”还是“高（低）层次消费者”，用户只需要了解 high-level consumer 使用的是 consumer group，而 low-level consumer 没有 group 的概念就可以了。

旧版本 consumer 的参数与新版本的出入不大，最重要的两个参数如下。

- group.id: 与新版本含义相同，指定 group 名称。
- zookeeper.connect: 旧版本无 bootstrap.servers 参数。相反，它需要连接 ZooKeeper，因此这个参数是必须要指定的。
- auto.offset.reset: 与新版本含义相同，不过当从最早位移读取时旧版本使用的是 smallest 而非 earliest。

下面分别讨论一下 high-level consumer 和 low-level consumer。

5.10.2 high-level consumer

和新版本 consumer 中使用 consumer group 的原理类似，high-level consumer 要求用户为多个 consumer 实例指定相同的 groupId，使其构成一个 consumer group 共同参与消费。consumer group 负责后续的组成员管理、错误检测、故障转移、负载均衡、rebalance 以及位移提交（用户也可以自行提交位移）。

但是和新版本不同的是，group 是依赖 ZooKeeper 来完成这些功能的，特别是位移的提交。high-level consumer 依靠向 ZooKeeper 下的特定节点（znode）写入位移信息完成位移提交动作。ZooKeeper 负责管理 group 的节点名称是/consumers，其下子节点列表如图 5.13 所示。

- /consumers/<groupId>/ids/<consumer id>: 记录了该 consumer 的订阅信息。同时该 znode 是一个临时节点（ephemeral node，如果 Kafka 与 ZooKeeper 会话过期，则自动删除该节点），因此也被 consumer 用来监听 consumer 的“存活”状态。一旦发现该节点会话失效，则认为 consumer 已死，开启新一轮 rebalance。
- /consumers/<groupId>/owners/<topic>/<partition>: 保存了 consumer 各个消费线程（后面会讨论）的 Id。consumer 在执行 rebalance 的时候必须要把消费该分区的线程 Id 保存在该节点下。
- /consumers/<groupId>/offsets/<topic>/<partition>: 保存该 group 消费指定分区的位移信息。

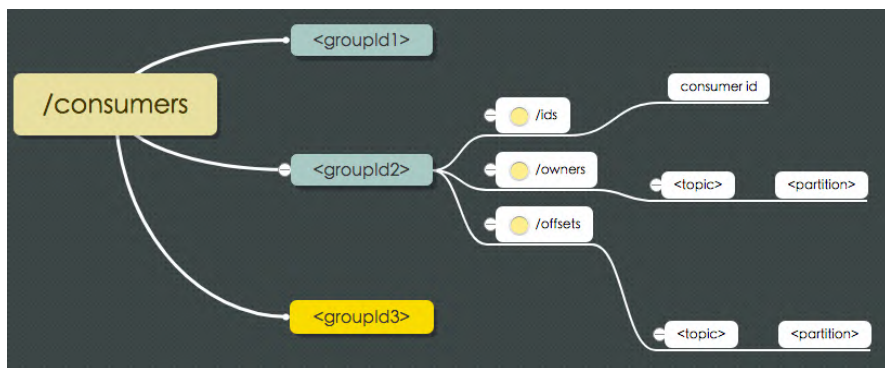


图 5.13 ZooKeeper consumer 节点

上面说到/owners znode时提到了消费线程，这是旧版本 consumer 与新版本 consumer 在设计上的又一个重大的区别。Java consumer 使用单线程轮询的方式来消费消息；Scala consumer 则使用多线程的方式，即用户可以指定多个线程来消费订阅的 topic。举一个例子来说明，假设某个 consumer group 订阅了一个 topic，该 topic 有 10 个分区，用户在使用旧版本 consumer 时指定使用 10 个线程来消费该 topic，那么每个线程都会被分配一个分区。若用户指定了 11 个线程，则有一个线程不会被分配到任何分区，从而造成资源浪费。因此在实际使用过程中应确保线程数不要超过订阅的分区总数。

值得注意的是，这里所说的多线程和 5.8 节“多线程消费实例”中的多线程不是一回事。这里的多线程是指当我们只创建一个 consumer 实例时，该 consumer 会在内部自动为我们创建多个线程进行消费，而 5.8 节中讨论的是我们手动地创建多个线程。Java consumer 本身还是单线程的设计（除了心跳线程）。

如果用户的逻辑很简单——不想关心位移的提交，只想消费数据，那么使用 high-level consumer 加自动提交位移就是最容易的方式。下面给出一个使用 high-level consumer 的完整样例，如下所示：

```
import kafka.consumer.Consumer;
import kafka.consumer.ConsumerConfig;
import kafka.consumer.ConsumerIterator;
import kafka.consumer.KafkaStream;
import kafka.javaapi.consumer.ConsumerConnector;

import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Properties;
import java.util.concurrent.ExecutorService;
```

```

import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class HighLevelConsumerSample {

    private final ConsumerConnector consumer;
    private final String topic;
    private ExecutorService executor;

    public HighLevelConsumerSample(String zkConnect, String groupId,
String topic) {
        this.topic = topic;
        consumer = Consumer.createJavaConsumerConnector(createConsumerConfig
(zkConnect, groupId));
    }

    public static void main(String[] args) {
        String zkConnect = "localhost:2181";
        String groupId = "test-group";
        String topic = "test-topic";
        int thread = Runtime.getRuntime().availableProcessors();

        HighLevelConsumerSample consumer = new HighLevelConsumerSample
(zkConnect, groupId, topic);
        consumer.run(thread);

        try {
            Thread.sleep(20000);
        } catch (InterruptedException e) {
            // 此处忽略该异常的处理
        }
        consumer.shutdown();
    }

    public void run(int threadCount) {
        Map<String, Integer> topicMap = new HashMap<>();
        topicMap.put(topic, threadCount); //指定线程数
        Map<String, List<KafkaStream<byte[], byte[]>>> consumerMap =
consumer.createMessageStreams(topicMap);
        List<KafkaStream<byte[], byte[]>> streams = consumerMap.get(topic);

        int threadNumber = 0;
        executor = Executors.newFixedThreadPool(threadCount);
        for (final KafkaStream stream : streams) {

```



```

        executor.submit(new ConsumerWork(stream, threadNumber++));
    }
}

public void shutdown() {
    if (consumer != null) {
        consumer.shutdown();
    }
    if (executor != null) {
        executor.shutdown();
    }
    try {
        if (!executor.awaitTermination(5000, TimeUnit.MILLISECONDS)) {
            System.err.println("Failed to gracefully shutdown.");
        }
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
}

private ConsumerConfig createConsumerConfig(String zkConnect, String
groupID) {
    Properties props = new Properties();
    props.put("group.id", groupID); //必须指定
    props.put("zookeeper.connect", zkConnect); //必须指定
    props.put("consumer.id", "test-consumer");
    props.put("auto.commit.enable", "true");
    props.put("auto.commit.interval.ms", "1000");
    props.put("auto.offset.reset", "smallest");
    return new ConsumerConfig(props);
}

}

class ConsumerWork implements Runnable {

    private final KafkaStream<byte[], byte[]> stream;
    private int threadNumber;

    public ConsumerWork(KafkaStream stream, int threadNumber) {
        this.stream = stream;
        this.threadNumber = threadNumber;
    }

    @Override

```

```

    public void run() {
        ConsumerIterator<byte[], byte[]> iter = stream.iterator();
        while (iter.hasNext()) {
            System.out.println(String.format("Thread %d consumed message: %s",
threadNumber, String.valueOf(iter.next().message())));
        }
        System.out.println(String.format("Thread %d finished, exiting...",
threadNumber));
    }
}

```

上面的代码中会创建一个固定大小的线程池用于消费订阅的分区。切记不要创建超过分区数的线程，否则会有线程无法消费任何分区。

5.10.3 low-level consumer

high-level consumer 简化了 consumer 的消费以及 consumer group 的管理，但如果用户想要细粒度地控制分区消费，low-level consumer 就派上了用场。low-level consumer 的典型使用场景包括：

- 消息“重演”，即需要重复读取历史数据。
- 只想消费部分分区的数据。
- 实现“精确一次”处理语义。

当然凡事有利有弊。low-level consumer 也有它自己的劣势，比如：

- 用户必须自己处理位移提交。
- 用户必须寻找分区的 leader broker。
- 用户必须自己处理 leader 变更。

首先，low-level consumer 把位移提交的工作完全交给用户来完成。用户可以把位移信息保存到任何地方，比如数据库、文件或是内存中。著名的 Apache Storm 流式处理框架的 storm-kafka 插件使用的就是 low-level consumer，它把位移保存在 ZooKeeper 中特定的位置下（与 high-level consumer 不同的位置！）。

其次，当使用 low-level consumer 时，用户必须在消费前写代码自行寻找待消费分区的 leader 所在的 broker，然后直接给该 broker 发送请求读取数据。而在 high-level consumer 中，这个工作是 group 帮用户自动完成的。

最后，由于位移提交由用户实现，用户可以利用它实现“精确一次”的处理语义。核心的

思想就是，要把数据处理和位移提交放入一个事务（transaction）中。若它们中任何一个操作失败，都可以回滚整个操作，从而实现精确一次的消息消费。

low-level consumer 的主要实现类是 SimpleConsumer。用户可以无差别使用 low-level consumer 和 SimpleConsumer 这两个名词。它们指代的是同一个意思。下面给出一个 SimpleConsumer 的完整样例：

```
import kafka.api.FetchRequest;
import kafka.api.FetchRequestBuilder;
import kafka.api.PartitionOffsetRequestInfo;
import kafka.cluster.BrokerEndPoint;
import kafka.common.ErrorMapping;
import kafka.common.TopicAndPartition;
import kafka.javaapi.*;
import kafka.javaapi.consumer.SimpleConsumer;
import kafka.message.MessageAndOffset;

import java.nio.ByteBuffer;
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class SimpleConsumerSample {

    private List<String> replicaBrokers = new ArrayList<>();

    public static void main(String args[]) {
        String topic = "test-topic";
        int partition = 0;
        String host = "localhost";
        int port = 9092;
        long maxReads = 1000L;

        SimpleConsumerSample consumer = new SimpleConsumerSample();
        List<String> seeds = new ArrayList<>();
        seeds.add(host);
        try {
            consumer.run(maxReads, topic, partition, seeds, port);
        } catch (Exception e) {
            System.out.println("Something wrong occurred: " + e);
            e.printStackTrace();
        }
    }
}
```

```

    }

    public void run(long maxReads, String topic, int partition, List<String>
seedBrokers, int port) throws Exception {
        // 确定给定 topic 分区的元数据信息, 包括领导者副本所在 broker、ISR 副本集合等
        PartitionMetadata metadata = findLeader(seedBrokers, port, topic,
partition);
        if (metadata == null) {
            System.out.println("Can't find metadata for Topic and Partition.
Exiting");
            return;
        }
        if (metadata.leader() == null) {
            System.out.println("Can't find Leader for Topic and Partition.
Exiting");
            return;
        }
        String leadBroker = metadata.leader().host();
        String clientName = "Client_" + topic + "_" + partition;

        SimpleConsumer consumer = new SimpleConsumer(leadBroker, port,
100000, 64 * 1024, clientName);
        long readOffset = getLastOffset(consumer, topic, partition, kafka.
api.OffsetRequest.EarliestTime(), clientName);

        int numErrors = 0;
        while (maxReads > 0) {
            if (consumer == null) {
                consumer = new SimpleConsumer(leadBroker, port, 100000,
64 * 1024, clientName);
            }
            FetchRequest req = new FetchRequestBuilder()
                .clientId(clientName)
                .addFetch(topic, partition, readOffset, 100000)
            // 增加 100000 的参数值将允许单次 Fetch 请求获取更多的 Kafka 消息
                .build();
            FetchResponse fetchResponse = consumer.fetch(req);

            if (fetchResponse.hasError()) {
                numErrors++;
                // 获取错误代码
                short code = fetchResponse.errorCode(topic, partition);
                System.out.println("Error fetching data from the Broker:"
+ leadBroker + " Reason: " + code);
            }
        }
    }
}

```

```

        if (numErrors > 5) break;
        if (code == ErrorMapping.OffsetOutOfRangeCode()) {
            // 此 Fetch 请求指定了无效位移, 重设位移为最新位移
            readOffset = getLastOffset(consumer, topic, partition,
kafka.api.OffsetRequest.LatestTime(), clientName);
            continue;
        }
        consumer.close();
        consumer = null;
        leadBroker = findNewLeader(leadBroker, topic, partition,
port);

        continue;
    }
    numErrors = 0;

    long numRead = 0;
    for (MessageAndOffset messageAndOffset : fetchResponse.messageSet
(topic, partition)) {
        long currentOffset = messageAndOffset.offset();
        if (currentOffset < readOffset) {
            System.out.println("Found an old offset: " +
currentOffset + " Expecting: " + readOffset);
            continue;
        }
        readOffset = messageAndOffset.nextOffset();
        ByteBuffer payload = messageAndOffset.message().payload();

        byte[] bytes = new byte[payload.limit()];
        payload.get(bytes);
        System.out.println(String.valueOf(messageAndOffset.offset())
+ ": " + new String(bytes, "UTF-8"));
        numRead++;
        maxReads--;
    }

    if (numRead == 0) {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException ie) {
        }
    }
}
if (consumer != null) consumer.close();
}

```

```

        public static long getLastOffset(SimpleConsumer consumer, String
topic, int partition, long whichTime, String clientName) {
            TopicAndPartition topicAndPartition = new TopicAndPartition
(topic, partition);
            Map<TopicAndPartition, PartitionOffsetRequestInfo> requestInfo =
new HashMap<>();
            requestInfo.put(topicAndPartition, new PartitionOffsetRequestInfo
(whichTime, 1));
            kafka.javaapi.OffsetRequest request = new kafka.javaapi.
OffsetRequest(
                requestInfo, kafka.api.OffsetRequest.CurrentVersion(),
clientName);
            OffsetResponse response = consumer.getOffsetsBefore(request);

            if (response.hasError()) {
                System.out.println("Error fetching data Offset Data the
Broker. Reason: " + response.errorCode(topic, partition) );
                return 0;
            }
            long[] offsets = response.offsets(topic, partition);
            return offsets[0];
        }

        private String findNewLeader(String a_oldLeader, String a_topic, int
a_partition, int a_port) throws Exception {
            for (int i = 0; i < 5; i++) {
                boolean goToSleep;
                PartitionMetadata metadata = findLeader(replicaBrokers,
a_port, a_topic, a_partition);
                if (metadata == null) {
                    goToSleep = true;
                } else if (metadata.leader() == null) {
                    goToSleep = true;
                } else if (a_oldLeader.equalsIgnoreCase(metadata.leader().
host()) && i == 0) {
                    // 第一次强制休眠 1 秒防止 ZooKeeper 依然保存旧的 leader 信息
                    // 之后则不再为此而休眠
                    goToSleep = true;
                } else {
                    return metadata.leader().host();
                }
                if (goToSleep) {
                    try {

```

```

        Thread.sleep(1000);
    } catch (InterruptedException ie) {
    }
}

System.err.println("Unable to find new leader after Broker
failure. Exiting");
throw new Exception("Unable to find new leader after Broker
failure. Exiting");
}

private PartitionMetadata findLeader(List<String> seedBrokers, int
port, String topic, int partition) {
    PartitionMetadata returnMetaData = null;
loop:
    for (String seed : seedBrokers) {
        SimpleConsumer consumer = null;
        try {
            consumer = new SimpleConsumer(seed, port, 100000, 64 *
1024, "leaderLookup");
            List<String> topics = Collections.singletonList(topic);
            TopicMetadataRequest req = new TopicMetadataRequest(topics);
            kafka.javaapi.TopicMetadataResponse resp = consumer.send(req);

            List<TopicMetadata> metaData = resp.topicsMetadata();
            for (TopicMetadata item : metaData) {
                for (PartitionMetadata part : item.partitionsMetadata()) {
                    if (part.partitionId() == partition) {
                        returnMetaData = part;
                        break loop;
                    }
                }
            }
        } catch (Exception e) {
            System.out.println("Error communicating with Broker [" +
seed + "] to find Leader for [" + topic + ", " + partition + "] Reason: " + e);
        } finally {
            if (consumer != null) consumer.close();
        }
    }
    if (returnMetaData != null) {
        replicaBrokers.clear();
        for (BrokerEndPoint replica : returnMetaData.replicas()) {
            replicaBrokers.add(replica.host());
        }
    }
}

```

```
        }  
    }  
    return returnMetaData;  
}  
}
```

5.11 本章小结

本章详细探讨了 Kafka 新版本 consumer 的各个方面，包括 consumer 的概要设计、consumer 程序的开发、参数设置以及 consumer 的各个子功能，如 rebalance、位移管理等。

另外本章还概要讨论了旧版本 consumer 并分别给出了 high-level consumer 和 low-level consumer 的完整样例。

相信各位读者在阅读本章之后可以结合自身业务目标的需求来开发实际的可供线上部署的 Kafka consumer 程序。

第 6 章

Kafka 设计原理

前 5 章涵盖了 Apache Kafka 在生产环境中使用的方方面面，详细讨论了如何安装部署 Kafka，如何使用 Kafka 提供的 Java 版本 producer API 发送消息，以及如何使用 Java 版本 consumer API 消费消息。本章将介绍 Kafka 内部设计原理，带领读者深入了解 Kafka 内部结构。

学习本章，你将了解到以下内容。

- broker 端设计架构。
- producer 端设计架构。
- consumer 端设计架构。
- broker 端交付语义保障。
- producer 端交付语义保障。
- consumer 端交付语义保障。

6.1 broker 端设计架构

broker 是 Apache Kafka 最重要的组件，本质上它是一个功能载体（或服务载体），承载了绝大多数的 Kafka 服务。事实上，大多数的消息队列框架都有 broker 或与之类似的角色。一个 broker 通常是以服务器的形式出现的，对用户而言，broker 的主要功能就是持久化消息以及将消息队列中的消息从发送端传输到消费端。从前几章的介绍中我们可知，Kafka 的 broker 负责持久化 producer 端发送的消息，同时还为 consumer 端提供消息。

国内通常将 broker 翻译成“代理人”或“经纪人”，笔者以为这些都过于突出该词在金融领域特别是投资领域内的角色，因此本章我们将始终使用英文单词 broker 来指代 Kafka 的

服务器。

如前所述，broker 是一个服务载体，如果不深入了解 broker 内部的设计原理，我们是无法掌握 Kafka 的设计精髓的。因此本节将详细讨论 broker 的各种设计思想，具体分为以下 8 个方面。

- 消息设计。
- 集群管理。
- 副本与 ISR 机制。
- 日志存储。
- 请求处理协议。
- controller 设计。
- broker 状态机。
- broker 通信原理。

6.1.1 消息设计

1. 消息格式

所谓消息引擎，定义消息格式必然是首当其冲的工作——使用什么数据结构来保存消息和消息队列是第一个要解决的问题。在详细考察 Kafka 消息格式之前，笔者尝试使用 Java 类自己定义了消息格式，下面先来思考一下这种方式的优劣。

笔者实现的 Java 消息类非常简单，如下面的代码所示：

```
public class Message implements Serializable {  
    private CRC32 crc;  
    private short magic;  
    private boolean codecEnabled;  
    private short codecClassOrdinal;  
    private String key;  
    private String value;  
}
```

Message 类由消息键（下称 key）和消息体（下称 value）再加上一组元数据字段组成（CRC32、magic 版本、压缩算法等）。使用 Java 类的方式定义 Kafka 消息似乎很简单，但这种实现方式的弊端是什么呢？

首先，由于使用了 Java 类来定义消息，因此必然受到 Java 对象开销所累。我们知道，在 Java 内存模型（Java Memory Model, JMM）中保存对象的开销其实相当大，有可能花费比消息大小大 2 倍的空间来保存数据（甚至更糟）。为了降低这种开销，JMM 通常会对用户自定

义的类进行字段重排，以试图减少内存占用。另外随着 Java 堆上的数据越来越多，垃圾回收（Garbage Collection，下称 GC）的性能下降得很快，从而整体上拖累应用程序的吞吐量。

这里多说一句，为什么上面的重排（reordering）会减少内存占用？这是因为 JMM 要求 Java 对象必须按照 8 字节对齐，未对齐的部分会填充空白字节进行补齐，该操作被称为 padding。若用户随意指定对象字段的顺序，那么由于每个字段类型占用字节数各异，可能会造成不必要的补齐，所以 JMM 会尝试对各个字段重排以期望降低整体的对象开销。

针对上面代码定义的类，JMM 会为字段进行重排变成下面的实现：

```
public class Message implements Serializable {  
    private short magic;  
    private short codecKlassOrdinal;  
    private boolean codecEnabled;  
    private CRC32 crc;  
    private String key;  
    private String body;  
}
```

即使这样，这个朴素的 Java 类依然要占用 40 字节的空间：16 字节对象头部（64 位 JVM 对象头部通常由两个 8 字节的 word 组成）+ 2 字节 magic + 2 字节 codec + 1 字节 boolean + 4 字节 CRC + 4 字节 String + 4 字节 String + 7 字节补齐= 40 字节。注意，两个 String 字段实际上是 Java 的引用类型。在对象实例化后，它们分别指向堆或常量池中的两个 String 对象，因此在计算内存占用时，它们只能被统计为 4 字节的引用类型。

由此可见，一条普通的 Kafka 消息，即使未初始化 key 和 value，也需要占用 40 字节的内存空间，而其中 7 字节更是完全被浪费掉的。这种朴素的实现方案其效率是十分低下的。显然，Kafka 不是这样设计的。Kafka 的实现方式本质上是使用 Java NIO 的 ByteBuffer 来保存消息，同时依赖文件系统提供的页缓存机制，而非依靠 Java 的堆缓存。毕竟在大部分情况下，我们在堆上保存的对象在写入文件系统后很有可能在操作系统的页缓存中仍保留着，从而会造成资源的浪费。

另外，ByteBuffer 是紧凑的二进制字节结构，而不需要 padding 操作，因此省去了很多不必要的对象开销。根据 Kafka 官网的测试数据，在一台 32GB 内存的机器上，Kafka 几乎可以用到 28~30GB 的物理内存而不用担心 Java GC 的糟糕性能。若使用 ByteBuffer 来保存相同的消息格式，经笔者测试，同一条消息比起纯 Java 堆的实现方案大概可节省近 40%的空间占用，好处不言而喻。除此之外，ByteBuffer 方案还有着非常好的扩展性。

2. 版本变迁

Kafka 社区于 2017 年 6 月底正式推出了 Kafka 0.11.0.0 版本，这是 Kafka 一个里程碑式

的大版本，特别是对于消息格式进行了改进和升级。下面详细探讨一下 Kafka 的消息版本变迁。

目前 Kafka 消息格式有 3 个版本：V0 版本、V1 版本和 V2 版本。值得注意的是，这 3 个版本与社区发布版本在时间线上并无严格对应，如此区分只是帮助读者了解 Kafka 重大版本变迁之间的联系。

(1) V0 版本

V0 版本主要指 Kafka 0.10.0.0 之前的版本，是 Kafka 最早的消息版本，其消息格式如图 6.1 所示。

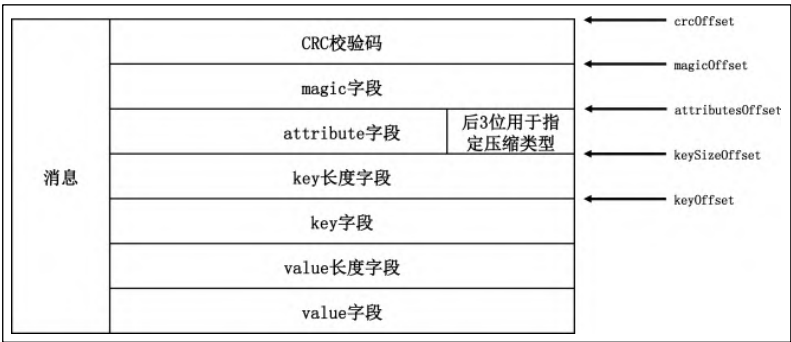


图 6.1 V0 版本消息格式

其中，各个字段的含义如下。

- CRC 校验码：4 字节的 CRC 校验码，用于确保消息在传输过程中不会被恶意篡改。
- magic：单字节的版本号。V0 版本 magic=0，V1 版本 magic=1，V2 版本 magic=2。
- attribute：单字节属性字段，目前只使用低 3 位表示消息的压缩类型。
- key 长度字段：4 字节的消息 key 长度信息。若未指定 key，则给该字段赋值为-1。
- key 值：消息 key，长度由上面的“key 长度字段”值指定。如果“key 长度字段”值是 -1，则无 key，消息没有该字段。
- value 长度字段：4 字节的消息长度。若未指定 value，则给该字段赋值-1。
- value 值：消息 value，长度由上面的“value 长度字段”值指定。如果“value 长度字段”值是-1，则无 value，消息没有该字段。

上面除去 key 值和 value 值之外的所有字段被统称为消息头部（message header）信息，总共占用 14 字节。这就是说，一条 Kafka V0 版本的消息长度再小也无法小于 14 字节，否则会被 Kafka 视为非法消息。

另外值得注意的是，上面消息格式中的 `attribute` 字段占用 1 字节，即共有 8 位。目前 V0 版本的消息只使用了低 3 位来指定压缩类型，其他 5 位留作以后扩展使用。当前支持的压缩类型以及对应值如下。

- 0x00: 未启用压缩。
- 0x01: GZIP。
- 0x02: Snappy。
- 0x03: LZ4。

根据上面的定义，假设一条消息使用了 LZ4 进行压缩，key 是 “key”，value 是 “hello”，那么该消息各个字段的值依次如下。

- CRC: 对所有字段进行 CRC 校验之后的 CRC 值。
- magic: 0。
- attribute: 0x03。
- key 长度字段: 3。
- key 字段: “key”。
- value 长度字段: 5。
- value 字段: “hello”。

故这条消息总的字节数 = $4 + 1 + 1 + 4 + 3 + 4 + 5 = 22$ 字节。若还有一条 Kafka 消息未指定 key，而 value 依然是 “hello”，则按照之前的定义，Kafka 会往 key 长度字段中写入 -1，然后总的字节数 = $4 + 1 + 1 + 4 + 4 + 5 = 19$ 字节。与第 1 条消息相比，少的 3 字节正是未指定 key 而节省的。同时我们也可以发现，即使不指定 key，Kafka 依然需要花费 4 字节来保存 -1。由此看来，该版本消息格式依然有改进的空间。

(2) V1 版本

随着 Kafka 功能的不断演进，人们日益发现 V0 版本消息在实际使用过程中存在着一些显著的弊端，比如：

- 由于没有消息的时间信息，Kafka 定期删除过期日志只能依靠日志段文件的“最近修改时间”，但这个时间极易受到外部操作的干扰。举一个例子，若不小心对日志段文件执行了 UNIX 的 `touch` 命令，该日志文件的最近修改时间就被更新了。一旦这个时间被“破坏”或者更改，Kafka 将无法对哪些消息过期做出正确判断。
- 很多流式处理框架都需要消息保存时间信息以便对消息执行时间窗口等聚合操作。

鉴于这些原因，社区于 Kafka 0.10.0.0 中改进了 V0 版本的消息格式，推出了 V1 版本的格式，主要的变化就是在消息中加入了时间戳字段，如图 6.2 所示。

V1 版本与 V0 版本的第一个差别是引入了 8 字节的时间戳字段，而其他字段的含义与 V0 版本相同，故此处不再赘述。加入 8 字节的时间戳后，V1 版本的消息头部膨胀到 22 字节。如果依然用 V0 版本的第 1 条消息（指定 key 的那条）来计算，该消息总字节数=22 + 3 + 5 = 30 字节，比 V0 版本多出来的 8 字节正是时间戳字段。未指定 key 消息总长度的计算方法与 V0 版本是一样的，它须占用 27 字节。

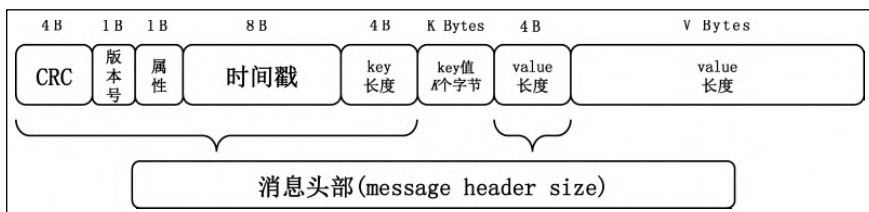


图 6.2 V1 版本消息格式

V1 版本与 V0 版本的第 2 个差别在于 attribute 字段的使用。由于增加了时间戳信息，V1 版本 attribute 字段的第 4 位被用于指定时间戳类型。当前支持两种时间戳类型：CREATE_TIME 和 LOG_APPEND_TIME。前者表示在消息创建时由 producer 指定时间戳，后者表示消息被发送到 broker 端时由 broker 指定时间戳。

在跳到 V2 版本之前，我们还需要提及一下 Kafka 的消息集合以及消息层次的概念。事实上，无论是哪个版本的 Kafka，它的消息层次都分为两层：消息集合（message set）和消息。我们之前给出了 V0 版本和 V1 版本的消息格式，但对于消息集合并未给出详细定义。

一个消息集合包含若干个日志项，而每个日志项都封装了实际的消息和一组元数据信息。Kafka 日志文件就是由一系列消息集合日志项构成的。Kafka 不会在消息层面上直接操作，它总是在消息集合上进行写入操作。

V0 版本和 V1 版本在消息集合上的设计没有任何差别，但它们与 V2 版本却有着很大的不同。故这里我们先来讲解一些 V2 版本之前的消息集合。

遵循 Kafka 社区的术语规范，V2 之前的版本更多使用的是日志项（log entry），而 V2 版本则使用消息批次（record batch）。为了便于理解，读者可以安全地认为它们指代的是同一个东西。

V0、V1 版本的日志项格式如图 6.3 所示。

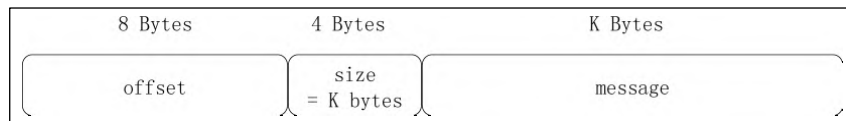


图 6.3 V0、V1 版本的日志项格式

每个消息集中的日志项由一条“浅层”消息和日志项头部组成。

- 浅层消息（shallow message）：如果没有启用消息压缩，那么这条浅层消息就是消息本身；否则，Kafka 会将多条消息压缩到一起统一封装进这条浅层消息的 value 字段。此时该浅层消息被称为包装消息（或外部消息，即 wrapper 消息），而 value 字段中包含的消息则被称为内部消息，即 inner 消息。V0、V1 版本中的日志项只能包含一条浅层消息。
- 日志项头部（log entry header）：如图 6.3 所示，头部由 8 字节的位移（offset）字段加上 4 字节的长度（size）字段构成。注意这里的 offset 非 consumer 端的 offset，它是指该消息在 Kafka 分区日志中的 offset。同样地，如果是未压缩消息，该 offset 就是消息的 offset；否则该字段表示 wrapper 消息中最后一条 inner 消息的 offset。因此，从 V0、V1 版本消息集合日志项中搜寻该日志项的起始位移（base offset 或 starting offset）是一件非常困难的事情，因为在该过程中 Kafka 需要深度遍历所有 inner 消息，这也就意味着 broker 端需要执行解压缩的操作，可见代价之高。

了解了日志项的格式之后，我们来实际计算一下消息集合的大小，还是以之前的两条 V1 版本的消息为例。假设第 1 条消息被“塞进”了一个消息集合，那么该消息集合的长度 = 8 (offset) + 4 (size) + 30 = 42 字节；而一个只包含第 2 条消息的消息集合的长度 = 8 (offset) + 4 (size) + 27 = 39 字节。下面做一个实验来验证一下。

①搭建一个 Kafka 0.10.2.0 版本单节点环境（搭建方法请参考第 3 章）。

②创建一个测试 topic——test，单分区，备份因子是 1，然后使用 console-producer 发送一条消息，key = “key”，value = “hello”。验证日志文件大小是 42 字节。

```
> bin/kafka-topics.sh --zookeeper localhost:2181 --create --partitions 1
--replication-factor 1 --topic test
Created topic "test".
> bin/kafka-console-producer.sh --broker-list localhost:9092 --topic
test --property parse.key=true --property key.separator=:
key:hello
^C
```

测试结果如下：

```
> pwd
/kafka-datalogs/kafka_1/test-0
> ll *.log
-rw-r--r-- 1 user staff 42 Jul 6 11:36 00000000000000000000.log
```

接下来，再使用 console-producer 脚本发送一条无 key 消息来验证该消息总字节数是 39 +

42 = 81 字节。

```
> bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test
hello
^C
> ll *.log
-rw-r--r--  1 user  staff   81 Jul  6 11:39 00000000000000000000.log
```

上面的输出结果证明了对消息日志项的计算方法是正确的。更重要的是，在了解了 V0、V1 版本的消息格式后，用户甚至可以编写程序从底层日志文件中直接读取 Kafka 消息而不需要通过 Kafka 的 API。有的时候，这会非常方便。

所谓“长江后浪推前浪”，新事物总是要取代旧事物的。人们逐渐发现了 V0、V1 版本消息集合在设计上的一些缺陷。

- 空间利用率不高：不论 key 和 value 长度是多少，它总是使用 4 字节固定长度来保存这部分信息。例如，这两个版本保存 100 或是 1000 都是使用 4 字节，但其实我们只需要 7 位就足以保存 100 这个数字了，也就是说，只用 1 字节就足够，另外 3 字节纯属浪费。设想若每条 Kafka 消息都浪费 3 字节，每天发送几十亿条 Kafka 消息的线上系统要浪费多少内存和磁盘空间。
- 只保存最新消息位移：如前所述，若启用压缩，这个版本中的 offset 是消息集合中最后一条消息的 offset。如果用户想要获取第 1 条消息的位移，必须要把所有的消息全部解压缩装入内存，然后反向遍历才能获取，显然这个代价是很大的。
- 冗余的消息级 CRC 校验：为每条消息都执行 CRC 校验有些“鸡肋”。即使在网络传输过程中没有出现恶意篡改，我们也不能想当然地认为在 producer 端发送的消息到 consumer 端时其 CRC 值是不变的。若用户指定时间戳类型是 LOG_APPEND_TIME，broker 将使用当前时间戳覆盖掉消息已有时间戳，那么当 broker 端对消息进行时间戳更新后，CRC 就需要重新计算从而发生变化；再如，broker 端进行消息格式转换（broker 端和 clients 端要求版本不一致时会发生消息格式转换，不过这对用户而言是完全透明的）也会带来 CRC 值的变化。鉴于这些情况，对每条消息都执行 CRC 校验实际上没有必要，不仅浪费空间，还占用了宝贵的 CPU 时间片。
- 未保存消息长度：每次需要单条消息的总字节数信息时都需要计算得出，没有使用单独字段来保存。每次计算时为了避免对现有数据结构的破坏，都需要大量的对象副本，解序列化效率很低。

鉴于这些缺陷，Kafka 社区于 0.11.0.0 版本重构了消息和消息集合格式的定义，大幅度的优化和改进使得 V2 版本与之前的版本有着很大的不同。下面就来看一下 V2 版本的消息格式。

(3) V2 版本

V2 版本依然分为消息和消息集合两个维度，只不过消息集合的提法被消息批次所取代。V2 版本中，它有一个专门的术语：**RecordBatch**。注意，大家不要和“producer 开发”一章中的 batch 相混淆——Java 版本 producer 中的 batch 是另外的含义，与这里的 batch 不是一个概念。

首先，我们来看 V2 版本的消息格式，如图 6.4 所示。

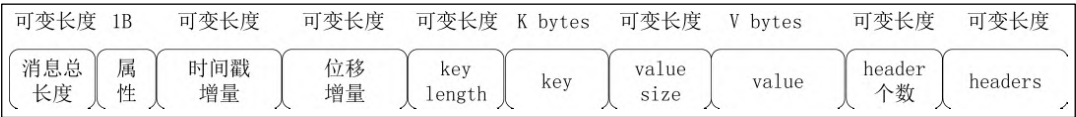


图 6.4 V2 版本消息格式

图 6.4 中“可变长度”表示 Kafka 会根据具体的值来确定到底需要几字节保存。为了在序列化时降低使用的字节数，V2 版本借鉴了 Google ProtoBuffer 中的 Zig-zag 编码方式，使得绝对值较小的整数占用比较少的字节。这通常是符合实际使用场景的，毕竟在生产环境中，key 或 value（特别是 key）非常大的情况其实并不多见。设想用户使用一个具有业务含义的字符串来标识 key（这是常见的 key 指定方法），那么该字符串长度一般不会太长，这样在大部分情况下只用 1 字节就能保存了。相比 V0、V1 版本中无差别地使用 4 字节来保存要节省 3 字节。

Zig-zag 编码方式主要的思想就是将一个有符号 32 位整数编码成一个无符号整数，同时各个数字围绕 0 依次编码，如下：

0 编码成 (➡) 0
 -1 ➡ 1
 1 ➡ 2
 -2 ➡ 3
 2 ➡ 4
 ...

由此可见，这种编码方式可使用较少的字节来保存绝对值很小的负数，比如-1 会被编码成 1，而不用像以前的做法去保存 32 位整数的补码。实际上，32 位补码中大部分都是 1，并没有包含太多有价值的信息，很多位都被浪费了。

由于可能使用多字节来编码一个数字，Zig-zag 会固定地将每个字节的第 1 位留作特殊用途，来表明该字节是否是某个数编码的最后一个字节，即最高位若是 1，则表明编码尚未结束，

还需要读取后面的字节来获取完整编码；若是 0，则表示下一个字节是新的编码。鉴于这个原因，Zig-zag 中每个字节只有 7 位可用于实际的编码任务，因此单个字节只能编码 0~127 之间的无符号整数。从上面的编码表中我们可以发现，所有正整数都会被编码为其 2 倍的数字，即 1 编码成 2，2 编码成 4，所以一旦消息的 key 或 value 长度超过了 $128/2=64$ ，长度信息就需要使用 2 字节来保存。正如我们前面说的，一般 key 的长度都不会超过 64，因此 V2 版本下大部分的消息只使用 1 字节便可以保存 key 长度信息。读者可以访问 <https://developers.google.com/protocol-buffers/docs/encoding> 进一步了解 Google PB 详细的编码规则。

V2 版本的消息格式变化之大不仅仅体现在引入了可变长度，同时它还新增、删除以及重构了一些消息字段，分别如下。

- 增加消息总长度字段：在消息格式的头部增加该字段，一次性计算出消息总字节数后保存在该字段中，而不需要像之前版本一样每次重新计算。Kafka 操作消息时可直接获取总字节数，直接创建出等大小的 ByteBuffer，然后分别装填其他字段，简化了消息处理过程。总字节数的引入还实现了消息遍历时的快速跳跃和过滤，省去了很多空间拷贝的开销。
- 保存时间戳增量（timestamp delta）：不再需要使用 8 字节来保存时间戳信息，而是使用一个可变长度保存与 batch 起始时间戳的差值。差值通常都是很小的，故需要的字节数也是很少的，从而节省了空间。假设一个 batch 中有 100 条消息，所有消息的时间戳差值都小于 64 毫秒（为什么用 64 来举例的原因，上面介绍 Zig-zag 时已经说到了），那么只需要 100 字节就能保存这些消息的时间信息。而 V0、V1 版本则需要 800 字节，整整浪费了 7 倍的空间。
- 保存位移增量（offset delta）：与时间戳增量类似，保存消息位移与外层 batch 起始位移的差值，而不再固定保存 8 字节的位移值，进一步节省消息总字节数。
- 增加消息头部（message headers）：这与之前本章介绍的消息头部概念完全不同。前面提及的消息头部属于元数据信息，对用户来说是透明的，而该字段是 0.11.0.0 版本新引入的，对用户是可见的。V2 版本中每条消息都必须有一个头部数组，里面的每个头部信息只包含两个字段：头部 key 和头部 value，类型分别是 String 和 byte[]。增加头部信息主要是为了满足用户的一些定制化需求，比如，做集群间的消息路由之用或承载消息的一些特定元数据信息。
- 去除消息级 CRC 校验：V2 版本不再为每条消息计算 CRC32 值，而是对整个消息 batch 进行 CRC 校验。
- 废弃 attribute 字段：V0、V1 版本格式都有一个 attribute 字段，V2 版本的消息正式废弃了这个字段。原先保存在 attribute 字段中的压缩类型、时间戳等信息都统一保存在外层的 batch 格式字段中，但 V2 版本依然保留了单字节的 attribute 字段留作以后扩展使用。

下面结合几个示例来计算 V2 版本的消息总字节数，我们依然使用之前的两条 Kafka 消息：一条消息 key 是 “key”，value 是 “hello”；另一条未指定 key，value 也是 “hello”。现在假设这两条消息都没有指定 headers，即 headers 数组长度是 0。结合图 6.4 可以计算第 1 条消息的长度 = 1(消息总长度) + 1(attribute) + 1(timestamp delta) + 1(offset delta) + 1(key length) + 3(key) + 1(value length) + 5(value) + 1(headers length) = 15 字节，同时消息总长度字段会保存值 15。然后计算第 2 条消息的总字节数，为了使问题更加真实，我们假设第 2 条消息的时间戳增量是 100，位移增量也是 100，同时 headers 数组中指定了一个 header，header key 是 “key”，header value 是 “hello”，故这条消息总字节数=1(消息总长度) + 1(attribute) + 2(timestamp delta) + 2(offset delta) + 1(key length) + 3(key) + 1(value length) + 5(value) + 1(headers length) + 1(头部 key 长度) + 3(头部 key) + 1(头部 value 长度) + 5(头部 value) = 27 字节。

说完了消息格式，下面讨论一下消息 batch。V2 版本的消息 batch 格式非常复杂，如图 6.5 所示。

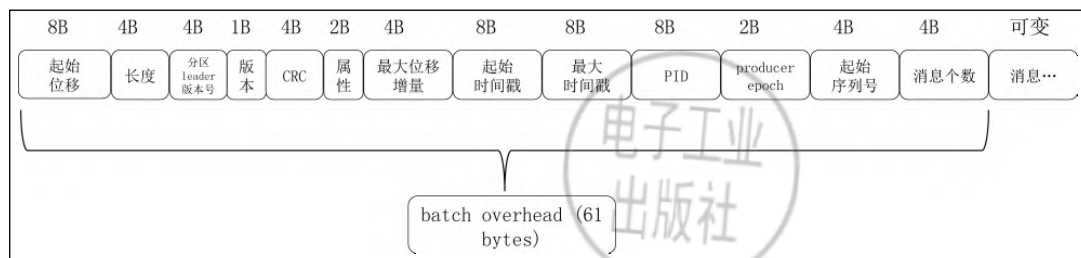


图 6.5 V2 版本消息 batch 格式

显然，这比 V0、V1 版本的日志项要复杂得多，下面简要介绍它们的主要区别。

- CRC 值从消息层面被移除，被放入 batch 这一层。
- batch 层面上增加了一个双字节 attribute 字段，同时废弃了消息级别的 attribute 字段。在这个双字节的 attribute 字段中，最低的 3 位依然保存压缩类型，第 4 位依然保存时间戳类型，而第 5、6 位分别保存 0.11.0.0 版本新引入的事务类型和控制类型。
- PID、producer epoch 和序列号等信息都是 0.11.0.0 版本为了实现幂等性 producer 和支持事务而引入的。PID 表示一个幂等性 producer 的 ID 值，producer epoch 表示某个 PID 携带的当前版本号，broker 使用 PID 和 epoch 来确定当前合法的 producer 实例，并以此阻止过期 producer 向 broker 生产消息。序列号的引入主要是为了实现消息生产的幂等性。Kafka 依靠它来辨别消息是否已成功提交，从而防止出现重复生产消息。
- V2 版本的消息 batch 开销 (overhead) 增加到 61 字节，看上去似乎比 V0、V1 版本的增加了不少，但 V2 版本的 batch 允许包含多条 Kafka 消息。因此这些新字段的开销实际上被摊到底层的每条消息上，故在总体上反而是节省了空间。

和之前一样，我们通过一个示例来说明如何计算 V2 版本的消息 batch 大小。我们仍然以之前的那两条 Kafka 消息为例。假设第 1 条消息被封装进一个 batch 中，那么该 batch 的总字节数=61 + 15 = 76 字节。下面做一个实验来验证：

```
> bin/kafka-topics.sh --create --topic test --zookeeper localhost:2181 -
-partitions 1 --replication-factor 1
Created topic "test".
> bin/kafka-console-producer.sh --topic test --broker-list localhost:9092 -
-property parse.key=true --property key.separator=:
>key:hello
^C
> pwd
kafka-datalogs/kafka_1/test-0
> ll *.log
-rw-r--r--  1 huxi  staff   76 Jul  6 14:38 00000000000000000000.log
```

测试结果表明，我们对 V2 版本消息 batch 的计算是正确有效的。如果用 V2 版本的格式与之前版本做比较，似乎 V2 版本占用的磁盘空间反而增加了，这是因为我们的实验中每个 batch 只包含一条消息。如果我们改用 Java API 程序批量发送消息，则可以马上发现两者的不同。图 6.6 和图 6.7 分别展示了批量发送未压缩消息和已压缩消息时两个版本磁盘空间占用的对比情况。

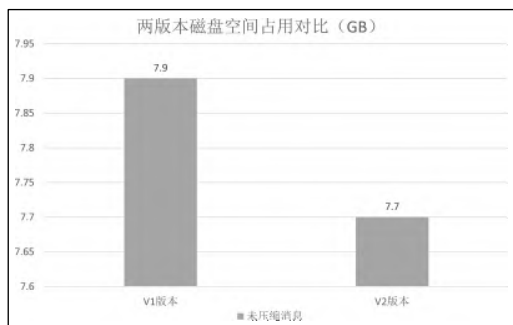


图 6.6 未压缩消息磁盘占用对比

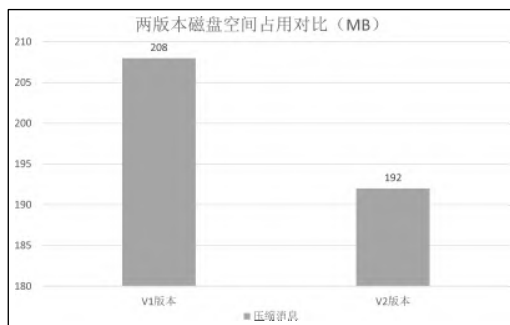


图 6.7 已压缩消息磁盘占用对比

由此可见，在未有任何调优的情况下，V2 版本消息格式确实可以节省磁盘空间。我们可以说 0.11.0.0 版本（及以后）的 Kafka 消息在支持事务、幂等性 producer 的同时还在一定程度上减少了网络 I/O 和磁盘 I/O 的开销。

6.1.2 集群管理

1. 成员管理

众所周知，Kafka 是分布式的消息引擎集群环境，它支持自动化的服务发现与成员管理。那么它是如何做到的呢？一言以蔽之，Kafka 是依赖 Apache ZooKeeper 实现的。每当一个 broker 启动时，它会将自己注册到 ZooKeeper 下的一个节点。图 6.8 展示了一个 Kafka 0.10.2.1 版本 broker 在 ZooKeeper 中注册的典型范例。

首先，每个 broker 在 ZooKeeper 下注册节点的路径是 `chroot/brokers/ids/<broker.id>`。如果没有配置 `chroot`，则路径是 `/brokers/ids/<broker.id>`。是否配置了 `chroot` 取决于 `server.properties` 中的 `zookeeper.connect` 参数是否设置了 `chroot`。图 6.8 使用 ZooKeeper 提供的客户端直接访问 ZooKeeper 服务器去获取该 broker 的注册信息。

```
[zk: localhost:2181(CONNECTED) 3] get /brokers/ids/0
{"listener_security_protocol_map":{"PLAINTEXT":{"PLAINTEXT":{"endpoints":["PLAINTEXT://localhost:9092"],"rack":"test-rack","jmx_port":9997,"host":"localhost","timestamp":"1499672525559","port":9092,"version":4}
cZxid = 0x1a
ctime = Mon Jul 10 15:42:32 CST 2017
mZxid = 0x1a
mtime = Mon Jul 10 15:42:32 CST 2017
pZxid = 0x1a
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x15d2b7316ad0000
dataLength = 209
numChildren = 0
[zk: localhost:2181(CONNECTED) 4] █
```

图 6.8 ZooKeeper 下 broker 注册信息

其次，broker 向 ZooKeeper 中注册的信息以 JSON 格式保存，其中包括的信息如下。

- `listener_security_protocol_map`: 此值指定了该 broker 与外界通信所用的安全协议类型。Kafka 当前支持 broker 使用不同的安全协议类型与 clients 和其他 broker 通信。比如，用户可以配置成与公网 clients 通信使用 SSL 传输数据，而与内网其他 broker 通信使用 PLAINTEXT。本例中笔者只配置唯一的一种通信方式 PLAINTEXT，即 broker 与外界的一切数据交互都以明码文本的方式传输。
- `endpoints`: 指定 broker 的服务 endpoint 列表，每个 endpoint 指明了传输协议类型、broker 主机名和端口信息。既然是列表，说明 endpoint 可以配置多个，每种协议类型都可以配置一个对应的 endpoint，只是端口号不能冲突。
- `rack`: 指定 broker 机架信息。和 Hadoop 机架感知原理类似，若设置了机架信息，Kafka 在分配副本时会考虑把某个分区的多个副本分配在多个机架架上。这样即使某个机架上的 broker 全部崩溃，也能保证其他机架上的副本可以正常工作。
- `jmx_port`: broker 的 JMX 监控端口，需要在启动 broker 前设置 `JMX_PORT` 环境变量。设置 JMX 端口后各种支持 JMX 的监控框架（如 Zabbix 等）才可以实时获取 Kafka 提供的各种 broker 端监控指标。

- host: broker 主机名或 IP 地址。
- port: broker 服务端口号。
- timestamp: broker 启动时间。
- version: broker 当前版本号。

上面最后一个字段 version 表征了 broker 的当前版本，注意，这不是 Kafka 的版本，而是 broker 注册信息的版本。Kafka 自开源以来总共提供了 4 个版本的 broker 信息，当前最新版本号是 4。图 6.9 至图 6.12 分别展示了 V1~V4 这 4 个版本的格式。对于想要自己写程序获取 broker 信息的用户而言，一定要注意所使用 Kafka 的版本并获取对应版本的 broker 注册信息。

```
{
  "version":1,
  "host":"localhost",
  "port":9092,
  "jmx_port":9999,
  "timestamp":"1499737197"
}
```

图 6.9 V1 版本 broker 注册信息

```
{
  "version":2,
  "host":"localhost",
  "port":9092,
  "jmx_port":9999,
  "timestamp":"1499737197",
  "endpoints":[
    "PLAINTEXT://host1:9092",
    "SSL://host1:9093"
  ]
}
```

图 6.10 V2 版本 broker 注册信息

```
{
  "version":3,
  "host":"localhost",
  "port":9092,
  "jmx_port":9999,
  "timestamp":"1499737197",
  "endpoints":[
    "PLAINTEXT://host1:9092",
    "SSL://host1:9093"
  ],
  "rack":"dc1"
}
```

图 6.11 V3 版本 broker 注册信息

```
{
  "version":4,
  "host":"localhost",
  "port":9092,
  "jmx_port":9999,
  "timestamp":"1499737197",
  "endpoints":[
    "CLIENT://host1:9092",
    "REPLICATION://host1:9093"
  ],
  "listener_security_protocol_map":{
    "CLIENT":"SSL",
    "REPLICATION":"PLAINTEXT"
  },
  "rack":"dc1"
}
```

图 6.12 V4 版本 broker 注册信息

最后，请注意图 6.8 中底部的 ephemeralOwner 值，该值不是 0，表示这是一个 ZooKeeper 中的临时节点（ephemeral node）。ZooKeeper 临时节点的生命周期和客户端会话绑定。如果客户端会话失效，该临时节点就会自动被清除掉。Kafka 正是利用 ZooKeeper 临时节点来管理 broker 生命周期的。broker 启动时在 ZooKeeper 中创建对应的临时节点，同时还会创建一个监听器（listener）监听该临时节点的状态；一旦 broker 启动后，监听器会自动同步整个集群信息到该 broker 上；而一旦该 broker 崩溃，它与 ZooKeeper 的会话就会失效，导致临时节点被

删除，监听器被触发，然后处理 broker 崩溃的后续事宜。这就是 Kafka 管理集群及其成员的主要流程。

2. ZooKeeper 路径

尽管新版本 producer 和 consumer 都已不再需要连接 ZooKeeper 了，但 Kafka 依然重度依赖于 ZooKeeper。ZooKeeper 从某种程度上甚至可以说是 Kafka 的“单点失效”组件。一旦 ZooKeeper 服务挂掉，Kafka 集群的很多组件也就无法正常工作。全面掌握 Kafka 使用到的各个 ZooKeeper 路径对于我们深入了解 Kafka 是大有裨益的。

图 6.13 涵盖了 Kafka 用到的各个 ZooKeeper 节点，我们会对每个路径做一个简要的说明。

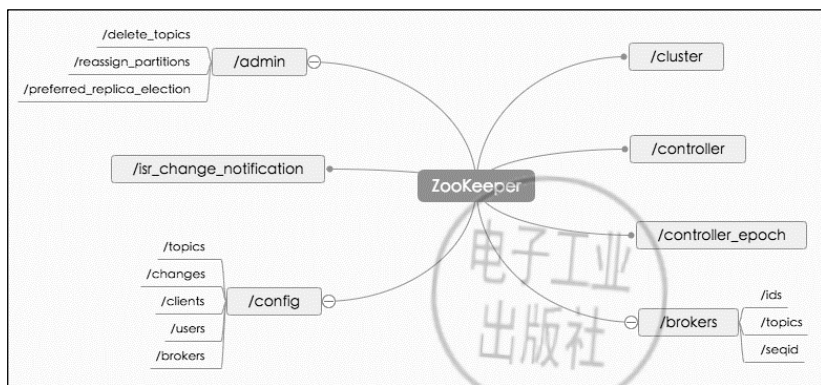


图 6.13 ZooKeeper 路径图

- `/brokers`: 里面保存了 Kafka 集群的所有信息，包括每台 broker 的注册信息，集群上所有 topic 的信息等。
- `/controller`: 保存了 Kafka controller 组件（controller 负责集群的领导者选举，6.1.7 节将会详细讨论 controller）的注册信息，同时也负责 controller 的动态选举。
- `/admin`: 保存管理脚本的输出结果，比如删除 topic，对分区进行重分配等操作。
- `/isr_change_notification`: 保存 ISR 列表发生变化的分区列表。controller 会注册一个监听器实时监控该节点下子节点的变更。
- `/config`: 保存了 Kafka 集群下各种资源的定制化配置信息，比如每个 topic 可能有自己专属的一组配置，那么就保存在 `/config/topics/<topic>` 下。
- `/cluster`: 保存了 Kafka 集群的简要信息，包括集群的 ID 信息和集群版本号。
- `/controller_epoch`: 保存了 controller 组件的版本号。Kafka 使用该版本号来隔离无效的 controller 请求。

6.1.3 副本与 ISR 设计

第1章中概要介绍了 Kafka 的副本、分区和 ISR 等概念。本节将深入介绍 Kafka 的副本及 ISR 工作原理。

首先回顾一下，一个 Kafka 分区本质上就是一个备份日志，即利用多份相同的备份共同提供冗余机制来保持系统高可用性。这些备份在 Kafka 中被称为副本（replica）。Kafka 把分区的所有副本均匀地分配到所有 broker 上，并从这些副本中挑选一个作为 leader 副本对外提供服务，而其他副本被称为 follower 副本，只能被动地向 leader 副本请求数据，从而保持与 leader 副本的同步，如图 6.14 所示。

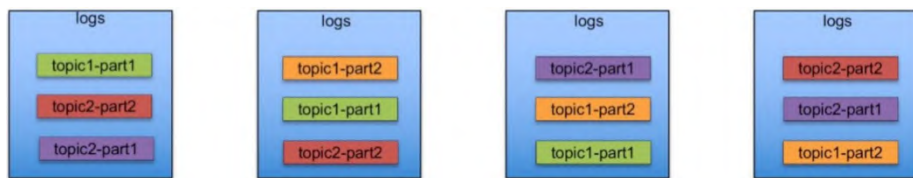


图 6.14 leader 和 follower 副本分配图

（图片来自 <https://www.slideshare.net/HadoopSummit/apache-kafka-best-practices>）

假如 leader 副本永远工作正常，那么其实不需要 follower 副本。但现实总是残酷的，Kafka leader 副本所在的 broker 可能因为各种各样的原因而随时宕机。一旦发生这种情况，follower 副本会竞相争夺成为新 leader 的权力。显然不是所有的 follower 都有资格去竞选 leader。前面说过，follower 被动地向 leader 请求数据。对于那些落后 leader 进度太多的 follower 而言，它们是没有资格竞选 leader 的，毕竟它们手中握有的数据太旧了，如果允许它们成为 leader，会造成数据丢失，而这对 clients 而言是灾难性的。鉴于这个原因，Kafka 引入了 ISR 的概念。

所谓 ISR，就是 Kafka 集群动态维护的一组同步副本集合（in-sync replicas）。每个 topic 分区都有自己的 ISR 列表，ISR 中的所有副本都与 leader 保持同步状态。值得注意的是，leader 副本总是包含在 ISR 中的，只有 ISR 中的副本才有资格被选举为 leader。而 producer 写入的一条 Kafka 消息只有被 ISR 中的所有副本都接收到，才被视为“已提交”状态。由此可见，若 ISR 中有 N 个副本，那么该分区最多可以忍受 $N-1$ 个副本崩溃而不丢失已提交消息。

1. follower 副本同步

follower 副本只做一件事情：向 leader 副本请求数据。在详细讲解 follower 副本如何请求数据之前，我们首先明确一些术语和概念以方便后续的讨论，如图 6.15 所示。

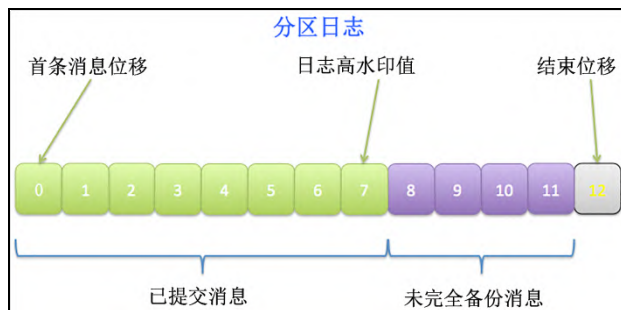


图 6.15 副本各种位置信息

图 6.15 中比较重要的位移信息如下。

- 起始位移（base offset）：表示该副本当前所含第一条消息的 offset。
- 高水印值（high watermark, HW）：副本高水印值。它保存了该副本最新一条已提交消息的位移。leader 分区的 HW 值决定了副本中已提交消息的范围，也确定了 consumer 能够获取的消息上限，超过 HW 值的所有消息都被视为“未提交成功的”，因而 consumer 是看不到的。另外值得注意的是，不是只有 leader 副本才有 HW 值。实际上每个 follower 副本都有 HW 值，只不过只有 leader 副本的 HW 值才能决定 clients 能看到的消息数量罢了。
- 日志末端位移（log end offset, LEO）：副本日志中下一条待写入消息的 offset。所有副本都需要维护自己的 LEO 信息。每当 leader 副本接收到 producer 端推送的消息，它会更新自己的 LEO（通常是加 1）。同样，follower 副本向 leader 副本请求到数据后也会增加自己的 LEO。事实上只有 ISR 中的所有副本都更新了对应的 LEO 之后，leader 副本才会向右移动 HW 值表明消息写入成功。整个流程如图 6.16 所示。

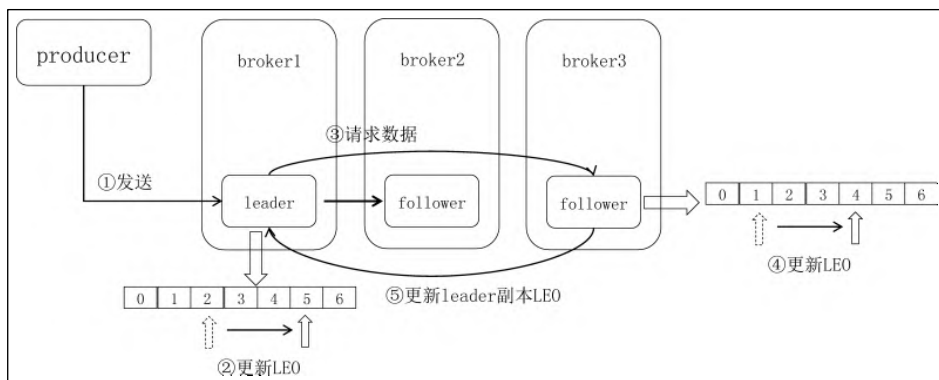


图 6.16 follower/leader 副本同步流程

为了形象化地说明图 6.16 的流程，下面结合一个具体的示例来说明。假设图 6.16 中的 Kafka 集群当前只有一个 topic，该 topic 只有一个分区，分区共有 3 个副本，因此 ISR 中也是这 3 个副本。该 topic 当前没有任何数据。由于没有任何数据，因此 3 个副本的 LEO 都是 0，HW 值是 0。

现有一个 producer 向 broker1 所在的 leader 副本发送了一条消息，接下来会发生什么呢？

(1) broker1 上的 leader 副本接收到消息，把自己的 LEO 值更新为 1。

(2) broker2 和 broker3 上的 follower 副本各自发送请求给 broker1。

(3) broker1 分别把该消息推送给 follower 副本。

(4) follower 副本接收到消息后各自更新自己的 LEO 为 1。

(5) leader 副本接收到其他 follower 副本的数据请求响应 (response) 之后，更新 HW 值为 1。此时位移为 0 的这条消息可以被 consumer 消费。

对于设置了 `acks=-1` (acks 的含义请参考第 4 章) 的 producer 而言，只有完整地做完上面所有的 5 步操作，producer 才能正常返回，这也标志着这条消息发送成功。

2. ISR 设计

有了副本、备份、同步等背景之后，我们可以聊聊 ISR 的设计了。之前我们一直说 ISR 是与 leader 同步的副本集合，但同步的具体含义是什么？follower 副本与 leader 副本不同步表示什么意思？不同步 (out-of-sync) 意味着 follower 副本无法追上 leader 副本的 LEO，而这又是什么意思？

对于如何界定 ISR，不同 Kafka 版本的界定方法不同，大体上可分为 0.9.0.0 版本之前和 0.9.0.0 版本 (含) 之后两种方法。鉴于现在依然有很多用户在使用 Kafka 0.8.2.x 版本，下面首先介绍在这个版本中是如何判定 ISR 的。

(1) 0.9.0.0 版本之前

0.9.0.0 版本之前，Kafka 提供了一个参数 `replica.lag.max.messages`，用于控制 follower 副本落后 leader 副本的消息数。一旦超过这个消息数，则视为该 follower 为“不同步”状态，从而需要被 Kafka “踢出”ISR。

我们举一个实际的例子。假设有一个单分区的 topic，副本数是 3，这 3 个副本分别保存在 broker1、broker2 和 broker3 上。leader 副本在 broker1 上，其他两个 broker 上的副本都是 follower 副本。现设置 `replica.lag.max.messages` 为 4，此时有一个 producer 每次都给这个 topic 发送 3 条消息，那么初始状态如图 6.17 所示。

初始状态下所有 follower 副本都是和 leader 副本同步的，所有 follower 都能追上 leader 的 LEO。现在假设 producer 生产了 1 条消息给 leader，而 broker3 上的 follower 副本经历了一次 Full GC，这时日志的状态则如图 6.18 所示。

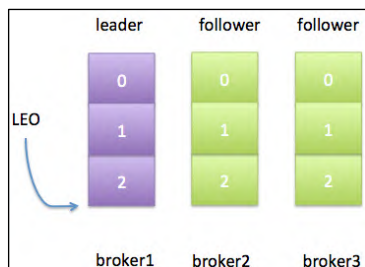


图 6.17 ISR 示例初始状态

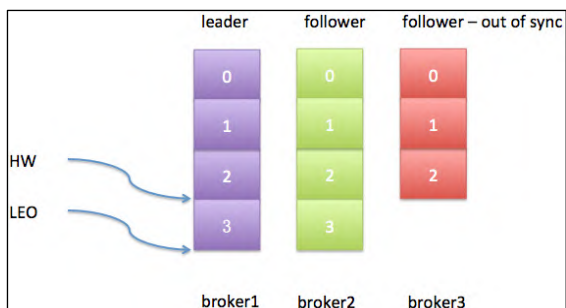


图 6.18 ISR 示例状态 1

此时，更新之后 leader 的 LEO 不再与 HW 值相等，但最新生产的这条消息不会被认为“已提交”，除非 broker3 上的 follower 副本被“踢出”ISR 或者追上 leader 的 LEO。由于 `replica.lag.max.messages` 被设置为 4，而 broker3 上的 follower 只落后 1 条消息，并不满足“不同步”条件，因此不会从 ISR 中移除。对于 broker3 上的副本而言，只需要追上 leader 的 LEO。如果我们假设 broker3 在执行 Full GC 停顿了 100 毫秒之后重新追上了 leader 的进度，那么此时的日志状态如图 6.19 所示。

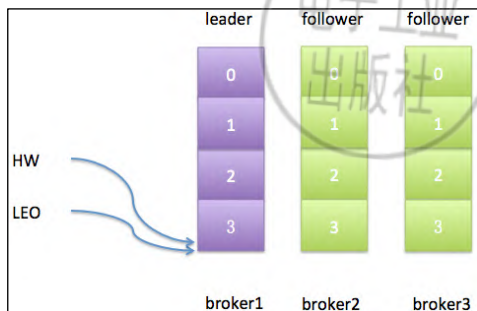


图 6.19 ISR 示例状态 2

好了，现在一切都恢复正常了。leader 的 HW 值与 LEO 再次重叠，而两个 follower 也与 leader 同步。那么，除了上面举例说明的 GC，还可能有哪些原因导致 follower 与 leader 不同步呢？归纳起来主要有如下 3 个原因。

- **请求速度追不上：** follower 副本在一段时间内都无法追上 leader 副本端的消息接收速度。比如 follower 副本所在 broker 的网络 I/O 开销过大导致备份消息的速度持续慢于从 leader 处获取消息的速度。
- **进程卡住：** follower 在一段时间内无法向 leader 请求数据，比如之前提到的频繁 GC 或程序 bug 等。

- **新创建的副本：**如果用户增加了副本数，那么新创建的 follower 副本在启动后全力追赶 leader 进度。在追赶进度这段时间内通常都是与 leader 不同步的。

上面的 `replica.lag.max.messages` 参数便是用于检测第一种情况的。另外，0.9.0.0 版本之前还提供了另一个参数 `replica.lag.time.max.ms` 用于检测另外两种情况。比如设置 `replica.lag.time.max.ms` 为 500 毫秒，若 follower 副本无法在 500 毫秒内向 leader 请求数据，那么该 follower 就会被视为“不同步”，即会被踢出 ISR。

0.9.0.0 版本之前的这种 ISR 方案在设计上有一些固有的缺陷。为了说明我们依然使用之前提到的例子。如果 producer 一次性发送消息的速度是 2 条/秒，即每个 producer batch 都包含 2 条消息。显然，此时设置 `replica.lag.max.messages=4` 是相当安全且合适的数值。为什么？因为在 leader 副本接收到 producer 发送过来的消息之后且 follower 副本开始备份这些消息之前，follower 副本落后 leader 的消息数不会超过 3 条。但如果 follower 副本落后 leader 的消息数超过 3 条，那么我们肯定希望 leader 把这个特别慢的 follower 副本踢出 ISR 以防止增加 producer 消息生产的延时。从这个简单的例子来看，这个参数似乎工作得很好，为什么说它是有缺陷的呢？

根本原因在于如果要正确设置这个参数的值，需要用户结合具体使用场景评估，而没有统一的设置方法。下面详细解释一下根本原因。首先，对于一个参数的设置，有一点是很重要的：用户应该对他们知道的参数进行设置，而不是对他们需要进行猜测的参数进行设置。对于该参数来说，我们只能猜测应该设置成哪些值，而不是根据需要进行设置。为什么？举一个例子，假设在刚才那个 topic 的环境中，producer 程序突然发起了一波消息生产的瞬时高峰流量，比如 producer 一次性发送 4 条消息，也就是说，消息数与 `replica.lag.max.messages` 值相等。此时，这两个 follower 副本都会被认为与 leader 副本不同步，从而被踢出 ISR，具体日志状态如图 6.20 所示。

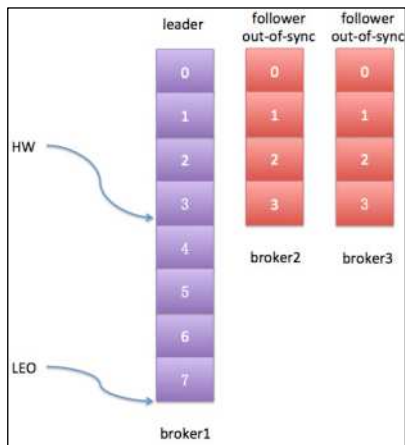


图 6.20 ISR 示例状态 3

从图 6.20 看，这两个 follower 副本与 leader 不再同步，但其实它们都处于存活状态（alive）且没有任何性能问题，下次 `FetchRequest` 时它们就能追上 leader 的 LEO，并重新被加入 ISR——于是就出现了这样的情况：它们不断地被踢出 ISR，然后重新加回 ISR，造成了与 leader 不同步、再同步、又不同步、再次同步的情况发生。想想就知道这是多大的开销！问题的关键就在 `replica.lag.max.messages` 这个参数上。用户通过猜测设置该值，猜测 producer 的速度，猜测 leader 副本的入站流量。

可能有用户会说该参数默认值是 4000 就应该足够使用了。但有一点需要注意的是，这个参数是全局的！即所有 topic 都受到这个参数的影响。假设集群中有两个 topic：t1 和 t2。若它们的流量差异非常巨大，t1 的消息生产者一次性生产 5000 条消息，直接就突破了 4000 这个默认值；而另一个 topic——t2，它的消息生产者一次性生产 10 条消息，那么 Kafka 就需要相当长的时间才能辨别出 t2 各个分区中那些滞后的副本。一旦出现有 broker 崩溃的情况，极易造成状态的不一致。

鉴于这些原因，在 0.9.0.0 版本及之后，Kafka 社区优化了 ISR 方案的设计。

（2）0.9.0.0 版本之后

自 0.9.0.0 版本之后，Kafka 去掉了之前的 `replica.lag.max.messages` 参数，改用统一的参数同时检测由于慢以及进程卡壳而导致的滞后（lagging）——即 follower 副本落后 leader 副本的时间间隔。这个唯一的参数就是 `replica.lag.time.max.ms`，默认值是 10 秒。对于“请求速度追不上”的情况，检测机制也发生了变化——如果一个 follower 副本落后 leader 的时间持续性地超过了这个参数值，那么该 follower 副本就是“不同步”的。这样即使出现刚刚提到的 producer 瞬时峰值流量，只要 follower 不是持续性落后，它就不会反复地在 ISR 中移进、移出。

6.1.4 水印（watermark）和 leader epoch

6.1.3 节简要介绍了 Kafka 提供的副本以及围绕副本实现的备份机制，其中谈到了水印并介绍了水印在备份机制中的作用。为了更加全面地了解 Kafka 备份机制的设计与完善，笔者打算专门花一节的篇幅来讨论水印及其工作原理。弄懂它后，读者就完全掌握了 Kafka 内部备份机制的原理。同时，笔者也会给出当前水印机制的一些弊端以及 Kafka 社区是如何改进它的。这也是它的替代者——leader epoch——的由来。

水印也被称为高水印或高水位，通常被用在流式处理领域（著名的框架如 Apache Storm、Apache Flink 和 Apache Spark 等），以表征元素或事件在基于时间层面上的进度。一个比较经典的表述为：流式系统保证在水印 t 时刻，创建时间（event time） $= t'$ 且 $t' \leq t$ 的所有事件都

已经到达或被观测到。在 Kafka 中，水印的概念反而与时间无关，而与位置信息相关。严格来说，它表示的就是位置信息，即位移（offset）。由于 Kafka 源码中使用的名字就是 highwatermark，故在本节中我们称之为高水印。

一个 Kafka 分区下通常存在多个副本（replica）用于实现数据冗余，进一步实现高可用性。如前所述，副本根据角色不同分为如下 3 类。

- leader 副本：响应 clients 端读/写请求的副本。
- follower 副本：被动地备份 leader 副本上的数据，不能响应 clients 端的读/写请求。
- ISR 副本集合：包含 leader 副本和所有与 leader 副本保持同步的 follower 副本。

每个 Kafka 副本对象都持有两个重要的属性：日志末端位移（log end offset，下称 LEO）和高水印（HW）。注意是所有的副本，而不止是 leader 副本。以下是这两个属性的解释。

- LEO：日志末端位移，记录了该副本对象底层日志文件中下一条消息的位移值。举一个例子，若 LEO=10，那么表示在该副本日志上已经保存了 10 条消息，位移范围是[0, 9]。另外，Kafka 对 leader 副本和 follower 副本的 LEO 更新机制是不同的，后面我们会详细讨论。
- HW：我们已经很熟悉的高水印值。任何一个副本对象的 HW 值一定不大于其 LEO 值，而小于或等于 HW 值的所有消息被认为是“已提交的”或“已备份的”（replicated），如图 6.15 所示。Kafka 对 leader 副本和 follower 副本的 HW 值更新机制也是不同的，同理后面内容将会讨论它们的不同。

如果把 LEO 和 HW 看作两个指针，那么它们定位的机制是不同的：任意时刻，HW 指向的是实实在在的消息，而 LEO 总是指向下一条待写入消息，也就是说 LEO 指向的位置上是没有消息的！如果重新查看图 6.15，我们很容易发现 HW 值是 7，这表示前 8 条消息（位移从 0 计数）都已经处于“已备份状态”；而 LEO 值是 12，则表示当前日志中写入了 11 条消息，而消息 8、9、10、11 尚未完全备份，即它们属于未提交消息。

之前我们提到过消费者无法消费未提交消息。如果换作以上名词来解读的话，那么这句话应该表述为：消费者无法消费分区 leader 副本上那些位移大于分区 HW 的消息。分区 HW 就是 leader 副本的 HW 值。

介绍完基本名词，下面分别讨论一下 LEO 和 HW 值的更新机制。

1. LEO 更新机制

我们分别介绍 leader 副本更新 LEO 机制和 follower 副本更新 LEO 机制。首先，笔者将阐述 Kafka 如何更新 follower 副本的 LEO 属性。follower 副本只是被动地向 leader 副本请求数据，

具体表现为 follower 副本不停地向 leader 副本所在的 broker 发送 FETCH 请求（关于 FETCH 请求及 Kafka 通信协议，我们会在 6.1.6 节中讨论），一旦获取消息，便写入自己的日志中进行备份。

那么 follower 副本的 LEO 是何时更新的呢？严格来说，Kafka 设计了两套 follower 副本 LEO 属性：一套 LEO 值保存在 follower 副本所在 broker 的缓存上；另一套 LEO 值保存在 leader 副本所在 broker 的缓存上，笔者称后者为 remote LEO——换句话说，leader 副本所在机器的缓存上保存了该分区下所有 follower 副本的 LEO 属性值（当然也包括它自己的 LEO）。

为什么要保存两套值呢？这是因为 Kafka 需要利用前者帮助 follower 副本自身更新 HW 值，而同时还需要使用后者来确定 leader 副本的 HW 值，即分区 HW。

（1）follower 副本端的 follower 副本 LEO 何时更新？

虽然有些拗口，但 follower 副本端的 follower 副本 LEO 值就是指该副本对象底层日志的 LEO 值，也就是说，每当新写入一条消息，其 LEO 值就会加 1。在 follower 发送 FETCH 请求后，leader 将数据返回给 follower，此时 follower 开始向底层 log 写数据，从而自动更新其 LEO 值。

（2）leader 副本端的 follower 副本 LEO 何时更新？

leader 副本端的 follower 副本 LEO 的更新发生在 leader 处理 follower FETCH 请求时。一旦 leader 接收到 follower 发送的 FETCH 请求，它首先会从自己的 log 中读取相应的数据，但是在给 follower 返回数据之前它先去更新 follower 的 LEO（即上面所说的第二套 LEO 值）。

接下来给出 leader 副本更新 LEO 的机制和时机。和 follower 更新 LEO 道理相同，leader 写 log 时就会自动更新它自己的 LEO 值。

2. HW 更新机制

同理，我们首先讨论 follower 副本 HW 属性的更新机制。follower 更新 HW 发生在其更新 LEO 之后，一旦 follower 向 log 写完数据，它就会尝试更新 HW 值。具体算法就是比较当前 LEO 值与 FETCH 响应中 leader 的 HW 值，取两者的小者作为新的 HW 值。这告诉我们一个事实：如果 follower 的 LEO 值超过了 leader 的 HW 值，那么 follower HW 值是不会越过 leader HW 值的。

比起 follower 副本的 HW 属性，我们更关心 leader 副本 HW 值的更新，因为它直接影响了分区数据对于 consumer 的可见性。在以下 4 种情况下，leader 会尝试更新分区 HW 值——切记是尝试，有可能因为不满足条件而不做任何更新。

- 副本成为 leader 副本时：当某个副本成为分区的 leader 副本，Kafka 会尝试更新分区 HW。这是显而易见的道理，毕竟分区 leader 发生了变更，这个副本的状态是一定要检查的。

- broker 出现崩溃导致副本被踢出 ISR 时：若有 broker 崩溃，则必须查看是否会波及此分区，因此检查分区 HW 值是否需要更新是有必要的。
- producer 向 leader 副本写入消息时：因为写入消息会更新 leader 的 LEO，故有必要再查看 HW 值是否也需要更新。
- leader 处理 follower FETCH 请求时：当 leader 处理 follower 的 FETCH 请求时，首先会从底层的 log 读取数据，之后再尝试更新分区 HW 值。

特别注意上面 4 个条件中的最后 2 个。它揭示了一个事实——当 Kafka broker 都正常工作时，分区 HW 值的更新时机有两个：leader 处理 PRODUCE 请求时和 leader 处理 FETCH 请求时。另外，leader 是如何更新它的 HW 值的呢？前面说过，leader broker 上保存了一套 follower 副本的 LEO 以及它自己的 LEO。当尝试确定分区 HW 时，它会选出所有满足条件的副本，比较它们的 LEO（当然也包括 leader 自己的 LEO），并选择最小的 LEO 值作为 HW 值。这里的满足条件主要是指副本要满足以下两个条件之一。

- 处于 ISR 中。
- 副本 LEO 落后于 leader LEO 的时长不大于 `replica.lag.time.max.ms` 参数值（默认值是 10 秒）。

乍看上去好像这两个条件说的是一回事，毕竟刚才定义 ISR 时也是用的这个参数。但某些情况下 Kafka 的确可能出现副本已经“追上”了 leader 的进度，但却不在 ISR 中的情况——比如，某个从 failure 中恢复的副本。如果 Kafka 只判断第一个条件，确定分区 HW 值时就不会考虑这些未在 ISR 中的副本，但这些副本已经具备了“立刻进入 ISR”的资格，因此就可能出现分区 HW 值越过 ISR 中副本 LEO 的情况——这肯定是不允许的，因为分区 HW 实际上就是 ISR 中所有副本 LEO 的最小值。

好了，理论部分介绍得足够多了。后续将用一个实际的例子来图解 Kafka 处理消息的全过程。通过对整个过程的研究，读者可以清晰地领会 Kafka 备份机制的原理。

为了讨论方便，我们假设有一个测试主题，单分区，副本因子是 2，即一个 leader 副本和一个 follower 副本。我们看看当生产者发送一条消息时，broker 端的副本到底会发生什么事情以及分区 HW 是如何被更新的。

3. 图解 Kafka 备份原理

图 6.21 是初始状态，稍微解释一下：初始时 leader 以及 follower 的 HW 和 LEO 都是 0（严格来说，源代码会初始化 LEO 为 -1，不过这不影响之后的讨论）。leader 中的 remote LEO 指的就是 leader 端保存的 follower LEO，也被初始化为 0。此时，producer 没有发送任何消息给 leader，而 follower 已经开始不断地给 leader 发送 FETCH 请求了，但因为没有任何数据，因此什么都

不会发生。值得一提的是，follower 发送过来的 FETCH 请求因为无数据而暂时被寄存在 leader 端的 purgatory 中，待 500 毫秒（replica.fetch.wait.max.ms 参数）超时后会强制完成。若在寄存期间 producer 端发送过来数据，那么 Kafka 会自动唤醒该 FETCH 请求，让 leader 继续处理。

purgatory 是 Kafka 暂存请求对象的地方。有一些请求由于各种各样的原因无法立即被处理，就会被 Kafka 放入 purgatory 中。虽然对 purgatory 的剖析不在本书讨论范围内，但其暂存的 FETCH 和 PRODUCE 请求的处理时机会影响 HW 值的更新，因此笔者将分两种情况来仔细分析。

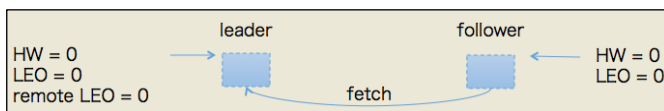


图 6.21 副本属性初始状态图

（1）情况 1：leader 副本写入消息后 follower 副本发送 FETCH 请求。

假设生产者给 topic 的某个分区发送了一条消息，此时状态如图 6.22 所示。

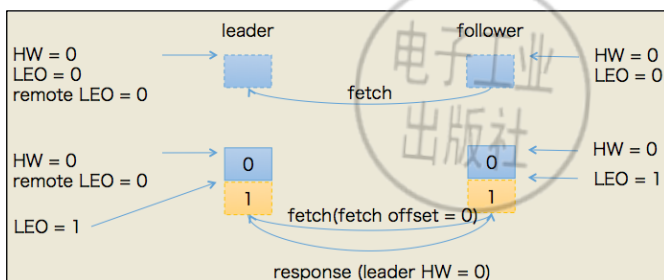


图 6.22 生产者发送消息后

如图 6.22 所示，leader 接收到生产消息的请求后，主要做如下两件事情。

①写入消息到底层日志，同时更新 leader 副本的 LEO 属性。

②尝试更新 leader 副本的 HW 值（满足 leader HW 更新条件的第 3 个条件）。我们已经假设此时 follower 尚未发送 FETCH 请求，那么 leader 端保存的 remote LEO 依然是 0，因此 leader 会比较它自己的 LEO 值和 remote LEO 值，发现最小值是 0，与当前 HW 值相同，故不会更新分区 HW 值。

所以，写入消息成功后，leader 端的 HW 值依然是 0，而 LEO 是 1，remote LEO 是 1。假设此时 follower 发送了 FETCH 请求（或者说 follower 早已发送了 FETCH 请求，只不过在 broker 的请求队列中排队），那么状态变更如图 6.23 所示。

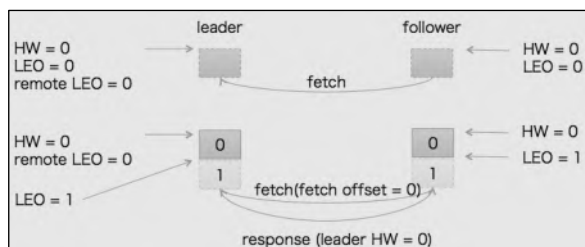


图 6.23 leader 副本写入消息后

此时 follower 发送 FETCH 请求，leader 端的处理逻辑依次如下。

①读取底层 log 数据。

②更新 remote LEO = 0：为什么是 0？因为此时 follower 还没有写入这条消息。leader 如何确认 follower 还未写入呢？这是通过 follower 发来的 FETCH 请求中的 fetch offset 来确定的。

③尝试更新分区 HW：此时 leader LEO = 1，remote LEO = 0，故分区 HW 值 = $\min(\text{leader LEO}, \text{follower remote LEO}) = 0$ 。

④把数据和当前分区 HW 值（依然是 0）发送给 follower 副本。

而 follower 副本接收到 FETCH response 后依次执行下列操作。

①写入本地 log（同时更新 follower LEO）。

②更新 follower HW——比较本地 LEO 和当前 leader HW[Offset1]后取较小值，故 follower HW = 0。

此时，第一轮 FETCH 请求处理结束，虽然 leader 和 follower 都已经在 log 中保存了这条消息，但分区 HW 值尚未被更新。实际上，它是在第二轮 FETCH RPC 中被更新的，如图 6.24 所示。

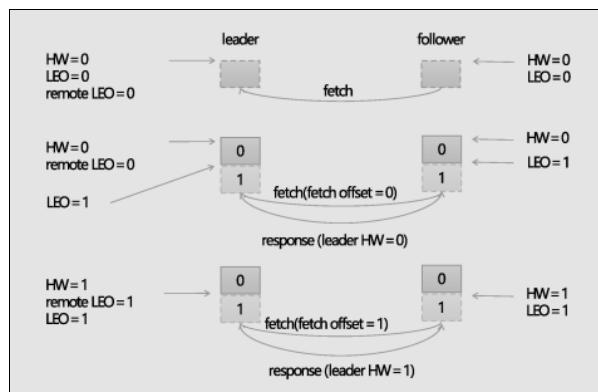


图 6.24 第二轮 FETCH 请求处理后

图 6.24 中, follower 发来了第二轮 FETCH 请求, leader 端接收到后仍然会依次执行下列操作。

①读取底层 log 数据。

②更新 remote LEO = 1 (这次为什么是 1? 因为这轮 FETCH RPC 携带的 fetch offset 是 1。那么为什么这轮携带的就是 1 呢? 因为上一轮结束后 follower LEO 被更新为 1)。

③尝试更新分区 HW——此时 leader LEO = 1, remote LEO = 1, 故分区 HW 值 = $\min(\text{leader LEO}, \text{follower remote LEO}) = 1$ 。注意分区 HW 值此时被更新了。

④把数据 (实际上没有数据) 和当前分区 HW 值 (已更新为 1) 发送给 follower 副本。

同样, follower 副本接收到 FETCH response 后依次执行下列操作。

①写入本地 log, 当然没东西可写, 故 follower LEO 也不会变化, 依然是 1。

②更新 follower HW——比较本地 LEO 和当前 leader LEO 后取较小值。由于此时两者都是 1, 故更新 follower HW = 1。

此时消息已经被成功地拷贝到 leader 和 follower 的 log 中且分区 HW 是 1, 表明 consumer 能够消费 offset = 0 的这条消息。这便是 Kafka 处理消息写入以及消息备份的完全流程。

下面来分析一下 PRODUCE 和 FETCH 请求交互的第二种情况。

(2) 情况 2: FETCH 请求保存在 purgatory 中时生产者发来消息。

这种情况实际上和第一种情况差不多。前面说过, 当 leader 无法立即满足 FETCH 返回要求的时候 (比如没有数据可返回), 那么该 FETCH 请求会被暂存到 leader 端的 purgatory 中, 待时机成熟时尝试再次处理它。不过 Kafka 不会无限期地将其缓存着, 默认会有一个超时时间 (500 毫秒), 一旦超时时间已过, 则这个请求会被强制完成。不过我们要讨论的场景是在寄存期间, producer 发送 PRODUCE 请求, 从而使之满足条件并被唤醒。此时, leader 端处理流程如下。

①leader 写入本地 log (同时自动更新 leader LEO)。

②尝试唤醒在 purgatory 中寄存的 FETCH 请求。

③尝试更新分区 HW。

唤醒后的 FETCH 请求的处理与第一种情况完全一致, 故这里不再赘述。

以上对图解流程的拆分其实就想说明一件事情: Kafka 使用 HW 值来决定副本备份的进度, 而 HW 值的更新通常需要另一轮 FETCH 请求才能完成, 故这种设计在本质上是存在缺陷的。

它们可能引起的问题如下。

- 备份数据丢失。
- 备份数据不一致。

下面逐一分析可能造成的问题。

4. 基于水印备份机制的缺陷

在 0.11.0.0 版本之前，Kafka 一直使用基于水印的备份机制，但在上面的分析中我们得知这种机制可能会引起两方面的问题。

(1) 数据丢失

如前所述，使用 HW 值来确定备份进度时其值的更新是在下一轮 RPC 中完成的。设想一下这样的场景：某 follower 发送了第二轮的 FETCH 请求给 leader，在接收到响应之后，它会首先写入本地日志——假设没有数据可写，故 follower LEO 不会发生变化。之后 follower 副本准备更新其 HW 值。此时故障发生了，follower 副本发生崩溃，那么这个时刻就可能造成数据丢失。图 6.25 给出了一个实例。

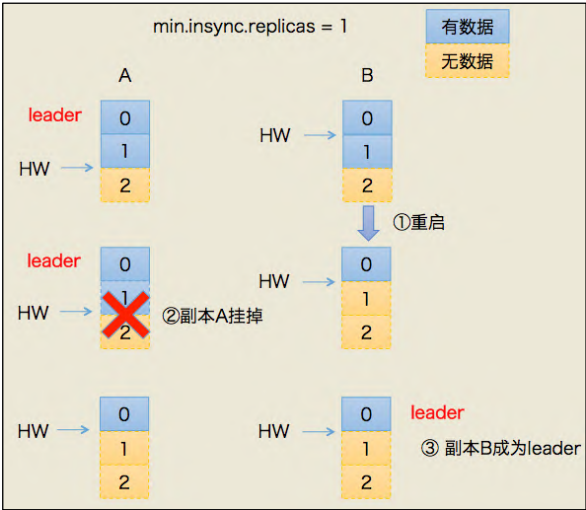


图 6.25 副本数据丢失实例

图 6.25 中有两个副本 A 和 B，开始状态为 A 是 leader。我们假设生产者端的参数 `min.insync.replicas` 被设置为 1，那么当生产者发送两条消息给 A 后，A 写入到底层 log，此时 Kafka 会通知生产者说这两条消息写入成功。

但在 broker 端，leader 和 follower 底层的 log 虽都写入了 2 条消息且分区 HW 已经被更新

到 2，但 follower HW 尚未被更新（因为其要额外多等一轮 FETCH 请求才会被更新）。若此时副本 B 所在的 broker 宕机，那么重启后 B 会自动把 LEO 调整到之前的 HW 值，故副本 B 会做日志截断（log truncation），将 offset = 1 的那条消息从 log 中删除，并调整 LEO = 1，此时 follower 副本底层 log 中就只有 1 条消息，即 offset = 0 的消息。

B 重启之后需要给 A 发 FETCH 请求，但若 A 所在 broker 机器此时宕机，那么 Kafka 会令 B 成为新的 leader，而当 A 重启回来后也会执行日志截断，将 HW 调整回 1。这样，位移=1 的消息就从两个副本的 log 中被删除，即永远丢失。

这个场景丢失数据的前提是在 `min.insync.replicas=1` 时，一旦消息被写入 leader 端，log 即被认为“已提交”，而延迟一轮 FETCH 请求更新 HW 值的设计使得 follower HW 值是异步延迟更新的，若在这个过程中 leader 发生变更，那么成为新 leader 的 follower 的 HW 值就有可能是过期的，使得 clients 端认为成功提交的消息被删除了。

（2）数据不一致/数据离散

除数据丢失风险之外，这种设计还有一个潜在的问题，即造成 leader 端 log 和 follower 端 log 的数据不一致，即数据离散的问题。举一个例子，假设 leader 端保存的消息序列是 `r1,r2,r3,r4,r5...`，而 follower 端保存的消息序列可能是 `r1,r3,r4,r5,r6...`。这也是非法的场景，因为顾名思义，follower 必须追随 leader，完整地备份 leader 端的数据。图 6.26 说明了这种场景是如何发生的。

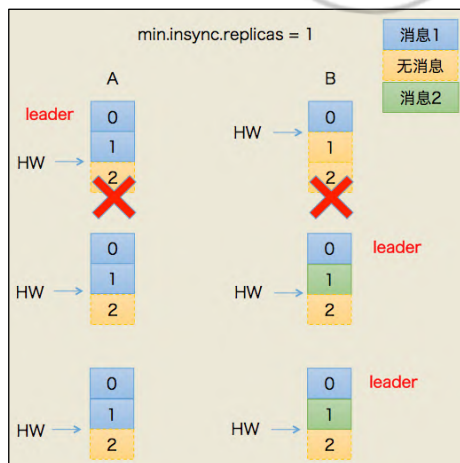


图 6.26 副本不一致实例

这种情况的初始状态与数据丢失场景有些许不同之处：A 依然是 leader，A 的 log 写入了 2 条消息，但 B 的 log 只写入了 1 条消息。分区 HW 更新到 2，但 B 的 HW 还是 1，同时生产者

端的 `min.insync.replicas = 1`。

这次我们让 A 和 B 所在机器同时挂掉，然后假设 B 先重启回来，因此成为 leader，分区 HW = 1。假设此时 producer 发送了第 3 条消息给 B，于是 B 的 log 中 `offset = 1` 的消息变成了绿色框表示的消息，同时分区 HW 更新到 2（A 还没有回来，就 B 一个副本，故可以直接更新 HW 而不用理会 A）之后 A 重启回来，需要执行日志截断，但发现此时分区 HW=2，而 A 之前的 HW 值也是 2，故不做任何调整。此后 A 和 B 将以这种状态继续正常工作。

显然，在这种场景下，A 和 B 底层 log 中保存在 `offset = 1` 的消息是不同的记录，从而引发不一致的情形出现。

5. 0.11.0.0 版本解决之道

针对上面提到的两个问题，Kafka 社区于 0.11.0.0 版本中正式引入 leader epoch 值彻底解决了基于水印备份机制的这两个弊端。

我们已然知晓，造成上述两个问题的根本原因在于 HW 值被用于衡量副本备份的成功与否，以及在出现崩溃时作为日志截断的依据，但 HW 值的更新是异步延迟的，特别是需要额外的 FETCH 请求处理流程才能更新，故这中间发生的任何崩溃都可能导致 HW 值的过期。鉴于这些原因，Kafka 0.11.0.0 引入了 leader epoch 来取代 HW 值。leader 端多开辟一段内存区域专门保存 leader 的 epoch 信息，这样即使出现上面的两个场景，Kafka 也能很好地规避这些问题。

所谓领导者 epoch (leader epoch)，实际上是一对值(epoch,offset)。epoch 表示 leader 的版本号，从 0 开始，当 leader 变更过 1 次时，epoch 就会加 1，而 offset 则对应于该 epoch 版本的 leader 写入第一条消息的位移。假设存在两对值(0, 0)和(1, 120)，那么表示第一个 leader 从位移 0 开始写入消息，共写了 120 条，即[0, 119]；而第二个 leader 版本号是 1，从位移 120 处开始写入消息。

每个 leader broker 中会保存这样一个缓存，并定期写入一个检查点文件中。当 leader 写底层 log 时，它会尝试更新整个缓存——如果这个 leader 首次写消息，则会在缓存中增加一个条目，否则就不做更新。而每次副本重新成为 leader 时会查询这部分缓存，获取对应 leader 版本的位移，这就不会发生数据不一致和丢失的情况。

下面依然使用图的方式来说明利用 leader epoch 如何规避上述两种情况。图 6.27 展示了如何应用 leader epoch 来规避数据丢失的问题。

图 6.27 左半边的文字部分已经给出了简要的流程描述，这里不详细展开具体的 leader epoch 实现细节（比如 `OffsetsForLeaderEpochRequest` 的实现），我们只需要知道每个副本都引入了新的状态来保存自己当 leader 时开始写入的第一条消息的 offset 以及 leader 版本。这样在

恢复的时候完全使用这些信息而非水位来判断是否需要截断日志。

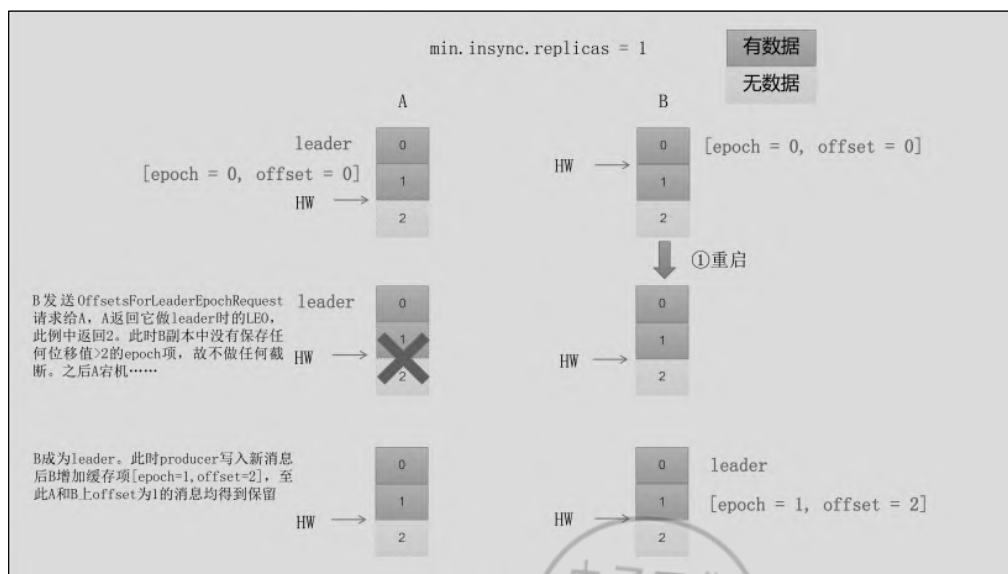


图 6.27 规避数据丢失

图 6.28 展示了如何避免数据不一致问题。

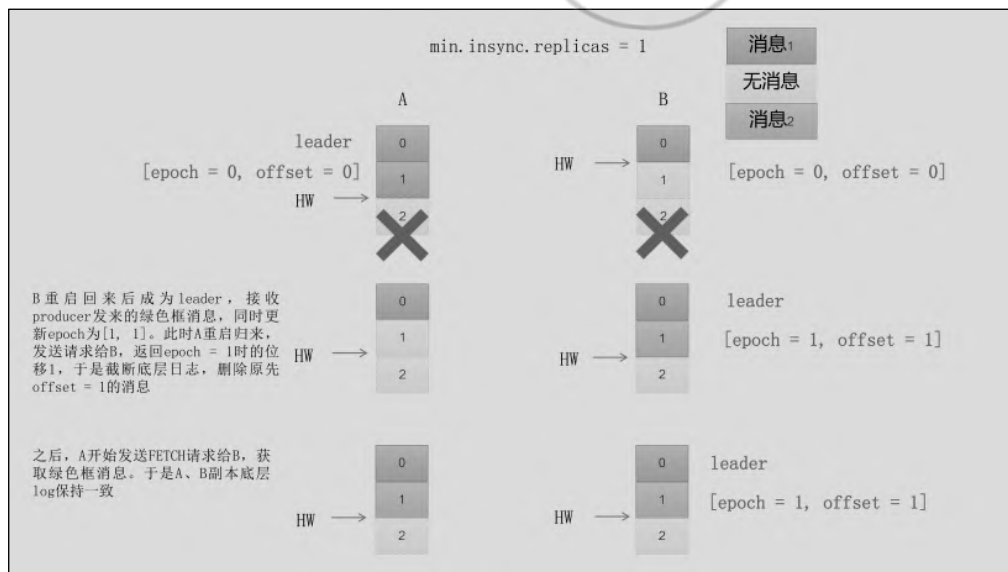


图 6.28 避免数据不一致

同样的道理，依靠 leader epoch 的信息可以有效地规避数据不一致的问题。

总结一下，0.11.0.0 版本的 Kafka 通过引入 leader epoch 解决了原先依赖水印来标识副本进度可能造成的数据丢失/数据不一致问题。有兴趣的读者可以阅读源代码进一步了解其中的工作原理。源代码的主要类是 `kafka.server.epoch.LeaderEpochCache.scala`（leader epoch 数据结构）、`kafka.server.checkpoints.LeaderEpochCheckpointFile`（checkpoint 检查点文件操作类），以及分布在 `Log.scala` 中的各种增删改查操作。

6.1.5 日志存储设计

1. Kafka 日志

什么是日志（log）？大多数人谈及日志时脑中浮现的可能是类似于图 6.29 这样的内容。我们都很熟悉这种日志格式——一系列松散结构化的请求日志、错误日志或其他数据。这种日志的主要用途就是方便人们阅读，而下面将要讨论的 Kafka 日志则属于另外一种类型：一类专门为程序访问的日志。

```
huxi:logs huxi$ tail -n 20 server.log
[2017-11-21 09:22:48,476] INFO [LogDirFailureHandler]: Starting (kafka.server.ReplicaManagersLogDirFailureHandler)
[2017-11-21 09:22:48,533] INFO [ExpirationReaper-0-topic]: Starting (kafka.server.DelayedOperationPurgatory$ExpiredOperationReaper)
[2017-11-21 09:22:48,538] INFO [ExpirationReaper-0-Heartbeat]: Starting (kafka.server.DelayedOperationPurgatory$ExpiredOperationReaper)
[2017-11-21 09:22:48,539] INFO Creating /controller (is it secure? false) (kafka.utils.ZKCheckedEphemeral)
[2017-11-21 09:22:48,539] INFO [ExpirationReaper-0-Rebalance]: Starting (kafka.server.DelayedOperationPurgatory$ExpiredOperationReaper)
[2017-11-21 09:22:48,546] INFO Result of znode creation is: OK (kafka.utils.ZKCheckedEphemeral)
[2017-11-21 09:22:48,564] INFO [GroupCoordinator 0]: Starting up. (kafka.coordinator.group.GroupCoordinator)
[2017-11-21 09:22:48,566] INFO [GroupCoordinator 0]: Startup complete. (kafka.coordinator.group.GroupCoordinator)
[2017-11-21 09:22:48,571] INFO [GroupMetadataManager brokerId=0] Removed 0 expired offsets in 6 milliseconds. (kafka.coordinator.group.GroupMetadataManager)
[2017-11-21 09:22:48,600] INFO [ProducerId Manager 0]: Acquired new producerId block (brokerId:0,blockStartProducerId:0,blockEndProducerId:999) by writing to Zk with path version 1 (kafka.coordinator.transaction.ProducerIdManager)
[2017-11-21 09:22:48,643] INFO [TransactionCoordinator id=0] Starting up. (kafka.coordinator.transaction.TransactionCoordinator)
[2017-11-21 09:22:48,646] INFO [Transaction Marker Channel Manager 0]: Starting (kafka.coordinator.transaction.TransactionMarkerChannelManager)
[2017-11-21 09:22:48,646] INFO [TransactionCoordinator id=0] Startup complete. (kafka.coordinator.transaction.TransactionCoordinator)
[2017-11-21 09:22:48,721] INFO Creating /brokers/ids/0 (is it secure? false) (kafka.utils.ZKCheckedEphemeral)
[2017-11-21 09:22:48,724] INFO Result of znode creation is: OK (kafka.utils.ZKCheckedEphemeral)
[2017-11-21 09:22:48,730] INFO Registered broker 0 at path /brokers/ids/0 with addresses: EndPoint(10.2.0.28,9092,ListenerName(PLAINTEXT),PLAINTEXT) (kafka.utils.ZKUtils)
[2017-11-21 09:22:48,735] WARN No meta.properties file under dir /Users/huxi/SourceCode/newenv/datalogs/kafka_1/meta.properties (kafka.server.BrokerMetadataCheckpoint)
[2017-11-21 09:22:48,781] INFO Kafka version : 1.0.0 (org.apache.kafka.common.utils.AppInfoParser)
[2017-11-21 09:22:48,786] INFO Kafka commitId : aaa7af6d4a11b29d (org.apache.kafka.common.utils.AppInfoParser)
[2017-11-21 09:22:48,788] INFO [KafkaServer id=0] started (kafka.server.KafkaServer)
```

图 6.29 access_log 日志片段

从某种意义上说，Kafka 日志的设计更像是关系型数据库中的记录，抑或是某些系统中所谓的提交日志（commit log）或日志（journal）。这些日志有一个共同的特点就是：只能按照时间顺序在日志尾部追加写入记录（record），如图 6.30 所示。注意，这里笔者使用的是 record 而不是 message，这说明 Kafka 其实并不是直接将原生消息写入日志文件的，相反，它会将消息和一些必要的元数据信息打包在一起封装成一个 record 写入日志。另外，这里的 record 就是 6.1.1 节中的消息集合或消息 batch。

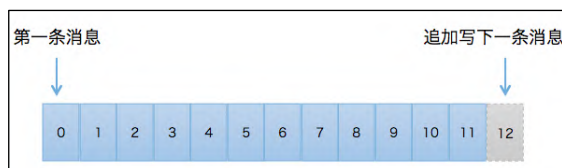


图 6.30 结构化 record

每个矩形都表示日志中的一条记录，日志记录按照被写入的顺序保存，读取日志以从左到右的方式进行。每条记录都会被分配一个唯一的且顺序增加的记录号作为定位该消息的唯一标识——这就是前面章节中提到的位移信息。记录中消息内容和格式的实现可能有多种方式，比如使用 XML 格式或 JSON 格式。如前所述，Kafka 则是自己定义了消息格式并且在写入日志前序列化紧凑的二进制字节数组来保存日志。

日志中记录的排序通常按照时间顺序，即位于日志左边部分的记录的发生时间通常要小于位于右边部分的记录。Kafka 自 0.10.0.0 版本开始在消息体中增加了时间戳信息。默认情况下，消息创建时间会被封装进消息中，因此，Kafka 记录大部分遵循按时间排序这一规则。当然，凡事皆有例外，Kafka 的 Java 版本 producer 确实支持用户为消息指定时间戳，用户完全可以打乱这种时间排序。只不过这样的话，时间戳索引文件可能会失效，因此，在实际中似乎并没有太多的使用场景。

Kafka 的日志设计都是以分区为单位的，即每个分区都有它自己的日志，该日志被称为分区日志（partition log）。producer 生产 Kafka 消息时需要确定该消息被发送到的分区，然后 Kafka broker 把该消息写入该分区对应的日志中，如图 6.31 所示。

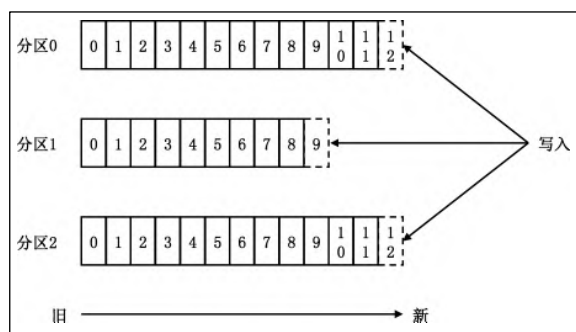


图 6.31 分区和分区日志

具体对每个日志而言，Kafka 又将其进一步细分成日志段文件（log segment file）以及日志段索引文件。可以这样说，每个分区日志都是由若干组日志段文件+索引文件构成的。图 6.32 给出了一个典型的 Kafka 分区日志构成。

```

bogon:test-0 huxi$ ll -h /Users/huxi/SourceCode/testenv/datalogs/kafka_1/test-0
total 5038024
drwxr-xr-x  11 huxi  staff   374B Jul 12 09:52 .
drwxr-xr-x   8 huxi  staff   272B Jul 12 09:53 ..
-rw-r--r--   1 huxi  staff   514K Jul 12 09:52 00000000000000000000.index
-rw-r--r--   1 huxi  staff   1.0G Jul 12 09:52 00000000000000000000.log
-rw-r--r--   1 huxi  staff   240K Jul 12 09:52 00000000000000000000.timeindex
-rw-r--r--   1 huxi  staff   512K Jul 12 09:52 0000000000004161281.index
-rw-r--r--   1 huxi  staff   1.0G Jul 12 09:52 0000000000004161281.log
-rw-r--r--   1 huxi  staff   177K Jul 12 09:52 0000000000004161281.timeindex
-rw-r--r--   1 huxi  staff   10M Jul 12 09:52 0000000000008749921.index
-rw-r--r--   1 huxi  staff   391M Jul 12 09:53 0000000000008749921.log
-rw-r--r--   1 huxi  staff   10M Jul 12 09:52 0000000000008749921.timeindex

```

图 6.32 分区日志实例

如图 6.32 所示，当前该分区日志下有 3 组日志段，图中的.log 文件就是日志段文件，而.index 和.timeindex 文件都是与日志段对应的索引文件。不难发现，每组日志段 3 个文件的文件名都是相同的。下面讨论一下这些文件的由来以及底层的文件格式。

2. 底层文件系统

创建 topic 时，Kafka 为该 topic 的每个分区在文件系统中创建了一个对应的子目录，名字就是<topic>-<分区号>。所以，倘若有一个 topic 名为 test，有两个分区，那么在文件系统中 Kafka 会创建两个子目录：test-0 和 test-1。每个日志子目录的文件构成都是如图 6.32 所示的结构，即若干组日志段+索引文件。

日志段文件，即后缀名是.log 的文件保存着真实的 Kafka 记录（6.1.1 节中详细讨论了 Kafka 记录或 Kafka 消息 batch 在不同版本间的格式对比）。每个.log 文件都包含了一段位移范围的 Kafka 记录。Kafka 使用该文件第一条记录对应的 offset 来命名此.log 文件。因此，每个新创建的 topic 分区一定有 offset 是 0 的.log 文件，即 00000000000000000000.log。虽然在 Kafka 内部 offset 是用 64 位来保存的，但目前对于日志段文件而言，Kafka 只使用 20 位数字来标识 offset。不过对于实际的线上环境而言，这通常是足够的。

我们仍然以图 6.32 中的分区日志为例。图中的第二组日志段对应的位移是 0000000000000004161281，这就说明该日志段的起始位移（第一条消息的位移）是 4161281。同时也说明 00000000000000000000.log 中包含的消息总数是 4161280 条。

细心的读者可能会发现图 6.32 中前两组日志段对应的.log 文件都是 1.0GB 大小，而第三组的.log 文件大小是 391MB。事实上，Kafka 每个日志段文件是有上限大小的，由 broker 端参数 log.segment.bytes 控制，默认就是 1GB 大小。因此，当日志段文件填满记录后，Kafka 会自动创建一组新的日志段文件和索引文件——这个过程被称为日志切分（log rolling）。日志切分后，新的日志文件被创建并开始承担保存记录的角色。图 6.32 中第三组日志段文件便是切分之后的结果，由于它没有被记录填满，因此它的大小不足 1GB。

一旦日志段被填满，它就不能再追加写入新消息了，而 Kafka 正在写入的分区日志段文

件被称为当前激活日志段（active log segment）或简称为当前日志段。图 6.32 中的 00000000000008749921.log 就是 active log segment。

当前日志段非常特殊，它不受任何 Kafka 后台任务的影响，比如定期日志清除任务和定期日志 compaction 任务。我们会在后面的章节中详细讨论这些维护任务。

3. 索引文件

除了.log 文件，Kafka 分区日志还包含两个特殊的文件.index 和.timeindex，它们都是索引文件，分别被称为位移索引文件和时间戳索引文件。前者可以帮助 broker 更快地定位记录所在的物理文件位置，而后者则是根据给定的时间戳查找对应的位移信息。在分别介绍这两种索引文件之前，下面先介绍它们的实现机制与原理。

它们都属于稀疏索引文件（sparse index file），每个索引文件都由若干条索引项（index entry）组成。Kafka 不会为每条消息记录都保存对应的索引项，而是待写入若干条记录后才增加一个索引项。broker 端参数 log.index.interval.bytes 设置了这个间隔到底是多大，默认值是 4KB，即 Kafka 分区至少写入了 4KB 数据后才会索引文件中增加一个索引项，故本质上它们是稀疏的。

不论是位移索引文件还是时间戳索引文件，它们中的索引项都按照某种规律进行升序排列。对于位移索引文件而言，它是按照位移顺序保存的；而时间戳索引文件则严格按照时间戳顺序保存。由于有了这种升序规律，Kafka 可以利用二分查找（binary search）算法来搜寻目标索引项，从而降低整体时间复杂度到 $O(\lg N)$ 。若没有索引文件，Kafka 搜寻记录的方式只能是从每个日志段文件的头部顺序扫描，因此，这种方案的时间复杂度是 $O(N)$ 。显然，引入索引文件可以极大地减少查找时间，减少 broker 端的 CPU 开销。

当前，索引文件支持两种打开方式：只读模式和读/写模式。对于非当前日志段而言，其对应的索引文件通常以只读方式打开，即只能读取索引文件中的内容而不能修改它。反之，当前日志段的索引文件必须要能被修改，因此总是以读/写模式打开的。当日志进行切分时，索引文件也需要进行切分。此时，Kafka 会关闭当前正在写入的索引文件，同时以读/写模式创建一个新的索引文件。broker 端参数 log.index.size.max.bytes 设置了索引文件的最大文件大小，默认值是 10MB。细心的读者可能会问：图 6.32 中的第 3 组索引文件中为什么当前日志段对应的索引文件是 10MB，已被切分过的那两组反而不是 10MB 呢？和日志段文件不同，索引文件的空间默认都是预先分配好的，而当对索引文件切分时，Kafka 会把该文件大小“裁剪”到真实的数据大小——这就是为什么前两组索引文件都是正常大小，反而当前日志段对应的索引文件是 10MB 的原因。

下面分别讨论一下位移索引文件和时间戳索引文件的格式。

(1) 位移索引文件

位移索引文件中的索引项格式如图 6.33 所示。

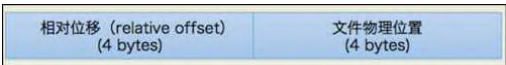


图 6.33 位移索引项格式

每个索引项固定地占用 8 字节的物理空间，同时 Kafka 强制要求索引文件必须是索引项大小的整数倍，即 8 的整数倍。因此，假设用户设置参数 `log.index.size.max.bytes` 为 300，那么 Kafka 在内部会“勒令”该文件大小为 296——即不大于 300 的最大的 8 的倍数。

注意图 6.33 中的相对位移——它保存的是与索引文件起始位移的差值。索引文件文件名中的位移就是该索引文件的起始位移。通过保存差值，我们只需要 4 字节而非保存整个位移的 8 字节。举一个简单的例子，假设位移索引文件名是 `00000000000000000050.index`，那么起始位移就是 50，于是位移=55 的消息在索引项中的相对位移就是 $55 - 50 = 5$ 。因此只保存相对位移可以节省很多磁盘空间。不过，这样做的结果就是后续在获取索引项时，Kafka 还需要把相对位移还原成绝对位移。当然，这对用户来说是透明的。

位移索引文件会强制保证索引项中的位移都是升序排列的，因此这种顺序性提升了查找的性能。有了位移索引文件，broker 可根据指定位移快速定位到记录的物理文件位置，或至少定位出离目标记录最近的低位文件位置。即使从该位置处扫描日志段文件，也要比从头扫描代价小得多。图 6.34 给出了 Kafka 使用位移索引文件的流程。值得注意的是，每个索引项是不保存图 6.34 中的逗号的，这里给出逗号只是方便阅读使用。

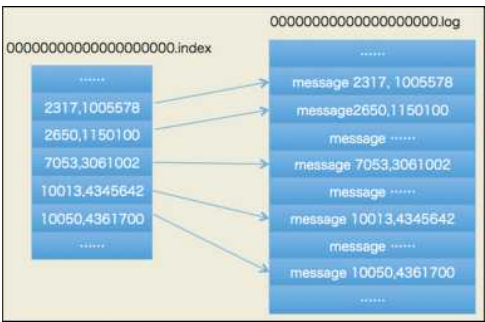


图 6.34 位移索引查找过程

假设 broker 端要查找位移为 7000 的消息，那么首先在位移索引文件中根据二分查找算法找到了小于 7000 的最大索引项：2650,1150100，然后 Kafka 会在对应的.log 文件中从第 1150100 字节处开始顺序搜寻记录，直至找到位移为 7000 的消息记录。如果用户想要增加索引项的密度，可以减少 broker 端参数 `log.index.interval.bytes` 的值。

(2) 时间戳索引文件

时间戳索引文件中索引项的格式如图 6.35 所示。

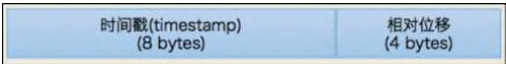


图 6.35 时间戳索引项格式

自 Kafka 0.10.0.0 版本在消息中加入时间戳信息后，很多用户都有这样的需求：想要查找某段时间内的消息记录。鉴于这个原因，Kafka 引入了时间戳索引文件，每个索引项固定占用

12 字节的物理空间，同时 Kafka 强制要求索引文件必须是索引项大小的整数倍，即 12 的整数倍。因此，假设用户设置参数 `log.index.size.max.bytes` 为 100，那么 Kafka 在内部会“勒令”该文件大小为 96——即不大于 100 的最大的 12 的倍数。时间戳索引项保存的也是相对位移值。

时间戳索引项保存的是时间戳与位移的映射关系。给定时间戳之后根据此索引文件只能找到不大于该时间戳的最大位移，稍后 Kafka 还需要拿着返回的位移再去位移索引文件中定位真实的物理文件位置。该索引文件中的时间戳一定是按照升序排列的。若消息 R2 在日志段中位于 R1 之前，但 R2 的时间戳小于 R1（这是可能的，因为 Java 版本 `producer` 允许用户手动指定时间戳），那么 R2 这条消息是会被记录在时间戳索引项中的，因为这会造成时间的乱序。目前 Kafka 还无力调整这种时间“错乱”的情况，而缺乏对应的索引项也会使得 `clients` 根据时间戳查找消息的结果不能完全准确，因而在实际场景中并不推荐 `producer` 端直接手动指定时间戳的用法。

图 6.36 给出了一个时间戳索引文件的部分索引项，下面我们结合一个实例来看看 Kafka 是如何应用时间戳索引文件的。假设我们想要寻找时间戳是 1499824275743 附近的消息，那么查找时间戳索引项定位到 1499824275740,7447 这个索引项，然后在位移索引文件中查找位移 =7447 对应的物理文件位置，至少是定位离 7447 最近的物理文件位置，从而找到 7053,3061002 索引项。最后就可以保证日志段文件第 3061002 字节处开始的消息就是满足该时间戳条件的消息。

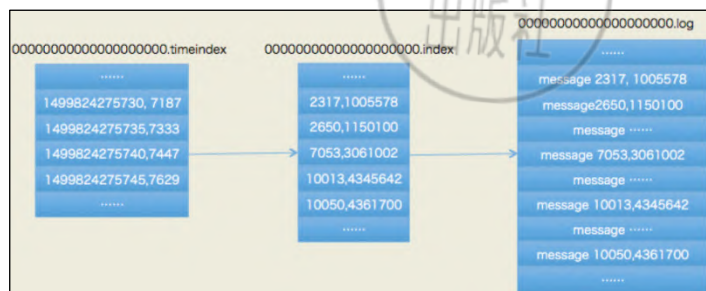


图 6.36 时间戳索引查找过程

4. 日志留存

Kafka 是会定期清除日志的，而且清除的单位是日志段文件，即删除符合清除策略的日志段文件和对应的两个索引文件。当前留存策略有如下两种。

- 基于时间的留存策略：Kafka 默认会清除 7 天前的日志段数据（包括索引文件）。Kafka 提供了 3 个 `broker` 端参数，其中 `log.retention.{hours|minutes|ms}` 用于配置清除日志的时间间隔，其中的 `ms` 优先级最高，`minutes` 次之，`hours` 优先级最低。
- 基于大小的留存策略：Kafka 默认只会为每个 `log` 保存 `log.retention.bytes` 参数值大小的字节数。默认值是 -1，表示 Kafka 不会对 `log` 进行大小方面的限制。

日志清除是一个异步过程，Kafka broker 启动后会创建单独的线程处理日志清除事宜。另外，一定要注意的是，日志清除对于当前日志段是不生效的。也就是说，Kafka 永远不会清除当前日志段。因此，若用户把日志段文件最大文件的大小设置得过大而导致没有出现日志切分，那么日志清除也就永远无法执行。用户观测的现象就是 Kafka 似乎从不对过期数据文件做清理。

在基于时间的清除策略中，0.10.0.0 版本之前 Kafka 使用日志段文件的最近修改时间（当前时间与最近修改时间的差值）来衡量日志段文件是否依然在留存时间窗口中，但文件的最近修改时间属性经常有可能被“无意”修改（比如执行了 touch 操作）。因此，在 0.10.0.0 版本引入时间戳字段后，该策略会计算当前时间戳与日志段首条消息的时间戳之差作为衡量日志段是否留存的依据。如果第一条消息没有时间戳信息，Kafka 才会使用最近修改时间的属性。

5. 日志 compaction

前面讨论的所有 topic 都有这样一个特点：clients 端通常需要访问和处理这种 topic 下的所有消息，但考虑这样一种应用场景，某个 Kafka topic 保存的是用户的邮箱地址，每次用户更新邮箱地址时都会发送一条 Kafka 消息。该消息的 key 就是用户 ID，而 value 保存了邮件地址信息。假设用户 ID 为 user123 的用户连续修改了 3 次邮件地址，那么就会产生 3 条对应的 Kafka 消息，如下：

(1) user123 => user123@kafka1.com

(2) user123 => user123@kafka2.com

(3) user123 => user123@kafka3.com

显然，在这种情况下用户只关心最近修改的邮件地址，即 value 是 user123@kafka3.com 的那条消息，而之前的其他消息都是“过期”的，可以放心删除。但前面提到的清除策略都无法实现这样的处理逻辑，因此，Kafka 社区引入了 log compaction。

比起日志压缩，笔者更倾向于把 log compaction 翻译成日志压实。log compaction 确保 Kafka topic 每个分区下的每条具有相同 key 的消息都至少保存最新 value 的消息。它提供了更细粒度化的留存策略。这也说明了如果要使用 log compaction，Kafka 消息必须要设置 key。无 key 消息是无法为其进行压实操作的。典型的 log compaction 使用场景如下。

- 数据库变更订阅：用户通常在多个数据系统存有数据，比如数据库、缓存、查询集群和 Hadoop 集群等。对数据库的所有变更都需要同步到其他数据系统中。在同步的过程中用户没必要同步所有数据，只需要同步最近的变更或增量变更。
- 事件溯源（event sourcing）：编织查询处理逻辑到应用设计中并使用变更日志保存应用状态。

- 高可用日志化（journaling）：将本地计算进程的变更实时记录到本地状态中，以便在出现崩溃时其他进程可以加载该状态，从而实现整体上的高可用。

图 6.37 给出了与 log compaction 相关的 Kafka log 结构。

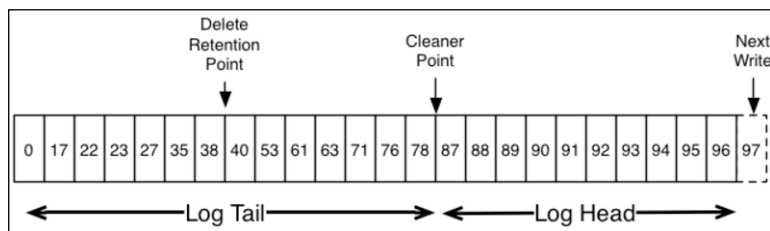


图 6.37 log compaction 相关的日志结构

为了实现 log compaction，Kafka 在逻辑上将每个 log 划分成 log tail 和 log head。log head 和普通的 Kafka log 没有区别。事实上，它就是 Kafka log 的一部分，在 log head 中所有 offset 都是连续递增的。但细心的读者会发现，log tail 中消息的位移则是不连续的，它已经是压实之后（compacted）的消息集合了。比如图 6.37 中 log tail 部分位移为 17 和 22 的消息彼此相邻，这表明经过了 compaction 之后，位移是 18、19、20、21 的这 4 条消息都被移除了。这告诉我们一个事实：log compaction 只会根据某种策略有选择性地移除 log 中的消息，而不会变更消息的 offset 值。

Kafka 有一个组件叫 Cleaner，它就是负责执行 compaction 操作的。Cleaner 负责从 log 中移除已废弃的消息。再次强调一下，如果一条消息的 key 是 K，位移是 O，只要日志中存在另外一条消息，key 也是 K，但位移是 O' 且 $O < O'$ ，即认为前面那条消息已被废弃。

log compaction 是 topic 级别的设置。一旦为某个 topic 启用了 log compaction，Kafka 会将该 topic 的日志在逻辑上划分成两部分：“已清理”部分和“未清理”部分，后者又可进一步划分成“可清理”部分和“不可清理”部分。

顾名思义，“不可清理”部分无法被 Kafka Cleaner 清理，而当前日志段永远属于“不可清理”部分。当前 Kafka 使用一些后台线程定期执行真正的清理任务。每个线程会挑选出“最脏”的日志段执行清理。衡量一个日志段“脏”的程度使用“脏”日志部分与总日志大小的比率。

在内部，Kafka 会构造一个哈希表来保存 key 与最新位移的映射关系。当执行 compaction 时，Cleaner 不断拷贝日志段中的数据，只不过它会无视那些 key 存在于哈希表中但具有较大位移值的消息，具体流程如图 6.38 所示。

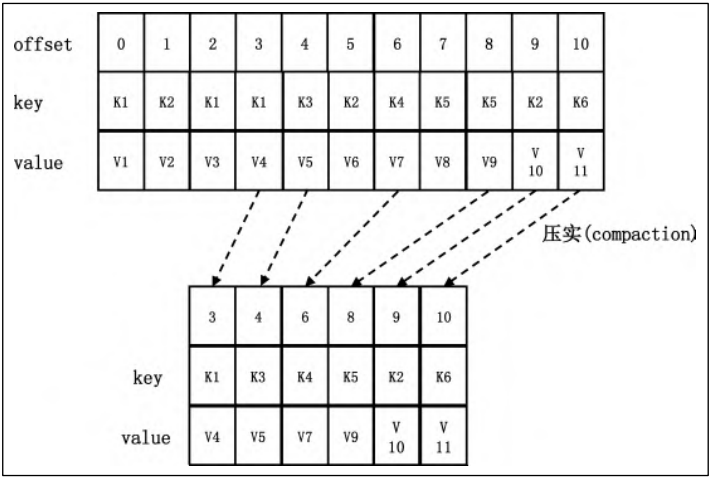


图 6.38 log compaction 执行过程

图 6.38 中，key 为 K1 的消息总共有 3 条，经过 compaction 之后 log 只保存了 Value=V4 的那条消息，因为该消息的位移最高，其他消息都被视为“可清理”的。

当前与 compaction 相关的 Kafka 参数如下。

- `log.cleanup.policy`：是否启用 log compaction。0.10.1.0 版本之前只有两种取值，即 `delete` 和 `compact`。其中 `delete` 是默认值，表示采用之前所说的留存策略；设置 `compact` 则表示启用 log compaction。自 0.10.1.0 版本开始，该参数支持同时指定两种策略，如 `log.cleanup.policy=delete,compact`，表示既为该 topic 执行普通的留存策略，也对其进行 log compaction。
- `log.cleaner.enable`：是否启用 log Cleaner。在 0.9.0.0 及之前的版本中该参数默认值是 `false`，即不启用 compaction。对于这些版本的用户来说，如果要启用 Cleaner，则必须显式地设置该参数=`true`。自 0.9.0.1 版本之后该参数便默认为 `true`。另外需要注意的是，如果要使用 log compaction，则必须将此参数设置为 `true`，否则即使用户设置 `log.cleanup.policy=compact`，Kafka 也不会执行清理任务。
- `log.cleaner.min.compaction.lag.ms`：默认值是 0，表示除了当前日志段，理论上所有的日志段都属于“可清理”部分，但有时候用户可能不想这么激进，用户可以设置此参数值来保护那些比某个时间新的日志段不被清理。假设设置此参数为 10 分钟，当前时间是下午 1 点钟，那么所有最大时间戳（通常都是最后一条消息的时间戳）在 12:50 之后的日志段都不可清理。

前面章节中屡次提到过 Kafka 新版本 consumer 使用 `__consumer_offsets` 内部 topic 来保存位

移信息。这个 topic 就是采用 log compaction 留存策略的，因为对于每一个 key（通常是 groupId + topic + 分区号）而言，我们只关心最新的位移值——这是非常典型的 log compaction 使用场景。Kafka 自动为该 topic 配置好了留存策略，因此这些对用户而言都是透明的。

6.1.6 通信协议（wire protocol）

1. 协议设计

所谓通信协议，就是实现 client-server 间或 server-server 间数据传输的一套规范。Kafka 的通信协议是基于 TCP 之上的二进制协议，这套协议提供的 API 表现为服务于不同功能的多种请求（request）类型以及对应的响应（response）。所有类型的请求和响应都是结构化的，由不同的初始类型构成。Kafka 使用这组协议完成各个功能的实现。

Kafka 客户端与 broker 传输数据时，首先需要创建一个连向特定 broker 的 Socket 连接，然后按照待发送请求类型要求的结构构造响应的请求二进制字节数组，之后发送给 broker 并等待从 broker 处接收响应。假如是像发送消息和消费消息这样的请求，clients 通常会一直维持与某些 broker 的长连接，从而创建 TCP 连接的开销被不断地摊薄给每条具体的请求。

实际使用过程中，单个 Kafka clients 通常需要同时连接多个 broker 服务器进行数据交互，但在每个 broker 之上只需要维护一个 Socket 连接用于数据传输。clients 可能会创建额外的 Socket 连接用于其他任务，如元数据获取以及组 rebalance 等。Kafka 自带的 Java clients（包括 Java 版本 producer 和 Java 版本 consumer）使用了类似于 epoll 的方式在单个连接上不停地轮询以传输数据。

broker 端需要确保在单个 Socket 连接上按照发送顺序对请求进行一一处理，然后依次返回对应的响应结果。单个 TCP 连接上某一时刻只能处理一条请求的做法正是为了保证不会出现请求乱序。当然这是 broker 端的做法，clients 端在实现时需要自行保证请求发送顺序。比如 Java 版本 producer 默认情况下会对 PRODUCE 请求（专门用于发送消息的请求）进行流水化处理，在内存中它允许多条未处理完成的请求同时排队等候被发送。这样做的好处是提升了 producer 端的吞吐量，但潜在的风险是 PRODUCE 请求发送乱序所导致的消息生产乱序。在实际应用中，用户可以通过设置参数 `max.in.flight.requests.per.connection=1` 来关闭这种流水化作业。

Kafka 通信协议中规定的请求发送流向有 3 种。除了上面所说的 clients 给 broker 发送请求之外，Kafka 集群中的 controller 也能够给其他 broker 发送请求，图 6.39 展示了一个可能的请求发送流向图。如图 6.39 所示，

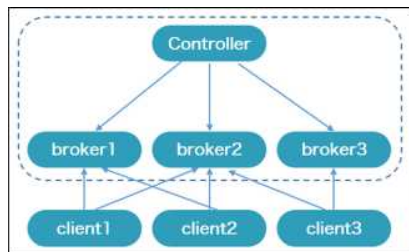


图 6.39 Kafka 请求发送流向图

clients 可能给多个 broker 发送请求，而 controller 会向所有 broker 发送请求。当然 clients 也可以给 controller 直接发送请求，只是在这种情况下 clients 只当其是一个普通的 broker 而已。第三种方式就是 follower 副本所在 broker 向 leader 副本所在 broker 发送请求，不过这只能是固定的 FETCH 请求。后面的“常见请求类型”部分将详细讨论 FETCH 请求。

Kafka 的 broker 端提供了一个可配置的参数用于限制 broker 端能够处理请求的最大字节数。一旦超过了该阈值，发生此请求的 Socket 连接就会被强制关闭。clients 观察到连接关闭后只能执行连接重建和请求重试等的逻辑。

2. 请求/响应结构

Kafka 协议提供的所有请求及其响应的结构体都是由固定格式组成的，它们统一构建于多种初始类型（primitive types）之上。这些初始类型如下。

- 固定长度初始类型：包括 int8、int16、int32 和 int64，分别表示有符号的单字节整数、双字节整数、4 字节整数和 8 字节整数。
- 可变长度初始类型：包括 bytes 和 string，由一个有符号整数 N 加上后续的 N 字节组成。N 表示它们的内容，若是-1，则表示内容为空。其中 string 类型使用 int16 来保存 N；bytes 使用 int32 来保存 N。
- 数组：用于处理结构体之类重复性数据结构。它们总是被编码成一个 int32 类型的整数 N 以及后续的 N 字节。同样，N 表示该数组的长度信息，而具体到里面的元素可以是其他的初始类型。在下面的请求结构体部分，我们使用[Array]的方式来表示数组。

所有的请求和响应都具有统一的格式，即 Size + Request/Response，其中的 Size 是 int32 表示的整数，表征了该请求或响应的长度信息。

请求又可划分成请求头部和请求体，请求体的格式因请求类型的不同而变化，但请求头部的结构是固定的——它由以下 4 个字段构成。

- api_key: 请求类型，以 int16 整数表示。
- api_version: 请求版本号，以 int16 整数表示。
- correlation_id: 与对应响应的关联号，实际中用于关联 response 与 request，方便用户调试和排错。该字段以 int32 整数表示。
- client_id: 表示发出此请求的 client ID。实际场景中用于区分集群上不同 clients 发送的请求。该字段是一个非空字符串。

同理，响应也可划分成响应头部和响应体，响应体的格式因其对应的请求类型的不同而变化，但响应头部的结构是固定的——它只有下面的这个字段。

- `correlation_id`: 该字段值就是上面请求头部中的 `correlation_id`。有了该字段, 用户就能够知道该响应对应于哪个请求了。

Kafka 推荐用户总是指定 `client_id` 和 `correlation_id`, 这样可以方便用户后续定位问题和 DEBUG。

鉴于每种请求类型的结构都不相同, 下面选取并结合几个重要且常用的请求类型加以说明。

3. 常见请求类型

Kafka 1.0.0 版本总共提供了多达 38 个请求类型, 完整的请求列表请参见 http://kafka.apache.org/protocol.html#protocol_api_keys。本书选取几个比较重要且常见的请求来说明它们的结构。

(1) PRODUCE 请求

这是编号为 0 的请求, 即 `api_key = 0`, 也就是 Kafka 通信协议中的第一个请求类型。顾名思义, 它实现消息的生产。`clients` 向 `broker` 发送 PRODUCE 请求并期待 `broker` 端返回响应表明消息生产是否成功。

当前, PRODUCE 请求共有 6 个版本, 这里只讨论最新版本的 PRODUCE 请求 (即 `api_version = 5` 的版本, 版本号从 0 开始)。读者可以参阅 http://kafka.apache.org/protocol.html#The_Messages_Produce 学习其他 3 个版本的格式。

V6 版本的 PRODUCE 请求 (版本号为 5) 格式为: 事务 Id + `acks` + `timeout` + [topic 数据]。

- 事务 Id: `transactional Id` 是 0.11.0.0 版本为支持事务引入的字段, 是 `string` 类型。如果不使用事务, 则该字段是 `null`。使用 0.11.0.0 版本的用户如果不使用事务型 `producer`, 则设置该字段为 `null` 即可。
- `acks`: 这就是第 4 章中介绍的 `acks`, 是 `int16` 类型, 表明 PRODUCE 请求从 `broker` 处返回之前必须被应答的 `broker` 数。当前只有 3 个取值: 0、1 和 -1 (同 `all`)。
- `timeout`: 以毫秒计算的请求超时时间, 默认值是 30 秒, 表示若 `broker` 端 30 秒内未发送响应给 `clients`, 则 `clients` 端视该请求超时。
- topic 数据: 这是 PRODUCE 请求主要的数据载体, 里面封装了要发送到 `broker` 端的消息数据。该字段可进一步划分成 `topic` + [partition + 消息集合]。
 - `topic`: 表征该 PRODUCE 请求要发送到哪些 `topic`。注意, `topic` 数据是数组类型, 故单个 PRODUCE 请求可同时生产多个 `topic` 的消息。
 - `partition`: 表征 PRODUCE 请求要发送消息到 `topic` 的哪些分区。
 - 消息集合: 即 6.1.1 节中讨论的消息集合, 封装了真正的消息数据。

以上就是 PRODUCE 请求的结构说明，下面来看看 PRODUCE 响应的结构：[responses] + throttle_time_ms。

- throttle_time_ms: 表示因超过了配额限制而延迟该请求处理的时间，以毫秒计。如果没有配置任何配额设置，则该字段恒为 0。
- [responses]: 每个 topic 都有对应的返回数据，即 response，每个 response 结构如下。
 - partition: 表示该 topic 分区号。
 - error_code: 该请求是否成功。
 - base_offset: 该消息集合的起始位移。
 - log_append_time: broker 端写入消息的时间。
 - log_start_offset: response 被创建时该分区日志总的起始位移。

(2) FETCH 请求

FETCH 请求是编号为 1 的请求，即 api_key = 1。它服务于消费消息，既包括 clients 向 broker 发送的 FETCH 请求，也包括分区 follower 副本发送给 leader 副本的 FETCH 请求。

当前，FETCH 请求共有 7 个版本，下面只讨论最新版本的 FETCH 请求，其格式为 replica_id + max_wait_time + min_bytes + max_bytes + isolation_level + [topics]。

- replica_id: int32 整数表征的副本 ID。该字段专门服务于 follower 给 leader 发送的 FETCH 请求。如果是正常的 clients 端 consumer 程序，则该字段是-1。
- max_wait_time: int32 类型的整数，表示等待响应返回的最大时间。
- min_bytes: int32 类型的整数，表示响应中包含数据的最小字节数。默认是 1 字节，表示只要 broker 端累积了超过 1 字节的数据，就可以返回响应。对于要提升 clients 端 TPS 的用户来说，可以适当增加此值，让 broker 积攒更多的数据之后再返回。
- max_bytes: int32 类型的整数，响应中包含数据的最大字节数。注意，这不是绝对不能突破的“硬界限”。若 FETCH 请求中要获取的第一个 topic 分区的单条消息大小已超过了这个阈值，那么 Kafka 已经允许 broker 返还这条消息给 clients。
- isolation_level: int8 类型的整数，0.11.0.0 版本新引入的字段，表示 FETCH 请求的隔离级别。当前只有两种隔离级别：READ_UNCOMMITTED(isolation_level=0)和 READ_COMMITTED(isolation_level=1)。对于非事务型的 consumer 而言，该字段固定是 0。
- [topics]: 数组类型，表征了要请求的 topic 数据，该数组中的每个元素结构如下。
 - topic。
 - 若干分区信息，每个分区信息都由 partition、fetch_offset、log_start_offset 和 max_bytes 构成。其中的 fetch_offset 表明 broker 需要从指定分区的哪个位移开始读取消息，而 log_start_offset 则是为 follower 发送的 FETCH 请求专用，表明

follower 副本最早可用的分区。特别要注意最后的 `max_bytes`，这和外层的 `max_bytes` 含义是不一样的，前者限定的是单个分区所能获取到的最大字节数，而后者限定的是整个 `FETCH` 请求能够获取到的最大字节数。

`FETCH` 请求对应的响应结构格式如下：`throttle_time_ms [responses]`。

- `throttle_time_ms`：和 `PRODUCE` 响应结构中的 `throttle_time_ms` 含义相同，同做节流之用。
- `[responses]`：返回的消息数据，是一个数组类型，每个元素结构如下。
 - `topic`：消息数据所属 topic。
 - `partition_header`：又可进一步细分为分区号、`error_code`、高水印值等字段。
 - 消息集合：真实的数据。

（3）METADATA 请求

clients 向 broker 发送 `METADATA` 请求以获取指定 topic 的元数据信息，其格式为 `[topics] + allow_auto_topic_creation`。

- `allow_auto_topic_creation`：布尔类型，指定了当 topic 不存在时是否允许自动创建该 topic。
- `[topics]`：数组类型，每个元素指定了 `METADATA` 请求想要获取元数据的 topic 名称。
与 `METADATA` 请求相比，`METADATA` 响应包含的数据非常丰富，其格式为 `throttle_time_ms + [brokers] + cluster_id + controller_id + [topic_metadata]`。
- `throttle_time_ms`：含义与前面相同，这里不再赘述。
- `[brokers]`：集群 broker 列表，每个 broker 信息包括 `node_id`、`host`、`port` 和 `rack`，即节点 ID、主机名或 IP 地址、端口和机架信息。
- `cluster_id`：该请求被发送的集群 ID。
- `controller_id`：该集群 controller 所在的 broker ID。
- `[topic_metadata]`：topic 元数据数组，每个元素表征了一个 topic 的所有元数据信息，如下。
 - `topic_error_code`：topic 错误码，表征该元数据是否有问题。
 - `topic`：topic 名。
 - `is_internal`：是否属于内部 topic。
 - `[partition_metadata]`：topic 下所有分区的元数据，包括每个分区的错误码、leader 信息、副本信息、ISR 信息等。

4. 请求处理流程

我们已经初步了解了 Kafka 的通信协议以及常见的请求类型，下面主要从两个方面讨论一下 Kafka 是如何处理这些请求的。

(1) clients 端

实际上, Kafka 并没有规定这些请求必须被如何处理, 它要求的只是 clients 端代码必须要构造符合格式的请求, 然后发送给 broker。每种语言的 clients 端代码必须自行实现对请求和响应的完整的生命周期管理。由于大部分用户依然在使用 Java 版本的 clients, 因此, 我们就以 Kafka 自带的 Java 版本 clients 端代码进行讨论。

图 6.40 给出了 Java 版本 clients 端的请求处理流程。由图可见, clients 发送请求前需要首先确定目标 broker, 即要搞清楚这个请求是发给哪个 broker 的。目前大部分请求都只能发给特定的 broker, 比如就 PRODUCE 和 FETCH 请求而言, clients 只能发送给特定分区的 leader broker。也有一类请求并不强制要求只发送到指定 broker。像之前说的 METADATA 请求就可以发送给集群中任意一个 broker, 因为每个 broker 都保存了相同的元数据缓存。

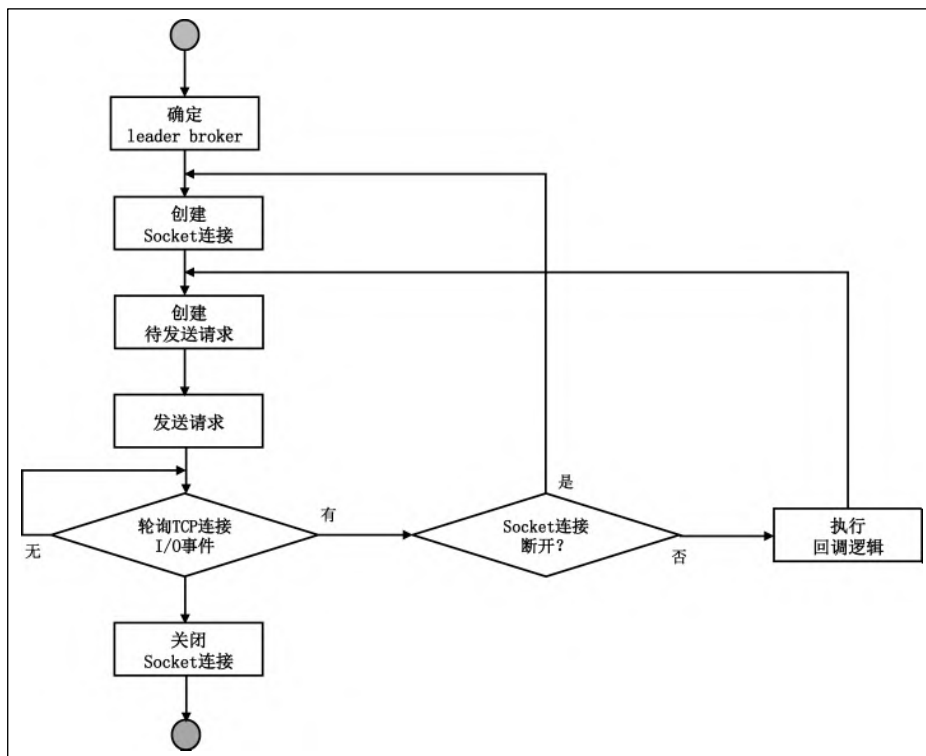


图 6.40 Java 版本 clients 端请求处理流程

确认了目标 broker 之后, Java 版本 clients 会创建与 broker 的连接并一直维持, 这样下次再向相同 broker 发送其他请求时就可以直接使用该连接而不需要重建。一旦 TCP 连接建立,

clients 端需要按照协议规定的格式构造不同的请求，然后发送出来。当前真正的 I/O 操作是在下面的轮询操作中执行的。前面说过，Java 版本 clients 采用了类似于 Linux select 和 epoll 的实现机制，在底层把 I/O 操作完全托管给 Java NIO 的 Selector，故在轮询这一步中 clients 会检查有无真正的 I/O 事件发生，比如发送请求或获取请求，甚至是连接重建或断开等。

若不是连接断开这样的 I/O 事件，clients 通常会对接收到的响应执行对应的回调逻辑，而如果是连接断开，则 clients 应该支持自动重建连接。这就是 Java 版本 clients 实现的完整请求处理逻辑。

(2) broker 端

broker 端的处理逻辑要简单很多，如图 6.41 所示。

每个 broker 启动时都会创建一个请求阻塞队列，专门用于接收从 clients 端发送过来的请求。同时，broker 还会创建若干个请求处理线程专门获取并处理该阻塞队列中的请求，如图 6.41 所示。每个线程实时地从该队列中拿出请求并判断是否关闭请求。在源代码中标识关闭该类线程的一类特殊请求被称为 AllDone 请求。严格来说，AllDone 不属于通信协议的一部分，它只作为 broker 端关闭的一个特殊标记被“塞入”队列中，这样请求处理线程一旦从队列中拿到 AllDone，则意识到 broker 要发起关闭操作，从而退出请求处理循环逻辑。

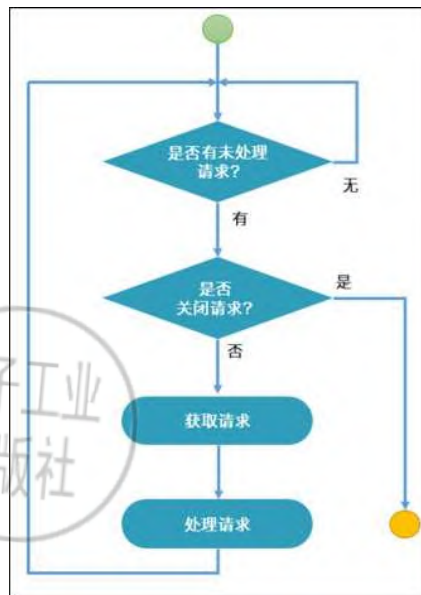


图 6.41 Java 版本 broker 端请求处理流程

5. 版本与兼容性

这套 Kafka 通信协议是区分版本的，这里的版本不是指 Kafka 的版本，而是指每种请求类型都有版本信息。比如之前提到的 PRODUCE 请求就有 6 个版本，而 FETCH 请求更是有 7 个版本。那么这些版本对用户有什么影响呢？

在 Kafka 0.10.2.0 之前，clients 端和 broker 端之间的兼容性是“单向”的，即高版本 Kafka 的 broker 可以处理低版本 clients 的请求。反过来，低版本的 broker 不能处理高版本 client 的请求。究其原因就是请求类型的版本不兼容。

由于升级 clients 远比升级 broker 简单得多，因此这个限制给很多用户带来了麻烦，甚至有很多人都不愿意去升级 broker 版本——毕竟在不停机的情况下正确升级 Kafka 服务器是一个不小的挑战。

自 Kafka 0.10.2.0 版本开始，社区对这个问题进行了优化。对于低版本 clients + 高版本（broker 版本大于 0.10.2.0 版本）的环境而言，Kafka 提供了一个命令可以查看当前 broker 支持的请求类型版本，然后自动选择 broker 支持的最高版本来构造请求。运行该命令的方法如下。

(1) 打开一个终端，进入 Kafka 安装目录下。

(2) 运行命令 `bin/kafka-broker-api-versions.sh --bootstrap-server localhost:9092`。

图 6.42 和图 6.43 分别展示了 Kafka 0.10.2.1 和 0.10.0.1 版本的 broker 所支持协议的版本范围，注意图中箭头所指的 FETCH 请求。

```
10.2.0.49:9092 (id: 0 rack: test-rack) -> {
  Produce(0): 0 to 2 [usable: 2],
  Fetch(1): 0 to 3 [usable: 3],
  Offsets(2): 0 to 1 [usable: 1],
  Metadata(3): 0 to 2 [usable: 2],
  LeaderAndIsr(4): 0 [usable: 0],
  StopReplica(5): 0 [usable: 0],
  UpdateMetadata(6): 0 to 3 [usable: 3],
  ControlledShutdown(7): 1 [usable: 1],
  OffsetCommit(8): 0 to 2 [usable: 2],
  OffsetFetch(9): 0 to 2 [usable: 2],
  GroupCoordinator(10): 0 [usable: 0],
  JoinGroup(11): 0 to 1 [usable: 1],
  Heartbeat(12): 0 [usable: 0],
  LeaveGroup(13): 0 [usable: 0],
  SyncGroup(14): 0 [usable: 0],
  DescribeGroups(15): 0 [usable: 0],
  ListGroups(16): 0 [usable: 0],
  SaslHandshake(17): 0 [usable: 0],
  ApiVersions(18): 0 [usable: 0],
  CreateTopics(19): 0 to 1 [usable: 1],
  DeleteTopics(20): 0 [usable: 0]
}
```

图 6.42 0.10.2.1 支持协议版本范围

```
10.2.0.49:9092 (id: 0 rack: null) -> {
  Produce(0): 0 to 2 [usable: 2],
  Fetch(1): 0 to 2 [usable: 2],
  Offsets(2): 0 [usable: 0],
  Metadata(3): 0 to 1 [usable: 1],
  LeaderAndIsr(4): 0 [usable: 0],
  StopReplica(5): 0 [usable: 0],
  UpdateMetadata(6): 0 to 2 [usable: 2],
  ControlledShutdown(7): 1 [usable: 1],
  OffsetCommit(8): 0 to 2 [usable: 2],
  OffsetFetch(9): 0 to 1 [usable: 1],
  GroupCoordinator(10): 0 [usable: 0],
  JoinGroup(11): 0 [usable: 0],
  Heartbeat(12): 0 [usable: 0],
  LeaveGroup(13): 0 [usable: 0],
  SyncGroup(14): 0 [usable: 0],
  DescribeGroups(15): 0 [usable: 0],
  ListGroups(16): 0 [usable: 0],
  SaslHandshake(17): 0 [usable: 0],
  ApiVersions(18): 0 [usable: 0],
  CreateTopics(19): UNSUPPORTED,
  DeleteTopics(20): UNSUPPORTED
}
```

图 6.43 0.10.0.1 支持协议版本范围

图 6.42 连接的是 0.10.2.1 版本的 broker，可以看到该版本的 Kafka 服务器支持的 FETCH 请求版本范围是 0~3，默认使用 3；而图 6.43 连接的是 0.10.0.1 版本的 Kafka，它只支持 0~2 版本的 FETCH 请求。因此，在编写客户端程序时需要根据这张表来确认 broker 支持的请求的最高版本，这样就间接实现了“低 broker 处理高 client 请求”的兼容性目标。

考虑到 Java 版本的 client 已被广泛使用，社区也改写了 Java clients 底层的网络客户端代码，其会自动判断连接的 broker 端所支持 client 请求的最高版本，并自动创建合乎标准的请求。因此，对于 FETCH 请求和 PRODUCE 请求而言，使用 Java 版本 clients 的用户不必自己实现这些细节，但对于其他第三方 clients 而言，则需要注意版本兼容性问题。

6. Java API 构造请求实例

虽然 Kafka 提供了大量的脚本工具用于各种功能的实现，但很多时候我们还是希望可以把某些功能以编程的方式嵌入另一个系统中。这时使用 Java API 的方式就显得异常灵活了。下面就结合上面所讲的协议类型给出一个 Java API 的实例，来构造 CreateTopics 请求类型实现程序 API 方式创建 topic 的目的。

该程序用到的 API 依赖于 kafka-clients，如果使用 Maven 构建，需要在 pom.xml 中增加以

下依赖，以 Kafka 0.11.0.0 版本为例：

```
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-clients</artifactId>
  <version>0.11.0.0</version>
</dependency>
```

如果使用 Gradle，则加上：

compile group: 'org.apache.kafka', name: 'kafka-clients', version: '0.11.0.0'

在构造具体的请求之前，我们首先需要实现底层的基本框架，即实现图 6.40 中的创建连接、发送请求、接收响应等基本功能。代码如下：

```
/**
 * 发送请求主方法
 * @param host          目标 broker 的主机名
 * @param port          目标 broker 的端口
 * @param request       请求对象
 * @param apiKey        请求类型
 * @return              序列化后的 response
 * @throws IOException
 */
public ByteBuffer send(String host, int port, AbstractRequest request,
    ApiKeys apiKey) throws IOException {
    Socket socket = connect(host, port);
    try {
        return send(request, apiKey, socket);
    } finally {
        socket.close();
    }
}

/**
 * 发送序列化请求并等待 response 返回
 * @param socket        连向目标 broker 的 Socket
 * @param request       序列化后的请求
 * @return              序列化后的 response
 * @throws IOException
 */
private byte[] issueRequestAndWaitForResponse(Socket socket, byte[]
    request) throws IOException {
    sendRequest(socket, request);
    return getResponse(socket);
}
```

```

/**
 * 发送序列化请求给 Socket
 * @param socket          连向目标 broker 的 Socket
 * @param request          序列化后的请求
 * @throws IOException
 */
private void sendRequest(Socket socket, byte[] request) throws IOException {
    DataOutputStream dos = new DataOutputStream(socket.getOutputStream());
    dos.writeInt(request.length);
    dos.write(request);
    dos.flush();
}

/**
 * 从给定 Socket 处获取 response
 * @param socket          连向目标 broker 的 Socket
 * @return                获取到的序列化后的 response
 * @throws IOException
 */
private byte[] getResponse(Socket socket) throws IOException {
    DataInputStream dis = null;
    try {
        dis = new DataInputStream(socket.getInputStream());
        byte[] response = new byte[dis.readInt()];
        dis.readFully(response);
        return response;
    } finally {
        if (dis != null) {
            dis.close();
        }
    }
}

/**
 * 创建 Socket 连接
 * @param hostname        目标 broker 主机名
 * @param port            目标 broker 服务端口, 比如 9092
 * @return                创建的 Socket 连接
 * @throws IOException
 */
private Socket connect(String hostName, int port) throws IOException {
    return new Socket(hostName, port);
}

```

```
/**
 * 向给定 Socket 发送请求
 * @param request          请求对象
 * @param apiKey           请求类型，即属于哪种请求
 * @param socket           连向目标 broker 的 Socket
 * @return                 序列化后的 response
 * @throws IOException
 */
private ByteBuffer send(AbstractRequest request, ApiKeys apiKey,
Socket socket) throws IOException {
    RequestHeader header = new RequestHeader(apiKey.id, request.version(),
"client-id", 0);
    ByteBuffer buffer = ByteBuffer.allocate(header.sizeOf() + request.
sizeOf());
    header.writeTo(buffer);
    request.writeTo(buffer);
    byte[] serializedRequest = buffer.array();
    byte[] response = issueRequestAndWaitForResponse(socket,
serializedRequest);
    ByteBuffer responseBuffer = ByteBuffer.wrap(response);
    ResponseHeader.parse(responseBuffer);
    return responseBuffer;
}
```

接下来构造具体的 CreateTopics 请求：

```
/**
 * 创建 topic
 * 由于只是样例代码，有些内容就硬编码到程序里面了（比如主机名和端口），读者
 * 可自行修改
 * @param topicName        topic 名
 * @param partitions       分区数
 * @param replicationFactor 副本数
 * @throws IOException
 */
public void createTopics(String topicName, int partitions, short
replicationFactor) throws IOException {
    Map<String, CreateTopicsRequest.TopicDetails> topics = new HashMap<>();
    // 插入多个元素便可同时创建多个 topic
    topics.put(topicName, new CreateTopicsRequest.TopicDetails(partitions,
replicationFactor));
    int creationTimeoutMs = 60000;
    CreateTopicsRequest request = new CreateTopicsRequest.Builder
(topics, creationTimeoutMs).build();
```

```
// 给 localhost:9092 的 Kafka broker 发送 CREATE_TOPICS 请求
ByteBuffer response = send("localhost", 9092, request, ApiKeys.
CREATE_TOPICS);
CreateTopicsResponse.parse(response, request.version());
}
```

6.1.7 controller 设计

1. controller 概览

在一个 Kafka 集群中，某个 broker 会被选举出来承担特殊的角色，即控制器（下称 controller）。顾名思义，引入 controller 就是用来管理和协调 Kafka 集群的。具体来说，就是管理集群中所有分区的状态并执行相应的管理操作。

每个 Kafka 集群任意时刻都只能有一个 controller。当集群启动时，所有 broker 都会参与 controller 的竞选，但最终只能由一个 broker 胜出。一旦 controller 在某个时刻崩溃，集群中剩余的 broker 会立刻得到通知，然后开启新一轮的 controller 选举。新选举出来的 controller 将承担起之前 controller 的所有工作。图 6.44 展示了 controller 的架构，后续我们会为读者一一详解其中的组件。

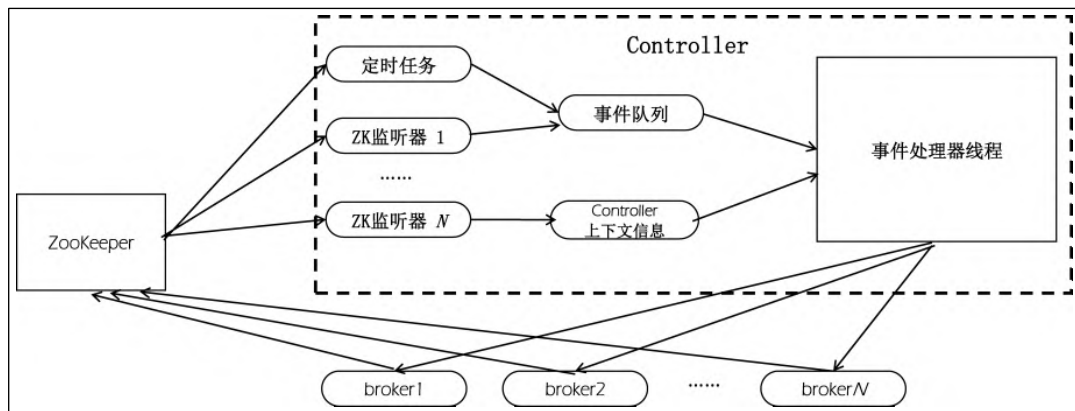


图 6.44 controller 架构

2. controller 管理状态

简单地说，controller 维护的状态分为两类：每台 broker 上的分区副本和每个分区的 leader 副本信息。从维度上看，这些状态又可分为副本状态和分区状态。controller 为了维护这两个状态专门引入了两个状态机，分别管理副本状态和分区状态。

（1）副本状态机（Replica State Machine）

当前，Kafka 为副本定义了 7 种状态以及每个状态之间的流转规则。这些状态分别如下。

- **NewReplica**: controller 创建副本时的最初状态。当处在这个状态时，副本只能成为 follower 副本。
- **OnlineReplica**: 启动副本后变更为该状态。在该状态下，副本既可以成为 follower 副本也可以成为 leader 副本。
- **OfflineReplica**: 一旦副本所在 broker 崩溃，该副本将变更为该状态。
- **ReplicaDeletionStarted**: 若开启了 topic 删除操作，topic 下所有分区的所有副本都会被删除。此时副本进入该状态。
- **ReplicaDeletionSuccessful**: 若副本成功响应了删除副本请求，则进入该状态。
- **ReplicaDeletionIneligible**: 若副本删除失败，则进入该状态。
- **NonExistentReplica**: 若副本被成功删除，则进入该状态。

副本状态机的流转规则如图 6.45 所示。当创建某个 topic 后，该 topic 下所有分区的所有副本都是 NonExistent 状态的，此时 controller 加载 ZooKeeper 中该 topic 每个分区的所有副本信息到内存中，同时将副本状态变更为 New，之后 controller 选择该分区副本列表中的第一个副本作为分区的 leader 副本并设置所有副本进入 ISR，然后在 ZooKeeper 中持久化该决定。

一旦确定了分区的 leader 和 ISR 之后，controller 会将这些信息以请求的方式发送给所有副本，同时将这些副本状态同步到集群的所有 broker 上以便让它们知晓。当这些都做完后，controller 会将分区的所有副本状态置为 Online，这也是副本正常工作的状态。

当开启了 topic 删除操作时，controller 会尝试停止所有副本，此时副本将停止向 leader 获取数据，但若停止的副本就是 leader 副本本身，则 controller 会设置该分区的 leader 为 NO_LEADER，之后副本进入 Offline 状态。一旦所有副本进入 Offline 状态，controller 需要将副本进一步变更到 ReplicaDeletionStarted 状态表明删除 topic 任务的开启。在这一步的状态流转中，controller 会给这些副本所在的 broker 发送请求，让它们删除本机上的副本数据。一旦删除成功，这些副本就变更到 ReplicaDeletionSuccessful 状态；如果有失败的副本，那么该副本进入 ReplicaDeletionIneligible 状态表明暂时还无法删除该副本，等待 controller 的重试。那些处于 ReplicaDeletionSuccessful 状态的副本稍后会被自动地变更到 NonExistent 终止状态，同时 controller 的上下文缓存会清除这些副本信息。这就是副本状态机操作副本状态流转的典型场景。

（2）分区状态机（Partition State Machine）

除了副本状态机，controller 还引入了分区状态机来负责集群下所有分区的状态管理，如

图 6.46 所示。乍一看似乎和副本状态机很类似，只不过分区状态机管理的对象是分区。

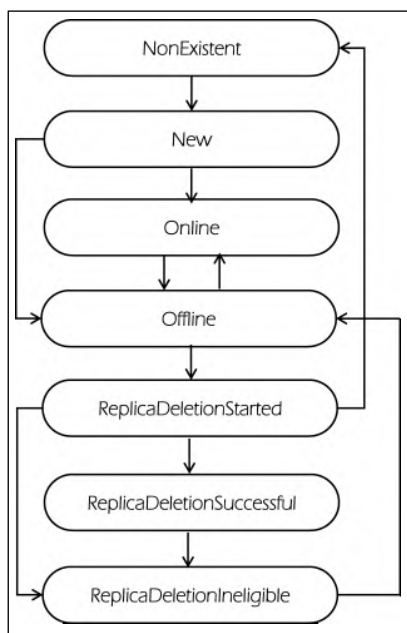


图 6.45 副本状态机

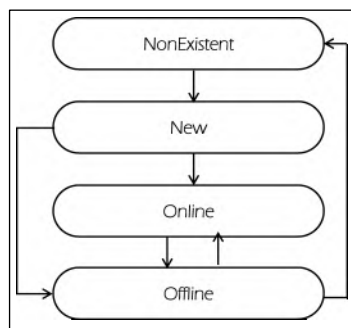


图 6.46 分区状态机

分区状态机只定义了 4 个分区状态，它们分别如下。

- **NonExistent**：表明不存在的分区或已删除的分区。
- **NewPartition**：一旦被创建，分区便处于该状态。此时，Kafka 已经为分区确定了副本列表，但尚未选举出 leader 和 ISR。
- **OnlinePartition**：一旦该分区的 leader 被选出，则进入此状态。这也是分区正常工作时的状态。
- **OfflinePartition**：在成功选举出 leader 后，若 leader 所在的 broker 宕机，则分区将进入该状态，表明无法正常工作了。

比起副本状态机 7 个状态，分区状态机的状态空间要小得多，分析它们之间状态的流转也要容易得多。当创建 topic 时，controller 负责创建分区对象，它首先会短暂地将所有分区状态设置为 NonExistent，之后马上读取 ZooKeeper 中（严格来说是从 controller 的上下文信息中读取，关于 controller 的上下文信息稍后会详细讨论）的副本分配方案，然后令分区状态进入 NewPartition。处于 NewPartition 状态的分区尚未有 leader 和 ISR，因此 controller 会初始化 leader 和 ISR 信息并设置分区状态为 OnlinePartition。此时分区开始正常工作。

若用户发起了删除 topic 操作或关闭 broker 操作，那么 controller 会令受影响的分区进入到 Offline 状态。如果是删除操作，则 controller 还会开启分区下面副本的删除操作并最终将分区状态设置为 NonExistent。而如果是关闭 broker 操作，则 controller 会判断该 broker 是否是分区的 leader。如果是则需要开启新一轮的 leader 选举并调整分区状态回 OnlinePartition。

3. controller 职责

对集群状态的维护只是 controller 保持运行状态一致性的一个基本要素，但却不是 controller 的职责所在。应该这样说，如果保持 controller 持续稳定地对外提供服务，就必须要求 controller 妥善地保存这些状态。实际上，controller 的职责相当多，包括如下职责。

- 更新集群元数据信息。
- 创建 topic。
- 删除 topic。
- 分区重分配。
- preferred leader 副本选举。
- topic 分区扩展。
- broker 加入集群。
- broker 崩溃。
- 受控关闭。
- controller leader 选举。



(1) 更新集群元数据

一个 clients 能够向集群中任意一台 broker 发送 METADATA 请求来查询 topic 的分区信息（比如 topic 有多少个分区、每个分区的 leader 在哪台 broker 上以及分区的副本列表）。随着集群的运行，这部分信息可能会发生变化，因此就需要 controller 提供一种机制，用于随时随地的把变更后的分区信息广播出去，同步给集群上所有的 broker。具体做法就是，当有分区信息发生变更时，controller 将变更后的信息封装进 UpdateMetadataRequests 请求（通信协议中的一种）中，然后发送给集群中的每个 broker，这样 clients 在请求数据时总是能够获取最新、最及时的分区信息。

(2) 创建 topic

controller 启动时会创建一个 ZooKeeper 的监听器，该监听器的唯一任务就是监控 ZooKeeper 节点/brokers/topics 下子节点的变更情况。当前一个 clients 或 admin 创建 topic 的方式主要有如下 3 种。

- 通过 kafka-topics 脚本的--create 创建。

- 构造 `CreateTopicsRequest` 请求创建。
- 配置 broker 端参数 `auto.create.topics.enable` 为 `true`，然后发送 `MetadataRequest` 请求。

无论上述哪种方式，其基本的原理都是在 ZooKeeper 的 `/brokers/topics` 下创建一个对应的 `znode`，然后把这个 `topic` 的分区以及对应的副本列表写入这个 `znode` 中。之前所说的 controller 监听器一旦监控到该目录下有新增 `znode`，就立即触发 `topic` 创建逻辑，即 controller 会为新建 `topic` 的每个分区确定 leader 和 ISR，然后更新集群的元数据信息。做完这些之后 controller 还会创建一个新的监听器用于监听 ZooKeeper 的 `/brokers/topics/<新增 topic>` 节点内容的变更。这样当 `topic` 分区发生变化的时候 controller 也能在第一时间得到通知。

（3）删除 topic

标准的 Kafka 删除 `topic` 方法有如下两种。

- 通过 `kafka-topics` 脚本的 `--delete` 来删除 `topic`。
- 构造 `DeleteTopicsRequest`。

这两种方式都是向 ZooKeeper 的 `/admin/delete_topics` 下新建一个 `znode`。controller 启动时会创建一个监听器专门监听该路径下的子节点变更情况，一旦发现有新增节点，则 controller 立即开启删除 `topic` 的逻辑。该逻辑主要分为两个阶段：①停止所有副本运行；②删除所有副本的日志数据。一旦完成这些操作，controller 会移除 `/admin/delete_topics/<待删除 topic>` 节点，这代表 `topic` 删除操作正式完成。

（4）分区重分配

分区重分配操作通常都是由 Kafka 集群的管理员发起的，旨在对 `topic` 的所有分区重新分配副本所在 broker 的位置，以期望实现更均匀的分配效果。在该操作中管理员需要手动制定分配方案并按照指定的格式写入 ZooKeeper 的 `/admin/reassign_partitions` 节点下。

分区副本重分配的过程实际上是先扩展再收缩的过程。controller 首先将分区副本集合进行扩展（旧副本集合与新副本集合的合集），等待它们全部与 leader 保持同步之后将 leader 设置为新分配方案中的副本，最后执行收缩阶段，将分区副本集合缩减成分配方案中的副本集合。

（5）preferred leader 选举

为了避免分区副本分配不均匀，Kafka 引入了 preferred 副本的概念。比如一个分区的副本列表是 `[1,2,3]`，那么 broker 1 就被称为该分区的 preferred leader，因为它位于副本列表的第一位。在集群运行的过程中，分区的 leader 因为各种各样的原因会发生变更，从而使得 leader 不再是 preferred leader，此时用户可以发起命令将这些分区的 leader 重新调整为 preferred leader。具体的方法有如下两种。

- 设置 broker 端参数 `auto.leader.rebalance.enable` 为 `true`，这样 controller 会定时地自动调整 preferred leader。
- 通过 `kafka-preferred-replica-election` 脚本手动触发。

在内部，这两种方法都向 ZooKeeper 的 `/admin/preferred_replica_election` 节点写入数据。同样地，controller 也会注册监听该目录的监听器。一旦被触发，controller 会将对应分区的 leader 调整回副本列表中的第一个副本，之后将此变更广播出去。

（6）topic 分区扩展

在 Kafka 集群的运行过程中，用户可能会发现某些 topic 的现有分区数不足以支撑 clients 的业务量，因此可能有增加分区的需求。当前增加分区主要使用 `kafka-topics` 脚本的 `--alter` 选项来完成。和创建 topic 一样它会向 ZooKeeper 的 `/brokers/topics/<topic>` 节点下写入新的分区目录。

前面说过，controller 在创建 topic 之后会注册一个新的监听器用于监听分区目录数据的变化。因此一旦增加了 topic 分区，该监听器会被触发，执行对应的分区创建任务（比如选举 leader 和 ISR 等），之后更新集群元数据信息。

（7）broker 加入集群

每个 broker 成功启动之后都会在 ZooKeeper 的 `/broker/ids` 下创建一个 `znode`，并写入 broker 的信息。如果要想让 Kafka 动态地维护 broker 列表，就必须注册一个 ZooKeeper 监听器时刻监控该目录下的数据变化。

每当有新 broker 加入集群时，该监听器会感知到变化，执行对应的 broker 启动任务，之后更新集群元数据信息并广而告之。

（8）broker 崩溃

由于当前 broker 在 ZooKeeper 中注册的 `znode` 是临时节点，因此一旦 broker 崩溃，broker 与 ZooKeeper 的会话会失效并导致临时节点被删除，故上面监控 broker 加入的那个监听器同样被用来监控那些因为崩溃而退出集群的 broker 列表。若发现有 broker 子目录“消失”，controller 便立即可知该 broker 退出集群，从而开启 broker 退出逻辑，最后更新集群元数据并同步到其他 broker 上。

（9）受控关闭

“优雅”地关闭 broker 是指通过 `kafka-server-stop` 脚本、`kill -9` 或 `kill -15` 的方式关闭 Kafka broker，而 broker 崩溃或强制退出通常都是以“掉电”或 `kill -9` 的方式实现的。前者被称为受控关闭。受控关闭能够最大限度地降低 broker 的不一致性。与其他 controller 功能中 controller 向其他 broker 发送请求不同的是，受控关闭是由即将关闭的 broker 向 controller 发送

请求的。请求的名字是 `ControlledShutdownRequest`。一旦发送完 `ControlledShutdownRequest`，待关闭 broker 将一直处于阻塞状态，直到接收到 broker 端发送的 `ControlledShutdownResponse`，表示关闭成功，或用完所有重试机会后强行退出。

controller 在处理完必要的 leader 重选举和 ISR 收缩调整之后，会给 broker 发送 `ControlledShutdownResponse` 表明该 broker 现在可以正常退出。

细心的读者可能会发现，之前所有 controller 的功能都是依托于 ZooKeeper 的帮助来实现的，具体来说就是依靠 ZooKeeper 的监听器功能实现的。但对于受控关闭而言，它依赖于 broker 端的 RPC 来实现，即 broker 直接发送请求给 controller，而没有借助 ZooKeeper。

（10）controller leader 选举

作为 Kafka 集群的重要组件，controller 必然要支持故障转移（fail-over）。若当前 controller 发生故障或显式关闭，Kafka 必须要能够保证及时选出新的 controller。当前，一个 Kafka 集群中发生 controller leader 选举的场景共有如下 4 种。

- 关闭 controller 所在 broker。
- 当前 controller 所在 broker 宕机或崩溃。
- 手动删除 ZooKeeper 的/controller 节点。
- 手动向 ZooKeeper 的/controller 节点写入新的 broker id。

在用户没有完全理解 controller 的选举机制之前，笔者不推荐用户使用后两种操作来触发 controller 的 leader 选举。

万变不离其宗的是，这 4 种操作变更实际上都是/controller 节点的内容，因此 controller 只需要做一件事情：创建一个监听该目录的监听器。/controller 本质上是一个临时节点，节点保存了当前 controller 所在的 broker id。集群首次启动时所有 broker 都会抢着创建该节点，但 ZooKeeper 保证了最终只能有一个 broker 胜出——胜出的那个 broker 即成为 controller。

一旦成为 controller，它会增加 controller 的版本号，即更新/controller_epoch 节点的值，然后履行上面所有的这些职责。

对于那些没有成为 controller 的 broker 们而言，它们不会甘心失败，而是继续监听/controller 节点的存活情况并随时准备竞选新的 controller。

4. controller 与 broker 间的通信

目前，controller 启动时会为集群中所有 broker 创建一个专属的 Socket 连接（也包括 controller 所在的 broker）。这就是说，若一个 Kafka 集群有 100 台 broker 机器，那么 controller 会创建 100 个 Socket 连接。虽说当前比较新的 Kafka 版本已经统一使用基于 Java NIO Selector

的网络连接库来实现这一功能，但 controller 依然会为每个 TCP 连接创建一个 RequestSendThread 线程。还是拿刚才的例子进行说明，100 台 broker 会创建 100 个 Socket 连接和 100 个 I/O 线程。

这些连接和线程被用于让 controller 给集群 broker 发送请求。在当前的设计下，controller 只能给 broker 发送 3 种请求，它们分别如下。

- **UpdateMetadataRequest:** 之前反复提到的更新集群元数据请求。该请求携带了集群当前最新的元数据信息。接收到该请求后，broker 会更新本地内存中的缓存信息，从而保证返还给 clients 的信息总是最新、最及时的。
- **LeaderAndIsrRequest:** 用于创建分区、副本，同时完成作为 leader 和作为 follower 角色各自的逻辑。
- **StopReplicaRequest:** 停止指定副本的数据请求操作，另外还负责删除副本数据的功能。

controller 通常都是发送请求给 broker 的，但在 controller 职责部分中我们提到的 ControlledShutdownRequest 请求是例外，该请求是待关闭 broker 通过 RPC 直接发给 controller 的，即请求的流向是反的。另外这个请求还有一个特别之处，controller 绝大多数的功能都是依托 ZooKeeper 来完成的，但只有这个请求是 broker 与 controller 直接进行交互完成的。

5. controller 组件

controller 到底是由哪些组件构成的呢？图 6.47 给出了一个完整的组件构成图。

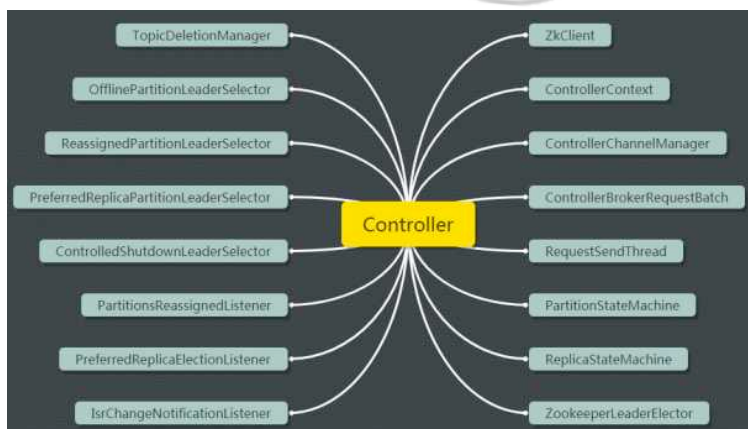


图 6.47 controller 组件构成图

乍一看 controller 的组件非常多，但其实可以分为数据类、基础功能类、状态机类、选举器和监听器等。

(1) 数据类组件——ControllerContext

ControllerContext 被称为 controller 上下文或 controller 缓存，可以说是 controller 最重要的数据组件了。它汇总了 ZooKeeper 中关于 Kafka 集群的所有元数据信息，是 controller 能够正确提供服务的基础。在 0.11.0.0 版本之前，Kafka controller 的设计是多线程的，因此保护好这个上下文，使其免受多线程并发修改就成了 controller 很重要的任务。事实上，0.11.0.0 之前的版本中，controller 被用户广为诟病的主要原因就是，使用了大量的同步机制来保护这个东西。这其中引入的复杂性令很多想要改进 controller 的社区开发人员望而却步。

如前所述，ControllerContext 里面的内容非常丰富，如图 6.48 所示。

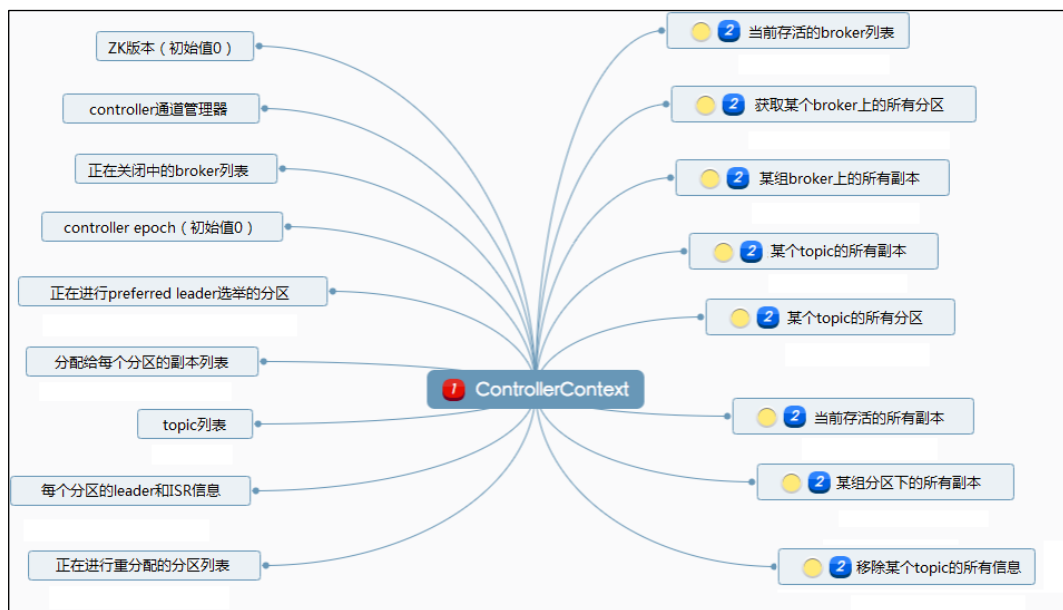


图 6.48 ControllerContext 组件构成

如图 6.48 所示，它几乎囊括了 Kafka 集群的一切信息。

(2) 基础功能类

笔者倾向于将以下组件归为基础功能类，它们分别如下。

- ZkClient：封装与 ZooKeeper 的各种交互 API。controller 与 ZooKeeper 的所有操作都交由该组件完成。
- ControllerChannelManager：前面说过了 controller 需要向其他 broker 发送请求，这个通道管理器即负责此事。

- **ControllerBrokerRequestBatch**: controller 将发往同一 broker 的各种请求按照类型分组, 稍后统一发送以提升效率。此组件就是用来管理请求 batch 的。
- **RequestSendThread**: 负责给其他 broker 发送请求的 I/O 线程。
- **ZookeeperLeaderElector**: 结合 ZooKeeper 负责 controller 的 leader 选举。

(3) 状态机类

属于这部分的组件有如下 3 个。

- **ReplicaStateMachine**: 副本状态机, 负责定义副本状态以及合法的副本状态流转。
- **PartitionStateMachine**: 分区状态机, 负责定义分区状态以及合法的分区状态流转。
- **TopicDeletionManager**: topic 删除状态机, 处理删除 topic 的各种状态流转以及相应的状态变更。

(4) 选举器类

当前, controller 提供了 4 个选举器用于各种情况下的 leader 选举。注意, 此处的 leader 选举是为分区进行 leader 选举而非选举 controller。它们分别如下。

- **OfflinePartitionLeaderSelector**: 负责常规性的分区 leader 选举。
- **ReassignPartitionLeaderSelector**: 负责用户发起分区重分配时的 leader 选举。
- **PreferredReplicaPartitionLeaderSelector**: 负责用户发起 preferred leader 选举时的 leader 选举。
- **ControlledShutdownLeaderSelector**: 负责 broker 在受控关闭后的 leader 选举。

(5) ZooKeeper 监听器

controller 自己维护的监听器实际上只有 3 个, 它们分别如下。

- **PartitionsReassignedListener**: 监听 ZooKeeper 下分区重分配路径的数据变更情况。
- **PreferredReplicaElectionListener**: 监听 ZooKeeper 下 preferred leader 选举路径的数据变更。
- **IsrChangeNotificationListener**: 监听 ZooKeeper 下 ISR 列表变更通知路径下的数据变化。

这里简单解释一下上面最后一个监听器的含义。Kafka 一旦发现 topic 分区的 ISR 发生了变化, 就会在 ZooKeeper 的 `/isr_change_notification` 节点下写入一个新的数据节点, 里面封装了集群中哪些 topic 的哪些分区对应的 ISR 发生了变更。该监听器监控到节点变化后会发起“更新元数据”请求给集群中的所有 broker。

事实上, controller 定义的监听器远不止这 3 个, 上面 3 个监听器只是 controller 自己维护的。更多的监听器交由各个状态机分别来维护。

6. 老版本 controller 设计缺陷

这里的老版本是指 Kafka 0.11.0.0 之前的版本。老版本 controller 设计中有以下设计缺陷。

- 多线程共享状态：编写正确的多线程程序一直是 Java 开发者的痛点。在 controller 的实现类 `KafkaController` 中创建了很多线程，比如之前提到的 `RequestSendThread` 线程，另外 `ZkClient` 也会创建单独的线程来处理 `ZooKeeper` 回调逻辑，这还不算 `TopicDeletionManager` 创建的线程和其他 I/O 线程等，而几乎所有这些线程都需要访问 controller 的上下文信息（`RequestSendThread` 只操作它们专属的请求队列，不会访问 `ControllerContext`），因此必要的多线程同步机制是一定需要的。当前 controller 是使用私有 `monitor` 锁来实现的，因此可以说没有并行度可言。
- 代码组织混乱：相信看过源代码的人对这一点一定深有体会。`KafkaController`、`PartitionStateMachine` 和 `ReplicaStateMachine` 每个都是 500 多行的大类且彼此混调的现象明显，比如 `KafkaController` 的 `stopOldReplicasOfReassignedPartition` 方法调用 `ReplicaStateMachine` 的 `handleStateChanges` 方法，而后者又会调用 `KafkaController` 的 `remoteReplicaFromIsr` 方法。类似的情况还发生在 `KafkaController` 和 `ControllerChannelManager` 之间。
- 管理类请求与数据类请求未分离：当前 broker 对入站请求类型不做任何优先级处理，不论是 `PRODUCE` 请求、`FETCH` 请求还是 controller 类的请求。这就可能造成一个问题，即 clients 发送的数据类请求积压导致 controller 推迟了管理类请求的处理。设想这样的场景，假设 controller 向 broker 广播了 leader 发生变更，于是新 leader 开始接收 clients 端请求，而同时老 leader 所在的 broker 由于出现了数据类请求的积压，使得它一直忙于处理这些请求而无法处理 controller 发来的 `LeaderAndIsrRequest` 请求，因此就会出现“双主”的情况——也就是所谓的脑裂。此时若 client 发送的一个 `PRODUCE` 请求未指定 `acks=-1`，那么因为日志水印截断的缘故，这个请求包含的消息就可能“丢失”了。现在社区中关于 controller 丢失数据的 bug，大部分是因为这个原因造成的。
- controller 同步写 `ZooKeeper` 且是一个分区一个分区地写：当前 controller 操作 `ZooKeeper` 是通过 `ZkClient` 来完成的。`ZkClient` 目前同步写入 `ZooKeeper`，而同步通常意味着性能不高。更为严重的是，controller 是一个分区一个分区进行写入的，对于分区数很多的集群来说，这无疑是一个巨大的性能瓶颈。如果用户仔细查看源代码，可以发现 `PartitionStateMachine` 的 `electLeaderForPartition` 就是一个分区一个分区地选举的。
- controller 一个分区一个分区地发送请求：controller 当前发送请求都是按照分级别别发送的，即一个分区一个分区地发送，没有任何 batch 或并行可言，效率很低。
- controller 给 broker 的请求无版本号信息：这里的版本号类似于 Java 版本 consumer 中的 `generation`，总之是要有一种机制告诉 controller broker 的版本信息。因为有些情况下

broker 会处理本已过期或失效的请求，从而导致 broker 状态不一致。举一个例子，如果一个 broker 正常关闭过程中“宕机”了，那么重启之后这个 broker 就有可能处理之前 controller 发送过来的 StopReplicaRequest，导致某些副本被置成 offline，进而无法使用。

- ZkClient 阻碍状态管理：controller 目前使用了 ZkClient 这个开源工具，它可以自动重建会话并使用特有的线程顺序处理所有的 ZooKeeper 监听消息。因为是顺序处理，它就有可能无法及时响应最新的状态变更，从而导致 Kafka 集群状态不一致。

鉴于这些缺陷，Kafka 社区在 0.11.0.0 版本中重构了 controller 的部分设计并着重完善了 controller 上下文的访问模型。

7. 新版本 controller

有些遗憾的是，新版本 controller 架构的改进相当有限。在笔者看来，0.11.0.0 版本的 controller 仅仅改造了 controller 多线程事件处理模型，即改为使用单线程基于事件队列的模型。即使是目前最新的 1.0.0 版本，controller 组件的变化也不是很大，足见其设计是很复杂的，改动起来并不容易。

不过切换到单线程事件队列模型还是能够极大地降低并发同步的开销。使用 Java LinkedBlockingQueue 可以很容易地在多线程间维护线程安全，因此再也不需要在多个线程上使用锁机制来保护 controller 状态了。

6.1.8 broker 请求处理

1. Reactor 模式

Kafka broker 处理请求的模式就是 Reactor 设计模式。根据维基百科的定义，Reactor 设计模式是一种事件处理模式，旨在处理多个输入源同时发送过来的请求。Reactor 模式中的服务处理器（service handler）或分发器（dispatcher）将入站请求（inbound request）按照多路复用的方式分发到对应的请求处理器（request handler）中，如图 6.49 所示。

本质上这和生产者-消费者的模式很像。外部的输入源就类似于 producer 角色，它们的工作就是生产“事件”发送给 Reactor 中的 dispatcher，具体而言是将事件放入 dispatcher 中的队列上；而 Reactor 通常会创建多个 request handler 线程专门消费 dispatcher 分发过来的事件。对于 Kafka 而言，该模型中的事件实际上对应于 Socket 连接通道（SocketChannel），即 broker 上每当有新的 Socket 连接通道被创建，dispatcher 都会将该连接分发给下面某个 request handler 来消费。

如图 6.49 所示，Reactor 模式中有很两个很重要的组件 acceptor 线程和 processor 线程。acceptor 线程实时地监听外部数据源发送过来的事件，并执行分发任务；processor 线程执行事件处理逻辑并将处理结果发送给 client。

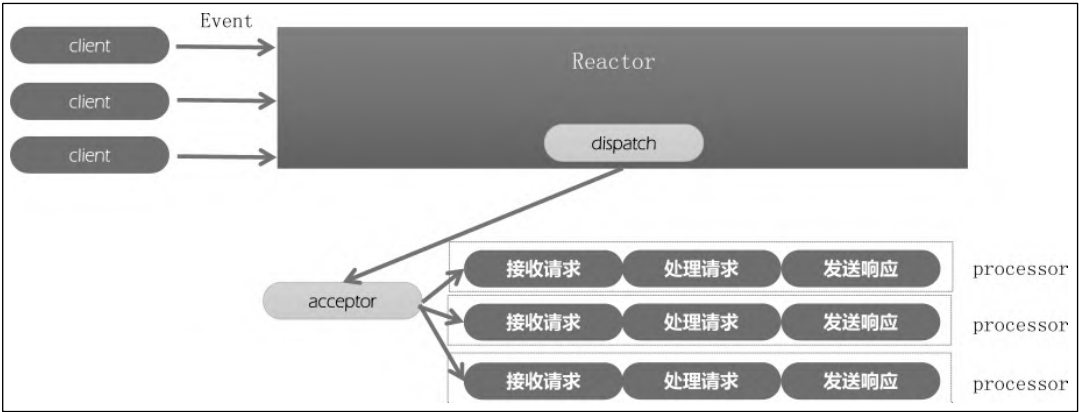


图 6.49 Reactor 模式

2. Kafka broker 请求处理

Kafka broker 请求处理实现了上面的 Reactor 模式。在 Kafka 中，每个 broker 都有一个 acceptor 线程和若干个 processor 线程。processor 线程的数量是可以配置的。num.network.threads 就用于控制该数量的 broker 端参数，默认值是 3，即每个 broker 都创建 3 个 processor 线程。值得注意的是，broker 会为用户配置的每组 listener 创建一组 processor 线程。还记得吧，前面章节中我们提到过每个 broker 可以同时设置多种通信安全协议，比如 PLAINTEXT 和 SSL，因此一旦某个 broker 同时配置了多套通信安全协议，那么 Kafka 会为每个协议都创建一组 processor 线程。

图 6.50 展示了 Kafka broker 端的请求处理流程。

我们之前提到过，在 Kafka 中对应于 Reactor 模式的“事件”实际上是连向 broker 的 Socket 连接通道，而不是 clients 端发送过来的真实请求。那么 Kafka 的 broker 到底是如何处理请求的呢？前面简单提到过多路复用（multiplexing）的概念，即使用很少的线程来维护多个连接。clients 端通常会保存与 broker 的长连接，因此不需要频繁地重建 Socket 连接，故 broker 端固定使用一个 acceptor 线程来唯一地监听入站连接。由于只做新连接监听这一件事情，acceptor 线程的处理逻辑是很轻量级的，在实际使用过程中通常也都不是系统瓶颈。这就是多路复用 in broker 端的第一个应用。

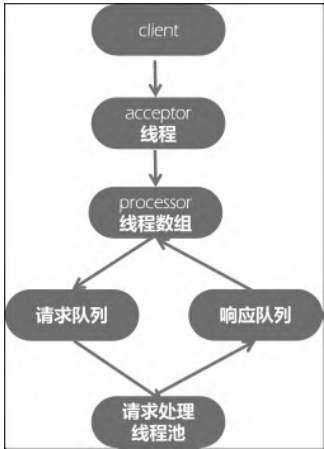


图 6.50 broker 请求处理流程

下面再来看看 processor 线程。processor 线程接收 acceptor 线程分配的新 Socket 连接通道，然后开始监听该通道上的数据传输。目前 broker 以线程数组而非线程池的方式来实现这组

processor，之后使用很简单的数组索引轮询方式依次给每个 processor 线程分配任务，实现了最均匀化的负载均衡。processor 线程实际上也不是处理请求的真正执行者，Kafka 创建了一个 KafkaRequestHandler 线程池专门地处理真正的请求。processor 线程一个重要的任务就是将 Socket 连接上接收到的请求放入请求队列中。每个 broker 启动时会创建一个全局唯一的请求队列，大小由 broker 端参数 `queued.max.requests` 控制，默认值是 500，表示每个 broker 最多只能保存 500 个未处理的请求。一旦超过该数字，clients 端发送给 broker 的请求将会被“阻塞”，直到该队列腾出空间。

记住，是 processor 线程把请求放入请求队列的，而 KafkaRequestHandler 线程池分配具体的线程从该队列中获取请求并执行真正的请求处理逻辑。该线程池的大小也是可以配置的，由 broker 端参数 `num.io.threads` 控制，默认是 8 个线程。

除了请求队列，每个 broker 还会创建与 processor 线程数等量的响应队列，即为每个 processor 线程都创建一个对应的响应队列。processor 线程的另一个很重要的任务就是实时处理各自响应队列中的响应结果。

那么，多路复用在 processor 线程中是如何体现的呢？设想一下，如果某个 broker 上存在多个 clients 端连接过来的 Socket，特别是 clients 数要远远大于 processor 线程数，那么每个 processor 线程都需要处理多个 Socket 连接通道上的数据。难道 processor 线程需要为每个 Socket 连接通道都创建一个对应的线程来处理吗？显然不是。Kafka 在设计上使用了 Java NIO 的 Selector+Channel+Buffer 的思想，在每个 processor 线程中维护一个 Selector 实例，并通过这个 Selector 来管理多个通道上的数据交互，这便是多路复用在 processor 线程上的应用。

下面给出一个完整的例子来帮助读者彻底理解 broker 端请求处理的全流程。假设 broker id 是 B1，client id 是 C1，一切 broker 端配置均遵循默认配置。完整的请求处理流程如下。

(1) 启动 B1。

①启动 acceptor 线程 A。

②启动 3 个 processor 线程 P1、P2 和 P3。

③创建 KafkaRequestHandler 线程池和 8 个请求处理线程 H1 ~ H8。

(2) B1 启动后，线程 A 不断轮询是否存在连向该 broker 的新连接；P1~P3 实时轮询线程 A 是否发送新的 Socket 连接通道以及请求队列和响应队列是否需要处理；H1~H8 则实时监控请求队列中的新请求。

(3) 此时，C1 开始向 B1 发送数据。首先 C1 会创建与该 broker 的 Socket 连接。

(4) 线程 A 监听到该 Socket 连接，接收之，然后将该连接发送给 P1~P3 中的一个，假设

是 P2。

(5) P2 下一次轮询开始时发现有 A 线程传过来的新连接，将其注册到 Selector 上并开始监听其上的入站请求。

(6) 现在 C1 开始给 B1 发送 PRODUCE 请求。

(7) P2 监听到有新的请求到来，故获取之，然后发送到请求队列中。

(8) 由于 H1~H8 实时监听请求队列，故必然有一个线程最早发现该 PRODUCE 请求的到来，这里假设是 H5。H5 线程将该请求从请求队列中取出并开始处理。

(9) H5 线程处理请求完成，将处理结果以响应的方式放入 P2 的响应队列。

(10) P2 监听到它的响应队列有新的响应到来，因而将响应从该队列中取出并通过对应的 Socket 连接通道发送给 C1。

(11) C1 接收到响应，标记本次 PRODUCE 请求处理过程结束。

以上便是一个典型的 broker 请求处理流程。

6.2 producer 端设计

6.2.1 producer 端基本数据结构

新版本客户端（包含新版本 producer 和新版本 consumer）重写了之前服务器端代码提供的很多数据结构，以摆脱对服务器端代码的依赖。其中有一些是理解新版本 producer 设计所必需的，它们包括（但不限于）如下这些。

1. ProducerRecord

一个 ProducerRecord 封装了一条待发送的消息（或称为记录）。虽然 Kafka 0.11.0.0 版本对消息格式进行了部分重构以支持事务和精确一次处理语义，但本节我们将以 0.10.2.1 版本的消息格式进行 producer 的说明。

ProducerRecord 由 5 个字段构成，它们分别如下。

- topic: 该消息所属的 topic。
- partition: 该消息所属的分区。
- key: 消息 key。
- value: 消息体。
- timestamp: 消息时间戳。

ProducerRecord 允许用户在创建消息对象的时候直接指定要发送的分区，这样 producer 后

续发送该消息时可以直接发送到指定分区，而不用先通过 `Partitioner` 计算目标分区。另外，我们还可以直接指定消息的时间戳——但一定要慎重使用这个功能，因为它有可能会令时间戳索引机制失效（笔者曾经直接指定时间戳故意打乱发送顺序进行测试，比如先发送消息的时间戳大于后发送消息的时间戳，最后发现通过时间戳定位消息时会发生混乱。为此笔者还特意开了一个 bug 报告，不过被认为是“当前不被支持的用法”）。

2. RecordMetadata

该数据结构表示 Kafka 服务器端返回给客户端的消息的元数据信息，包含如下内容。

- `offset`: 消息在分区日志中的位移信息。
- `timestamp`: 消息时间戳。
- `topic/partition`: 所属 topic 的分区。
- `checksum`: 消息 CRC32 码。
- `serializedKeySize`: 序列化后的消息 key 字节数。
- `serializedValueSize`: 序列化后的消息 value 字节数。

上面的元数据信息的前 3 项信息是比较重要的，`producer` 端可以使用这些信息做一些消息发送成功之后的处理，比如写入日志等。

6.2.2 工作流程

如果把 `producer` 统一看成一个盒子，那么整个 `producer` 端的工作原理便如图 6.51 所示。大体上来说，用户首先构建待发送的消息对象 `ProducerRecord`，然后调用 `KafkaProducer#send` 方法进行发送。`KafkaProducer` 接收到消息后首先对其进行序列化，然后结合本地缓存的元数据信息一起发送给 `partitioner` 去确定目标分区，最后追加写入内存中的消息缓冲池（`accumulator`）。此时 `KafkaProducer#send` 方法成功返回。

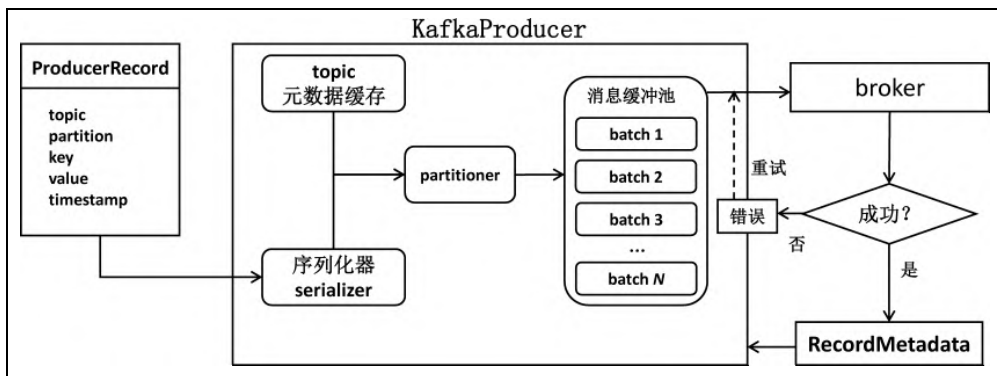


图 6.51 producer 端的工作原理

KafkaProducer 中还有一个专门的 Sender I/O 线程负责将缓冲池中的消息分批次发送给对应的 broker，完成真正的消息发送逻辑。

图 6.51 中其实并没有深入展开 producer 的工作原理。这里笔者打算详细说说 producer 内部到底是如何工作的，也就是梳理一下当用户调用 `KafkaProducer.send(ProducerRecord, Callback)` 时 Kafka 内部都发生了什么事情。

第一步：序列化+计算目标分区。

这是 `KafkaProducer#send` 逻辑的第一步，即为待发送消息进行序列化并计算目标分区，如图 6.52 所示。

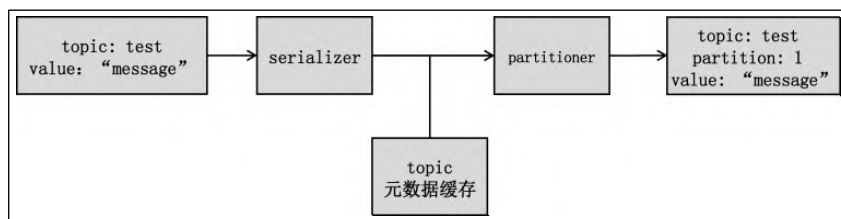


图 6.52 producer 序列化

如图 6.52 所示，一条所属 topic 是 "test"，消息体是 "message" 的消息被序列化之后结合 KafkaProducer 缓存的元数据（比如该 topic 分区数信息等）共同传给后面的 Partitioner 实现类进行目标分区的计算。

第二步：追加写入消息缓冲区（accumulator）。

producer 创建时会创建一个默认 32MB（由 `buffer.memory` 参数指定）的 accumulator 缓冲区，专门保存待发送的消息。除了之前在“关键参数”部分中提到的 `linger.ms` 和 `batch.size` 等参数之外，该数据结构中还包含了一个特别重要的集合信息：消息批次信息（batches）。该集合本质上是一个 `HashMap`，里面分别保存了每个 topic 分区下的 batch 队列，即前面说的批次是按照 topic 分区进行分组的。这样发往不同分区的信息保存在对应分区下的 batch 队列中。举一个简单的例子，假设消息 M1、M2 被发送到 test 的 0 分区但属于不同的 batch，M3 被发送到 test 的 1 分区，那么 batches 中包含的信息就是 `{"test-0" -> [batch1, batch2], "test-1" -> [batch3]}`。

单个 topic 分区下的 batch 队列中保存的是若干个消息批次，每个 batch 中最重要的 3 个组件如下。

- `compressor`：负责执行追加写入操作。
- `batch` 缓冲区：由 `batch.size` 参数控制，消息被真正追加写入的地方。
- `thunks`：保存消息回调逻辑的集合。

这一步的目的就是将待发送的消息写入消息缓冲池中，具体流程如图 6.53 所示。

这一步执行完毕之后理论上讲 `KafkaProducer.send` 方法就执行完毕了，用户主线程所做的事情就是等待 `Sender` 线程发送消息并执行返回结果。

第三步：`Sender` 线程预处理及消息发送。

此时，该 `Sender` 线程登场了。严格来说，`Sender` 线程自 `KafkaProducer` 创建后就一直都在运行着。它的工作流程基本上是如下这样的。

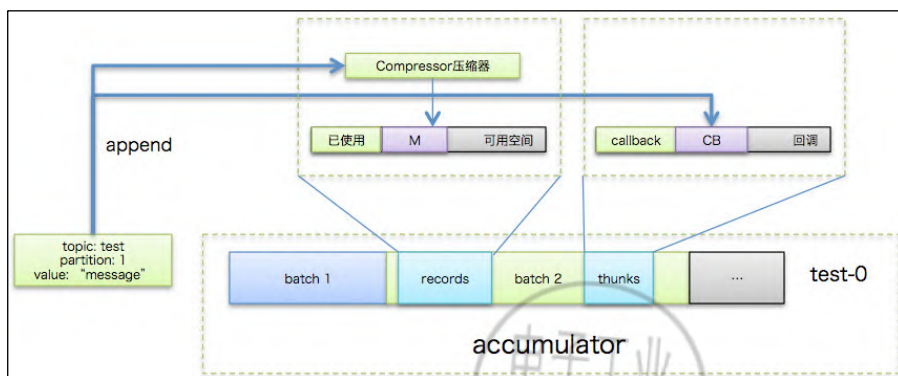


图 6.53 producer append 消息

- (1) 不断轮询缓冲区寻找已做好发送准备的分区。
- (2) 将轮询获得的各个 batch 按照目标分区所在的 leader broker 进行分组。
- (3) 将分组后的 batch 通过底层创建的 Socket 连接发送给各个 broker。
- (4) 等待服务器端发送 response 回来。

为了说明上的方便，笔者给出图 6.54 来详细解释 `Sender` 线程的工作原理。

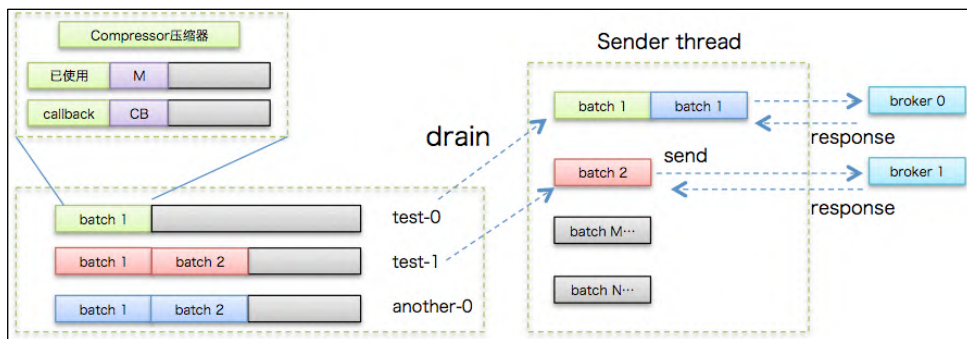


图 6.54 producer Sender 线程发送消息

第四步：Sender 线程处理 response。

图 6.54 中 Sender 线程会发送 PRODUCE 请求给对应的 broker，broker 处理完毕之后发送对应的 PRODUCE response。一旦 Sender 线程接收到 response，将依次（按照消息发送顺序）调用 batch 中的回调方法，如图 6.55 所示。

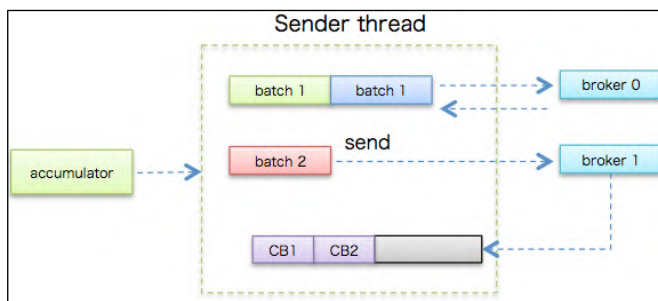


图 6.55 producer Sender 处理 response

做完这一步，producer 发送消息的工作就可以算作 100%完成了。通过这 4 步我们可以看到新版本 producer 发送事件完全是异步过程。因此在调优 producer 前我们需要搞清楚性能瓶颈到底是在用户主线程上还是在 Sender 线程上。

6.3 consumer 端设计

6.3.1 consumer group 状态机

在第 5 章中我们介绍了新版本 consumer 的概要设计。本节将重点介绍新版本 consumer 中 consumer group 的设计与管理。

如前所述，新版本 `consumer` 依赖于 `broker` 端的组协调者（`coordinator`）来管理组内的所有 `consumer` 实例并负责把分配方案下发到每个 `consumer` 上。分配方案是由组内的 `leader consumer` 根据指定的分区分配策略指定的。该分配策略必须是组内所有 `consumer` 都支持的。事实上，如果所有 `consumer` 协调在一起无法选出共同的分区策略，那么 `coordinator` 就会抛出异常。这种设计就确保了每个 `consumer group` 总有一个一致性的分配策略，同时还能确保每个 `consumer` 只能为它拥有的分区提交位移。

Kafka 社区刚刚推出分区分配的操作是在 consumer 端执行的而非 broker 端，这样做的好处主要有如下两点。

- 便于维护与升级：如果在 broker 端实现，那么分配策略的变动势必要重启整个 Kafka

集群。生产环境中重启服务器的代价是很高的。

- 便于实现自定义策略：不同的策略由不同的逻辑实现。coordinator 端代码不容易实现灵活可定制的分配逻辑。

基于以上两点优势，新版 consumer 对 group 下所有成员的分区分配工作由 consumer 代码实现。这样做的另一个好处就是，解耦了组管理与分区分配，coordinator 负责组管理工作，而 consumer 程序负责分区分配。

当前，Kafka 为每个 consumer group 定义了 5 个状态，它们分别如下。

- Empty: 该状态表明 group 下没有任何 active consumer，但可能包含位移信息。每个 group 创建时便处于 Empty 状态。若 group 工作了一段时间后所有 consumer 都离开组，那么 group 也会处于该状态。由于可能包含位移信息，处于此状态下的 group 依然可以响应 OffsetFetch 请求，即返回 clients 端对应的位移信息。
- PreparingRebalance: 该状态表明 group 正在准备进行 group rebalance。此时，group 已经“受理”了部分成员发送过来的 JoinGroup 请求，同时等待其他成员发送 JoinGroup 请求，直到所有成员都成功加入组或超时。由于该状态下的 group 依然可能保存有位移信息，因此 clients 依然可以发起 OffsetFetch 请求去获取位移，甚至还可以发起 OffsetCommit 请求去提交位移。
- AwaitingSync: 该状态表明所有成员都已经加入组并等待 leader consumer 发送分区分配方案。同样地，此时依然可以获取位移，但若提交位移，coordinator 将会抛出 REBALANCE_IN_PROGRESS 异常来表明该 group 正在进行 rebalance。
- Stable: 该状态表明 group 开始正常消费。此时 group 必须响应 clients 发送过来的任何请求，比如位移提交请求、位移获取请求、心跳请求等。
- Dead: 该状态表明 group 已经彻底废弃，group 内没有任何 active 成员并且 group 的所有元数据信息都已被删除。处于此状态的 group 不会响应任何请求。严格来说，coordinator 会返回 UNKNOWN_MEMBER_ID 异常。

任意时刻，所有 consumer group 必然处于上面 5 个状态中的一个。这 5 个状态之间的状态流转如图 6.56 所示。当 group 首次创建时，coordinator 会设置该 group 状态为 Empty，当有新的成员加入组时，组状态变更为 PreparingRebalance。此时 group 会等待一段时间让更多的组成员加入（在 Kafka 0.10.1.0 之后，这段时间由 consumer 端参数 max.poll.interval.ms 指定）。如果所有成员都及时地加入了组，那么组状态变更为 AwaitingSync。此时 leader consumer 开始分配消费方案。

在分配方案确定后，leader consumer 将分配方案以 SyncGroup 请求的方式发送给 coordinator，然后 coordinator 把方案下发给下面的所有组成员。此时组状态进入到 Stable，表

明 group 开始正常消费数据。

当 group 处于 Stable 时，若所有成员都离组，那么此时 group 状态会首先调整为 PreparingRebalance，然后变更为 Empty，最后等待元数据过期被移除后 group 变更为 Dead。

以上就是一个 consumer group 典型的状态生命周期流转。下面根据每个状态详细讨论一下状态流转的条件。

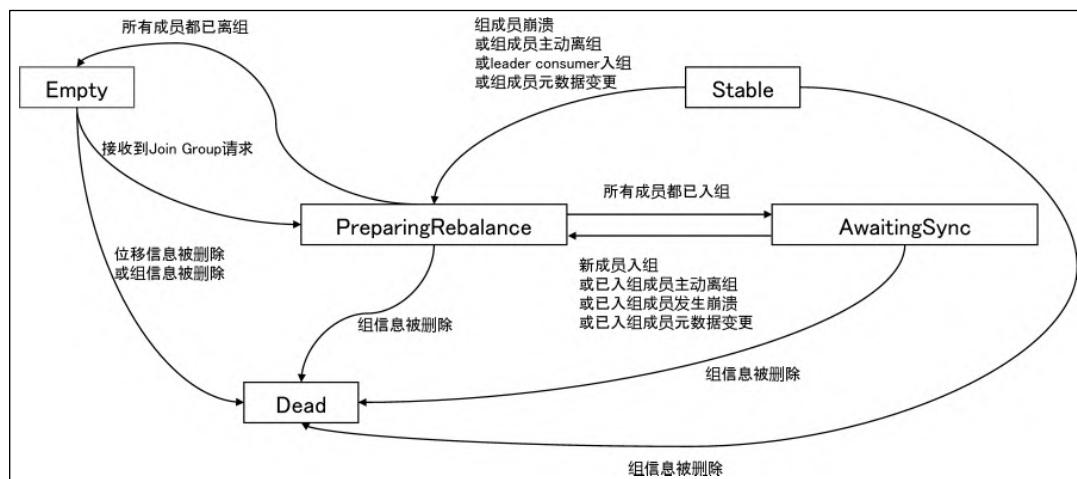


图 6.56 consumer group 状态机

- Empty 与 PreparingRebalance: Empty 状态的 group 下没有任何 active consumer。此时当有一个 consumer 加入时，Empty 变为 PreparingRebalance。反之，处于 PreparingRebalance 状态的 group 中，当所有成员都离开组时，PreparingRebalance 变为 Empty。
- Empty 与 Dead: Empty 状态下的 group 依然可能保存 group 元数据信息甚至位移信息。Kafka 默认会在 1 天（这个时间可通过 broker 端参数 `offsets.retention.minutes` 配置）后删除这些数据。一旦这些数据被删除，则 group 进入 Dead 状态。
- PreparingRebalance 与 AwaitingSync: 在 PreparingRebalance 状态时，若成员在规定时间内（`max.poll.interval.ms`）完成加入组操作，那么 group 进入 AwaitingSync 状态。若有的组成员很慢，没能在这段时间内加入组，那么规定时间一过 group 依然会进入 AwaitingSync，当那个慢 consumer 加入组时，group 又会重新变更为 PreparingRebalance。因此在实际应用中一定要谨慎设置 `max.poll.interval.ms` 参数。对于处于 AwaitingSync 状态的 group 而言，当已加入成员崩溃、主动离组或元数据信息发生变更时，group 也会重新进入 PreparingRebalance。
- AwaitingSync 与 Stable: 在 coordinator 成功下发了 leader consumer 所做的分配方案后，

group 进入到 Stable 状态开始正常工作。

- Stable 与 PreparingRebalance: group 正常工作时, 当有成员发生崩溃或主动离组, 抑或是 leader consumer 重新加入组, 再或是有成员元数据发生变更, 则 group 会直接进入 PreparingRebalance 开启新一轮 rebalance。在实际场景中, 最常见的 rebalance 都是因为处于 Stable 状态的 group 下的 consumer 消费处理逻辑过重且 session 超时, 从而被“踢出” group 而导致的。
- 其他状态与 Dead: 前面章节说过新版本 consumer group 的位移保存在内部 topic——__consumer_offsets 下的某个分区。当这个分区 leader 所在的 broker 发生了崩溃时, 就必须对该分区进行迁移, 从而导致 coordinator 的变更。当出现这种情况时, 不论 group 处于哪种状态, 都必须直接将 group 设置为 Dead 状态。

6.3.2 group 管理协议

介绍完 consumer 的状态机, 下面看看 coordinator 的组管理是如何工作的。coordinator 的组管理协议由两个阶段构成, 即组成员加入阶段和状态同步阶段。第一个阶段用于为 group 指定 active 成员并从它们之中选出 leader consumer; 第二个阶段则让 leader consumer 制定分配方案并同步到其他组成员中。从 consumer 的角度来看, 第一个阶段是收集所有 consumer 的 topic 订阅信息, 而第二个阶段则利用这些信息给每个 consumer 分配要消费的分区。每个 group 下的 leader consumer 通常都是第一个加入 group 的 consumer。

下面分别讨论每个阶段的工作原理。

阶段 1: 加入组。

这一阶段的主要目的就是让成员加入 group。所用到的协议请求类型是 JoinGroup。当确定了该 group 对应的 coordinator 之后, 每个成员都要显式地发送 JoinGroup 请求给 coordinator。该请求封装了 consumer 各自的订阅信息、成员 id 等元数据。coordinator 会持有这些 JoinGroup 请求一段时间, 直到所有组成员都发送了 JoinGroup 请求。此时, coordinator 选择其中的一个 consumer 作为 leader, 然后给所有组成员发送对应的 response。

coordinator 拿到所有成员的 JoinGroup 请求后会去获取所有成员都支持的协议类型。如果有成员指定了一个与其他成员都不兼容的协议类型, 则该成员将被拒绝加入组。值得注意的是, 这里的协议类型不是 6.1.6 节中提到的通信协议, 而是 consumer group 端支持的分配策略。举一个例子说明一下, 1.0.0 版本的 Kafka 默认支持 3 种分配策略 (range、round-robin 和 sticky), 如果有一个 consumer 指定了自定义的策略而其他 consumer 都不支持该策略, 那么这个 consumer 就不被允许加入组。

coordinator 处理 JoinGroup 请求后会把所有 consumer 成员的元数据信息封装进一个数组，然后以 JoinGroup response 的方式发给 group 的 leader consumer。切记，它只会给 leader consumer 发送这样的信息，给其他成员只会发送一个空数组。这样做的好处在于：①非 leader consumer 本来也不需要知晓这部分信息；②极大地减少了网络 I/O 传输的开销。

leader consumer 通过 JoinGroup response 知晓了 group 下所有成员的订阅情况，这样它就有足够的信息开始制定分配方案了。

阶段 2：同步组状态信息。

group 所有成员都加入组之后，leader consumer 根据指定的分配策略进行分区的分配。当前最新版本的 Kafka 1.0.0 默认提供了 3 种分配策略，即 range、round-robin 和 sticky。sticky 策略是 Kafka 社区于 0.11.0.0 版本最新推出的，可以最大限度地实现分区负载的均匀分配以及 rebalance 之后最少的分配变动。

在这一阶段中，group 所有成员都需要显式地给 coordinator 发送 SyncGroup 请求。不过只有 leader consumer 的 SyncGroup 请求中会包含它的分配方案。coordinator 接收到 leader 的 SyncGroup 请求后取出分配方案并单独抽取出每个 consumer 对应的分区，然后把分区封装进 SyncGroup 的 response，发送给各个 consumer。这样每个 consumer 都只会得到属于自己的那一部分分区，而不会知晓其他 consumer 的分配方案。

在所有 consumer 成员都收到 SyncGroup response 之后，coordinator 将 group 状态设置为 Stable，此时组开始正常工作，每个成员按照 coordinator 发过来的方案开始消费指定的分区。

6.3.3 rebalance 场景剖析

了解了 coordinator 的组成员管理协议，下面结合 4 个具体的场景来讨论 rebalance 的工作原理。

场景 1：新成员加入组（如图 6.57 所示）。

场景 2：成员发生崩溃。

组成员崩溃和组成员主动离开是两种不同的场景。因为在崩溃时成员并不会主动告知 coordinator 此事，coordinator 有可能需要一个完整的 session.timeout 周期才检测到这一崩溃，这必然会造成 consumer 的滞后。可以说离开组是主动地发起 rebalance，而崩溃则是被动地发起 rebalance，如图 6.58 所示。

场景 3：成员主动离组（如图 6.59 所示）。

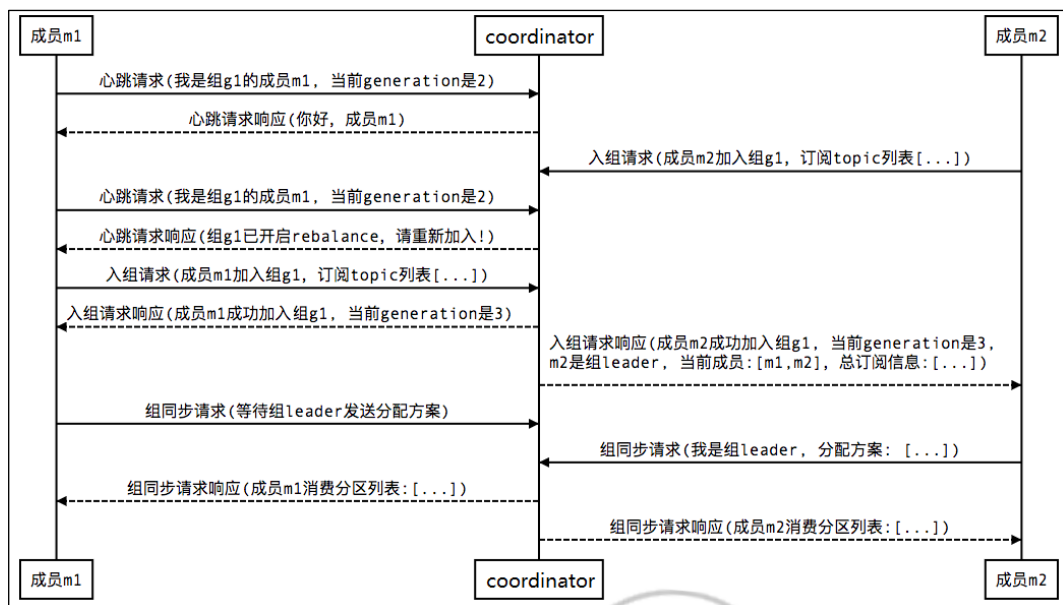


图 6.57 新成员加入组

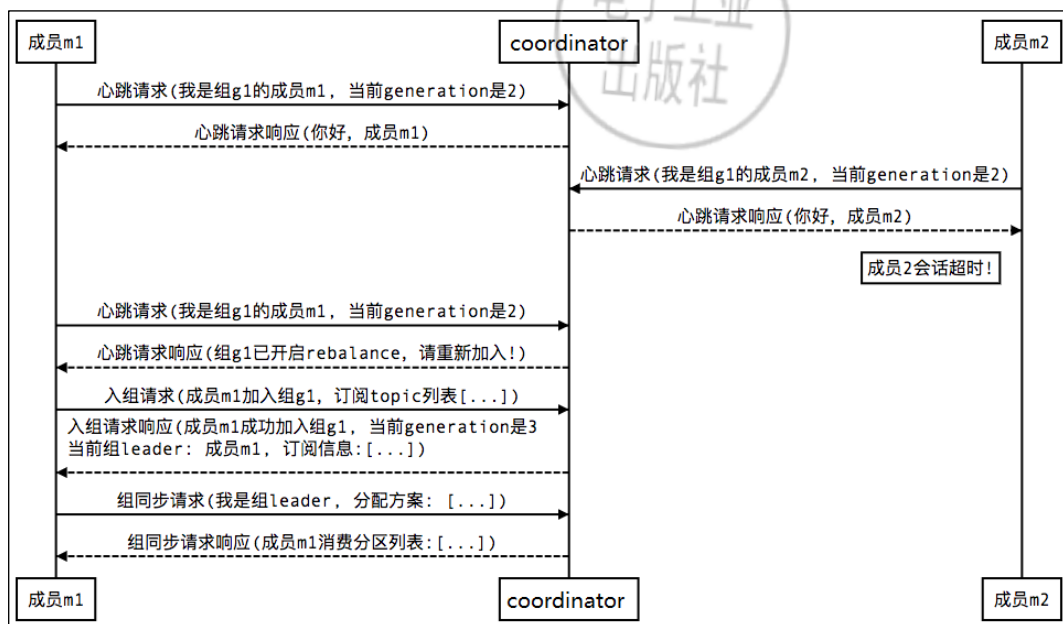


图 6.58 成员发生崩溃

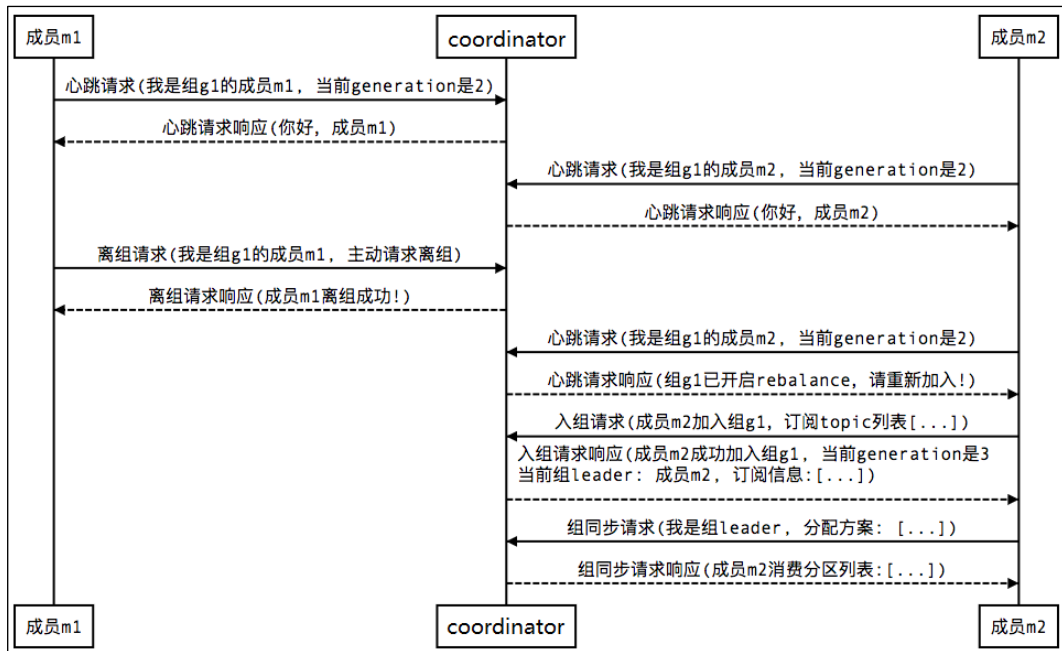


图 6.59 成员主动离组

场景 4：成员提交位移（如图 6.60 所示）。

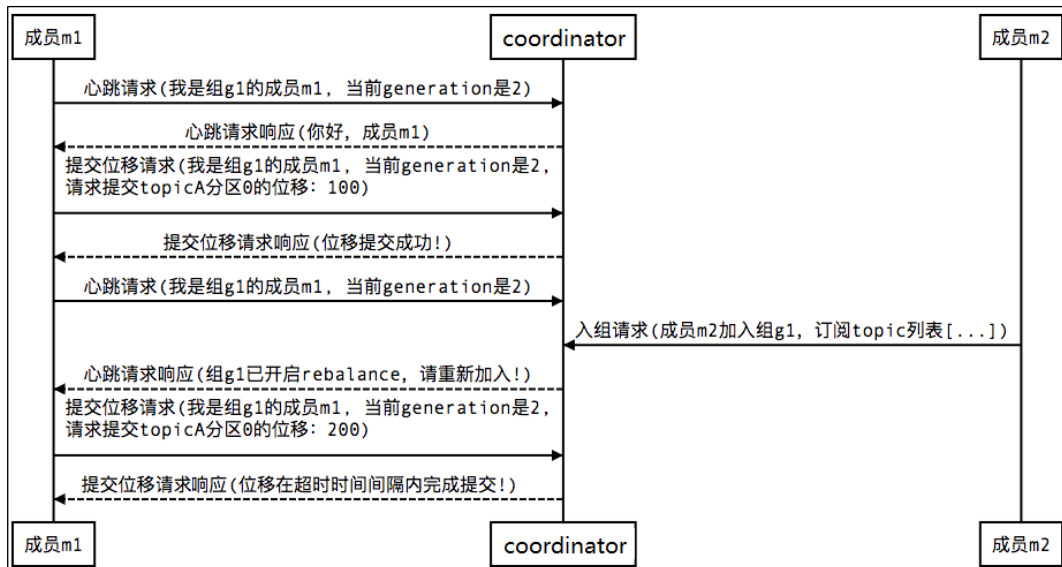


图 6.60 成员提交位移

6.4 实现精确一次处理语义

前面章节中我们反复提到过 Apache Kafka 的消息交付语义（message delivery semantic）。本节我们结合最新的 Kafka 1.0.0 版本引入的各种新功能来说明一下 Kafka 是如何达到精确一次处理语义（exactly-once semantics，简称 EOS）的。

6.4.1 消息交付语义

为了深入理解 Kafka 是如何支持 EOS 的，我们重温一下 clients 端常见的 3 种消息交付语义。它们分别如下。

- 最多一次（at most once）：消息可能丢失也可能被处理，但最多只会被处理一次。
- 至少一次（at least once）：消息不会丢失，但可能被处理多次。
- 精确一次（exactly once）：消息被处理且只会被处理一次。

在开始讨论 Kafka 的消息交付语义之前，我们必须明确这样一个观点：这 3 种消息交付语义没有好坏之分。不是说 at most once 就是不好的，而 exactly once 总是对的。具体使用哪种语义要结合用户实际的业务需求来定，没有必要过分追求精确一次语义。一个典型的例子就是统计网页的 PV（page view）和 UV（user view）。在这种场景下，结果的不完全准确通常并不会影响我们利用它们所做的决策，所以没有必要引入很复杂的数据结构或功能来实现 EOS。

下面分别从 producer 和 consumer 两个角度来分析 Kafka 的消息交付语义。对 producer 而言，Kafka 引入已提交消息（committed message）的概念。一旦消息被成功地提交到日志文件，只要至少存在一个可用的包含该消息的副本，那么这条消息就永远不会丢失。由此可见 Kafka producer 提供的不是 at most once 语义，但它是 at least once 还是 exactly once 呢？

在 0.11.0.0 版本之前，Kafka producer 默认提供的是 at least once 语义。设想这样的一个场景，当 producer 向 broker 发送新消息后，分区 leader 副本所在的 broker 成功地将该消息写入本地磁盘，然后发送响应给 producer。此时假设网络出现故障导致该响应没有发送成功，那么未接到响应的 producer 会认为该消息请求失败从而开启重试操作。若重试后网络恢复正常，那么显然同一条消息被写入到日志两次。在比较极端的情况下，同一条消息可能会被发送多次。具体流程如图 6.61 所示。

因此我们说 Kafka producer 默认提供的就是 at least once 语义。

好消息是 Kafka 0.11.0.0 版本推出了幂等性 producer 和对事务的支持，从而完美地解决了这种消息重复发送的问题。我们会在后面详细探讨具体的设计细节。

对 consumer 端而言，我们知道，相同日志下所有的副本都应该有相同的内容以及相同的

当前位移值。consumer 通过 consumer 位移自行控制和标记日志读取的进度。如果 consumer 程序崩溃，那么替代它的新程序实例就必须接管这个 consumer 位移，即从崩溃时读取位置继续开始消费。若要判断 consumer 到底支持什么交付语义，位移提交的时机就显得至关重要。

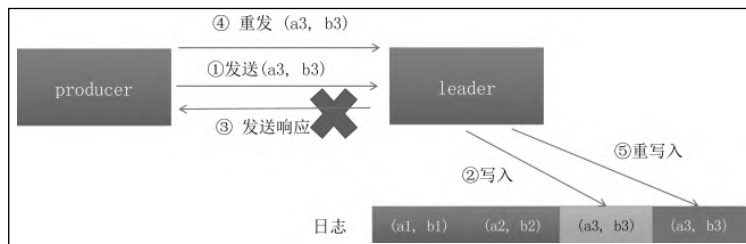


图 6.61 producer 重试导致消息重复发送

一种方式是 consumer 首先获取若干消息，然后提交位移，之后再开始处理消息。这种方法下若 consumer 在提交位移后处理消息前崩溃，那么它实现的就是 at most once 语义，因为消息有可能不被处理，就算处理了最多也只会是一次。

另一种方式是 consumer 获取了若干消息，处理到最后提交位移。显然，consumer 保证只有在消息被处理完成后才提交位移，因此它实现的就是 at least once 语义，因为消息处理过程中如果出现错误从而引发重试，那么某些消息就可能被处理多次。

那么如何实现 consumer 端的 EOS 呢？主要是依赖 0.11.0.0 版本引入的事务。后面我们会详细展开。

6.4.2 幂等性 producer (idempotent producer)

幂等性 producer 是 Apache Kafka 0.11.0.0 版本用于实现 EOS 的第一个利器。若一个操作执行多次的结果与只运行一次的结果是相同的，那么我们称该操作为幂等操作。比如将变量 a 的值设置成 1 这个操作就是幂等操作，无论该操作执行多少次，a 的值始终是 1，与只运行一次该操作的效果是相同的。反之，将当前金额加 100 这样的操作就不是幂等的，因为该操作依赖于之前的金额，即操作有前置状态。

0.11.0.0 版本引入的幂等性 producer 表示它的发送操作是幂等的。瞬时的发送错误可能导致 producer 端出现重试，同一条消息被 producer 发送多次，但在 broker 端这条消息只会被写入日志一次。对于单个 topic 分区而言，这种 producer 提供的幂等性消除了各种错误导致的重复消息。如果要启用幂等性 producer 以及获取其提供的 EOS 语义，用户需要显式地设置 producer 端的新参数 enable.idempotence 为 true。

幂等性 producer 的设计思路类似于 TCP 的工作方式。发送到 broker 端的每批消息都会被

赋予一个序列号（sequence number）用于消息去重。但是和 TCP 不同的是，这个序列号不会被丢弃，相反 Kafka 会把它们保存在底层日志中，这样即使分区的 leader 副本挂掉，新选出来的 leader broker 也能执行消息去重工作。保存序列号只需要额外几字节，因此整体上对 Kafka 消息保存开销的影响并不大。

除了序列号，Kafka 还会为每个 producer 实例分配一个 producer id（下称 PID）。producer 在初始化时必须分配一个 PID。PID 分配的过程对用户来说是完全透明的，因此不会为用户所见。消息要被发送到每个分区都有对应的序列号值，它们总是从 0 开始并且严格单调增加。对于 PID、分区和序列号的关系，用户可以设想一个 Map，key 就是（PID，分区号），value 就是序列号。即每对（PID，分区号）都有对应的序列号值。若发送消息的序列号小于或等于 broker 端保存的序列号，那么 broker 会拒绝这条消息的写入操作。

这种设计确保了即使出现重试操作，每条消息也只會被保存在日志中一次。不过，由于每个新的 producer 实例都会被分配不同的 PID，当前设计只能保证单个 producer 实例的 EOS 语义，而无法实现多个 producer 实例一起提供 EOS 语义。这一点要特别注意。

我们依然使用图 6.61 的例子来说明。引入了序列号和 PID 之后，producer 应付重复发送的流程如图 6.62 所示。

如图 6.62 所示，当 producer 端重发消息时，由于 broker 端已经成功写入了该消息，因此消息所在的 seq（序列号）值不大于 broker 端当前保存的最大 seq 值，因此 broker 将拒绝该 PRODUCE 请求，从而成功实现了消息去重。

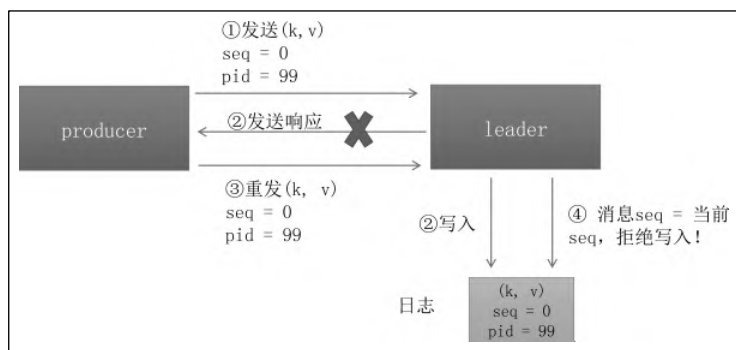


图 6.62 幂等性 producer 消息去重

6.4.3 事务（transaction）

对事务的支持是 Kafka 实现 EOS 的第二个利器。引入事务使得 clients 端程序（无论是 producer 还是 consumer）能够将一组消息放入一个原子性单元中统一处理。

处于事务中的这组消息能够从多个分区中消费，也可以发送到多个分区中去。重要的是不论是发送还是消费，Kafka 都能保证它们是原子性的，即所有的写入操作要么全部成功，要么全部失败。当然对于 consumer 而言，EOS 语义的支持要弱一些，这是由 consumer 本身的特性决定的。也就是说，consumer 有可能以原子性的方式消费这批消息，也有可能是非原子性的。设想 consumer 总是需要 replay 某些消息，如果是这样的使用场景，那么对于 EOS 的支持就要弱很多。

Kafka 为实现事务要求应用程序必须提供一个唯一的 id 来表征事务。这个 id 被称为事务 id，或 TransactionalId，它必须在应用程序所有的会话上是唯一的。值得注意的是，TransactionalId 与上面所说的 PID 是不同的，前者是由用户显式提供的，而后者是 producer 自行分配的。

当提供了 TransactionalId 后，Kafka 就能确保：

- 跨应用程序会话间的幂等发送语义。具体的做法与新版本 consumer 的 generation 概念类似，使用具有版本含义的 generation 来隔离旧事务的操作。
- 支持跨会话间的事务恢复。如果某个 producer 实例挂掉了，Kafka 能够保证下一个实例首先完成之前未完成的事务，从而总是保证状态的一致性。

如果以 consumer 的角度而言，如前所述，事务的支持要弱一些，原因如下。

- 对于 compacted 的 topic 而言，事务中的消息可能已经被删除了。
- 事务可能跨多个日志段（log segment），因此若老的日志段被删除，用户将丢失事务中的部分消息。
- consumer 程序可能使用 seek 方法定位事务中的任意位置，也可能造成部分消息的丢失。
- consumer 可能选择不消费事务中的所有消息，即无法保证读取事务的全部消息。

下面分别讨论一下事务是如何在 producer 和 consumer 端实现的。图 6.63 给出了原子性写入多个分区的流程。

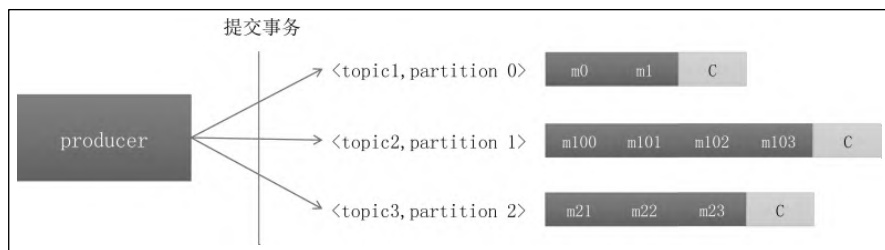


图 6.63 跨分区事务提交

图 6.63 中的 C 是一类特殊的消息，即控制消息（control message）。事务控制消息和普通的 Kafka 消息一样，只是在消息属性字段（attribute field）中专门使用 1 位来表征它是 control

message。当前的 control message 总有两类——COMMIT 和 ABORT，分别表示事务提交和事务终止。将 control message 保存到 Kafka 日志中的目的就是让 consumer 能够识别事务边界，从而整体读取某个事务下的所有消息，如图 6.64 所示。

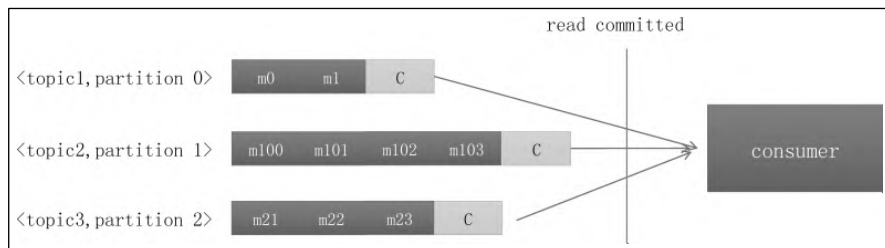


图 6.64 consumer 读取消息

那么，用户如何应用事务 API 呢？下面的代码给出了一个典型的事务 API 使用范例：

```
producer.initTransactions();
try {
    producer.beginTransaction();
    producer.send(record0);
    producer.send(record1);
    producer.sendOffsetToTxn(...);
    producer.sendOffsetsToTransaction();
} catch (ProducerFencedException e) {
    producer.close();
} catch (KafkaException e) {
    producer.abortTransaction();
}
```

就 producer 而言，开启事务的第一步就是初始化事务状态 `initTransactions()`，然后调用 `beginTransaction` 正式开始事务，上面的代码中的 `sendOffsetsToTransaction` 方法适用于事务中同时包含消息生产和消息消费的场景。当调用该方法时，Kafka 会确保只有事务被提交成功才消费给定的位移。

6.5 本章小结

本章我们详细探讨了 Kafka 的内部设计原理，包括 broker 端的设计架构、producer 设计、consumer 设计以及新版本对 EOS 的支持。

相信各位读者对于 Kafka 内部运行机制有了一定的了解。第 7 章我们将学习如何实际操作和管理 Kafka 集群以及管理集群的一些常见命令与技巧。

第 7 章

管理 Kafka 集群

本章重点讲解 Apache Kafka 集群的运维管理、参数配置、工具脚本和常见问题的排查等内容，将基本涵盖 Kafka 运维相关的各个方面，从实践角度诠释主要脚本工具的使用方法。本章内容也为第 8 章“监控 Kafka 集群”内容的展开奠定了基础。

之前所有章节的内容都是围绕 Kafka 的功能主线展开的，主要从一般用户的角度出发，本章则以管理员的视角阐述 Kafka 集群的管理。

学习本章，你将了解到以下内容。

- 集群与 topic 管理。
- 消费者管理。
- 常用脚本工具。
- 集群管理 API。
- MirrorMaker 的使用。
- Kafka 安全。
- 常见问题。

7.1 集群管理

7.1.1 启动 broker

启动 Kafka 服务器（下称 broker）需要首先启动 ZooKeeper 服务器，后者的配置和启动方法参见第 3 章 3.2.2 节，这里不再赘述。至于启动 Kafka 的 broker，本节要强调一些在第 3 章中

未曾提及的关键要点：首先，在生产环境中强烈推荐使用 `-daemon` 参数启动服务器，如下面的代码所示：

```
bin/kafka-server-start.sh -daemon <path>/server.properties
```

笔者碰到过很多用户抱怨他们的 **Kafka** 集群会定期自动关闭但却无法找到原因，从日志上看 **broker** 是正常关闭而非异常崩溃。经询问才知道他们的启动命令通常都是这样的：

```
bin/kafka-server-start.sh <path>/server.properties &
```

虽然通过添加 `&` 符号使该命令在后台运行，但当用户从会话登出（`log out`）时该进程会被自动 `kill` 掉，从而导致 **broker** 关闭。一个实际的案例就是，有用户抱怨他们的 **Kafka** 集群每天 7~8 点左右自动关闭，最后发现原因是该用户使用上面的命令启动 **Kafka**，而他每天都 7~8 点下班关电脑回家，会话登出从而引发 **broker** 进程关闭。

因此在生产环境中，笔者建议读者使用 `nohup ... &` 或加 `-daemon` 参数的方式启动 **Kafka** 集群，即：

```
nohup bin/kafka-server-start.sh <path>/server.properties &
```

如果查阅 **Kafka** 启动脚本源码，读者会发现 `-daemon` 的作用就是以 `nohup...&` 的方式启动 **broker**，因此两者的效果是一样的。

启动 **broker** 之后，笔者建议用户查看启动日志以确保 **broker** 启动成功。服务器日志通常保存在 **kafka** 安装目录的 `logs` 子目录下，名字是 `server.log`。用户可查询该日志文件，如果发现诸如 “[**Kafka Server ****], started (kafka.server.KafkaServer)” 之类的输出项，则表示 **broker** 进程启动成功。

7.1.2 关闭 broker

有些令人惊讶的是，比起 **broker** 的启动，用户对于如何正确关闭 **Kafka broker** 有着更多的疑问。笔者认为这主要是因为社区的文档并没有给出很好的示范所致。官方文档的 `Get Started` 部分并未给出如何关闭 **broker** 的方法。

正确关闭 **broker** 的方法在笔者看来分以下两种情况讨论。

1. 前台方式启动 broker 进程时

所谓的前台（`foreground`）启动 **broker** 是指，在 **Linux** 终端（`terminal`）不加 `nohup` 或 `-daemon` 参数的方式直接启动 **broker**。这种启动方式多用于本地测试或 `DEBUG`。关闭以这种方式启动的 **broker** 进程很简单，只需要在该终端上按下 `Ctrl + C` 组合键发出 `SIGINT` 信号终止进程即可。**Kafka broker** 进程会捕捉 `SIGINT` 信号并执行 **broker** 关闭逻辑。

2. 后台方式启动 broker 进程时

当使用前面的 `-daemon` 或 `nohup` 方式启动 broker 后，正确关闭 broker 的方式就是使用 Kafka 自带的 `kafka-server-stop.sh` 脚本。该脚本位于 Kafka 安装目录的 `bin` 子目录下。如果是 Windows 平台，该脚本位于 Kafka 安装目录的 `bin/windows` 下。此脚本使用方法很简单，直接运行以下命令即可：

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

一定要注意，该脚本会搜寻当前机器中所有的 Kafka broker 进程，然后关闭它们。这就意味着，如果用户的机器上存在多个 broker 进程，该脚本会全部关闭它们。这个脚本当前并不支持以给定参数的方式单独关闭某个 broker。

另外笔者并不推荐用户直接使用 `kill -9` 方式“杀死”broker 进程，因为这通常会造成一些 broker 状态的不一致。当前 broker 在关闭过程中如果出现错误会进行重试。若重试次数用完了依然无法正常关闭 broker 进程，Kafka 会进行强制关闭。让 Kafka 自行强制结束要比用户使用 `kill -9` 来得“温柔”，因此当用户发现关闭 broker 时有似乎“卡住”的情况，请耐心等待，并打开 `server.log` 实时监控关闭进度。通常一段时间之后，broker 进程总会成功结束。事实上，现在 broker 无法关闭的情况已经非常罕见了。

不过凡事不可能尽善尽美。社区中也有一些用户提交 bug 报告，宣称 `kafka-server-stop` 脚本无法关闭 broker——主要的原因在于该脚本是依靠操作系统命令（如 `ps ax` 和 `grep`）来获取 Kafka broker 的进程号（PID）的，一旦用户的 Kafka 安装目录的路径过长，则有可能令该脚本失效从而无法正确获取 broker 的 PID。如若用户碰到这种情况，可以采用下面的办法来关闭 broker。

- （1）如果机器上安装了 JDK，可运行 `jps` 命令直接获取 Kafka 的 PID，否则转到下一步。
- （2）运行 `ps ax | grep -i 'kafka\Kafka' | grep java | grep -v grep | awk '{print $1}'` 命令自行寻找 Kafka 的 PID。
- （3）运行 `kill -s TERM $PID` 关闭 broker。

实际上，以上 3 步就是 `kafka-server-stop` 脚本执行的逻辑。

7.1.3 设置 JMX 端口

Kafka 提供了丰富的 JMX 指标用于实时监控集群运行的健康程度。我们会在第 8 章中详细介绍如何监控 Kafka 集群以及如何利用这些指标。不过若要使用它们，用户必须在启动 broker 前就首先设置 JMX 端口。

我们依然分两种情况讨论如何设置 JMX 端口。首先依然是以前台方式运行 broker 的场景。如果是前台运行，那么只需要在执行启动命令前设置 JMX_PORT 环境变量，如下面的代码设置 JMX 端口为 9997：

```
JMX_PORT=9997 bin/kafka-server-start.sh <path>/server.properties
```

以后台方式运行 broker 的设置方法是类似的，依然在启动 broker 前设置 JMX_PORT 环境变量：

```
export JMX_PORT=9997 bin/kafka-server-start.sh -daemon <path>/  
server.properties
```

7.1.4 增加 broker

由于 Kafka 集群的服务发现交由 ZooKeeper 来处理，因此向 Kafka 集群增加新的 broker 服务器非常容易。用户只需要为新增 broker 设置一个唯一的 broker.id，然后启动即可。Kafka 集群能自动地发现新启动的 broker 并同步所有的元数据信息，主要包括当前集群有哪些主题（下称 topic）以及 topic 都有哪些分区等。

唯一有缺憾的是，新增的 broker 不会自动被分配任何已有的 topic 分区，因此用户必须手动执行分区重分配（本章后续部分会详细讨论）操作才能使它们为已有 topic 服务。当然，在这些 broker 启动之后新创建的 topic 分区还是可能分配给这些 broker 的。用户需要仔细地对各个 broker 上的负载进行规划，避免出现负载极度不均匀的情况。

7.1.5 升级 broker 版本

随着 Apache Kafka 社区不断推出新的版本，用户需要不断地升级已有生产环境中的 Kafka 版本才能享受新版本带来的各种新功能改进和性能提升。升级 Kafka 集群的版本实际上非常简单，核心步骤只需要 4 步，但通常情况下用户都想要最大程度缩短服务的宕机时间（downtime），特别是最好不要干扰集群上 producer 和 consumer 的正常运转。

本节笔者将结合一个具体的实例来模拟 Kafka 集群的版本升级过程，同时还确保不影响 clients 端程序的运行。

首先简要介绍一下本例中的源 Kafka 集群是一个双 broker 集群环境，版本是 0.10.0.0，目标是把集群升级到 0.10.2.0 版本。实际上本例中使用的方法适用于大部分的版本升级。

图 7.1 给出了测试环境下的目录文件构成。

在图 7.1 中，两个 broker 的 data.dirs 分别设置为 data/broker1 和 data/broker2，读取的配置文件分别是 configs/server.properties 和 configs/server2.properties。另外笔者创建了两个启动脚本

startBroker1.sh 和 startBroker2.sh, 它们的内容基本相同, 如下:

```
CURRENT_PATH=<your_path>/kafka_2.11-0.10.0.0
cd $CURRENT_PATH
JMX_PORT=9997 (或 9998) bin/kafka-server-start.
sh ../configs/server.properties
```

启动 broker 之后, 我们创建一个测试 topic, 名为 test, 双分区且副本因子是 2, 然后使用 kafka-topics.sh 来查询测试 topic 详情, 如图 7.2 所示。

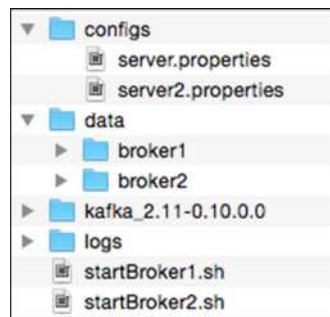


图 7.1 测试环境下的目录文件构成

Topic:test	PartitionCount:2	ReplicationFactor:2	Configs:
Topic: test	Partition: 0	Leader: 0	Replicas: 0,1 Isr: 0,1
Topic: test	Partition: 1	Leader: 1	Replicas: 1,0 Isr: 1,0

图 7.2 测试 topic 详情

由图 7.2 可见, 测试 topic 一切正常。下面我们分别启动一个生产者（下称 producer）和一个消费者（下称 consumer）来模拟真实的线上环境。下面的代码是一个很简单的 producer 程序, 每 1 秒发送一条 Kafka 消息, 然后打印提交成功的消息数和提交失败的消息数。这里, 我们依赖提交失败的消息数来确保升级流程不会影响 producer。

```
Properties props = new Properties();
    props.put("bootstrap.servers", "localhost:9092,localhost:9093");
    props.put("acks", "all");
    props.put("retries", Integer.MAX_VALUE);
    props.put("batch.size", 16384);
    props.put("linger.ms", 1);
    props.put("buffer.memory", 33554432);
    props.put("key.serializer", "org.apache.kafka.common.serialization.
StringSerializer");
    props.put("value.serializer", "org.apache.kafka.common.serialization.
StringSerializer");

    Producer<String, String> producer = new KafkaProducer<>(props);
    final AtomicInteger success = new AtomicInteger(0);
    final AtomicInteger failed = new AtomicInteger(0);
    try {
        while (true) {
            producer.send(new ProducerRecord<String, String>("test",
"a message"), new Callback() {
                @Override
                public void onCompletion(RecordMetadata metadata,
```

```
Exception exception) {
    if (exception != null) {
        System.out.println("Current failed count: "
+ failed.incrementAndGet());
    } else
        System.out.println("Current success count:
" + success.incrementAndGet() + ", failed: " + failed.get());
    }
});
Thread.sleep(2000);

}
} finally {
    producer.close();
}
```

现在启动 **consumer**。为简单起见，本例使用 **console-consumer** 脚本。需要注意的是，由于升级前的版本是 0.10.0.0，所以运行脚本时需要显式加上 **--new-consumer** 表明我们使用的是新版本 **consumer**（注意：这个版本必须加上 **--new-consumer** 才能启用新版本 **consumer**），如下：

```
bin/kafka-console-consumer.sh --topic test --from-beginning --new-consumer
--bootstrap-server localhost:9092,localhost:9093
```

此时，应该能够看到 **consumer** 和 **producer** 都可以正常工作了。现在开始升级 **Kafka broker** 版本。在执行完升级流程中的每一步后，我们都要确保 **producer** 和 **consumer** 程序依然在正常执行且没有出现错误。

第一步：更新 **broker** 间通信版本和消息版本。

这一步需要向所有 **broker** 的 **server.properties** 中增加下面两行：

```
inter.broker.protocol.version=0.10.0
log.message.format.version=0.10.0
```

如果用户不是从 0.10.0.0 版本升级，那么只需要填写用户环境下当前 **broker** 的版本。

第二步：依次更新代码，重启所有 **broker**。

下载 0.10.2.0 版本的 **Kafka** 二进制包，覆盖已有的目录，然后依次重启所有 **broker**，比如首先重启 **broker1**，然后重启 **broker2**。用户也可以选择不覆盖安装目录，而是单独放置在全新的目录下。只要确保使用目标版本的二进制程序即可。

第三步：再次更新 **broker** 间通信版本和消息版本。

由于本例要升级到 0.10.2.0 版本，故这里写 0.10.2。若要升级到其他版本，则用户需要自行调整目标版本号。比如要升级到 0.10.1.0 版本，则填写 0.10.1 即可。

```
inter.broker.protocol.version=0.10.2
log.message.format.version=0.10.2
```

第四步：再次依次重启 broker。

这一步依然是轮流重启 broker。

至此，Kafka 集群的升级工作就完成了。在上面的每一步中，producer 都应该是可以正常工作的，即提交失败的消息数应该始终是 0；而 consumer 可能会出现位移提交失败的警告从而造成消息的重复消费——而这一点通常由下游处理子系统来负责去重，因此也不是太大的问题。至于 broker 端，则会出现短暂的 `org.apache.kafka.common.errors.NotLeaderForPartitionException` 异常，表明内部 topic `__consumer_offsets` 各分区的 leader 在轮流重启过程中出现短暂的不可用。鉴于这个异常是瞬时发生且通常可以自行恢复的，因此也不必太在意。

通过以上的实例，我们几乎实现了零宕机时间下的 Kafka 集群版本升级。当然在实际线上环境中需要考虑的因素还有很多，远远不止 clients 端的影响这一项。不过总体而言，升级 Kafka 版本的流程只需要这 4 步就能完成。

7.2 topic 管理

7.2.1 创建 topic

严格来说，Kafka 创建 topic 的途径当前总共有如下 4 种。

- 通过执行 `kafka-topics.sh(bat)` 命令行工具创建。
- 通过显式发送 `CreateTopicsRequest` 请求创建 topic。
- 通过发送 `MetadataRequest` 请求且 broker 端设置了 `auto.create.topics.enable` 为 `true`。
- 通过向 ZooKeeper 的 `/brokers/topics` 路径下写入以 topic 名称命名的子节点。

当然，除了上面 4 种方法还有其他一些比较“偏门”的创建方式，比如写 Java/Scala 程序直接调用 `TopicCommand` 的 `createTopic` 方法或调用 `AdminUtils` 的 `createTopic` 方法，但鉴于这些方法并没有显式地记录在官方文档上，笔者并不推荐用户使用这些非寻常的途径。

另外，上面的第 4 个方法实际上也是不推荐的。官方社区推荐用户使用前两种方式来创建 topic。本节将主要展示如何使用 `kafka-topics` 脚本，至于 `CreateTopicsRequest` 请求发送的方法将留到 7.7 节。

如果不加任何参数执行 `kafka-topics` 脚本，用户将得到该脚本提供的所有功能帮助信息，如表 7.1 所示。

表 7.1 kafka-topics 脚本命令行参数列表

参 数 名	参 数 含 义
--alter	用于修改 topic 的信息，比如分区数、副本因子
--config <key=value>	设置 topic 级别的参数，比如 cleanup.policy 等
--create	创建 topic
--delete	删除 topic
--delete-config <name>	删除 topic 级别的参数
--describe	列出 topic 详情
--disable-rack-aware	创建 topic 时不考虑机架信息
--force	无效参数，当前未使用
--help	打印帮助信息
--if-exists	若设置，脚本只会对已存在的 topic 执行操作
--if-not-exists	若设置，当创建已存在的同名 topic 时不会抛出错误
--list	列出集群当前所有 topic
--partitions <分区数>	创建或修改 topic 时指定分区数
--replica-assignment	手动指定分区分配 CSV 方案，副本之间使用冒号分割。比如指定双分区方案为 0:1:2,3:4:5，表示分区 1 的 3 个副本在 broker 0、1、2 上，分区 2 在 broker 3、4、5 上
--replication-factor	指定副本因子
--topic	指定 topic 名称
--topics-with-overrides	展示 topic 详情时不显示具体的分区信息
--unavailable-partitions	只显示 topic 不可用的分区（即没有 leader 的分区）
--under-replicated-partitions	只显示副本数不足的分区信息
--zookeeper	（必填项）指定连接的 ZooKeeper 信息

下面我们来演示两个例子，一个使用自动分区分配来创建 topic，另一个使用手动分区分配来创建 topic。kafka-topics 脚本以及后续提到的所有脚本都位于 Kafka 目录的 bin 子目录下，如果是运行 Windows 平台版本的脚本，则位于 Kafka 目录的 bin/windows/子目录下。

首先我们来创建一个 topic，名为 test-topic，6 个分区，每个分区 3 个副本，同时指定该 topic 的日志留存时间是 3 天，命令如下：

```
> bin/kafka-topics.sh --create --zookeeper localhost:2181 --partitions 6 --
replication-factor 3 --topic test-topic --config delete.retention.ms=259200000
Created topic "test-topic".
```

看到 Created topic "test-topic"的字样通常表明 topic 创建成功。下面我们来演示如何手动指定分区在集群上的分配。假设我们有一个 3 台 broker 构成的 Kafka 集群，broker id 分别是 0、1、2。现在我们来创建一个 topic，名为 test-topic2，分区数是 4，副本因子是 2。我们手动分配方案如下。

分区 1: 0、1

分区 2: 1、2

分区 3: 0、2

分区 4: 1、2

那么 topic 创建命令如下:

```
> bin/kafka-topics.sh --create --zookeeper localhost:2181 --topic test-  
topic2 --replica-assignment 0:1,1:2,0:2,1:2  
Created topic "test-topic2".
```

值得注意的是, 若指定 `--replica-assignment`, 用户不必再指定 `--partitions` 和 `--replication-factor`, 因为脚本可以从手动分配方案中计算出 topic 的分区数和副本因子。

7.2.2 删除 topic

与创建 topic 类似, 删除 topic 当前有如下 3 种方式。

- 使用 `kafka-topics` 脚本: 这是最常见也是最“地道”的方式。
- 构造 `DeleteTopicsRequest` 请求: 这种方式需要手动编写程序实现, 7.7 节将详细讨论它的实现。
- 直接向 ZooKeeper 的 `/admin/delete_topics` 下写入子节点: 不推荐的方式。在实际场景中谨慎使用。

经常有一些用户采用“暴力删除法”来移除 topic 数据, 即手动删除 topic 底层的日志文件以及 ZooKeeper 中与 topic 相关的所有 `znode`。事实上, 笔者并不推荐这种删除方式, 除非上面的 3 种方式全部失效。毕竟这种方式只能保证物理文件被删除干净, 但在集群 broker 机器的内存中(特别是 controller 所在的 broker 上)依然保存有已删除 topic 的信息。

因此最“正宗”的删除方式依然是执行 `kafka-topics.sh(bat) --delete` 命令。该命令执行之后通常是立即返回, 因此往往给用户一个错觉, 仿佛 topic 已经被删除。但其实它只是在后台开启一个异步删除的任务, 所以用户需要多等待一段时间才能观察到 topic 数据被完全删除。

无论用户采用上面的哪种方法, 一定要首先确保 broker 端参数 `delete.topic.enable` 被设置为 `true`, 否则 Kafka 是不会删除 topic 的。截止到最新的 1.0.0 版本之前, 该参数的默认值依然是 `false`, 表明默认情况下 Kafka 集群是不允许删除 topic 的。因此如果使用 1.0.0 之前的版本, 一定不要忘记将其设置为 `true`。社区于 1.0.0 版本将该参数默认值调整为 `true`, 表明了 topic 删除功能已经相对完善, 读者应该始终坚持使用 `kafka-topics` 命令来删除 topic。

下面我们以“创建 topic”一节中的测试 topic 为例演示如何删除 test-topic，命令如下：

```
> bin/kafka-topics.sh --delete --zookeeper localhost:2181 --topic test-
topic
Topic test-topic is marked for deletion.
Note: This will have no impact if delete.topic.enable is not set to true.
```

由上面的输出可知，该命令执行返回之后 test-topic 仅仅被标记为“待删除”状态，而且它还会“好心”地提醒用户不要忘记设置 delete.topic.enable=true，否则该命令不会有任何效果。

看到这个输出，就表明 topic 删除任务开始了，但通常都未结束。令人遗憾的是，当前 Kafka 删除 topic 的逻辑是由 controller 在后台默默地完成的，用户无法感知进度也不能知晓删除是否成功。唯一的确认方法就是使用 kafka-topics.sh(bat) -list 命令去查询 topic 列表。若被删除 topic 不在列表中则表明删除成功。

7.2.3 查询 topic 列表

Kafka 自带的 kafka-topics 脚本运行用户查询当前集群的 topic 列表，命令如下：

```
> bin/kafka-topics.sh --zookeeper localhost:2181 --list
test
test-topic2
```

输出比较简单，就是 topic 名称的列表。上面的输出表明当前 Kafka 集群总共有两个 topic，即 test 和 test-topic2。

7.2.4 查询 topic 详情

除了查询 topic 列表，kafka-topics 脚本还可以查询单个或所有 topic 的详情，比如若要查询 test-topic2 的详情，则可以执行下面的命令：

```
> bin/kafka-topics.sh --zookeeper localhost:2181 --describe --topic test-
topic2
Topic:test-topic2 PartitionCount:4 ReplicationFactor:2 Configs:
Topic: test-topic2 Partition: 0 Leader: 0 Replicas: 0,1 Isr: 0,1
Topic: test-topic2 Partition: 1 Leader: 1 Replicas: 1,2 Isr: 1,2
Topic: test-topic2 Partition: 2 Leader: 0 Replicas: 0,2 Isr: 0,2
Topic: test-topic2 Partition: 3 Leader: 1 Replicas: 1,2 Isr: 1,2
```

输出结果表明 test-topic2 有 4 个分区，副本因子是 2，以及每个分区当前的分配情况。如果用户没有指定命令结尾处的--topic test-topic2，则表明查询集群上所有 topic 的详情，如下：

```
> bin/kafka-topics.sh --zookeeper localhost:2181 --describe
Topic:test PartitionCount:1 ReplicationFactor:1 Configs:cleanup.policy
= compact
```

```

Topic: test Partition: 0 Leader: 0 Replicas: 0 Isr: 0
Topic: test-topic2 PartitionCount:4 ReplicationFactor:2 Configs:
Topic: test-topic2 Partition: 0 Leader: 0 Replicas: 0,1 Isr: 0,1
Topic: test-topic2 Partition: 1 Leader: 1 Replicas: 1,2 Isr: 1,2
Topic: test-topic2 Partition: 2 Leader: 0 Replicas: 0,2 Isr: 0,2
Topic: test-topic2 Partition: 3 Leader: 1 Replicas: 1,2 Isr: 1,2

```

可见，该命令把集群上所有的 topic 详情都展示了出来。

7.2.5 修改 topic

topic 被创建后，Kafka 依然允许用户对 topic 的某些参数和设置进行修改，比如分区数、副本因子和 topic 级别参数等。本节演示如何使用自带的命令来修改 topic。

首先，我们来展示如何为 topic 增加分区。以上面的 test-topic2 为例，当前的分区数是 4，若要增加到 10，则可以运行下面的命令：

```

> bin/kafka-topics.sh --alter --zookeeper localhost:2181 --partitions 10
--topic test-topic2
WARNING: If partitions are increased for a topic that has a key, the
partition logic or ordering of the messages will be affected
Adding partitions succeeded!

```

注意上面输出结果中的警告信息：当 topic 的消息有 key 时，根据 key 确定目标分区的算法就会因分区数的增加而发生变更。这对有些用户而言可能是不可忍受的，因此在增加分区前一定要考虑清楚。

另外一定要牢记的是，当前 Kafka 不支持减少分区数。有兴趣的读者可以指定一个小的分区数（小于 10 即可）再次运行上面的命令，Kafka 会抛出异常提示 The number of partitions for a topic can only be increased。

除了分区数，用户可以使用 kafka-configs 脚本为已有 topic 设置 topic 级别的参数。实际上，kafka-topics.sh(bat) --alter 虽然也可以实现这样的目的，但 Kafka 社区已经不推荐用户使用 --alter 来设置，读者应该始终使用 kafka-configs 脚本来设置它们。我们首先来看看 kafka-configs 脚本命令行参数及其含义，如表 7.2 所示。

表 7.2 kafka-configs 脚本命令行参数及其含义

参 数 名	参 数 含 义
--add-config k1=v1,k2=v2,...	设置 topic 级别参数
--alter	修改 topic 或其他实体（entity）类型
--delete-config	删除指定的参数
--entity-default	主要与配额（quota）设置有关，用于 Kafka 配置的默认配额值

续表

参 数 名	参 数 含 义
--describe	列出给定实体类型的参数详情
--entity-name	指定实体名称，若是 topic 类型，则是 topic 名称
--entity-type	指定实体类型，总共有 4 类实体类型：user、topic、clients、broker
--help	打印帮助信息
--zookeeper	（必填项）指定连接的 ZooKeeper 信息

在给出 kafka-configs 脚本运行实例之前，我们必须说明一下该脚本功能涵盖的范围。当前 Kafka 定义了 4 类实体（entity），它们分别是 topic、user、clients 和 broker。每类实体都能设置专属的配置参数。kafka-configs 脚本就是为这 4 类实体进行设置的工具。不过本节我们只关注与 topic 相关的功能部分。

首先，我们来看看如何为已有 topic 增加一个 topic 级别的参数。假设我们要为上面的 test-topic2 设置 cleanup.policy=compact，命令如下：

```

> bin/kafka-configs.sh --zookeeper localhost:2181 --alter --entity-type
topics --entity-name test-topic2 --add-config cleanup.policy=compact
Completed Updating config for entity: topic 'test-topic2'.
    
```

```

然后使用 kafka-configs 的--describe 选项来确认参数是否增加成功：

> bin/kafka-configs.sh --zookeeper localhost:2181 --describe --entity-
type topics --entity-name test-topic2
Configs for topic 'test-topic2' are cleanup.policy=compact
    
```

7.3 topic 动态配置管理

7.3.1 增加 topic 配置

截止到最新的 1.0.0 版本，Kafka 支持的 topic 级别的参数共有 24 个，完整的参数列表及其含义请参见官网 <https://kafka.apache.org/documentation/#topic-config> 中的说明。表 7.3 给出了几个常见且重要的 topic 级别参数及其含义。

表 7.3 常见且重要的 topic 级别参数及其含义

参 数 名	参 数 含 义
cleanup.policy	指定 topic 的留存策略，可以是 compact、delete 或同时指定两者
compression.type	指定该 topic 消息的压缩类型
max.message.bytes	指定 broker 端能够接收该 topic 消息的最大长度
min.insync.replicas	指定 ISR 中需要接收 topic 消息的最少 broker 数，与 producer 端参数 acks=-1 配合使用

续表

参 数 名	参 数 含 义
preallocate	是否为该 topic 的日志文件提前分配存储空间
retention.ms	指定持有该 topic 单个分区消息的最长时间
segment.bytes	指定该 topic 日志段文件的大小
unclean.leader.election.enable	是否为 topic 启用 unclean 领导者选举

与 broker 端参数不同的是, 以上这些 topic 级别的参数可以动态修改而无须重启 broker。下面我们来演示如何为一个 topic 动态地增加若干个参数设置。我们首先创建一个测试单分区且副本因子是 1 的测试 topic, 然后为其动态地增加 preallocate 和 segment.bytes 两个参数设置, 如下面的代码所示:

```
> bin/kafka-topics.sh --create --zookeeper localhost:2181 --partitions 1
--replication-factor 1 --topic test
Created topic "test".
> bin/kafka-configs.sh --zookeeper localhost:2181 --alter --entity-type
topics --entity-name test --add-config preallocate = true,segment.bytes
= 104857600
Completed Updating config for entity: topic 'test'.
> bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic test
Topic:test PartitionCount:1ReplicationFactor:1
Configs:segment.bytes=104857600,preallocate=true
Topic: test Partition: 0Leader: 0 Replicas: 0 Isr: 0
```

最后一条命令显示 preallocate 和 segment.bytes 都被动态地设置了。

7.3.2 查看 topic 配置

当前有两种方式可以查看 topic 配置, 一种是使用 kafka-topics 脚本, 另一种是使用 kafka-configs 脚本。我们先使用 kafka-topics 脚本来查看 topic 配置, 如下面的代码所示:

```
> bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic test
Topic:test PartitionCount:1ReplicationFactor:1
Configs:segment.bytes=104857600,preallocate=true
Topic: test Partition: 0Leader: 0 Replicas: 0 Isr: 0
```

接下来使用 kafka-configs 脚本来查看:

```
> bin/kafka-configs.sh --describe --zookeeper localhost:2181 --entity-
type topics --entity-name test
Configs for topic 'test' are segment.bytes=104857600,preallocate=true
```

7.3.3 删除 topic 配置

我们依然使用 `kafka-configs` 脚本删除 `topic` 配置。以上面的测试 `topic` 为例，删除 `preallocate` 配置：

```
> bin/kafka-configs.sh --zookeeper localhost:2181 --alter --entity-type
topics --entity-name test --delete-config preallocate
Completed Updating config for entity: topic 'test'.
> bin/kafka-configs.sh --describe --zookeeper localhost:2181 --entity-
type topics --entity-name test
Configs for topic 'test' are segment.bytes=104857600
```

可见，`preallocate` 配置已经被删除了。

7.4 consumer 相关管理

7.4.1 查询消费者组

生产环境对于消费者组（`consumer group`）的运行情况有着很强的监控需求。大多数用户选择使用第三方的监控框架（第 8 章详细展开）来监控消费者组。实际上 `Kafka` 默认自带了一些工具脚本来监控并管理消费者组的执行情况。下面首先来看看如何查询消费者组。

`Kafka` 提供的用于查询消费者组的脚本是 `kafka-consumer-groups.sh(bat)`，如果 `Kafka` 环境中没有发现这个脚本，则说明你使用的是比较老的 `Kafka` 版本，请尽快升级到最新版本。在后面我们也会简要介绍老版本查询消费者组的方法。

`kafka-consumer-groups.sh` 位于 `Kafka` 路径的 `bin/`子目录下，而 `Windows` 平台的 `bat` 脚本位于 `Kafka` 路径的 `/bin/windows/`子目录下。该脚本主要的参数及其含义如表 7.4 所示。

表 7.4 `kafka-consumer-groups` 脚本主要的参数及其含义

参 数 名	参 数 含 义
<code>--bootstrap-server</code>	指定 <code>Kafka</code> 集群的 <code>broker</code> 列表， <code>CSV</code> 格式，查询新版本消费者组时使用
<code>--list</code>	列出集群当前所有消费者组
<code>--describe</code>	查询消费者组详情（包括消费滞后情况等）
<code>--group</code>	指定消费者组名称
<code>--zookeeper</code>	指定 <code>ZooKeeper</code> 连接信息，查询老版本消费者组时使用（已不推荐使用）
<code>--reset-offsets</code>	重新设置消费者组位移

特别注意表 7.4 中 `--bootstrap-server` 和 `--zookeeper` 的区别，两者不可同时指定。前面章节中我们提到过 `Kafka` 目前提供了新旧两个版本的 `consumer`（事实上，截止到目前最新的 `Kafka`

1.0.0 版本，老版本 consumer 依然只是被标记为 deprecated，而没有被移除）。如果用户使用了新版本 consumer（即 Java consumer），那么就需要指定--bootstrap-server 来查询和管理消费者组，反之则需要指定--zookeeper。

下面给出一个实例来演示该命令的使用。首先在一个 Kafka 单节点测试环境中创建一个测试 topic，名为 test，单分区，副本因子是 1。之后使用自带的 kafka-producer-perf-test.sh 脚本（该脚本主要用于为 producer 应用做性能测试，在这里我们仅仅利用它为我们生产测试消息）为该 topic 生产 500 万条消息，命令如下：

```
> cd /usr/huxi/kafka-0.11/
> bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic test
Created topic "test".
> bin/kafka-producer-perf-test.sh --topic test --throughput -1 --num-records 5000000 --record-size 100 --producer-props bootstrap.servers=localhost:9092 acks=-1
587880 records sent, 117576.0 records/sec (11.21 MB/sec), 1194.1 ms avg latency, 1854.0 max latency.
916268 records sent, 183180.3 records/sec (17.47 MB/sec), 1649.9 ms avg latency, 1805.0 max latency.
983459 records sent, 196456.1 records/sec (18.74 MB/sec), 1612.9 ms avg latency, 1891.0 max latency.
1119028 records sent, 223805.6 records/sec (21.34 MB/sec), 1343.1 ms avg latency, 1492.0 max latency.
5000000 records sent, 208272.587162 records/sec (19.86 MB/sec), 1326.70 ms avg latency, 1891.00 ms max latency, 1353 ms 50th, 1792 ms 95th, 1844 ms 99th, 1871 ms 99.9th.
```

然后，我们分别打开两个终端，使用 kafka-console-consumer.sh 创建两个测试消费者组，组名分别是 test-group1 和 test-group2：

```
> bin/kafka-console-consumer.sh --bootstrap-server=localhost:9092 --topic test --from-beginning --consumer-property group.id=test-group1
> bin/kafka-console-consumer.sh --bootstrap-server=localhost:9092 --topic test --consumer-property group.id=test-group2
```

为了模拟不同的消费进度，test-group1 指定了--from-beginning，表示从头消费 topic；而 test-group2 则未指定--from-beginning，表示从 topic 的最新位移处开始消费。

下面使用 kafka-consumer-groups.sh 来查询消费者组的情况，我们以 Kafka 0.11.0.0 版本为例。由于我们使用的是新版本 consumer，故调用 kafka-consumer-groups.sh 时需要指定--bootstrap-server，如下：

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


Note: This will only show information about consumers that use the Java consumer API (non-ZooKeeper-based consumers).

```
test-group2
test-group1
```

如上所见，该命令正确地列出了当前集群中的消费者组。下面分别查询一下两个消费者组的详情，即分别运行以下命令：

```
> bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --describe --group test-group1
```

Note: This will only show information about consumers that use the Java consumer API (non-ZooKeeper-based consumers).

Note: This will only show information about consumers that use the Java consumer API (non-ZooKeeper-based consumers).

TOPIC	PARTITION	CURRENT-OFFSET	LOG-END-OFFSET	LAG	CONSUMER-ID	HOST	CLIENT-ID
test	0	5000000	5000000	0	consumer-1-d558eded-da9c-4503-98f8-82a107df805a	/127.0.0.1	consumer-1

```
> bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --describe --group test-group2
```

Note: This will only show information about consumers that use the Java consumer API (non-ZooKeeper-based consumers).

Note: This will only show information about consumers that use the Java consumer API (non-ZooKeeper-based consumers).

Consumer group 'test-group2' has no active members.

TOPIC	PARTITION	CURRENT-OFFSET	LOG-END-OFFSET	LAG	CONSUMER-ID	HOST	CLIENT-ID
test	0	5000000	5000000	0	-	-	-

消费者组的详情展示包括如下几个方面的内容。

- **TOPIC**：表示该消费者组消费了哪些 topic。本例中只有一个 topic——test。
- **PARTITION**：表示该消费者组消费的是哪个分区。本例中 test 只有一个分区，故输出只有 1 行。
- **CURRENT-OFFSET**：表示消费者组最新消费的位移值。本例中两个消费者组都是 5000000。
- **LOG-END-OFFSET**：表示 topic 所有分区当前的日志终端位移值。本例中是 5000000，因为我们生产了 5 百万条消息。
- **LAG**：表示消费滞后进度，该值等于 LOG-END-OFFSET 与 CURRENT-OFFSET 的差值，通常都是大于或等于 0 的正数。若该值接近 LOG-END-OFFSET 则表明该消费者组消费滞后严重。本例中该值是 0 表示不存在消费滞后的情况。特别需要注意的是，若该值小于 0，则通常表明存在消费数据丢失的情况——即有些消息未被消费就被 consumer 直接跳过了。若出现这种情况，我们需要确认 broker 端参数 `unclean.leader.election.enable` 的值是否被设置成了 `true`，并进一步研究是否有可能出现了因 `unclean` 领导者选举而造成的数据丢失。

- **CONSUMER-ID**: 表示 consumer 的 ID，通常是 Kafka 自动生成的。如果该列没有值，通常表明此 consumer 目前尚未处于运行中。
- **HOST**: 表示 consumer 所在的 broker 主机信息。如果该列没有值，通常表明此 consumer 目前尚未处于运行中。
- **CLIENT-ID**: 表示用户指定或系统自动生成的一个标识 consumer 所在客户端的 ID。该 ID 通常用于定位和调试问题。

由前面的输出可知，test-group1 后 3 列值都存在说明 test-group1 依然是 active 的消费者组，即正在进行消费的 consumer group。它当前的消费滞后是 0，表示它完全地消费了 test 的所有消息。而对于 test-group2 而言，后 3 列是空或-，表示它当前不是 active 的，组成员已全部退出，但它当前的消费滞后也是 0，也完整地消费了 test 的所有消息。

上面说的是查询新版本消费者组，下面来演示如何查询老版本的消费者组。首先构造一个老版本的消费者组 test-group3，运行命令如下：

```
> bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic test
--from-beginning --consumer-property group.id=test-group3
```

然后使用 kafka-consumer-groups.sh --zookeeper 查询该消费者组的状态：

```
> bin/kafka-consumer-groups.sh --zookeeper localhost:2181 --list
Note: This will only show information about consumers that use ZooKeeper
(not those using the Java consumer API).

test-group3

> bin/kafka-consumer-groups.sh --zookeeper localhost:2181 --describe --
group test-group3
Note: This will only show information about consumers that use ZooKeeper
(not those using the Java consumer API).

TOPIC PARTITION CURRENT-OFFSET LOG-END-OFFSET LAG CONSUMER-ID
test 0 5000000 5000000 0 test-group3_bogon-
1502240586849-26736ee7
```

由于这里查询的是老版本消费者组，所以执行命令时需要指定--zookeeper 而非--bootstrap-server。同时我们也可以看到 test-group1 和 test-group2 没有出现在列表中。这表明 Kafka 对新旧两个版本是分别管理的，互不干扰。

7.4.2 重设消费者组位移

这是 0.11.0.0 版本提供的新功能并且只适用于新版本消费者组。在此版本之前，如果要为

已有的消费者组调整位移，就必须手动编写 Java 程序调用 `KafkaConsumer#seek` 方法，这样费时费力不说，还容易出错。0.11.0.0 版本丰富了 `kafka-consumer-groups` 脚本的功能，用户可以直接使用该脚本很方便地为已有的 consumer group 重新设置位移，但前提是 consumer group 不能处于运行状态，也就是说它必须是 `inactive` 的。

总体来说，重设位移的流程由 3 步组成，如图 7.3 所示。

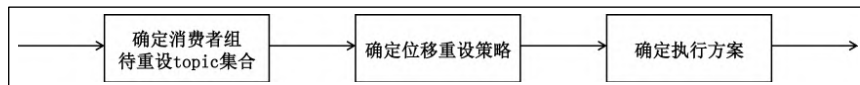


图 7.3 重设消费者组位移流程

第一步是确定消费者组下 topic 的作用域，当前支持 3 种作用域，它们分别如下。

- `--all-topics`: 为消费者组下所有 topic 的所有分区调整位移。
- `--topic t1, --topic t2`: 为指定的若干个 topic 的所有分区调整位移。
- `--topic t1:0,1,2`: 为 topic 的指定分区调整位移。

确定了 topic 作用域之后，第二步就是确定位移重设策略。当前支持如下 8 种设置规则。

- `--to-earliest`: 把位移调整到分区当前最早位移处。
- `--to-latest`: 把位移调整到分区当前最新位移处。
- `--to-current`: 把位移调整到分区当前位移处。
- `--to-offset <offset>`: 把位移调整到指定位移处。
- `--shift-by N`: 把位移调整到当前位移+N 处。N 可以是负值。
- `--to-datetime <datetime>`: 把位移调整到大于给定时间的最早位移处。datetime 格式是 `yyyy-MM-ddTHH:mm:ss.xxx`，比如 `2017-08-04T00:00:00.000`。
- `--by-duration <duration>`: 把位移调整到距离当前时间指定间隔的位移处。duration 格式是 `PnDTnHnMnS`，比如 `PT0H5M0S`。
- `--from-file <file>`: 从 CSV 文件中读取位移调整策略。

最后一步是确定执行方案，当前支持如下 3 种方案。

- 不加任何参数：只是打印位移调整方案，不实际执行。
- `--execute`: 执行真正的位移调整。
- `--export`: 把位移调整方案保存成 CSV 格式并输出到控制台，方便用户保存成 CSV 文件，供后续结合 `--from-file` 参数使用。

本节将重点演示除去 `--from-file` 的 7 种位移重设策略。首先我们依然要创建一个测试 topic，名为 `test`，5 个分区，然后发送 5000000 条消息，如下：

```
> bin/kafka-topics.sh --zookeeper localhost:2181 --create --partitions 5
--replication-factor 1 --topic test

Created topic "test".

> bin/kafka-producer-perf-test.sh --topic test --num-records 5000000 --
throughput -1 --record-size 100 --producer-props bootstrap.servers=localhost:9092
acks=-1

1439666 records sent, 287760.5 records/sec (27.44 MB/sec), 75.7 ms avg
latency, 317.0 max latency.
1541123 records sent, 308163.0 records/sec (29.39 MB/sec), 136.4 ms avg
latency, 480.0 max latency.
1878025 records sent, 375529.9 records/sec (35.81 MB/sec), 58.2 ms avg
latency, 600.0 max latency.
5000000 records sent, 319529.652352 records/sec (30.47 MB/sec), 86.33 ms
avg latency, 600.00 ms max latency, 38 ms 50th, 319 ms 95th, 516 ms 99th,
591 ms 99.9th.
```

然后在该终端下启动一个 **console consumer** 程序，组名设置为 **test-group**：

```
> bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --
topic test --from-beginning --consumer-property group.id=test-group

.....
```

待运行一段时间后按下 **Ctrl+C** 组合键关闭 **console consumer** 程序，将 **test-group** 设置为 **inactive** 状态。之后运行 **kafka-consumer-groups.sh** 脚本，确定当前 **group** 的消费进度：

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

Note: This will only show information about consumers that use the Java
consumer API (non-ZooKeeper-based consumers).
```

TOPIC	PARTITION	CURRENT-OFFSET	LOG-END-OFFSET	LAG	CONSUMER-ID	HOST
test	0	1000000	1000000	0	consumer-1-8688633a-2f88-4c41-89ca-fd0cd6d19ec7	127.0.0.1
test	1	1000000	1000000	0	consumer-1-8688633a-2f88-4c41-89ca-fd0cd6d19ec7	127.0.0.1
test	2	1000000	1000000	0	consumer-1-8688633a-2f88-4c41-89ca-fd0cd6d19ec7	127.0.0.1
test	3	1000000	1000000	0	consumer-1-8688633a-2f88-4c41-89ca-fd0cd6d19ec7	127.0.0.1
test	4	1000000	1000000	0	consumer-1-8688633a-2f88-4c41-89ca-fd0cd6d19ec7	127.0.0.1

由上面的输出可知，当前 5 个分区 LAG 列的值都是 0，表示全部消费完毕。下面我们演示一下如何重设位移：

```
--to-earliest
> bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --group
test-group --reset-offsets --all-topics --to-earliest --execute
Note: This will only show information about consumers that use the Java
consumer API (non-ZooKeeper-based consumers).

TOPIC PARTITION NEW-OFFSET
test 0 0
test 1 0
test 4 0
test 3 0
test 2 0
```

可见，所有分区的位移都已经被重设到 0，即当前最早位移处。

--to-latest

接着上面的状态，我们执行下面的命令：

```
> bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --group
test-group --reset-offsets --all-topics --to-latest --execute
Note: This will only show information about consumers that use the Java
consumer API (non-ZooKeeper-based consumers).

TOPIC PARTITION NEW-OFFSET
test 0 1000000
test 1 1000000
test 4 1000000
test 3 1000000
test 2 1000000
```

即所有分区的位移都已经被重设为最新位移，即 1000000。

--to-offset <offset>

下面将所有分区的位移都重设到当前的一半，即 $1000000/2=500000$ ：

```
> bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --group
test-group --reset-offsets --all-topics --to-offset 500000 --execute
Note: This will only show information about consumers that use the Java
consumer API (non-ZooKeeper-based consumers).
```

```

TOPIC PARTITION NEW-OFFSET
test 0 500000
test 1 500000
test 4 500000
test 3 500000
test 2 500000
--to-current
> bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --group
test-group --reset-offsets --all-topics --to-current --execute
Note: This will only show information about consumers that use the Java
consumer API (non-ZooKeeper-based consumers).

TOPIC PARTITION NEW-OFFSET
test 0 500000
test 1 500000
test 4 500000
test 3 500000
test 2 500000

```

输出表明所有分区的位移都已经被移动到当前位移（实际上位移未移动，距上一步没有变动）：

```

--shift-by N
> bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --group
test-group --reset-offsets --all-topics --shift-by -100000 --execute
Note: This will only show information about consumers that use the Java
consumer API (non-ZooKeeper-based consumers).

TOPIC PARTITION NEW-OFFSET
test 0 400000
test 1 400000
test 4 400000
test 3 400000
test 2 400000

```

输出表明所有分区的位移被移动到 $(500000 - 100000) = 400000$ 处：

```

--to-datetime
> bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --group
test-group --reset-offsets --all-topics --to-datetime 2017-08-
04T14:30:00.000
Note: This will only show information about consumers that use the Java
consumer API (non-ZooKeeper-based consumers).

```

```
TOPIC PARTITION NEW-OFFSET
test 0 1000000
test 1 1000000
test 4 1000000
test 3 1000000
test 2 1000000
```

将所有分区的位移调整为 2017 年 8 月 4 日 14: 30 之后的最早位移:

```
--by-duration
> bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --group
test-group --reset-offsets --all-topics --by-duration PT0H30M0S
Note: This will only show information about consumers that use the Java
consumer API (non-ZooKeeper-based consumers).
```

```
TOPIC PARTITION NEW-OFFSET
test 0 0
test 1 0
test 4 0
test 3 0
test 2 0
```

将所有分区位移调整为 30 分钟之前的最早位移。

7.4.3 删除消费者组

很多用户都有这样的需求, 即删除已有消费者组的信息。Kafka 自带的 `kafka-consumer-groups` 脚本便提供了这样的功能, 帮助用户删除处于 `inactive` 状态的老版本消费者组信息。切记, 该命令只能删除老版本消费者组的元数据信息。我们依然以 `test-group3` 为例, 首先查询老版本消费者组:

```
> bin/kafka-consumer-groups.sh --zookeeper localhost:2181 --list
Note: This will only show information about consumers that use ZooKeeper
(not those using the Java consumer API).

test-group3
```

可见, 当前集群中只有一个老版本消费者组 `test-group3`, 下面执行删除命令:

```
> bin/kafka-consumer-groups.sh --zookeeper localhost:2181 --delete --
group test-group3
Note: This will only show information about consumers that use ZooKeeper
(not those using the Java consumer API).
```

```
Error: Delete for group 'test-group3' failed because its consumers are still active.
```

命令执行失败了，提示该消费者组依然处于 `active` 状态。因此我们首先要关闭 `test-group3`，本例中直接按下 `Ctrl+C` 组合键退出 `console-consumer` 程序即可。之后再次运行上面的命令：

```
> bin/kafka-consumer-groups.sh --zookeeper localhost:2181 --delete --group test-group3
Note: This will only show information about consumers that use ZooKeeper (not those using the Java consumer API).

Deleted all consumer group information for group 'test-group3' in zookeeper.
```

这次命令成功运行，`test-group3` 的元数据信息被成功删除。

很多用户可能会有这样的疑问：我应该如何移除新版本消费者组的信息呢？事实上，新版本消费者组过期数据的移除完全不需要用户操作，`Kafka` 会定期地移除过期消费者组的数据，而这对用户是完全透明的。如果用户查询 `Kafka` 官网上的 `broker` 端参数列表，就会发现一个名为 `offsets.retention.minutes` 的参数，默认值是 1440 分钟，即 1 天。该参数控制 `Kafka` 何时移除 `inactive` 消费者组位移的信息。默认情况下，对于一个新版本的消费者组而言，即使它所有的成员都已经退出组，它的位移信息也不会马上被移除，`Kafka` 会在最后一个成员退出组的 1 天之后删除该组的所有信息，因此不需要用户显式地手动删除。不过用户的确可以设置该参数来间接影响这个过程。

7.4.4 kafka-consumer-offset-checker

鉴于很多用户依然在使用老版本的 `Kafka` 以及消费者组，这里还是要提一下 `kafka-consumer-offset-checker` 脚本。实际上该脚本已经被新版本 `Kafka` 标记为不推荐使用（`deprecated`），而且该脚本在 1.0.0 版本已经被正式移除了。`Kafka` 更希望用户尽快升级到新版本并始终坚持使用 `kafka-consumer-groups` 脚本。

`kafka-consumer-offset-checker.sh(bat)` 只能查询老版本消费者组。具体使用方法与 `kafka-consumer-groups` 类似。我们依然以 `test-group3` 为例。若要查询消费者组列表以及单个消费者组详情，需要运行以下命令：

```
> bin/kafka-consumer-offset-checker.sh --zookeeper localhost:2181 --group test-group3
[2017-08-09 09:49:00,349] WARN WARNING: ConsumerOffsetChecker is deprecated and will be dropped in releases following 0.9.0. Use ConsumerGroupCommand instead. (kafka.tools.ConsumerOffsetChecker$)
Group      Topic    Pid    Offset    logSize    Lag    Owner
test-group3 test     0      2132231   5000000    2867769 none
```


7.5 topic 分区管理

7.5.1 preferred leader 选举

在一个 Kafka 集群中，broker 服务器宕机或崩溃是不可避免的。一旦发生这种情况，该 broker 上的那些 leader 副本将变为不可用，因此就必然要求 Kafka 把这些分区的 leader 转移到其他的 broker 上。即使崩溃 broker 重启回来，其上的副本也只能作为 follower 副本加入 ISR（ISR 的含义参见第 6 章）中，不能再对外提供服务。

随着集群的不断运行，这种 leader 的不均衡现象开始出现，即集群中的一小部分 broker 上承载了大量的分区 leader 副本。为了校正这种情况，Kafka 引入了首选副本（preferred replica，实际上这个名词国内目前没有特别权威的译法，而且笔者也不太赞成翻译成首选副本，因此在下面的讨论中将始终使用 preferred replica 来指代）的概念。

假设我们为一个分区分配了 3 个副本，它们分别是 0、1、2。那么节点 0 就是该分区的 preferred replica，并且通常情况下是不会发生变更的。选择节点 0 的原因仅仅是它是副本列表中的第一个副本。

Kafka 提供了两种方式帮助用户把指定分区的 leader 调整回它们的 preferred replica。这个过程被称为 preferred leader 选举。下面就来具体地看一下如何操作。第一种方式是使用 Kafka 自带的 kafka-preferred-replica-election.sh(bat)脚本。和之前所有的脚本相同，.sh 脚本位于 Kafka 路径的/bin 子目录下，而.bat 脚本位于 Kafka 路径的/bin/windows 子目录下。

如果不加参数地运行 kafka-preferred-replica-election 脚本，将得到此工具的帮助信息，它的参数及其含义如表 7.5 所示。

表 7.5 kafka-preferred-replica-election 脚本常见参数及其含义

参 数 名	参 数 含 义
--zookeeper	指定 ZooKeeper 连接信息
--path-to-json-file	指定 json 文件路径，该文件包含了要为哪些分区执行 preferred leader 选举。也可不指定该参数，若不指定则表明为集群中所有分区都执行 preferred leader 选举

清楚了该脚本的参数含义，下面我们结合一个具体的实例来演示如何利用 kafka-preferred-replica-election.sh 执行 preferred leader 选举。

首先，搭建一个 3 节点的 Kafka 环境，具体方法参见第 3 章，这里不再赘述。然后我们创建一个 3 个分区、副本因子是 3 的测试 topic——test-topic。命令如下：

```

    > bin/kafka-topics.sh --create --zookeeper localhost:2181 --partitions 3
    --replication-factor 3 --topic test-topic

```

```
Created topic "test-topic".
> bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic
test-topic
Topic:test-topic PartitionCount:3ReplicationFactor:3 Configs:
  Topic: test-topicPartition: 0Leader: 1    Replicas: 1,0,2 Isr: 1,0,2
  Topic: test-topicPartition: 1Leader: 2    Replicas: 2,1,0 Isr: 2,1,0
  Topic: test-topicPartition: 2Leader: 0    Replicas: 0,2,1 Isr: 0,2,1
```

由上面的输出可知，当前的3个分区的 leader 副本已经是它们的 preferred replica 了。现在我们依次关闭 broker1 和 broker2 来模拟集群服务器崩溃的场景。这样做的目的是为了让所有分区的 leader 都变更到 broker0 上，从而造成集群 leader 分布不均匀的情况出现。执行完这些操作之后我们再次 describe 一下 topic，输入如下：

```
> bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic
test-topic
Topic:test-topic PartitionCount:3ReplicationFactor:3 Configs:
  Topic: test-topicPartition: 0Leader: 0    Replicas: 1,0,2 Isr: 0,1,2
  Topic: test-topicPartition: 1Leader: 0    Replicas: 2,1,0 Isr: 0,1,2
  Topic: test-topicPartition: 2Leader: 0    Replicas: 0,2,1 Isr: 0,1,2
```

果然，3个分区的 leader 现在都变成了 broker0。现在我们开始执行 kafka-preferred-replica-election.sh 脚本来调整它们的 leader。本例中由于分区 2 的 leader 已经是 preferred replica 了，故真正需要调整的分只分区 0 和分区 1。因此我们首先构造 json 文件：

```
echo '{"partitions":[{"topic":"test-topic","partition":0}, {"topic":"test-
topic","partition":1}]}' > preferred-leader-plan.json
```

之后，执行 kafka-preferred-replica-election.sh 脚本命令：

```
> bin/kafka-preferred-replica-election.sh --zookeeper localhost:2181 --
path-to-json-file <path>/preferred-leader-plan.json
Created preferred replica election path with {"version":1,"partitions":
[{"topic":"test-topic","partition":0}, {"topic":"test-topic","partition":1}]}
Successfully started preferred replica election for partitions
Set([test-topic,0], [test-topic,1])
```

由此可见，命令执行成功。下面再次 describe 一下 topic，来验证分区 0 和分区 1 的 leader 都已经调整回它们的 preferred replica：

```
> bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic
test-topic
Topic:test-topic PartitionCount:3ReplicationFactor:3 Configs:
  Topic: test-topicPartition: 0Leader: 1    Replicas: 1,0,2 Isr: 0,1,2
  Topic: test-topicPartition: 1Leader: 2    Replicas: 2,1,0 Isr: 0,1,2
  Topic: test-topicPartition: 2Leader: 0    Replicas: 0,2,1 Isr: 0,1,2
```

上面的输出表明目前 test-topic 的所有分区都已经调整成它们的 preferred replica 了。

当然，对于本例这个测试场景而言，用户完全可以不指定 `--path-to-json-file` 参数直接运行 `kafka-preferred-replica-election.sh` 脚本。这样做的效果就是 Kafka 对集群上所有 topic 的所有分区都执行了 preferred leader 选举。这种用法在实际的线上环境中一定要慎用，因为对所有分区进行 leader 的转移是一项成本很高的工作，对 clients 端程序必然有一定的影响。因此如非必需，最好还是小批量地对部分分区执行该操作，至少也要避开业务高峰时段进行。

除了 `kafka-preferred-replica-election.sh(bat)` 脚本帮助用户手动进行 preferred leader 选举之外，Kafka 还提供了一个 broker 端参数 `auto.leader.rebalance.enable` 来帮助用户自动地执行此项操作。该参数默认值是 `true`，表明每台 broker 启动后都会在后台自动地定期执行 preferred leader 选举。与之关联的还有两个 broker 端参数 `leader.imbalance.check.interval.seconds` 和 `leader.imbalance.per.broker.percentage`。前者控制阶段性操作的时间间隔，当前默认值是 300 秒，即 Kafka 每 5 分钟就会尝试在后台运行一个 preferred leader 选举；后者用于确定需要执行 preferred leader 选举的目标分区，当前默认值是 10，表示若 broker 上 leader 不均衡程度超过了 10%，则 Kafka 需要为该 broker 上的分区执行 preferred leader 选举。Kafka 计算不均衡程度的逻辑实际上非常简单——该 broker 上的 leader 不是 preferred replica 的分区数/broker 上总的分区数。

7.5.2 分区重分配

7.1.3 节中提到过新增的 broker 是不会自动地分担已有 topic 的负载的，它只会对增加 broker 后新创建的 topic 生效。如果要让新增 broker 为已有的 topic 服务，用户必须手动地调整已有 topic 的分区分布，将一部分分区搬移到新增 broker 上。这就是所谓的分区重分配操作（partition reassignment）。Kafka 提供了分区重分配脚本工具 `kafka-reassign-partitions.sh(bat)`。该工具所在路径与其他脚本工具相同，此处不再赘述。

用户使用该工具时需要提供一组需要执行分区重分配的 topic 列表以及对应的一组 broker。该脚本接到用户这些信息后会尝试制作一个重分配方案，它会力求保证均匀地分配给定 topic 的所有分区到目标 broker 上。

下面我们结合一个实例来演示如何利用该脚本工具执行分区的重分配。假设我们有一个 4 节点的 Kafka 集群（具体搭建方法请参考第 3 章），然后创建两个测试 topic——foo1 和 foo2。它们都是 3 个分区，副本因子都是 2。之后我们把这两个 topic 的分区都搬移到新的 broker 节点上，即 broker5 和 broker6。

首先要构造一个 json 文件，表明要执行分区重分配的 topic 列表：

```
> cat topics-to-move.json
{"topics": [{"topic": "foo1"}, {"topic": "foo2"}], "version": 1}
```

现在执行 `kafka-reassign-partitions.sh` 脚本先产生一个分配方案：

```
> bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --topics-to-move-json-file topics-to-move.json --broker-list "5,6" --generate
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]}]}

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]}]}
}
```

如上可见，该工具给出了当前的分区分布情况以及一个候选的重分配方案。值得注意的是，此时分区重分配并没有真正执行，它仅告诉用户一个可能的方案而已。用户最好把当前的分布情况保存下来以备后续的 `rollback`，同时把新的候选方案保存成另一个新的 json 文件 `expand-cluster-reassignment.json`。做完这些之后，我们就可以执行真正的分区重分配了：

```
> bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --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] }
  ]
}
```

命令成功执行，用户可以指定`--verify`参数来验证分区重分配执行成功：

```
> bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --
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
```

除了让该工具提供候选方案之外，用户还可以执行指定分配方案，因此也就不再需要使用`--generate`参数让其生成分配方案了。我们依然使用上面的例子，假设我们要把 `foo1` 分区 0 的两个副本搬移到 `broker5` 和 `broker6` 上，同时把 `foo2` 分区 1 的两个副本搬移到 `broker2` 和 `broker3` 上。若是这样的话，我们可以自行编写重分配方案：

```
> cat custom-reassignment.json
{
  "version": 1,
  "partitions": [
    { "topic": "foo1", "partition": 0, "replicas": [5, 6] },
    { "topic": "foo2", "partition": 1, "replicas": [2, 3] }
  ]
}
```

然后，使用该 `json` 文件和`--execute`参数来开启重分配操作：

```
> bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --
reassignment-json-file custom-reassignment.json --execute
Current partition replica assignment

{
  "version": 1,
  "partitions": [
    { "topic": "foo1", "partition": 0, "replicas": [1, 2] },
    { "topic": "foo2", "partition": 1, "replicas": [3, 4] }
  ]
}

Save this to use as the --reassignment-json-file option during rollback
```

```
Successfully started reassignment of partitions
{"version":1,
"partitions":[{"topic":"foo1","partition":0,"replicas":[5,6]},
               {"topic":"foo2","partition":1,"replicas":[2,3]}]
}
```

执行完之后，依旧使用--verify 进行验证：

```
> bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --
reassignment-json-file custom-reassignment.json --verify
Status of partition reassignment:
Reassignment of partition [foo1,0] completed successfully
Reassignment of partition [foo2,1] completed successfully
```

在实际生产环境中，用户一定要谨慎地发起分区重分配操作，因为分区在不同 broker 间进行数据迁移会极大地占用 broker 机器的带宽资源，从而显著地影响 clients 端业务应用的性能。如果可能的话，尽量在非高峰业务时段执行重分配操作。

7.5.3 增加副本因子

Kafka 支持为已有 topic 的分区增加副本因子（replication factor），具体的方法就是使用 kafka-reassign-partitions.sh 并且为 topic 分区增加额外的副本。下面就来结合一个具体的实例进行演示。

假设我们在一个 3 节点的 Kafka 集群中创建一个单分区、副本因子是 1 的测试 topic——test，如下：

```
> bin/kafka-topics.sh --create --topic test --zookeeper localhost:2181 -
-partitions 1 --replication-factor 1
Created topic "test".
> bin/kafka-topics.sh --describe --topic test --zookeeper localhost:2181
Topic:test PartitionCount:1ReplicationFactor:1 Configs:
Topic: test Partition: 0Leader: 2 Replicas: 2 Isr: 2
```

可见，目前副本因子是 1。现在我们把它重设成 3。首先创建 json 文件：

```
echo '{"version":1,"partitions":[{"topic":"test","partition":0,"replicas"
:[0,1,2]}]}' > increase-replication-factor.json
```

然后执行重分配工作：

```
> bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --reassignment-
json-file increase-replication-factor.json --execute
Current partition replica assignment

{"version":1,"partitions":[{"topic":"test","partition":0,"replicas":[2]}]}
```

Save this to use as the `--reassignment-json-file` option during rollback
Successfully started reassignment of partitions.

之后，我们使用`--verify`来验证重分配是否成功：

```
> bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --reassignment-  
json-file increase-replication-factor.json --verify  
Status of partition reassignment:  
Reassignment of partition [test,0] completed successfully
```

最后使用 `kafka-topics` 脚本来验证该 topic 的副本因子已被设置成 3：

```
> bin/kafka-topics.sh --describe --topic test --zookeeper localhost:2181  
Topic:test PartitionCount:1ReplicationFactor:3 Configs:  
Topic: test Partition: 0Leader: 2 Replicas: 0,1,2 Isr: 2,0,1
```

可见，test 的副本因子已经被调整为 3 了。

7.6 Kafka 常见脚本工具

7.6.1 kafka-console-producer 脚本

`kafka-console-producer` 脚本与 `kafka-console-consumer` 脚本也许是用户最常用的两个 Kafka 工具脚本。它们允许用户在控制台上方便地对 Kafka 集群进行 producer 和 consumer 的测试。

`kafka-console-producer` 脚本从控制台读取标准输入，然后将其发送到指定的 Kafka topic 上。该脚本位于 Kafka 路径的/bin 子目录下（Windows 平台位于 Kafka 路径的/bin/windows 下），其主要参数及其含义如表 7.6 所示。

表 7.6 kafka-console-producer 脚本常见参数及其含义

参 数 名	参 数 含 义
<code>--broker-list</code>	指定 Kafka 集群连接信息。如果是多台 broker 需要以 CSV 格式指定，比如 <code>k1:port1,k2:port,k3:port……</code>
<code>--topic</code>	指定 producer 将消息发送到哪个 topic
<code>--producer-property</code>	指定 producer 的其他定制属性，如 <code>acks</code> 、 <code>compression.type</code> 等
<code>--producer.config</code>	将 producer 其他定制属性保存在文件中，指定给该 producer
<code>--compression-codec</code>	指定 producer 消息压缩类型
<code>--timeout</code>	指定 producer 的 <code>linger.ms</code> 值
<code>--request-required-acks</code>	指定 producer 的 <code>acks</code> 值
<code>--max-memory-bytes</code>	指定 producer 的 <code>buffer.memory</code> 值
<code>--message-send-max-retries</code>	指定 producer 的 <code>retries</code> 值
<code>--max-partition-memory-bytes</code>	指定 producer 的 <code>batch.size</code> 值

表 7.6 给出了该脚本最常用的参数。用户可以运行不加参数的 `kafka-console-producer` 脚本以获取完整的用法。

下面我们来演示一下如何使用 `kafka-console-producer` 脚本。假设我们有一个单节点的 Kafka 环境，首先创建一个测试 topic：

```
> bin/kafka-topics.sh --create --zookeeper localhost:2181 --topic test -
-partitions 4 --replication-factor 1
Created topic "test".
```

然后运行 `kafka-console-producer` 脚本向 `test` 发送消息，指定 `producer` 的 `acks` 为 `all`，使用 LZ4 进行消息压缩，把失败重试次数设置为 10 次，`linger.ms` 设置为 3 秒，需要执行的命令如下：

```
> bin/kafka-console-producer.sh --broker-list localhost:9092 --topic
test --compression-codec lz4 --request-required-acks all --timeout 3000 --
message-send-max-retries 10
>hello, Kafka
>This is a test message
```

需要特别注意的是，当前 `kafka-console-producer` 脚本（至少截止到 1.0.0 版本）不支持用户指定自定义序列化类，而是硬编码使用了 `ByteArraySerializer`，因此用户不需要手工指定 `serializer`，反正指定了也不会使用。

表 7.6 中只是罗列了一部分 `producer` 的属性，如果用户要修改其他的定制属性，则需要使用 `--producer-property` 或 `--producer.config` 来指定。假设我们要指定 `producer` 的 `connections.max.idle.ms` 属性为 300000。该参数不属于常用参数范畴，它控制了 `producer` 端空闲 Socket 连接需要关闭的最长时间。本例中设置为 300000，即如果 `producer` 端维护的 Socket 连接在 5 分钟内没有任何请求，则关闭该 Socket 连接。如果要在 `console producer` 中设置该属性，则需要指定 `--producer-property` 或 `--producer.config`，如下列命令所示：

```
$ bin/kafka-console-producer.sh --broker-list localhost:9092 --topic
test --producer-property connections.max.idle.ms=300000
>hello, Kafka
>
```

用户也可以创建一个文本文件，写入 `connections.max.idle.ms=300000`，然后在调用脚本时指定 `--producer.config <文件路径>` 来设定。不过需要注意的是，`--producer-property` 的优先级要高于 `--producer.config`。如果 `--producer-property` 和 `--producer.config` 中指定了相同的参数，那么以 `--producer-property` 中设定的值为准。

7.6.2 kafka-console-consumer 脚本

和 `kafka-console-producer` 类似，`kafka-console-consumer` 脚本也是运行在控制台上的，主要

从 Kafka topic 中读取消息并写入标准输出。

kafka-console-consumer 脚本位于 Kafka 路径的/bin 子目录下（Windows 平台位于 Kafka 路径的/bin/windows 下），其主要参数及其含义如表 7.7 所示。

表 7.7 kafka-console-consumer 脚本主要参数及其含义

参 数 名	参 数 含 义
--bootstrap-server	指定 Kafka 集群连接信息。如果是多台 broker，需要以 CSV 格式指定，比如 k1:port1,k2:port,k3:port……
--topic	指定 consumer 消费的 topic
--from-beginning	类似于设置 consumer 属性 auto.offset.reset=earliest，即从当前最早位移处开始消费
--zookeeper	指定使用老版本 consumer，不可与--bootstrap-server 同时使用
--consumer-property	指定 consumer 端参数
--consumer.config	以文件方式指定 consumer 端参数
--partition	指定要消费的特定分区

用户可以运行不加参数的 kafka-console-consumer 脚本以获取完整的用法。下面来演示如何使用 kafka-console-consumer 脚本。我们依然以上面的 test topic 为例。首先演示从头消费 test:

```
$ bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic test --from-beginning
hello, world
This is a test message
hello, Kafka
```

细心的读者可能会发现，我们在上面的命令中没有指定 consumer 的 group 名。实际上 kafka-console-consumer 脚本会帮助用户自动生成一个 group ID，因此 Kafka 认为运行 kafka-console-consumer 脚本的用户通常只是为了测试集群，已创建的 consumer group 也不会再被使用了，因此它会“好心”地替用户合成一个 group ID。

如果用户一定要指定 console consumer 的 group ID，那也是有办法的，如下列命令所示：

```
$ bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic test --from-beginning --consumer-property group.id=test-group
hello, Kafka
This is a test message
hello, Kafka
```

由于目前 Kafka 并存两个版本的 consumer，故 kafka-console-consumer 脚本也支持使用老版本 consumer 进行消费。若要使用 old consumer，用户必须指定--zookeeper 而不是--bootstrap-server，如下：

```
$ bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic test --from-beginning --consumer-property group.id=old-consumer-group
```

```
Using the ConsoleConsumer with old consumer is deprecated and will be
removed in a future major release. Consider using the new consumer by
passing [bootstrap-server] instead of [zookeeper].
```

```
hello, Kafka
This is a test message
hello, Kafka
```

正如上面的输出提示的那样，已经不再推荐使用老版本的 `consumer` 了，Kafka 社区总是鼓励用户始终使用新版本 `consumer`。

以上所有的用法都是基于 `consumer group` 来实现的，但 `kafka-console-consumer` 脚本也支持基于 `standalone consumer` 的 `consumer` 实现（`consumer group` 与 `standalone consumer` 的区别请参见第 5 章），即 `kafka-console-consumer` 脚本允许用户指定特定的分区进行消费。以 `test` topic 为例，假设用户想要单独消费 `test` 分区 0 的数据，那么只需要执行以下命令：

```
$ bin/kafka-console-consumer.sbootstrap-server localhost:9092 --topic
test --from-beginning --partition 0
hello, Kafka
This is a test message
hello, Kafka
```

在操作完上述命令后，用户可以单独开一个终端执行 `kafka-consumer-groups --list` 命令，这时便会发现上面的 `console consumer` 并没有创建新的 `consumer group`，可见它是基于 `standalone consumer` 的。

7.6.3 kafka-run-class 脚本

到目前为止，本章中出现的所有 Kafka 脚本工具虽然实现了各自不同的功能，但底层都是使用 `kafka-run-class` 脚本来实现的。比如 `kafka-console-consumer.sh` 就是使用 `kafka-run-class.sh kafka.tools.ConsoleConsumer <参数列表>` 实现的，而 `kafka-console-producer.sh` 则使用的是 `kafka-run-class.sh kafka.tools.ConsoleProducer <参数列表>`。

`kafka-run-class.sh(bat)` 是一个通用的脚本，它允许用户直接指定一个可执行的 Java 类和一组可选的参数列表，从而调用该类实现的逻辑。之前谈到的所有脚本，仅仅是 Kafka 社区开发人员帮助用户做了一层封装。事实上，还有很多其他有用的功能并没有被封装成单独的 Shell 脚本提供给用户，因此用户在使用这些功能时就需要显式地调用 `kafka-run-class` 脚本自行实现。

有些遗憾的是，目前 Kafka 官网文档对于这个现状并没有太多的说明。只有读过 Kafka 源码的人才可能清楚地说出目前到底有多少个这样的工具。笔者会在下面的几节中详细列举几个未记录在官网文档中但却很有用的工具。

7.6.4 查看消息元数据

很多用户都有这样的需求，即查询特定 topic 的消息元数据。所谓的元数据信息包括消息的位移、创建时间戳、压缩类型、字节数等。Kafka 确实提供了这样的功能，只是没有记录在官方文档上——这个工具就是 `kafka.tools.DumpLogSegments`。下面我们结合一个具体的实例来演示一下如何使用该工具查询 Kafka 的日志消息。

首先，创建两个测试 topic——t1 和 t2，之后为 t1 生产若干条未压缩消息，为 t2 生产若干条 LZ4 和 Snappy 压缩的消息，如下：

```
$ bin/kafka-topics.sh -zookeeper localhost:2181 --create --partitions 1
--replication-factor 1 --topic t1
Created topic "t1".

$ bin/kafka-topics.sh -zookeeper localhost:2181 --create --partitions 1
--replication-factor 1 --topic t2
Created topic "t2".

$ bin/kafka-console-producer.sh --broker-list localhost:9092 --topic t1
>message1 for t1
>message2 for t1
>message3 for t1
>message4 for t1
>^C

$ bin/kafka-console-producer.sh --broker-list localhost:9092 --topic t2
--compression-codec lz4
>message1 for t2
>message2 for t2
>message3 for t2
>message4 for t2
```

然后使用 `DumpLogSegments` 查看 t1 日志中的消息：

```
$ bin/kafka-run-class.sh kafka.tools.DumpLogSegments --files ../datalogs/
kafka_1/t1-0/00000000000000000000.log
Dumping ../datalogs/kafka_1/t1-0/00000000000000000000.log
Starting offset: 0
baseOffset: 0 lastOffset: 0 baseSequence: -1 lastSequence: -1 producerId:
-1 producerEpoch: -1 partitionLeaderEpoch: 0 isTransactional: false position:
0 CreateTime: 1502417400743 isValid: true size: 83 magic: 2 compresscodec:
NONE crc: 2602149835
baseOffset: 1 lastOffset: 1 baseSequence: -1 lastSequence: -1 producerId:
-1 producerEpoch: -1 partitionLeaderEpoch: 0 isTransactional: false position:
83 CreateTime: 1502417406693 isValid: true size: 83 magic: 2 compresscodec:
```

```
NONE crc: 4201639679
  baseOffset: 2 lastOffset: 2 baseSequence: -1 lastSequence: -1 producerId:
-1 producerEpoch: -1 partitionLeaderEpoch: 0 isTransactional: false position:
166 CreateTime: 1502417410162 invalid: true size: 83 magic: 2 compresscodec:
NONE crc: 1859000275
  baseOffset: 3 lastOffset: 3 baseSequence: -1 lastSequence: -1 producerId:
-1 producerEpoch: -1 partitionLeaderEpoch: 0 isTransactional: false position:
249 CreateTime: 1502417414698 invalid: true size: 83 magic: 2 compresscodec:
NONE crc: 366766691
```

上面的命令中`--files`参数指定了该 topic 日志段文件的完整路径。从输出可知，当前共有 4 条消息（console producer 生产的 4 条），每行表示一条消息，详细地列出了该消息的位移信息、物理文件位置、消息长度、压缩类型、CRC 码等元数据信息。下面我们使用相同的命令来查看 t2 的情况：

```
$ bin/kafka-run-class.sh kafka.tools.DumpLogSegments --files ../datalogs/
kafka_1/t2-0/00000000000000000000.log --print-data-log
Dumping ../datalogs/kafka_1/t2-0/00000000000000000000.log
Starting offset: 0
  baseOffset: 0 lastOffset: 1 baseSequence: -1 lastSequence: -1 producerId:
-1 producerEpoch: -1 partitionLeaderEpoch: 0 isTransactional: false position:
0 CreateTime: 1502417944716 invalid: true size: 119 magic: 2 compresscodec:
LZ4 crc: 2725016247
  baseOffset: 2 lastOffset: 2 baseSequence: -1 lastSequence: -1 producerId:
-1 producerEpoch: -1 partitionLeaderEpoch: 0 isTransactional: false position:
119 CreateTime: 1502417946995 invalid: true size: 107 magic: 2 compresscodec:
LZ4 crc: 695385804
  baseOffset: 3 lastOffset: 3 baseSequence: -1 lastSequence: -1 producerId:
-1 producerEpoch: -1 partitionLeaderEpoch: 0 isTransactional: false position:
226 CreateTime: 1502417949777 invalid: true size: 115 magic: 2 compresscodec:
LZ4 crc: 2275841639
```

细心的读者可能会觉得奇怪，console producer 明明为 t2 发送了 4 条消息，为什么打印出来的只有 3 行元数据呢？这是因为 t2 启动了消息压缩，使得多条消息可能被封装进一条外层消息（wrapper message）中，而 DumpLogSegments 默认只显示 wrapper message。本例中第一条 wrapper message 实际上包含了两条内部消息（inner message）。何以见得？我们可以看到第 1 行中的 baseOffset 是 0，而 lastOffset 是 1，故这条 wrapper message 实际上包含了两条消息。

DumpLogSegments 工具为开启消息压缩的日志提供了`--deep-iteration`参数来深度遍历被压缩消息，如下：

```
$ bin/kafka-run-class.sh kafka.tools.DumpLogSegments --files ../datalogs/
kafka_1/t2-0/00000000000000000000.log --deep-iteration
Dumping ../datalogs/kafka_1/t2-0/00000000000000000000.log
```

```
Starting offset: 0
offset: 0 position: 0 CreateTime: 1502417944005 invalid: true keysize: -1
valuesize: 11 magic: 2 compresscodec: LZ4 producerId: -1 sequence: -1
isTransactional: false headerKeys: []
offset: 1 position: 0 CreateTime: 1502417944716 invalid: true keysize: -1
valuesize: 17 magic: 2 compresscodec: LZ4 producerId: -1 sequence: -1
isTransactional: false headerKeys: []
offset: 2 position: 119 CreateTime: 1502417946995 invalid: true keysize: -1
valuesize: 24 magic: 2 compresscodec: LZ4 producerId: -1 sequence: -1
isTransactional: false headerKeys: []
offset: 3 position: 226 CreateTime: 1502417949777 invalid: true keysize: -1
valuesize: 33 magic: 2 compresscodec: LZ4 producerId: -1 sequence: -1
isTransactional: false headerKeys: []
```

显然，加上`--deep-iteration`之后 `DumpLogSegments` 正确地打印出了 4 条消息。这 4 条消息都是使用 LZ4 进行压缩的，且位移值一定是连续的。

如果用户为 `t1`（未压缩 topic）使用`--deep-iteration`，会发现输出和原来是一样的。这是因为对于未压缩的 topic 而言，`--deep-iteration` 的效果等同于普通遍历，两者没有区别。

`DumpLogSegments` 不不仅能够查询日志段文件，它还能够查询 Kafka 支持的索引文件类型，比如下列命令是用来查询 `t1` 的第一个日志段对应的位移索引文件的：

```
$ bin/kafka-run-class.sh kafka.tools.DumpLogSegments --files ../datalogs/
kafka_1/t1-0/00000000000000000000.index
Dumping ../datalogs/kafka_1/t1-0/00000000000000000000.index
offset: 0 position: 0
```

由于本例只生产了 4 条消息，没有产生任何索引项，故该索引文件实际上是空的，没有任何输出结果。在实际生产环境中，该命令会将索引文件的每一行打印出来。对于位移索引文件而言，每一行数据的格式就是[位移 相对物理文件位置]。

7.6.5 获取 topic 当前消息数

这也是一个非常实际的需求。用户总是希望能实时了解当前总共为 topic 生产了多少条消息。Kafka 提供了 `GetShellOffset` 类帮助用户实时计算特定 topic 总的消息数。

为了演示如何做到这点，下面首先创建一个多分区的测试 topic——`test`，如下：

```
$ bin/kafka-topics.sh -zookeeper localhost:2181 --create --partitions 5
--replication-factor 1 --topic test
Created topic "test".
```

然后使用 `kafka-producer-perf-test.sh` 脚本生产 50 万条消息：

```
$ bin/kafka-producer-perf-test.sh --topic test --throughput -1 --record-size 10 --num-records 500000 --producer-props bootstrap.servers=localhost:9092
500000 records sent, 257069.408740 records/sec (2.45 MB/sec), 6.52 ms avg latency, 197.00 ms max latency, 5 ms 50th, 19 ms 95th, 38 ms 99th, 53 ms 99.9th.
```

现在使用 `GetShellOffset` 类来帮助我们统计当前的消息总数：

```
$ bin/kafka-run-class.sh kafka.tools.GetOffsetShell --broker-list localhost:9092 --topic test --time -1
test:2:100000
test:4:100000
test:1:100000
test:3:100000
test:0:100000

$ bin/kafka-run-class.sh kafka.tools.GetOffsetShell --broker-list localhost:9092 --topic test --time -2
test:2:0
test:4:0
test:1:0
test:3:0
test:0:0
```

上面 `--time -1` 命令表示要获取指定 `topic` 所有分区当前的最大位移；而 `--time -2` 表示获取当前最早位移。我们将两个命令的输出结果相减便可得到所有分区当前的消息总数。有的读者可能会问为什么需要相减，直接使用 `--time -1` 的结果可以吗？我们知道，随着集群的不断运行，`topic` 的数据可能会被移除一部分，因此 `--time -1` 的结果其实表示的是历史上该 `topic` 生产的最大消息数。如果用户要统计当前的消息总数就必须减去 `--time -2` 的结果。

本例中由于未发生任何消息删除操作，故 `--time -2` 的结果全是 0，表示最早位移都是 0。测试 `topic` 当前的消息总数等于历史上发送的消息总数，即为 $5 \times (100000 - 0) = 50$ 万，与我们 `producer` 生产的消息数吻合。

7.6.6 查询__consumer_offsets

在第 5 章中我们谈到过新版本 `consumer` 的位移保存在 `Kafka` 的内部 `topic` 中，即 `__consumer_offsets`。很多用户对于该 `topic` 的消息内容很感兴趣，想要查询一下该 `topic` 到底是什么样子的。`Kafka` 其实提供 `kafka-simple-consumer-shell.sh(bat)` 脚本来满足用户的好奇心。下面就来演示一下如何查询 `__consumer_offsets`。

依然使用 7.6.3 节中的测试 `topic`，首先创建一个 `consumer` 进行消费：

```
$ bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --
topic test --from-beginning --consumer-property group.id=test-group
SSXVJHPDQ
SSXVJHPDQ
.....
```

上面的 SSXVJHPDQ 实际上是 kafka-producer-perf-test 脚本自动生产的消息，本例中我们不关心消息的具体内容，只关心这个 consumer 的位移是怎么保存的。

由于我们的 group ID 是 test-group，可以计算一下该 ID 对应的哈希值是 627841412，对其进行 mod 50（求模）得到 12，表明 test-group 的位移保存在 __consumer_offsets 的分区 12 上。这里多提一句，为什么对 50 求模？因为默认情况下 __consumer_offsets 的分区数是 50。用户可以配置 broker 端参数 offsets.topic.num.partitions 来修改分区数。

既然确定了目标分区，下面就使用 kafka-simple-consumer-shell 脚本来查询 test-group 的位移信息：

（0.11.0.0 版本及以后）

```
bin/kafka-simple-consumer-shell.sh --topic __consumer_offsets --partition 12 --
broker-list localhost:9092 --formatter "kafka.coordinator.group.GroupMetadataManager\
$OffsetsMessageFormatter"
```

（0.11.0.0 之前的版本）

```
bin/kafka-simple-consumer-shell.sh --topic __consumer_offsets --partition
12 --broker-list localhost:9092 --formatter "kafka.coordinator.GroupMetadataManager\
$OffsetsMessageFormatter"
```

输入如下：

```
[test-group,test,0]::[OffsetMetadata[100000,NO_METADATA],CommitTime
1502420155393,ExpirationTime 1502506555393]
[test-group,test,1]::[OffsetMetadata[100000,NO_METADATA],CommitTime
1502420155393,ExpirationTime 1502506555393]
[test-group,test,4]::[OffsetMetadata[100000,NO_METADATA],CommitTime
1502420155393,ExpirationTime 1502506555393]
[test-group,test,3]::[OffsetMetadata[100000,NO_METADATA],CommitTime
1502420155393,ExpirationTime 1502506555393]
[test-group,test,2]::[OffsetMetadata[100000,NO_METADATA],CommitTime
1502420155393,ExpirationTime 1502506555393]
[test-group,test,0]::[OffsetMetadata[100000,NO_METADATA],CommitTime
1502420160396,ExpirationTime 1502506560396]
[test-group,test,1]::[OffsetMetadata[100000,NO_METADATA],CommitTime
1502420160396,ExpirationTime 1502506560396]
[test-group,test,4]::[OffsetMetadata[100000,NO_METADATA],CommitTime
```

```
1502420160396,ExpirationTime 1502506560396]
  [test-group,test,3]::[OffsetMetadata[100000,NO_METADATA],CommitTime
1502420160396,ExpirationTime 1502506560396]
  [test-group,test,2]::[OffsetMetadata[100000,NO_METADATA],CommitTime
1502420160396,ExpirationTime 1502506560396]
  . . . . .
```

可见，test-group 的位移信息的确保存在 __consumer_offsets 的分区 12 中，每一行展示了某个分区当前的已提交位移、提交时间等。

7.7 API 方式管理集群

除了使用 Kafka 自带的工具脚本，很多用户都有编写 Java 程序来管理集群的需求，这就需要使用 Kafka 开放的各种 API 来帮助实现这样的功能。坦率地说，在 API 使用方法的介绍方面，Kafka 官方文档做得并不好。有许多公共 API 并未显式地记录下来，只有读过源代码才能知道它们的存在。本节将为读者介绍几个这样的 API 使用实例。

谈到 Kafka 的 API，严格来说应该分为服务器端的 API 和客户端的 API。服务器端 API 主要是指利用 Kafka 服务器端代码（kafka_core）实现的各种 API 功能；反之，客户端 API 则是由客户端代码（kafka_clients）提供的。下面在使用过程中我们会显式区分两者。

另外，Kafka 0.11.0.0 版本正式推出了新版本的客户端 API，旨在统一原来散落在各个客户端 API 上的功能，交由统一的 KafkaAdminClient 来提供。我们会在本节后面介绍该类的具体使用方法。下面首先来看看如何应用服务器端 API 来管理集群。

7.7.1 服务器端 API 管理 topic

如前所述，Kafka 官方提供了两个脚本来管理 topic，包括 topic 的增删改查。其中 kafka-topics.sh(bat)脚本负责 topic 的创建与删除，kafka-configs.sh(bat)脚本负责 topic 参数的修改和查询，但很多用户都更倾向于使用程序 API 的方式对 topic 进行操作。

如果要使用服务器端 API，在编写程序时必须显式地增加 Kafka 服务器端代码依赖，如下：

Maven 版本

```
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka_2.11</artifactId>
  <version>Kafka 版本（比如 0.11.0.1 或 0.10.2.1）</version>
</dependency>
```


Gradle 版本

```
compile group: 'org.apache.kafka', name: 'kafka_2.11', version: 'Kafka 版本 (比如 0.11.0.1 或 0.10.2.1) '
```

上面的 `kafka_2.11` 中的 2.11 表示编译 Kafka 的 Scala 语言版本。这里笔者推荐使用 Scala 2.11 进行编译，即 `kafka_2.11`。当然使用 0.10.1.1 之后版本的用户也可以使用 `kafka_2.12`，即使用 Scala 2.12 版本进行编译。

成功添加了依赖之后，就可以使用服务器端的 API——`AdminUtils` 类来管理 topic 了。首先，看看如何创建 topic，如下面的代码所示：

```
ZkUtils zkUtils = ZkUtils.apply("localhost:2181", 30000, 30000, JaasUtils.isZkSecurityEnabled());
// 创建一个单分区、单副本、名为 t1 的 topic
AdminUtils.createTopic(zkUtils, "t1", 1, 1, new Properties(), RackAwareMode.Enforced$.MODULE$);
zkUtils.close();
```

在使用 `AdminUtils` 之前，我们必须构造一个 `ZkUtils` 创建与 ZooKeeper 的连接。本例中 ZooKeeper 连接信息是 `localhost:2181`，后面两个 30000 分别表示 ZooKeeper 的会话超时时间（session timeout）和连接超时时间（connection timeout），我们将其都设置成 30 秒。

在连接上 ZooKeeper 后，上面的代码调用 `AdminUtils.createTopic` 来创建 topic，名字是 `t1`，一个分区且副本因子也是 1。由于未指定任何 topic 级别的参数，故上面代码直接传入空的 `Properties` 对象。另外，`RackAwareMode.Enforced$.MODULE$` 实际上等同于指定了 `RackAwareMode.Enforced`，即在为 topic 制定副本分配方案时需要考虑 broker 的机架位置。

执行完 `createTopic` 后，上面的代码显式地关闭了 `zkUtils`，即与 ZooKeeper 的连接。当然，如果用户要复用这个 `zkUtils`，可以考虑暂不关闭。

以上就是使用 `AdminUtils.createTopic` 创建 topic 的一个代码示例片段。下面来看看如何删除 topic：

```
ZkUtils zkUtils = ZkUtils.apply("localhost:2181", 30000, 30000, JaasUtils.isZkSecurityEnabled());
// 删除 topic 't1'
AdminUtils.deleteTopic(zkUtils, "t1");
zkUtils.close();
```

依然非常简单，只需要调用 `AdminUtils.deleteTopic` 方法传入指定的 topic 即可。不过需要特别注意的是：不管是创建 topic 还是删除 topic，目前 Kafka 实现的方式都是后台异步操作的，而且没有提供任何回调机制或返回任何结果给用户，所以用户除了捕获异常以及查询 topic 状

态之外似乎并没有特别好的办法可以检测操作是否成功。

下面来看看如何使用 API 查询 topic 级别属性：

```
ZkUtils zkUtils = ZkUtils.apply("localhost:2181", 30000, 30000, JaasUtils.
isZkSecurityEnabled());
// 获取 topic 'test' 的属性
Properties props = AdminUtils.fetchEntityConfig(zkUtils, ConfigType.Topic(),
"test");
// 查询 topic-level 属性
Iterator it = props.entrySet().iterator();
while(it.hasNext()){
    Map.Entry entry=(Map.Entry)it.next();
    Object key = entry.getKey();
    Object value = entry.getValue();
    System.out.println(key + " = " + value);
}
zkUtils.close();
```

这次使用的是 `AdminUtils.fetchEntityConfig` 方法遍历输出 topic 的所有 topic 级别参数。下面演示如何变更 topic-level 参数：

```
ZkUtils zkUtils = ZkUtils.apply("localhost:2181", 30000, 30000, JaasUtils.
isZkSecurityEnabled());
Properties props = AdminUtils.fetchEntityConfig(zkUtils, ConfigType.Topic(),
"test");
// 增加 topic 级别属性
props.put("min.cleanable.dirty.ratio", "0.3");
// 删除 topic 级别属性
props.remove("max.message.bytes");
// 修改 topic 'test' 的属性
AdminUtils.changeTopicConfig(zkUtils, "test", props);
zkUtils.close();
```

上面的代码删除了 topic-level 参数 `max.message.bytes`，同时增加了另一个参数 `min.cleanable.dirty.ratio`。本例不关心两个参数的具体含义，用户只需要了解如何使用这些 API 来为特定 topic 修改 topic-level 参数。

7.7.2 服务器端 API 管理位移

这里的位移主要是指新版本 consumer 的位移信息。笔者碰到过很多用户实际上并不愿意使用自带的 `kafka-consumer-groups` 脚本查询 consumer group 的状态信息；相反地，他们更愿意编写 Java 程序定期查询 consumer 状态。本节将演示如何利用服务器端 API 查询位移。

首先编写 Java 程序查询当前集群下的所有 consumer group 信息:

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092"); // 设置 Kafka 集群连接信息
AdminClient client = AdminClient.create(props);
Map<Node, List<GroupOverview>> groups = JavaConversions.mapAsJavaMap(
    (client.listAllGroups()));

for(Map.Entry<Node, List<GroupOverview>> entry : groups.entrySet()) {
    Iterator<GroupOverview> groupOverviewList =
        JavaConversions.asJavaIterator(entry.getValue().iterator());
    while (groupOverviewList.hasNext()) {
        GroupOverview overview = groupOverviewList.next();
        System.out.println(overview.groupId());
    }
}
client.close();
```

下面演示如何查询给定 consumer group 的位移信息:

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
AdminClient client = AdminClient.create(props);

String groupId = "a1";
Map<TopicPartition, Object> offsets = JavaConversions.mapAsJavaMap(client.
    listGroupOffsets(groupId));
Long offset = (Long) offsets.get(new TopicPartition("test", 0));
System.out.println(offset);

client.close();
```

7.7.3 客户端 API 管理 topic

Kafka 自己实现了一套二进制协议 (binary protocol) 用于各种功能的实现, 比如发送消息、获取消息、提交位移以及创建 topic 等。具体协议规范参见 <https://kafka.apache.org/protocol>。这套协议的具体使用流程如下。

- 客户端创建对应协议的请求。
- 客户端发送请求给对应的 broker。
- broker 处理请求, 并发送 response 给客户端。

虽然 Kafka 提供的大量的脚本工具用于各种功能的实现, 但很多时候我们还是希望可以把某些功能以编程的方式嵌入另一个系统中。这时使用 Java API 的方式就显得异常灵活了。本节

将尝试给出一个 Java API 底层框架的范例，同时也会针对“创建 topic”和“查看位移”这两个主要功能给出对应的例子。

既然是客户端 API，则必须显式地添加客户端依赖：

Maven

```
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-clients</artifactId>
  <version>你的Kafka clients 版本（比如0.10.2.1）</version>
</dependency>
```

Gradle

```
compile group: 'org.apache.kafka', name: 'kafka-clients', version: '你的
Kafka clients 版本（比如0.10.2.1）'
```

首先构建底层发送请求框架：

```
/**
 * 发送请求主方法
 * @param host          目标 broker 的主机名
 * @param port          目标 broker 的端口
 * @param request       请求对象
 * @param apiKey        请求类型
 * @return              序列化后的 response
 * @throws IOException
 */
public ByteBuffer send(String host, int port, AbstractRequest request,
    ApiKeys apiKey) throws IOException {
    Socket socket = connect(host, port);
    try {
        return send(request, apiKey, socket);
    } finally {
        socket.close();
    }
}

/**
 * 发送序列化请求并等待 response 返回
 * @param socket        连向目标 broker 的 Socket
 * @param request       序列化后的请求
 * @return              序列化后的 response
 * @throws IOException
 */
```

```
private byte[] issueRequestAndWaitForResponse(Socket socket, byte[]
request) throws IOException {
    sendRequest(socket, request);
    return getResponse(socket);
}

/**
 * 发送序列化请求给 Socket
 * @param socket          连向目标 broker 的 Socket
 * @param request         序列化后的请求
 * @throws IOException
 */
private void sendRequest(Socket socket, byte[] request) throws
IOException {
    DataOutputStream dos = new DataOutputStream(socket.getOutputStream());
    dos.writeInt(request.length);
    dos.write(request);
    dos.flush();
}

/**
 * 从给定 Socket 处获取 response
 * @param socket          连向目标 broker 的 Socket
 * @return                获取到的序列化后的 response
 * @throws IOException
 */
private byte[] getResponse(Socket socket) throws IOException {
    DataInputStream dis = null;
    try {
        dis = new DataInputStream(socket.getInputStream());
        byte[] response = new byte[dis.readInt()];
        dis.readFully(response);
        return response;
    } finally {
        if (dis != null) {
            dis.close();
        }
    }
}

/**
 * 创建 Socket 连接
 * @param hostname        目标 broker 主机名
 * @param port            目标 broker 服务端口, 比如 9092

```

```

    * @return                创建的 Socket 连接
    * @throws IOException
    */
    private Socket connect(String hostName, int port) throws IOException {
        return new Socket(hostName, port);
    }

    /**
     * 向给定 Socket 发送请求
     * @param request          请求对象
     * @param apiKey           请求类型, 即属于哪种请求
     * @param socket           连向目标 broker 的 Socket
     * @return                 序列化后的 response
     * @throws IOException
     */
    private ByteBuffer send(AbstractRequest request, ApiKeys apiKey,
        Socket socket) throws IOException {
        RequestHeader header = new RequestHeader(apiKey.id, request.
version(), "client-id", 0);
        ByteBuffer buffer = ByteBuffer.allocate(header.sizeOf() + request.
sizeof());
        header.writeTo(buffer);
        request.writeTo(buffer);
        byte[] serializedRequest = buffer.array();
        byte[] response = issueRequestAndWaitForResponse(socket,
serializedRequest);
        ByteBuffer responseBuffer = ByteBuffer.wrap(response);
        ResponseHeader.parse(responseBuffer);
        return responseBuffer;
    }
}

```

然后构建 CreateTopics 请求来创建 topic:

```

/**
 * 创建 topic
 * 由于只是样例代码, 有些东西可以硬编码到程序里 (比如主机名和端口)
 * @param topicName        topic 名
 * @param partitions       分区数
 * @param replicationFactor 副本数
 * @throws IOException
 */
    public void createTopics(String topicName, int partitions, short
replicationFactor) throws IOException {
        Map<String, CreateTopicsRequest.TopicDetails> topics = new
HashMap<>();
    }
}

```

```
// 插入多个元素便可同时创建多个 topic
topics.put(topicName, new CreateTopicsRequest.TopicDetails(partitions,
replicationFactor));
int creationTimeoutMs = 60000;
CreateTopicsRequest request = new CreateTopicsRequest.Builder
(topics, creationTimeoutMs).build();
ByteBuffer response = send("localhost", 9092, request, ApiKeys.
CREATE_TOPICS);
CreateTopicsResponse.parse(response, request.version());
}
```

下面演示如何构造 DeleteTopicsRequest 来删除一组 topic:

```
public void deleteTopics(Set<String> topics) {
    int deleteTimeoutMs = 30000;
    DeleteTopicsRequest request = new DeleteTopicsRequest.
Builder(topics, deleteTimeoutMs).build();
    ByteBuffer response = send("localhost", 9092, request, ApiKeys.
DELETE_TOPICS);
    DeleteTopicsResponse.parse(response, request.version());
}
```

7.7.4 客户端 API 查看位移

同样地，我们使用相同的底层请求发送框架构造特定的请求类型，分别查询某个 consumer group 下所有 topic 的位移信息以及特定 topic 的位移信息：

```
/**
 * 获取某个 consumer group 下所有 topic 分区的位移信息
 * @param groupId      group id
 * @return              (topic 分区→分区信息) 的 map
 * @throws IOException
 */
public Map<TopicPartition, OffsetFetchResponse.PartitionData>
getAllOffsetsForGroup (String groupId) throws IOException {
    OffsetFetchRequest request = new OffsetFetchRequest.Builder
(groupId, null).setVersion((short)2).build();
    ByteBuffer response = send("localhost", 9092, request, ApiKeys.
OFFSET_FETCH);
    OffsetFetchResponse resp = OffsetFetchResponse.parse(response,
request.version());
    return resp.responseData();
}
```

再获取特定 topic 的位移信息：

```

/**
 * 获取某个 consumer group 下的某个 topic 分区的位移
 * @param groupID      group id
 * @param topic        topic 名
 * @param parititon    分区号
 * @throws IOException
 */
public void getOffsetForPartition(String groupID, String topic, int
parititon) throws IOException {
    TopicPartition tp = new TopicPartition(topic, parititon);
    OffsetFetchRequest request = new OffsetFetchRequest.Builder
(groupID, singletonList(tp)).setVersion((short)2).build();
    ByteBuffer response = send("localhost", 9092, request, ApiKeys.
OFFSET_FETCH);
    OffsetFetchResponse resp = OffsetFetchResponse.parse(response,
request.version());
    OffsetFetchResponse.PartitionData partitionData = resp.responseData().
get(tp);
    System.out.println(partitionData.offset);
}

```

7.7.5 0.11.0.0 版本客户端 API

自 0.11.0.0 版本起，Kafka 社区推出了 `AdminClient` 和 `KafkaAdminClient`，意在统一所有的集群管理 API。使用 0.11.0.0 及以后版本的用户应该始终使用这个类来管理集群。

虽然和原先服务器端的 `AdminClient` 类同名，但这个工具是属于客户端的，因此只需要在管理程序项目中添加 `kafka_clients` 依赖即可，比如在 Gradle 中增加 `compile group: 'org.apache.kafka', name: 'kafka-clients', version: '1.0.0'`。

该工具提供的所有功能如下。

- 创建 topic。
- 查询所有 topic。
- 查询单个 topic 详情。
- 删除 topic。
- 修改 config（包括 BROKER 和 TOPIC 资源的 config）。
- 查询资源 config 详情。
- 创建 ACL。
- 查询 ACL 详情。
- 删除 ACL。

- 查询整个集群详情。

用户使用该类的方式与 Java clients 的使用方式一致，不用连接 ZooKeeper，而是直接给定集群中的 broker 列表。另外，该类是线程安全的，因此可以放心地在多个线程中使用该类的实例。AdminClient 的实现机制与 7.7.4 节中的客户端 API 实现原理完全一样，都是在后台自行构建 Kafka 的各种请求后发送，只不过 AdminClient 帮用户实现了所有的细节，用户不再自己编写底层的各种功能代码。

下面来演示一下上面列表中除了 ACL 以外的所有功能。由于代码很简单且添加了足够的注释，故不再对该类做过多说明：

```
public class AdminClientTest {

    private static final String TEST_TOPIC = "test-topic";//测试Kafka topic

    public static void main(String[] args) throws Exception {
        Properties props = new Properties();
        props.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:
9092,localhost:9093"); // Kafka 集群连接信息

        try (AdminClient client = AdminClient.create(props)) {
            describeCluster(client); // 描述集群信息
            createTopics(client);    // 创建 topic
            listAllTopics(client);   // 查询集群所有 topic
            describeTopics(client);  // 查询 topic 信息
            alterConfigs(client);    // 修改 topic 参数配置信息
            describeConfig(client);  // 查询所有配置信息
            deleteTopics(client);    // 删除 topic
        }
    }

    /**
     * 获取 Kafka 集群信息
     * @param client      AdminClient 客户端
     * @throws ExecutionException
     * @throws InterruptedException
     */
    public static void describeCluster(AdminClient client) throws
ExecutionException, InterruptedException {
        DescribeClusterResult ret = client.describeCluster();
        System.out.println(String.format("Cluster id: %s, controller: %s",
ret.clusterId().get(), ret.controller().get()));
        System.out.println("Current cluster nodes info: ");
    }
}
```

```

        for (Node node : ret.nodes().get()) {
            System.out.println(node);
        }
    }

    /**
     * 获取测试 topic 的 topic 级别参数配置信息
     * @param client      AdminClient 客户端
     */
    public static void describeConfig(AdminClient client) throws
    ExecutionException, InterruptedException {
        DescribeConfigsResult ret = client.describeConfigs(Collections.
    singleton(new ConfigResource(ConfigResource.Type.TOPIC, TEST_TOPIC)));
        Map<ConfigResource, Config> configs = ret.all().get();
        for (Map.Entry<ConfigResource, Config> entry : configs.entrySet()) {
            ConfigResource key = entry.getKey();
            Config value = entry.getValue();
            System.out.println(String.format("Resource type: %s, resource
    name: %s", key.type(), key.name()));
            Collection<ConfigEntry> configEntries = value.entries();
            for (ConfigEntry each : configEntries) {
                System.out.println(each.name() + " = " + each.value());
            }
        }
    }

    /**
     * 修改 topic 级别参数
     * @param client      AdminClient 客户端
     */
    public static void alterConfigs(AdminClient client) throws
    ExecutionException, InterruptedException {
        Config topicConfig = new Config(Arrays.asList(new ConfigEntry
    ("cleanup.policy", "compact")));
        client.alterConfigs(Collections.singletonMap(
            new ConfigResource(ConfigResource.Type.TOPIC, TEST_TOPIC),
    topicConfig)).all().get();
    }

    /**
     * 删除给定 topic
     * @param client      AdminClient 客户端
     */
    public static void deleteTopics(AdminClient client) throws ExecutionException,

```

```

InterruptedException {
    KafkaFuture<Void> futures = client.deleteTopics(Arrays.asList
(TEST_TOPIC)).all();
    futures.get();
}

/**
 * 获取 topic 详情数据，包括分区数据（如 leader、ISR 等）
 * @param client      AdminClient 客户端
 * @throws ExecutionException
 * @throws InterruptedException
 */
public static void describeTopics(AdminClient client) throws
ExecutionException, InterruptedException {
    DescribeTopicsResult ret = client.describeTopics(Arrays.asList
(TEST_TOPIC, "__consumer_offsets"));
    Map<String, TopicDescription> topics = ret.all().get();
    for (Map.Entry<String, TopicDescription> entry : topics.entrySet()) {
        System.out.println(entry.getKey() + " ==> " + entry.getValue());
    }
}

/**
 * 创建 topic
 * @param client      AdminClient 客户端
 */
public static void createTopics(AdminClient client) throws
ExecutionException, InterruptedException {
    NewTopic newTopic = new NewTopic(TEST_TOPIC, 3, (short)3);
    CreateTopicsResult ret = client.createTopics(Arrays.asList(newTopic));
    ret.all().get();
}

/**
 * 获取集群 topic 列表
 * @param client      AdminClient 客户端
 * @throws ExecutionException
 * @throws InterruptedException
 */
public static void listAllTopics(AdminClient client) throws
ExecutionException, InterruptedException {
    ListTopicsOptions options = new ListTopicsOptions();
    options.listInternal(true); // includes internal topics such as
__consumer_offsets

```

```

    ListTopicsResult topics = client.listTopics(options);
    Set<String> topicNames = topics.names().get();
    System.out.println("Current topics in this cluster: " + topicNames);
}
}

```

7.8 MirrorMaker

7.8.1 概要介绍

对于 Kafka 企业级用户而言，一个常见的痛点就是跨机房或跨数据中心（data center，DC）的数据传输。大型企业通常在多个数据中心部署 Kafka 集群。这里的数据中心可能是企业拥有的自建机房，也可能是公有云厂商的不同机房。在多个机房部署 Kafka 集群的优势如下。

- 实现灾备。
- 较近的地理位置可缩短延时以及用户响应时间。
- 实现负载均衡，即每个数据中心上的集群可能只保存部分数据集合并。
- 区别隔离不同优先级的数据处理。

图 7.4 给出了 MirrorMaker 实现灾备的框架图。

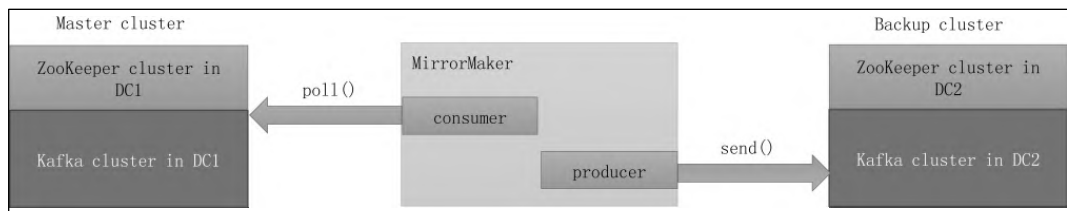


图 7.4 MirrorMaker 实现灾备的框架图

尽管部署方式各种各样，但跨机房部署方案有一个共同的特点：数据需要能够从一个 Kafka 集群被拷贝到另一个集群，而且还必须支持双向拷贝：某次传输的源集群（source cluster）可能是下次传输的目标集群（target cluster）。

为了实现这样的需求，Kafka 默认提供了一个工具 MirrorMaker，用来帮助用户实现数据在两个 Kafka 集群间的拷贝。就具体实现而言，MirrorMaker 仅仅是一个 consumer + producer 的混合物。对于源集群而言，它是一个 consumer；而对于目标集群而言，它又是一个 producer。MirrorMaker 读取源集群指定 topic 的数据，然后写入目标集群中的同名 topic 下。用户可以运行多个 MirrorMaker 实例增加整体数据拷贝的吞吐量，同时还提升了容错性。毕竟当一个实例

崩溃后，其他实例能够自动地承担起它的负载。

在实际生产环境中，源集群和目标集群是完全独立的两套环境：它们上的 topic 可能设置了不同的分区数且有不同的位移值。基于这个原因，MirrorMaker 工具并不能完美地实现容错性——因为 consumer 的位移值可能是不同的。不过，MirrorMaker 依然会保存并使用消息的 key 来执行分区任务。

7.8.2 主要参数

MirrorMaker 的脚本名是 kafka-mirror-maker.sh(bat)，位于 Kafka 安装目录的/bin 子目录下（Windows 平台下则位于 Kafka 安装目录的/bin/windows 子目录下）。表 7.8 给出了 0.11.0.0 版本的 MirrorMaker 脚本参数及其含义。

表 7.8 0.11.0.0 版本的 MirrorMaker 脚本参数及其含义

参 数 名	参 数 含 义
-- whitelist	指定一个正则表达式，指定拷贝源集群中的哪些 topic。比如 a b 表示拷贝源集群上两个 topic 的数据 a 和 b。注意，当使用新版本 consumer 时必须指定该参数
--blacklist	指定一个正则表达式，屏蔽指定 topic 的拷贝。注意，该参数只适用于老版本 consumer
--abort.on.send.failure	若设置为 true，当发送失败时则关闭 MirrorMaker
--consumer.config	指定 MirrorMaker 下 consumer 的属性文件，至少要在文件中指定 bootstrap.servers
--producer.config	指定 MirrorMaker 下 producer 的属性文件
--consumer.rebalance.listener	指定 MirrorMaker 使用的 consumer rebalance 监听器类
--rebalance.listener.args	指定 MirrorMaker 使用的 consumer rebalance 监听器的参数，与 consumer.rebalance.listener 一同使用
--message.handler	指定消息处理器类。消息处理器在 consumer 获取消息与 producer 发送消息之间被调用
--message.handler.args	指定消息处理器类的参数，与 message.handler 一同使用
--num.streams	指定 MirrorMaker 线程数。默认值是 1，即启动一个线程执行数据拷贝
--offset.commit.interval.ms	设定 MirrorMaker 位移提交间隔，默认值是 1 分钟
--help	打印帮助信息

在实际使用中，经常被使用的参数是 whitelist、consumer.config 和 producer.config。如果要实现某些特定的拷贝逻辑（比如拷贝指定分区数据或有选择性地拷贝数据等），那么就需要实现特定的消息处理器并使用 message.handler 进行指定。

一个典型的命令执行如下。该命令读取由 consumer.properties 文件指定的源 Kafka 集群，去读取名为 topicA 和 topicB 主题的消息，并写入到由 producer.properties 文件指定的目标 Kafka 集群。

```

    bin/kafka-mirror-maker.sh --consumer.config consumer.properties --producer.config producer.properties --whitelist topicA|topicB

```

7.8.3 使用实例

下面我们结合一个具体的实例来看看如何应用 MirrorMaker 工具在集群间进行数据拷贝。在本例中我们搭建 3 个测试 Kafka 集群，分别用 k1、k2 和 k3 表示。三套环境都是单节点的 Kafka 集群，端口号分别是 9092、9093 和 9094，在 ZooKeeper 上 chroot 分别是 /k1、/k2 和 /k3。

本例的目标是在 k1 上生产消息，然后使用 MirrorMaker 将消息拷贝到 k2 和 k3 上，最后在 k3 上运行 consumer 以验证消息是否被成功拷贝，如图 7.5 所示。

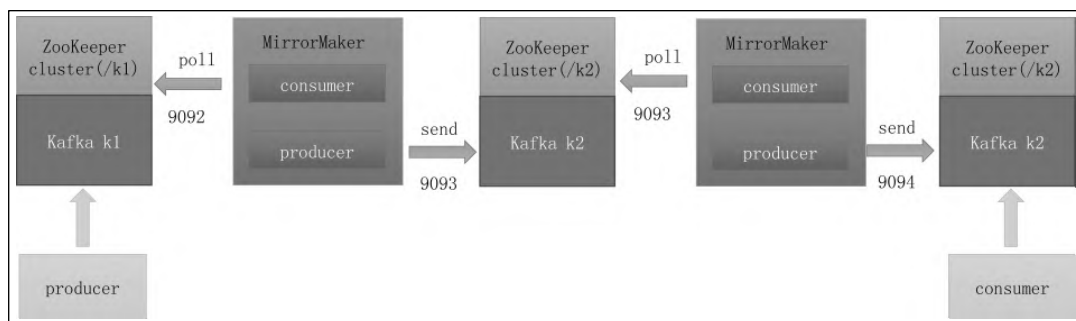


图 7.5 MirrorMaker 使用实例架构

现在开始搭建环境。分别设置三套环境使用不同的 ZooKeeper chroot，如下：

```
zookeeper.connect=localhost:2181/k1
zookeeper.connect=localhost:2181/k2
zookeeper.connect=localhost:2181/k3
```

然后分别启动 k1、k2 和 k3：

```
$ bin/kafka-server-start.sh ../config_files/server1.properties
$ bin/kafka-server-start.sh ../config_files/server2.properties
$ bin/kafka-server-start.sh ../config_files/server3.properties
```

现在在 k1 上创建一个测试 topic：

```
$ bin/kafka-topics.sh --create --zookeeper localhost:2181/k1 --topic
test --partitions 1 --replication-factor 1
Created topic "test".
```

随后接连启动两个 MirrorMaker 工具连接 k1 与 k2、k2 与 k3。首先创建两个 consumer.config 文件 mm-consumer-1st.config 和 mm-consumer-2nd.config：

```
bootstrap.servers=localhost:9092
client.id=mm1.k1Andk2
group.id=mm1.k1Andk2.consumer
partition.assignment.strategy=org.apache.kafka.clients.consumer.RoundRob
```

```
inAssignor

bootstrap.servers=localhost:9093
client.id=mm1.k2Andk3
group.id=mm1.k2Andk3.consumer
partition.assignment.strategy=org.apache.kafka.clients.consumer.RoundRob
inAssignor
```

然后创建两个对应的 `producer.config` 文件 `mm-producer-1st.config` 和 `mm-producer-2nd.config`:

```
bootstrap.servers=localhost:9093
client.id=mm1.k1Andk2

bootstrap.servers=localhost:9094
client.id=mm1.k2Andk3
```

下面分别启动两个 `MirrorMaker`:

```
$ bin/kafka-mirror-maker.sh --consumer.config mm-consumer-1st.config --
producer.config mm-producer-1st.config --whitelist test

$ bin/kafka-mirror-maker.sh --consumer.config mm-consumer-2nd.config --
producer.config mm-producer-2nd.config --whitelist test
```

做好这些之后，我们分别打开两个终端，在 `k1` 环境上使用 `console producer` 模拟生产一些消息，以及在 `k3` 环境上使用 `console consumer` 验证消息是否拷贝成功：

```
$ bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test
>hello
>another

$ bin/kafka-console-consumer.sh --bootstrap-server localhost:9094 --
topic test --from-beginning
hello
another
```

上述输出表明 `console producer` 生产的消息已经成功地从 `k1` 集群经由两个 `MirrorMaker` 拷贝到 `k3` 集群。

7.9 Kafka 安全

很多企业或组织对于安全性都有着很高的要求。在 0.9.0.0 版本之前，Kafka 并未提供任何形式的安全配置，用户只能通过粗粒度的网络配置“一刀切式”地限制对 Kafka 集群的访问。

随着对 Kafka 集群提供安全配置的呼声越来越高，社区终于在 0.9.0.0 版本正式添加了安全特性，并在 0.10.0.0 版本中进一步完善。目前 Kafka 包括的安全特性如下。

- 连接认证机制，包含服务器端与客户端（生产者/消费者）连接、服务器间连接以及服务器与工具间连接。支持的认证机制包括 SSL（TLS）或 SASL。
- 服务器与 ZooKeeper 之间连接的认证机制。
- 基于 SSL 的连接通道数据传输加密。
- 客户端读/写授权。
- 支持可插拔的授权服务和与外部授权服务的集成。

这里简单普及一下认证（authentication）和授权（authorization）的区别。认证就是证明你是谁的过程。当访问 Kafka 服务时你必须显式地提供身份信息来证明你的身份是合法的，而授权则是验证你能访问哪些服务的过程。举一个简单的例子，在一个配置了认证和授权的 Kafka 集群中，只有合法的生产者（已认证）才被允许向 Kafka 集群发送 PRODUCE 请求，而发给 broker 的 PRODUCE 请求是否会被真正处理则需要该生产者有对应的权限——通常生产者必须拥有主题写入的权限（topic write），即该生产者必须是已授权的。

鉴于认证和授权是最重要的安全配置且 Kafka 安全特性还在不断地演进中，本节将主要关注如何配置认证机制和基于 ACL 的授权机制，同时关于如何启用 SSL 信道加密，也会给出一个实例。

我们首先来看看如何开启认证机制和授权机制。

7.9.1 SASL+ACL

如前所述，自 0.9.0.0 版本引入安全配置之后，Kafka 一直在完善安全特性的功能。当前 Kafka 安全主要包含三大功能：认证（authentication）、信道加密（encryption）和授权（authorization），而其中的认证机制主要是指配置 SASL，而授权是通过 ACL 接口命令来完成的。

在生产环境中，用户若要使用 SASL，通常会配置 Kerberos，但对一些小公司而言，他们的用户系统并不复杂（即需要访问 Kafka 集群服务的用户数并不是很多），显然使用 Kerberos 有些大材小用，而且由于运行在内网环境，SSL 加密也不是很必要。因此一个基于明文传输（PLAINTEXT）的 SASL 集群环境足以应付一般的使用场景。本节我们将给出一个可运行的实例来演示一下如何在不使用 Kerberos 的情况下配置 SASL + ACL 以构建安全的 Kafka 集群。读者可以参考官网资料，很容易地将这个实例中的配置扩展到 Kerberos 使用场景中。

若要开启 SASL 和 ACL 机制，我们需要在 broker 端进行两个方面的设置。首先是创建包

含所有认证用户信息的 JAAS 文件。在本例中，我们使用 Kafka 最新版本 1.0.0，并假设有 3 个用户 admin、reader 和 writer，其中 admin 是集群管理员，reader 用户负责读取 Kafka 集群中的 topic 数据，而 writer 用户则负责向 Kafka 集群写入消息。为简单起见，我们假设这 3 个用户的密码分别与各自用户名相同（在实际场景中，管理员需要单独把密码发送给各用户），因此我们可以如下这样编写 JAAS 文件：

```
KafkaServer {  
    org.apache.kafka.common.security.plain.PlainLoginModule required  
        username="admin"  
        password="admin"  
        user_admin="admin"  
        user_reader="reader"  
        user_writer="writer";  
};
```

切记不要忘记这段配置中最后一行和倒数第 2 行结尾的分号！如果不写 JAAS 文件将被视为无效。现在我们保存这个文件为 jaas.conf，之后我们需要把这个文件的完整路径作为一个 JVM 参数传递给 Kafka 的启动脚本。不过由于 bin/kafka-server-start.sh 只接收 server.properties 的位置，不接收其他任何参数，故需要修改 Kafka 启动脚本，具体做法如下：

```
$ cd kafka_2.12-1.0.0/  
# 备份一份新的启动脚本，并命名为 secured-kafka-server-start.sh  
$ cp bin/kafka-server-start.sh bin/secured-kafka-server-start.sh
```

之后，修改新启动脚本，如下：

```
# 将该文件中的这一行：  
exec $base_dir/kafka-run-class.sh $EXTRA_ARGS kafka.Kafka "$@"  
# 修改为下面这行，然后保存退出  
exec $base_dir/kafka-run-class.sh $EXTRA_ARGS -Djava.security.auth.login.config=<你的路径>/jaas.conf kafka.Kafka "$@"
```

做完上面的步骤后，我们就在 bin/目录下做好了一份新的 Kafka 启动脚本。接下来开始修改 broker 启动所需的 server.properties 文件，你至少需要配置（或修改）以下这些参数：

```
# 配置 ACL 入口类  
authorizer.class.name=kafka.security.auth.SimpleAclAuthorizer  
# 本例使用 SASL_PLAINTEXT  
listeners=SASL_PLAINTEXT://:9092  
security.inter.broker.protocol= SASL_PLAINTEXT  
sasl.mechanism.inter.broker.protocol=PLAIN  
sasl.enabled.mechanisms=PLAIN  
# 设置本例中 admin 为超级用户  
super.users=User:admin
```

现在启动 broker 服务器：

```
$ bin/zookeeper-server-start.sh -daemon config/zookeeper.properties
$ bin/secured-kafka-server-start.sh config/server.properties
```

如果一切正常，你应该可以看到 Kafka 已成功启动的日志信息：

```
INFO [KafkaServer id=0] started (kafka.server.KafkaServer)
```

如果你发现 `unable to find LoginModule class: *****` 之类的报错，那么检查 `jaas.conf` 是否存在非法字符（比如中文字符等）。

此时，broker 端的认证已经开启且授权 ACL 接口也建立起来。下面开始为客户端开启认证。不过在开始之前，首先创建一个测试 topic 供后续使用。该 topic 名为 `test`，单分区，副本因子是 1：

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

topic 创建成功了！细心的读者可能会问，我们不是启用了 ACL 吗，为什么客户端（这里指广义客户端，包含工具脚本）还能成功创建 topic？有这样的疑问是正常的，因为当我们开启 ACL 后，理论上所有客户端程序和工具脚本都不能再访问任何 Kafka 资源（topic、集群信息、消费者组等），除非用户显式地启动 broker 前在 `server.properties` 中设置 `allow.everyone.if.no.acl.found=true`。

顺便提一句，`allow.everyone.if.no.acl.found` 参数是 broker 端参数，不过有些奇怪的是，它并未记录在官网的 broker 端参数列表中。当设置它为 `true` 时，整个 ACL 机制将改为黑名单机制，即只有在黑名单中的用户才无法访问资源，非黑名单用户可畅通无阻地访问任何 Kafka 资源；当参数为 `false` 时，也就是它的默认值时，ACL 机制是白名单机制，只有白名单用户才能访问设定的资源，其他任何用户都属于未授权用户。

既然我们未设置 `allow.everyone.if.no.acl.found`，也没有配置 ACL，那么为什么能够创建 topic 呢？这是因为当前 `kafka-topics.sh` 脚本是直接连接 ZooKeeper 的，完全绕过了 ACL 审查机制，故不受 ACL 的限制。所以无论是否配置了 ACL，用户总是可以使用 `kafka-topics` 来管理 topic。所以在实际使用过程中，最好对能连接 ZooKeeper 的用户也增加认证机制。

言归正传，本例中我们的目的是要测试是否用户 `writer` 向 `test` topic 写入消息以及用户 `reader` 从 `test` topic 读取消息。

首先启动一个 `console-consumer` 和一个 `console-producer` 来看看当前的状况：

```
$ bin/kafka-console-producer.sh --broker-list localhost:9092 --topic
test
>hello, kafka
```

```
[2017-11-22 10:20:52,003] WARN [Producer clientId=console-producer] Bootstrap
broker localhost:9092 (id: -1 rack: null) disconnected (org.apache.kafka.
clients.NetworkClient)
[2017-11-22 10:20:52,061] WARN [Producer clientId=console-producer]
Bootstrap broker localhost:9092 (id: -1 rack: null) disconnected (org.apache.
kafka.clients.NetworkClient)
[2017-11-22 10:20:52,119] WARN [Producer clientId=console-producer]
Bootstrap broker localhost:9092 (id: -1 rack: null) disconnected (org.apache.
kafka.clients.NetworkClient)
[2017-11-22 10:20:52,232] WARN [Producer clientId=console-producer]
Bootstrap broker localhost:9092 (id: -1 rack: null) disconnected (org.apache.
kafka.clients.NetworkClient)
.....
```

再启动一个消费者：

```
$ bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --
topic test --from-beginning
[2017-11-22 10:22:01,330] WARN [Consumer clientId=consumer-1, groupId=
console-consumer-52595] Bootstrap broker localhost:9092 (id: -1 rack: null)
disconnected (org.apache.kafka.clients.NetworkClient)
[2017-11-22 10:22:01,387] WARN [Consumer clientId=consumer-1, groupId=
console-consumer-52595] Bootstrap broker localhost:9092 (id: -1 rack: null)
disconnected (org.apache.kafka.clients.NetworkClient)
[2017-11-22 10:22:01,500] WARN [Consumer clientId=consumer-1, groupId=
console-consumer-52595] Bootstrap broker localhost:9092 (id: -1 rack: null)
disconnected (org.apache.kafka.clients.NetworkClient)
[2017-11-22 10:22:01,556] WARN [Consumer clientId=consumer-1, groupId=
console-consumer-52595] Bootstrap broker localhost:9092 (id: -1 rack: null)
disconnected (org.apache.kafka.clients.NetworkClient)
.....
```

可见，当前生产者和消费者都无法工作，报的错误就是无法连接 **broker**（localhost:9092）。这是因为我们开启了认证机制。下面我们为 **writer** 用户配置认证信息，首先需要创建一个属于用户 **writer** 的 **JAAS** 文件，该文件中指定了用户 **writer** 的连接信息，如下：

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

同样注意末尾的分号。现在把上述内容保存成 **writer_jaas.conf** 文件。和 **broker** 端类似，我们需要拷贝一份新的 **bin/kafka-console-producer.sh** 脚本，并将该 **JAAS** 文件作为一个 **JVM** 参数传给它：

```
$ cp bin/kafka-console-producer.sh bin/writer-kafka-console-producer.sh
$ vi bin/writer-kafka-console-producer.sh
# 把该文件中的这一行
exec $(dirname $0)/kafka-run-class.sh kafka.tools.ConsoleProducer "$@"
# 修改为下面这行，然后保存退出
exec $(dirname $0)/kafka-run-class.sh -Djava.security.auth.login.config=
<你的路径>/writer_jaas.conf kafka.tools.ConsoleProducer "$@"
```

现在使用新的 `console-producer` 脚本创建生产者：

```
bin/writer-kafka-console-producer.sh --broker-list localhost:9092 --topic
test --producer-property security.protocol=SASL_PLAINTEXT --producer-property
sasl.mechanism=PLAIN
```

注意命令结尾处的两个生产者属性 `security.protocol` 和 `sasl.mechanism`，需要分别设置它们为 `SASL_PLAINTEXT` 和 `PLAIN`。设置为 `SASL_PLAINTEXT` 表示开启 SASL，而由于本例不使用 Kerberos，故这里设置 `sasl.mechanism=PLAIN`。启动生产者后尝试输入一条消息，得到以下输出：

```
$ bin/writer-kafka-console-producer.sh --broker-list localhost:9092 --
topic test --producer-property security.protocol=SASL_PLAINTEXT --producer-
property sasl.mechanism=PLAIN
>hello, kafka
[2017-11-22 10:35:28,996] WARN [Producer clientId=console-producer] Error
while fetching metadata with correlation id 1 : {test=TOPIC_AUTHORIZATION_FAILED}
(org.apache.kafka.clients.NetworkClient)
[2017-11-22 10:35:28,997] ERROR Error when sending message to topic test
with key: null, value: 12 bytes with error: (org.apache.kafka.clients.producer.
internals.ErrorLoggingCallback)
org.apache.kafka.common.errors.TopicAuthorizationException: Not authorized
to access topics: [test]
```

依然有错误，不过错误变成“topic 授权失败”了。这说明我们运行的 `console producer` 通过了认证，但是没有通过授权，因此需要配置 ACL 来让用户 `writer` 有权限写入 `topic`。

在设置具体的 ACL 规则之前，首先简单学习一下 Kafka ACL 的格式。根据官网的介绍，Kafka 中一条 ACL 的格式为“Principal P is [Allowed/Denied] Operation O From Host H On Resource R”，含义描述如下。

- `principal`: 表示一个 Kafka user。
- `operation`: 表示一个具体的操作类型，如 `WRITE`、`READ`、`DESCRIBE` 等。完整的操作列表详见 <http://docs.confluent.io/current/kafka/authorization.html#overview>。
- `Host`: 表示连向 Kafka 集群的 client 的 IP 地址，如果是“*”则表示所有 IP。注意，当

前 Kafka 不支持主机名，只能指定 IP 地址。

- **Resource:** 表示一种 Kafka 资源类型。当前共有 4 种类型 TOPIC、CLUSTER、GROUP 和 TRANSACTIONID。

既然是生产者，我们需要赋给它对应 topic 的写入权限，故执行以下命令创建对应的 ACL 规则：

```
$ bin/kafka-acls.sh --authorizer kafka.security.auth.SimpleAclAuthorizer
--authorizer-properties zookeeper.connect=localhost:2181 --add --allow-
principal User:writer --operation Write --topic test
Adding ACLs for resource `Topic:test`:
    User:writer has Allow permission for operations: Write from hosts: *

Current ACLs for resource `Topic:test`:
    User:writer has Allow permission for operations: Write from hosts: *
```

再次尝试发送消息：

```
$ bin/writer-kafka-console-producer.sh --broker-list localhost:9092 --
topic test --producer-property security.protocol=SASL_PLAINTEXT --producer-
property sasl.mechanism=PLAIN
>hello,kafka
>this is a test message
>kafka security example
```

生产者发送消息成功了。下面我们来配置消费者，即用户 reader 的安全配置。和用户 writer 类似，首先创建 reader 用户的 JAAS 文件：

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

然后拷贝一份新的 console consumer 脚本来指定上面的 reader_jaas.conf：

```
$ cp bin/kafka-console-consumer.sh bin/reader-kafka-console-consumer.sh
$ vi bin/reader-kafka-console-consumer.sh
# 把该文件中的这一行
exec $(dirname $0)/kafka-run-class.sh kafka.tools.ConsoleConsumer "$@"
# 修改为下面这行，然后保存退出
exec $(dirname $0)/kafka-run-class.sh -Djava.security.auth.login.config
=<你的路径>/reader_jaas.conf kafka.tools.ConsoleConsumer "$@"
```

然后创建一个 consumer.config 为该 console producer 指定以下 3 个属性：

```
security.protocol=SASL_PLAINTEXT
```

```
sasl.mechanism=PLAIN
group.id=test-group
```

现在运行 **console consumer**:

```
$ bin/reader-kafka-console-consumer.sh --bootstrap-server localhost:9092
--topic test --from-beginning --consumer.config consumer.config
[2017-11-22 10:42:58,202] WARN [Consumer clientId=consumer-1, groupId=
test-group] Error while fetching metadata with correlation id 2 :
{test=TOPIC_AUTHORIZATION_FAILED} (org.apache.kafka.clients.NetworkClient)
[2017-11-22 10:42:58,205] ERROR Unknown error when running consumer:
(kafka.tools.ConsoleConsumer$)
org.apache.kafka.common.errors.TopicAuthorizationException: Not authorized
to access topics: [test]
```

又报错了，同样是 **topic** 授权失败，由于要读取 **topic** 数据，故我们需要赋予用户 **readertopic** 写入的 **ACL** 规则：

```
$ bin/kafka-acls.sh --authorizer kafka.security.auth.SimpleAclAuthorizer --
authorizer-properties zookeeper.connect=localhost:2181 --add --allow-principal
User:reader --operation Read --topic testc
Adding ACLs for resource `Topic:testc`:
User:reader has Allow permission for operations: Read from hosts: *

Current ACLs for resource `Topic:testc`:
User:reader has Allow permission for operations: Read from hosts: *
```

重新运行 **console consumer** 程序：

```
$ bin/reader-kafka-console-consumer.sh --bootstrap-server localhost:9092
--topic test --from-beginning --consumer.config consumer.config
[2017-11-22 10:47:56,892] ERROR Unknown error when running consumer:
(kafka.tools.ConsoleConsumer$)
org.apache.kafka.common.errors.GroupAuthorizationException: Not authorized
to access group: test-group
```

依然有错误，不过这次的错误变成了消费者组授权失败，这表明用户 **reader** 无权访问 **test-group** 消费者组——同样是授权的问题，我们还需要设置 **ACL** 规则来解决，即赋予用户 **reader** 消费者组的读权限：

```
$ bin/kafka-acls.sh --authorizer kafka.security.auth.SimpleAclAuthorizer
--authorizer-properties zookeeper.connect=localhost:2181 --add --allow-principal
User:reader --operation Read --group test-group
Adding ACLs for resource `Group:test-group`:
User:reader has Allow permission for operations: Read from hosts: *

Current ACLs for resource `Group:test-group`:
```

```
User:reader has Allow permission for operations: Read from hosts: *
```

再一次运行消费者程序:

```
$ bin/reader-kafka-console-consumer.sh --bootstrap-server localhost:9092
--topic test --from-beginning --consumer.config consumer.config
hello,kafka
this is a test message
kafka security example
```

这次 reader 用户成功地读取了用户 writer 生产的消息。此时用户 writer 和 reader 都可以正常地工作了。

最后我们使用 admin 用户查询消费者组的消费状态。同样地, 为 admin 用户创建对应的 JAAS 文件 admin_jaas.conf:

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

然后拷贝一份新的 kafka-consumer-groups.sh 脚本:

```
$ cp bin/kafka-consumer-groups.sh bin/admin-kafka-consumer-groups.sh
$ vi bin/admin-kafka-consumer-groups.sh
# 将该文件中的这一行
exec $(dirname $0)/kafka-run-class.sh kafka.admin.ConsumerGroupCommand
"$@"
# 修改为下面这行, 然后保存退出
exec $(dirname $0)/kafka-run-class.sh -Djava.security.auth.login.config=
<你的路径>/admin_jaas.conf kafka.admin.ConsumerGroupCommand "$@"
```

同时也需要为其设置以下两个参数, 将其保存到 admin_sasl.config 文件中:

```
security.protocol=SASL_PLAINTEXT
sasl.mechanism=PLAIN
```

然后运行 admin-kafka-consumer-groups.sh 脚本查询消费者组消费进度:

```
$ bin/kafka-admin-consumer-groups.sh --bootstrap-server localhost:9092 -
-group test-group --describe --command-config admin_jaas.config
Note: This will not show information about old Zookeeper-based consumers.

Consumer group 'test-group' has no active members.

TOPIC PARTITION CURRENT-OFFSET LOG-END-OFFSET LAG CONSUMER-ID HOST
CLIENT-ID
test 0 3 3 0 - - -
```

至此，所有用户都已经能正常地通过 SASL 认证机制及对应的 ACL 授权规则来访问特定的 Kafka 资源了。

7.9.2 SSL 加密

比起认证和授权，SSL 信道加密在很多生产环境甚至是一个必选项。为了避免在 Kafka 中明文传输敏感的公司业务数据，开启 SSL 就显得至关重要了。本节依然以一个实例的形式给出 Kafka 配置 SSL 的具体步骤。

总体来说，配置 SSL 的整体流程如图 7.6 所示。本例会为读者提供一个方便的 Shell 脚本（`setup_ssl_for_servers.sh`），自动地执行前 7 步的工作。这样用户实际上只需要做 3 件事即可。

- 运行 `setup_ssl_for_servers.sh` 脚本（本例使用这个名字）。
- 配置 broker 的 `server.properties`。
- 配置 clients 的特定属性。

关于 `setup_ssl_for_servers.sh` 脚本的实际内容，首先来看第一部分。

第一步：设置环境变量。

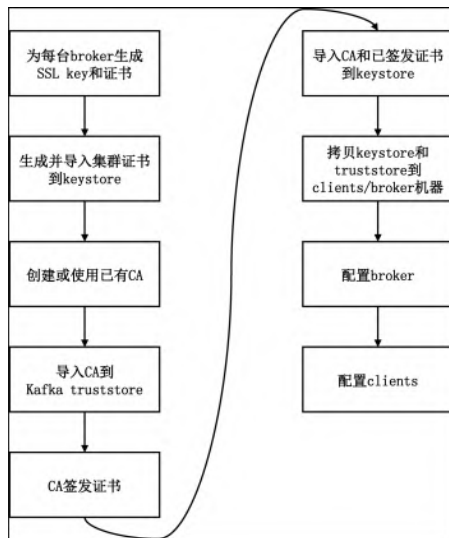


图 7.6 SSL 配置流程

```
#!/bin/bash
echo "1. 设置环境变量"
##### 设置环境变量 #####
BASE_DIR=/mnt/disk/kafka_ssl # SSL 各种生成文件的基础路径
CERT_OUTPUT_PATH="$BASE_DIR/certificates" # 证书文件生成路径
PASSWORD=kafka1234567 # 密码
KEY_STORE="$CERT_OUTPUT_PATH/kafka.keystore" # Kafka keystore 文件路径
TRUST_STORE="$CERT_OUTPUT_PATH/kafka.truststore" # Kafka truststore 文件路径
KEY_PASSWORD=$PASSWORD # keystore 的 key 密码
STORE_PASSWORD=$PASSWORD # keystore 的 store 密码
TRUST_KEY_PASSWORD=$PASSWORD # truststore 的 key 密码
TRUST_STORE_PASSWORD=$PASSWORD # truststore 的 store 密码
CLUSTER_NAME=test-cluster # 指定别名
CERT_AUTH_FILE="$CERT_OUTPUT_PATH/ca-cert" # CA 证书文件路径
CLUSTER_CERT_FILE="$CERT_OUTPUT_PATH/${CLUSTER_NAME}-cert" # 集群证书文件路径
DAYS_VALID=365 # key 有效期
D_NAME="CN=Xi Hu, OU=YourDept, O=YourCompany, L=Beijing, ST=Beijing,
```


Apache Kafka 实战

```
C=CN" # distinguished name
#####

mkdir -p $CERT_OUTPUT_PATH
```

第二步：创建集群证书到 keystore。

```
echo "2. 创建集群证书到 keystore....."
keytool -keystore $KEY_STORE -alias $CLUSTER_NAME -validity $DAYS_VALID
-genkey -keyalg RSA \
-storepass $STORE_PASSWORD -keypass $KEY_PASSWORD -dname "$DNAME"
```

第三步：创建 CA。

```
echo "3. 创建 CA....."
openssl req -new -x509 -keyout $CERT_OUTPUT_PATH/ca-key -out "$CERT_AUTH_
FILE" -days "$DAYS_VALID" \
-passin pass:"$PASSWORD" -passout pass:"$PASSWORD" \
-subj "/C=CN/ST=Beijing/L=Beijing/O=YourCompany/CN=Xi Hu"
```

第四步：导入 CA 文件到 truststore。

```
echo "4. 导入 CA 文件到 truststore....."
keytool -keystore "$TRUST_STORE" -alias CARoot \
-import -file "$CERT_AUTH_FILE" -storepass "$TRUST_STORE_PASSWORD" -
keypass "$TRUST_KEY_PASS" -noprompt
```

第五步：从 keystore 中导出集群证书。

```
echo "5. 从 key store 中导出集群证书....."
keytool -keystore "$KEY_STORE" -alias "$CLUSTER_NAME" -certreq -file
"$CLUSTER_CERT_FILE" -storepass "$STORE_PASSWORD" -keypass "$KEY_PASSWORD" -
noprompt
```

第六步：签发证书。

```
echo "6. 签发证书....."
openssl x509 -req -CA "$CERT_AUTH_FILE" -CAkey $CERT_OUTPUT_PATH/ca-key
-in "$CLUSTER_CERT_FILE" \
-out "${CLUSTER_CERT_FILE}-signed" \
-days "$DAYS_VALID" -CAcreateserial -passin pass:"$PASSWORD"
```

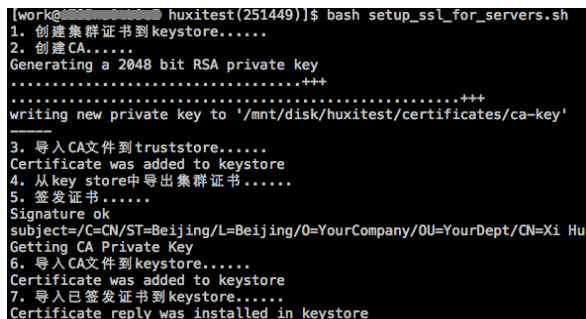
第七步：导出 CA 文件到 keystore。

```
echo "7. 导入 CA 文件到 keystore....."
keytool -keystore "$KEY_STORE" -alias CARoot -import -file "$CERT_AUTH_
FILE" -storepass "$STORE_PASSWORD" \
-keypass "$KEY_PASSWORD" -noprompt
```

第八步：导入已签发证书到 keystore。

```
echo "8. 导入已签发证书到 keystore....."
keytool -keystore "$KEY_STORE" -alias "${CLUSTER_NAME}" -import -file
"${CLUSTER_CERT_FILE}-signed" \
-storepass "$STORE_PASSWORD" -keypass "$KEY_PASSWORD" -noprompt
```

接下来，我们在 broker 机器上执行该 Shell 脚本（SSL 配置脚本），如图 7.7 所示。



```
[work@huxitest(251449)]$ bash setup_ssl_for_servers.sh
1. 创建集群证书到 keystore.....
2. 创建CA.....
Generating a 2048 bit RSA private key
.....+++
writing new private key to '/mnt/disk/huxitest/certificates/ca-key'
3. 导入CA文件到truststore.....
Certificate was added to keystore
4. 从key store中导出集群证书.....
5. 签发证书.....
Signature ok
subject=C=CN/ST=Beijing/L=Beijing/O=YourCompany/OU=YourDept/CN=Xi Hu
Getting CA Private Key
6. 导入CA文件到 keystore.....
Certificate was added to keystore
7. 导入已签发证书到 keystore.....
Certificate reply was installed in keystore
```

图 7.7 执行 SSL 配置脚本

现在到对应的目录下检查生成的文件列表。

- ca-cert: CA 文件，不要把该文件拷贝到别的 broker 机器上。
- test-cluster-cert-signed: CA 已签发的 Kafka 证书文件，不要把该文件拷贝到别的 broker 机器上。
- test-cluster-cert: Kafka 认证文件（包含公钥和私钥），不要把该文件拷贝到别的 broker 机器上。
- kafka.keystore: Kafka 的 keystore 文件，所有 clients 端和 broker 机器上都需要。
- kafka.truststore: Kafka 的 truststore 文件，所有 clients 端和 broker 机器上都需要。

脚本执行成功后，我们还需要对相应的 Kafka 参数进行设置，与 SSL 相关的部分如下。

- listeners=PLAINTEXT://:9092,SSL://:9093 # 这里为 Kafka broker 配置了两个 listeners，一个是明文传输，另一个使用 SSL 加密进行数据传输。连接 9093 端口则会进行 SSL 加密。
- advertised.listeners=PLAINTEXT://公网 IP:9092,SSL://公网 IP:9093 # 如果是云上环境，即 clients 通过公网（或外网）去连接 broker，那么 advertised.listeners 就必须配置成所在机器的公网 IP。
- ssl.keystore.location=/mnt/disk/huxitest/certificates/kafka.keystore # 提供 SSL keystore 的文件。
- ssl.keystore.password=kafka1234567 # 提供 keystore 密码。
- ssl.truststore.location=/mnt/disk/huxitest/certificates/kafka.truststore # 提供 SSL truststore 的

文件。

- `ssl.truststore.password=kafka1234567` # 提供 truststore 密码。
- `ssl.key.password=kafka1234567` # keystore 中的私钥密码。
- `ssl.client.auth=required` # 设置 clients 也要开启认证。

配置好 broker 后，我们可以尝试启动 broker。如果一切正常，broker 应该会正常启动。之后创建测试 topic——test:

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

至此，broker 端的 SSL 配置就做完了。为了让客户端程序使用 SSL 信道给 broker 端发送请求，我们还需要配置客户端程序。

先来创建一个生产者，依然使用 Kafka 自带的 `kafka-console-producer.sh` 脚本。首先为其创建一个 `producer.config` 文件，内容如下：

```
bootstrap.servers=kafka1:9093    # 指定 9093 端口，即使用 SSL 监听器端口
security.protocol=SSL
ssl.truststore.location=<你的路径>/kafka.truststore # 指定 truststore 文件
ssl.truststore.password=kafka1234567
ssl.keystore.password=kafka1234567
ssl.keystore.location=<你的路径>/kafka.keystore # 指定 keystore 文件
```

然后运行脚本开始发送消息：

```
$ bin/kafka-console-producer.sh --broker-list kafka1:9093 --topic test -
-producer.config producer.config
>hello, world
>hello, Kafka
>a test message
```

注意此生产者连接的端口号是 9093。如果一切正常，消息发送应该是成功的。下面使用 `kafka-console-consumer` 脚本来验证消费者。同理，我们也需要创建一个 `consumer.config` 文件：

```
security.protocol=SSL
group.id=test-group
ssl.truststore.location=<你的路径>/kafka.truststore # 指定 truststore 文件
ssl.truststore.password=kafka1234567
ssl.keystore.password=kafka1234567
ssl.keystore.location=<你的路径>/kafka.keystore # 指定 keystore 文件
```

然后创建消费者来验证：

```
$ bin/kafka-console-consumer.sh --bootstrap-server kafka1:9093 --topic
```

```
test --from-beginning --consumer.config consumer.config
hello, world
hello, Kafka
a test message
```

可见，SSL 设置成功了。目前客户端和服务端通信皆使用 SSL 进行加密。值得一提的是，启用了 SSL 之后 Kafka 的吞吐量（特别是客户端）通常有 10%~40% 的下降，因此用户需要权衡 SSL 和性能之间的收益。

7.10 常见问题

(1) UnknownTopicOrPartitionException

```
org.apache.kafka.common.errors.UnknownTopicOrPartitionException: This server
does not host this topic-partition.
```

该异常表示请求的分区数据不在抛出该异常的 broker 上。Kafka 抛出该异常的原因通常有如下 3 个。

- follower 副本所在的 broker 在另一个 broker 成为 leader 之前率先完成了成为 follower 的操作，使得 follower 从 leader 拉取数据时发现 leader broker 上还未准备好数据，从而抛出 UnknownTopicOrPartitionException。不过该问题通常会在下一轮 RPC 中自动恢复。
- producer 向不存在的 topic 发送数据时，broker 会封装该异常返回给 producer。不过由于该异常属于可重试类异常（RetriableException），因此在打开“自动创建 topic 开关”的前提下，producer 端会自动创建 topic，然后重试发送请求。若用户持续地在 producer 端看到该异常，检查 auto.create.topics.enable 参数，另外也可适当地增大 retries 值。
- 当启用 ACL 后，Kafka 对于未授权操作中的 topic 一律返回该异常，而非“无权访问”之类的错误。这主要是基于不对外暴露 topic 是否存在的考虑。

总而言之，UnknownTopicOrPartitionException 异常属于可重试异常，通常是能够自动修复的，所以偶尔在日志中看到这个警告也不用太过在意。但若持续地观测到该异常则需要进行相应的处理。若是 broker 端抛出，则检查对应 topic 的分区部署情况（使用 kafka-topics 脚本）；若是 clients 端抛出，则检查 clients 端代码——查看是否向不存在的 topic 发送了请求。

(2) LEADER_NOT_AVAILABLE

```
LEADER_NOT_AVAILABLE: There is no leader for this topic-partition as we
are in the middle of a leadership election.
```

顾名思义，就是对应分区当前没有 leader。没有 leader 的原因可能有很多，比如正在进行新 leader 的选举，抑或是该 topic 正在被删除，使得 leader broker 是-1，即无 leader。在实际环境中，该异常通常都是瞬时出现的，比如当 producer 向一个不存在的 topic 发送消息时，若 `auto.create.topics.enable=true`，broker 会自动创建该 topic，然后向 clients 端返回 `LEADER_NOT_AVAILABLE` 异常，显式告知 producer 要重新请求最新的元数据，因此在 producer 端就会瞬时地抛出若干条 `LEADER_NOT_AVAILABLE` 警告。不过一旦 producer 刷新了元数据，该异常警告就会消失。

不过若用户持续地发现该异常，则通常需要使用 `kafka-topics` 脚本检查分区的 leader 信息。如果 leader 是-1，那么可以考虑使用 `kafka-preferred-replica-election.sh` 手动调整一下 leader 选举。如果依然不好用，则需要检查 controller broker 的存活情况。

（3）NotLeaderForPartitionException

```
NotLeaderForPartitionException: This server is not the leader for that topic-partition.
```

该异常主要是指当前 broker 已不是对应分区的 leader broker，这通常发生在 leader 变更的情况下。当 leader 从一个 broker 切换到另一个 broker 时，原有的 clients 或 follower 依然有可能在向老的 leader 请求数据。在这种情况下，Kafka 就会抛出这个异常。

和之前的异常类似，该异常通常都应该是瞬时的。如果用户持续地观测到该异常，就需要查询 broker 端和 clients 端的日志来进一步定位问题。

（4）TimeoutException

```
TimeoutException: ****
```

在生产环境下，`TimeoutException` 异常十分常见，它表示的就是请求超时，而请求可以是各种类型的请求，比如请求元数据信息或者生产消息请求。若碰到该异常，有一个很简单的解决办法。用户需要定位该异常是哪里抛出来的，比如是从 producer 端，broker 端，还是 consumer 端？是哪里抛出的就增加哪里 `request.timeout.ms` 参数的值。这种方法通常能很好地规避这一问题。若增加 `request.timeout.ms` 值依然不管用，则需要考虑用户环境中的 broker 或 clients 是否负载过重，导致任务堆积不能被处理，这就可能需要从架构方面来进行改造。

（5）RecordTooLargeException

在实际生产环境中该异常多见于 producer 端，通常是因为 producer 应用的后台发送线程无法匹配用户主线程的消息创建速率。解决的思路有两个：第一就是在出现该异常后尽量避免共享相同的 producer 实例；其次就是适当增加 `request.timeout.ms` 以及适当减少 `batch.size`。

当 producer 端无法从 Kafka 集群获取元数据时，也会抛出这个异常，特别是对那些未正确配置连接信息的 producer 而言。此时用户需要仔细检查 bootstrap.servers 连接设置，查看是否存在无法连接的情况。另外如果 Kafka broker 搭建在云环境主机上，通常都需要设置 broker 端参数 advertised.listeners。

```
RecordTooLargeException: The request included a message larger than the
max message size the server will accept.
```

顾名思义，该异常“抱怨”的是消息过长导致 Kafka 无法处理。当前若要让 Kafka 集群处理大消息，总共有 3 个参数需要调整。

- broker 端参数 message.max.bytes：设置 broker 端能处理的最大消息长度。
- producer 端参数 max.request.size：设置 producer 端能处理的最大消息长度。
- consumer 端参数 fetch.max.bytes（新版本）、fetch.message.max.bytes（旧版本）：设置 consumer 端能处理的最大消息长度。
- broker 端参数 socket.request.max.bytes：设置 broker 端 Socket 请求的最大字节数。通常用户不需要额外配置该参数，但如果向 Kafka 发送超过 100MB 的超大消息，则必须要调整该参数的值。

当前，这些参数的默认值（socket.request.max.bytes 的默认值是 100MB）大约都在 1MB 左右，对于有些生产环境的大消息而言，用户需要同时调整上面这些参数。

（6）NetworkException

```
NetworkException: The server disconnected before a response was received.
```

这个异常是 producer 端抛出的，主要是因为 producer 在工作过程中断开了与某些 broker 的连接，从而使得发送到这些 broker 的 PRODUCE 请求失败。此时 producer 会抛出该异常。

NetworkException 是可重试的异常，因此通常情况下它都是瞬时出现的，但若用户在 producer 端持续地观测到了该异常，则需要检查 producer 与对应 broker 节点的连通性以及 broker 节点的存活情况。

（7）ILLEGAL_GENERATION

```
ILLEGAL_GENERATION: Specified group generation id is not valid.
```

这是新版本 consumer 抛出的异常，表明当前 consumer 错过了 consumer group 正在进行的 rebalance，原因是该 consumer 花费了大量的时间处理 poll()返回的数据。用户需要适当减少 max.poll.records 值以及（或）增加 max.poll.interval.ms 值。对于没有这两个参数的老版本 Kafka 而言，则需要减少 max.partition.fetch.bytes 参数的值。当然优化消息处理的逻辑也是不错的办法——比如把 poll 回来的消息放入单独的线程进行处理。

7.11 本章小结

本章我们详细探讨了 Kafka 集群的各种管理方法，包括 Kafka 自带的各种工具脚本以及如何使用 API 编程的方式对 Kafka 集群进行管理。另外针对常见的线上问题我们也给出了简要的问题原因定位以及解决办法。

相信各位读者对于 Kafka 集群管理已经有了一定的了解。第 8 章我们将学习如何监控 Kafka 集群，包括如何使用自带的工具脚本以及利用第三方的监控框架。



第 8 章

监控 Kafka 集群

对于运维人员而言，维护生产环境的一个重要内容就是监控集群的运行与状态。当出现问题时，系统管理员能做的只是从海量的日志中尝试定位异常行为发生的根本原因。这种排查问题的方法通常是低效率的，因此需要我们引入完善的监控指标以及框架来帮助管理 Kafka 集群。

本章将重点讲解 Apache Kafka 集群监控指标以及如何监控 Kafka 集群，同时也会讨论如何利用这些指标诊断问题以及目前主流的第三方图形化监控框架的使用方法。值得注意的是，如无显式说明，本章将结合 1.0.0 版本的 Kafka 来讲述如何对 Kafka 进行监控。

学习本章，你将了解到以下内容。

- 集群健康度检查。
- MBean 监控。
- broker 端 JMX 监控。
- clients 端 JMX 监控。
- OS 监控。
- 主流监控框架。

8.1 集群健康度检查

在深入查看一个 Kafka 生产集群环境的各个组件之前，运维人员有必要在全局对集群的健康度检查高屋建瓴般地做一个详尽的“顶层设计”。笔者认为，对于任何一个生产集群而言，以下几个方面都是应该重点关注的内容。

- 所有 broker 的执行状态，包括运行状态、所属版本、底层日志路径磁盘使用情况、所

在机器的物理负载情况、系统日志是否有严重错误等。

- ZooKeeper 运行状态，包括版本、底层文件系统使用情况（特别是快照所在磁盘空间使用）、所在机器物理负载情况等。
- 集群中所有主题（topic）分布以及分区状态，包括所有 topic 的分区情况以及每个分区 leader 副本的存活情况等。
- 客户端应用（clients application）运行状态，包括客户端应用负载分析、有无消费者（consumer 端）消费滞后、有无生产者（producer 端）超时等。
- 版本匹配性，全面了解集群中所有 clients 端应用程序 API 版本与 broker 端版本的适配性。
- 集群中定时作业的运行状态，全面了解当前集群中有哪些大的定时作业（如 preferred leader 选举）或当前正在手动执行哪些耗时作业（如分区重分配）。

上述内容需要系统管理员在每天的运维管理任务中逐条地进行核实和校验，以确保生产环境不会出现严重的问题。下面就来详细地展开介绍。

8.2 MBean 监控

如官网所述，Kafka 使用基于 yammer metrics 的监控指标体系来统计 broker 端和 clients 端的各种监控指标（metric）。说到 yammer metrics，其官网给出了这样的一句话：

yammer metrics 是一个 Java 库，它使得你能够对生产环境代码所做之事具有无与伦比的洞察力。

当前，虽然该项目已经进化到 3.x 版本，但即使是最新版本的 Kafka 依然在使用 2.2 版本的 yammer metrics。不过这并不会妨碍我们对其使用方法的讨论。

8.2.1 监控指标

目前，Kafka 默认提供了超多的监控指标。无一例外，用户皆使用 JMX 接口访问这些指标。JMX，即 Java Management Extension，Java 管理扩展，是一套为各种应用程序、设备或系统等植入管理功能的框架。JMX 本身是跨平台的，因此具有高度的灵活性，可以无缝集成进各种系统中。

虽然有一小部分功能是以 Shell 脚本或直接运行 Java 类来提供的，但是 Kafka 集群大部分的运行表现都可由 JMX 指标来表征，因此搭建一套能够访问 JMX 的监控框架或系统对于监控 Kafka 集群而言至关重要。

Kafka 的每个监控指标都是以 JMX MBean 的形式定义的。虽然 JMX 规范不在本书的讨论范围内，但这里依然简单介绍一些 MBean 的概念。MBean 代表一个被管理的资源实例，它表示管理资源的 Java 对象。每个 MBean 的管理接口如下。

- 属性值。
- 能够执行的操作。
- 能发出的通知事件。
- 构建器。

对于 Kafka 而言，我们只关心上述列表中的第一项，即 MBean 的属性值，它包含了 Kafka 各种监控指标的真实值。MBean 需要注册之后才能使用，在注册时必须指定 MBean 的名称。在 Kafka 中，所有 MBean 的命名规范有着统一的格式 `xxx.type=xxx,{attr}=xxx`。第一个等式中的 xxx 通常表示所属的 Kafka 组件，比如 `kafka.server`、`kafka.producer` 或 `kafka.consumer` 等；第二个等式中的 xxx 和前面的 attr 通常表示该 MBean 的范围，比如 `topic=test` 表示该 MBean 的作用范围是名为 test 的 topic，即它是一个 topic 级别的 MBean，表征该 topic 下的某个性能指标。

对于 Java 用户而言，访问 JMX 接口最简单的方式就是使用 JDK 自带的 JConsole 工具，当然用户也可以使用其他外部的监控系统。无论使用哪种工具，用户至少需要指定连接 JMX 的 IP 和端口。图 8.1 给出了 JConsole 启动画面，我们需要指定 Kafka broker 和 clients 应用的 IP 地址和 JMX 端口信息。本例中我们监控本机运行的 broker，JMX 端口是 9997。



图 8.1 JConsole 启动画面

图 8.2 展示了成功连入 Kafka broker JMX 接口后查询测试 topic test1 的 LEO 指标（LEO 表示 Log End Offset，是该 topic 指定分区当前日志的末端位移）。图 8.2 中显示该 JMX 指标值是 5000000，即 Kafka 向该 topic 的分区 0 生产了 500 万条消息。

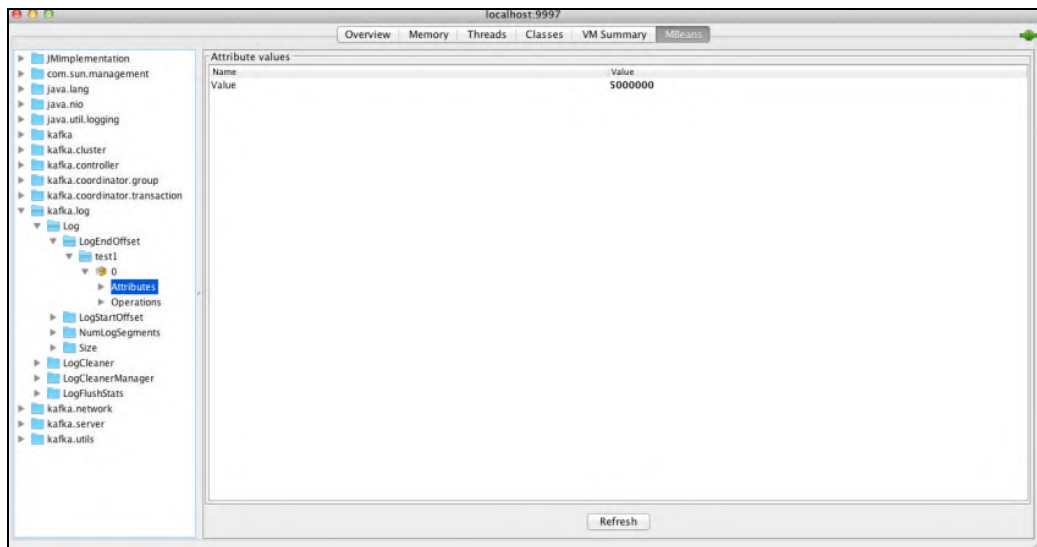


图 8.2 在 JConsole 中查询 JMX 指标

8.2.2 指标分类

如前所述，Kafka JMX MBean 的命名规范中第一项即指定该 MBean 所属的 Kafka 组件。截止到最新的 1.0.0 版本，用于 JMX 监控的指标共有 5 大类，它们分别如下。

- Kafka broker 端指标，具体又可细分为如下几个指标。
 - kafka.server: 服务器端 JMX 指标。
 - kafka.network: 网络相关 JMX 指标。
 - kafka.log: 分区日志相关 JMX 指标。
 - kafka.Controller: controller 相关 JMX 指标。
- clients 端常用指标：包括了所有 clients 端公共的一些指标。这里的 clients 包括 producer、consumer、connect 和 streams。后两者是 Kafka 中新加入的用于流式数据处理的组件。我们会在第 10 章中详细介绍它们。
- producer 端指标：定义了 Java 版本 producer API 调用过程中的各种 JMX 统计指标，MBean 命名通常以 kafka.producer 开头。
- consumer 端指标：定义了 Java 版本 consumer API 调用过程中的各种 JMX 统计指标，MBean 命名通常以 kafka.consumer 开头。值得注意的是，老版本的 consumer 也有一些 JMX 指标，MBean 命名也是以 kafka.consumer 开头的。
- Streams 端指标：定义了 Kafka Streams 的 JMX 指标，MBean 命名以 kafka.streams 开头。

在实际使用过程中，最常用的就是前 4 类指标。用户需要经常查询官网中各类指标的定义列表，获取其对应的 MBean 名称，然后在监控框架中实时地对它们进行观测。

8.2.3 定义和查询 JMX 端口

若要查询 JMX 指标，必须首先连接到 JMX 接口。在连接过程中用户需要指定连接的 IP 地址以及对应的端口信息。需要特别注意的是，这里的端口信息不是 Kafka broker 的启动端口，也不是 clients 应用运行所需的各种端口，而是专门为访问 JMX 接口而设置的 JMX 端口，而且在启动 broker 和 clients 端应用时用户需要提前设置好这个端口。

具体设置方法是在启动 broker 和 clients 应用之前设置 JMX_PORT 环境变量。如下面的代码中在启动 broker 之前我们设置 JMX 端口为 9997：

```
JMX_PORT=9997 bin/kafka-server-start.sh config/server.properties
```

用户可将上面这行代码写入一个 Shell 脚本中，然后使用该脚本启动 Kafka broker，这样 broker 在启动时就已经成功地设置了 JMX 端口。

对于一个陌生的 Kafka 集群而言，用户如何能够确定该 Kafka broker 在启动时是否设置了 JMX 端口呢？实际上，broker 在启动过程中始终会将 JMX 端口信息连同其他 broker 信息都写入 ZooKeeper 中的对应位置，即/brokers/ids/<broker ID>节点中。图 8.3 和图 8.4 分别给出了 ZooKeeper 中保存的两个 broker 启动信息，一个未指定 JMX 端口，另一个指定了 9997 作为 JMX 端口。

```
[zk: localhost:2181(CONNECTED) 3] get /brokers/ids/0
{"listener_security_protocol_map":{"PLAINTEXT":"PLAINTEXT"},"endpoints":["PLAINTEXT://bogon:9092"],"jmx_port":-1,
"host":"bogon","timestamp":"1505099469239","port":9092,"version":4}
```

图 8.3 未指定 JMX 端口的 broker

```
{"listener_security_protocol_map":{"PLAINTEXT":"PLAINTEXT"},"endpoints":["PLAINTEXT://bogon:9092"],"jmx_port":9997,
"host":"bogon","timestamp":"1505097351994","port":9092,"version":4}
```

图 8.4 指定了 JMX 端口的 broker

如图 8.3 所示，如果未指定 JMX 端口，ZooKeeper 中保存的 jmx_port 值是 -1。通过这个信息，用户就能判断出集群中 broker 是否设置了 JMX 端口。

需要说明的是，上面两张图的 broker 信息格式是最新版本 Kafka 的 broker JSON 格式。对于老版本 Kafka 而言，它保存的信息没有目前这么丰富，但 jmx_port 信息是一直存在的。

8.3 broker 端 JMX 监控

Kafka 提供了很多 broker 端的 JMX 监控指标，完整的列表可参见官网 <https://kafka.apache.org/documentation/#monitoring>。本节我们将重点介绍一些常用的 broker 端 JMX 指标。

在和社区开发人员的交流中，他们告诉笔者目前提供的每个监控指标几乎都是他们在开发过程中“踩过的坑”，里面充满了 Kafka 代码调试和 bug 修复的“血泪史”，由此可见这些监控指标的分量。在实际生产环境中，用户一定要对 broker 端 JMX 指标做监控，特别是本节要详细展开的常见 metrics。

8.3.1 消息入站/出站速率

消息入站/出站速率对应的 Mbean 名称如表 8.1 所示，它们的单位都是字节/秒，分别表示 Kafka broker 每秒接收/发送的消息字节数。

所谓接收消息，即 broker 接收来自 producer 端发送的消息或 follower broker 接收来自 leader broker 发送的消息（严格来说，是 follower broker 主动向 leader 拉取消息）；而发送消息则是指 leader broker 需要将消息备份到 follower broker。

表 8.1 消息入站/出站速率 MBean

JMX 指标	MBean 名称
Bytes in rate（消息入站速率）	kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec
Bytes out rate（消息出站速率）	kafka.server:type=BrokerTopicMetrics,name=BytesOutPerSec

这两个 MBean 都支持 topic 级别的统计，即为当前 broker 处理的各个 topic 分别计算入站速率和出站速率。若要单独查询某个 topic 的消息速率，需要将 topic 名称加入表 8.1 中的 MBean 名称中。假设用户查询名为 test 的 topic 的入站速率，则需要指定 MBean 名为 kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec,topic=test。另外，Kafka 为这两组 MBean 定义了多个属性，如表 8.2 所示。

表 8.2 消息入站/出站速率 MBean 属性

属 性 名	属 性 含 义
Count	计算该 broker 处理过的总消息字节数
EventType	值固定是字符串“bytes”，表明该 MBean 计算的数值类型是字节数
FifteenMinuteRate	统计过去 15 分钟内的消息速率
FiveMinuteRate	统计过去 5 分钟内的消息速率
MeanRate	统计平均消息速率
OneMinuteRate	统计过去 1 分钟内的消息速率
RateUnit	定义该 MBean 的时间单位，值是 SECONDS，即秒

在实际生产环境中，用户需要实时观测***Rate 的属性值，如果发现消息速率（不论是入站速率还是出站速率）接近 broker 所在机器的网络带宽，则这通常表明该 broker 可能是一个瓶颈，应考虑是否将该 broker 的负载转移到其他 broker 上。在第 9 章中将会详细讨论关于如何调优 Kafka 集群。图 8.5 使用 JConsole 监控笔者测试集群中某台 broker 的消息速率。

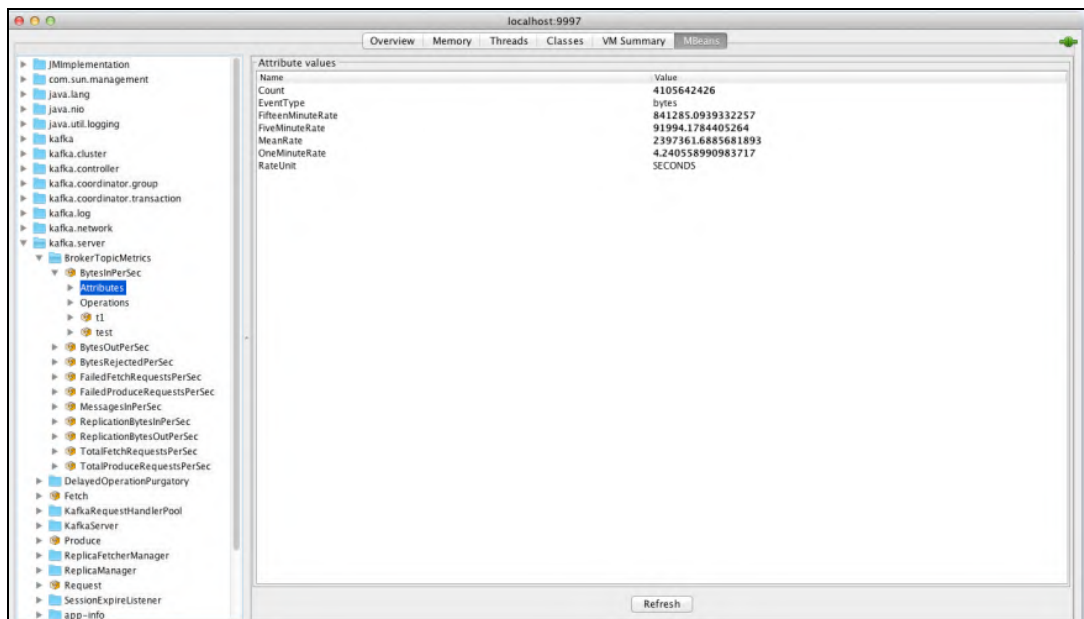


图 8.5 测试 broker 的消息速率 JMX 指标

如图 8.5 所示，在笔者的测试环境中，这台 broker 总共接收到的 topic 字节数为 5498992362 字节，大约为 5.2GB 左右，而平均的消息入站速率是 10.3MB/s（10778190 字节），即 83Mb/s，远低于 1Gb/s 的带宽（假设是千兆位网卡），故笔者测试环境中这台 broker 的网络负载还远远没有饱和，有很大的提升空间。

很多用户在实际监控过程中可能会发现消息出站速率是消息入站速率的好几倍。这是正常情况，因为对于一台 broker 而言，假设它是某个分区的 leader，而该 topic 的副本因子是 3，那么就意味着该 broker 接收到 producer 发来的一条消息后需要将该条消息发送到其他两台 broker 上，故统计出来的 BytesOutRate 指标值的确有可能是 BytesInRate 指标值的数倍。

8.3.2 controller 存活 JMX 指标

前面章节中介绍了 Kafka controller 在整个集群中的作用与地位。一旦 controller 挂掉了，整个集群的运行都会受到非常严重的影响，因此在实际生产环境中一定要实时监控 controller

的存活状态。

controller 的 JMX 指标中表征 controller 是否存活的 JMX 指标是 `kafka.controller:type=KafkaController,name=ActiveControllerCount`，即当前集群中 active controller 的数量，该 MBean 值应该始终等于 1 才是正常情况，毕竟在目前 Kafka 的设计中任意时刻只允许有一个运行中的 controller。一旦长时间发现该值为 0，则表示 controller 已被关闭且未能自行进行选举。这通常都是很严重的故障，需要用户马上介入处理，因为 controller 无法再处理任何的 broker 间请求以及用户发起的管理任务。此时，查看 controller 所在 broker 的 controller 日志（`controller.log` 和 `state-change.log`）来定位问题。若一时无法找出根本原因，可以先行恢复 controller，有如下两种方法。

- 删除 ZooKeeper 节点/controller：删除此 znode 可触发 controller 重新选举。
- 重启 controller 所在 broker：若上面的方法失效，则可考虑使用该方法。不过通常对业务都有较大冲击，实际使用时需要谨慎考虑影响。

8.3.3 备份不足的分区分数

笔者认为这是最重要的 broker 端 JMX 指标。该指标给出了当前 broker 上分区 leader 副本备份不足的程度。所谓备份不足（under replicated）的分区是指那些当前 ISR 中副本数少于副本因子的分区。举一个例子，假设一个 topic 的副本因子是 3，有 10 个分区。若某一时刻，有 5 个分区的 ISR 副本数是 2，则该 topic 上副本不足的分区分数就是 5。只不过该 MBean 统计的是整个集群上所有 topic 的情况。值得注意的是，该指标的统计规则有两个条件，除了计算备份不足之外，还要保证统计的分区的 leader 副本就位于该 broker 上才行。这就是说，如果该 broker 上某个分区的 follower 副本备份不足，那么这个分区是不会计入此 broker 对应的 MBean 值的。

此 MBean 的名称是 `kafka.server:type=ReplicaManager,name=UnderReplicatedPartitions`。在实际生产环境中，此值应该越小越好，0 是最好的情况。一旦发现此值长时间大于 0，用户需要检查该 broker 作为 leader 的那些分区当前的副本备份情况，是否出现了 follower 副本长时间无法追上 leader 副本或 broker 发生崩溃的情况。

如果发现该 MBean 值在持续波动但未发现有 broker 崩溃的情况，这也可能是性能问题，即有 follower 副本经常被踢出 ISR 的情况发生。此时需要进一步地查看出问题的 follower broker 日志来详细定位问题。

8.3.4 leader 分区分数

Kafka 的 MBean 指标 `kafka.server:type=ReplicaManager,name=LeaderCount` 统计了该 broker 是多少个分区的 leader 副本。单看某个 broker 上的该指标可能意义不大，而需要在整个集群上

查看这一指标。

一个比较理想的情况是，集群上所有 broker 的该 MBean 值都相差无几，这表明整个负载被均匀地分配到了所有 broker 上，毕竟只有分区的 leader 副本才能服务 clients 端的请求。

虽然 Kafka 在创建 topic 分区时会尽量地做到负载均衡，但集群长时间运行之后，的确有可能出现某些 broker 上承载过多分区 leader 角色的情况。一旦用户发现某些 broker 的该 MBean 值异常高，则可以使用 `bin/kafka-reassign-partitions.sh` 或 `bin/windows/kafka-reassign-partitions.bat` 脚本对现有的分区分布进行重调整。

8.3.5 ISR 变化速率

这里的 ISR 变化指的是 ISR 扩张（expansion）和收缩（shrink）。当有新的 follower 副本追上了 leader 副本的进度时，它会被加入 ISR 中；反之，若存在 follower 副本无法追上 leader 副本进度，则会被“踢出”ISR。ISR 经常变化通常会带来一些性能问题，比如造成 broker 上负载流量的经常性波动等。另外它还可能预示着经常有 broker 挂掉的情形出现。

这组 MBean 对应的名称分别是 `kafka.server:type=ReplicaManager,name=IsrExpandsPerSec` 和 `kafka.server:type=ReplicaManager,name=IsrShrinksPerSec`。在实际场景中，这两个 MBean 的平均值应该接近于 0 才是正常情况，这表明没有经常性的 ISR 变化。

一旦发现该组值长时间远大于 0，则需要查看集群上各个 broker 是否有经常性的“宕机”出现，另外对于老版本的 Kafka 而言（0.10.0.0 版本之前），可能需要仔细调优 `replica.lag.max.messages` 参数。

8.3.6 broker I/O 工作处理线程空闲率

默认情况下，每台 broker 都会创建 8 个工作线程（由参数 `num.io.threads` 指定）来处理外界发往该 broker 的各种请求。为了实时监控该 broker 上的工作负载情况，Kafka 提供了 `kafka.server:type=KafkaRequestHandlerPool,name=RequestHandlerAvgIdlePercent` 来统计这些线程总的空闲率。

该 MBean 值是一个百分比值，表示所有 I/O 工作线程处于空闲状态的时长与总运行时长的比值。在实际使用场景中建议该值不要小于 30%，因为一旦发现某台 broker 的这个 MBean 值降到了很低的水平，则说明此 broker 承担的负载过重，需要考虑将一部分负载转移到其他 broker 上。

8.3.7 broker 网络处理线程空闲率

默认情况下，每台 broker 都会创建 3 个网络处理线程（由参数 `num.network.threads` 指定）来处理网络接收请求和发送响应。为了实时监控这些 processor 线程的工作运行状态，Kafka 提供了 `kafka.network:type=SocketServer,name=NetworkProcessorAvgIdlePercent` 来统计这些线程总的空闲率。

该 MBean 值是一个百分比值，表示所有网络处理线程处于空闲状态的时长与总运行时长的比值。在实际使用场景中建议该值不要小于 30%，因为一旦发现某台 broker 的这个 MBean 值降到了很低的水平，则说明此 broker 无法及时地从网络中接收请求或发送响应，可能造成请求堆积。如果 broker 上的 CPU 资源比较充足，可以考虑增加该参数的值；否则需要增加 broker 请求队列的大小以缓存更多的进站请求和出站响应（增加 `queued.max.requests` 参数值）。

8.3.8 单个 topic 总字节数

经常有用户有这样的疑问，如何计算一个 topic 总的消息字节数？Kafka 提供了专门的 JMX 指标可以实现这样的统计。具体的 JMX MBean 名称是 `kafka.log:type=Log,name=Size,topic=<topic>,partition=<partition id>`。

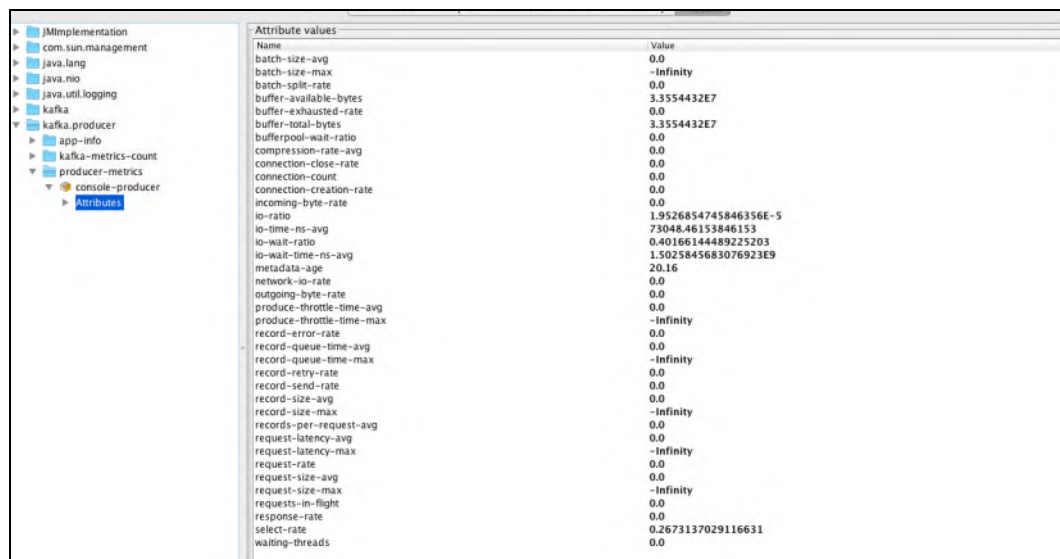
假设一个名为 test 的 topic 有 10 个分区，那么用户可以查询 `kafka.log:type=Log,name=Size,topic=test,partition=0` 到 `kafka.log:type=Log,name=Size,topic=test,partition=9` 的字节数，然后相加便可以得出 test 总的字节数。

8.4 clients 端 JMX 监控

8.4.1 producer 端 JMX 监控

Java 版本 producer 端内置了屈指可数的几个 JMX 指标，与 broker 端指标不同的是，每个 producer 端的 JMX 指标（甚至是 consumer 端指标）都有一个对应的 client ID，用于标识该指标统计的是哪个 producer 的数据。

与 broker 端 JMX 指标的另一个显著的不同在于，相比 broker 端单个 MBean 只定义单个属性不同，Java 版本 producer 的 JMX 指标非常少，但是定义的属性却十分多。如果以 0.11.0.0 版本为例，在启动一个 console producer 之后，通过 JConsole 我们可以发现 `producer-metrics` 下只有一个名为 `console-producer` 的 MBean，但其下的属性却有将近 40 个，如图 8.6 所示。



Name	Value
batch-size-avg	0.0
batch-size-max	-Infinity
batch-split-rate	0.0
buffer-available-bytes	3.3554432E7
buffer-exhausted-rate	0.0
buffer-total-bytes	3.3554432E7
bufferpool-wait-ratio	0.0
compression-rate-avg	0.0
connection-close-rate	0.0
connection-count	0.0
connection-creation-rate	0.0
incoming-byte-rate	0.0
io-ratio	1.9526854745846356E-5
io-time-ns-avg	73048.46153846153
io-wait-ratio	0.40166144489225203
io-wait-time-ns-avg	1.5025845683076923E9
metadata-age	20.16
network-io-rate	0.0
outgoing-byte-rate	0.0
produce-throttle-time-avg	0.0
produce-throttle-time-max	-Infinity
record-error-rate	0.0
record-queue-time-avg	0.0
record-queue-time-max	-Infinity
record-retry-rate	0.0
record-send-rate	0.0
record-size-avg	0.0
record-size-max	-Infinity
records-per-request-avg	0.0
request-latency-avg	0.0
request-latency-max	-Infinity
request-rate	0.0
request-size-avg	0.0
request-size-max	-Infinity
requests-in-flight	0.0
response-rate	0.0
select-rate	0.2673137029116631
waiting-threads	0.0

图 8.6 producer 端 JMX MBean 属性列表

这种设计极大地方便了用户实时观测 producer 应用运行情况，免去了用户需要在多个 MBean 之间切换查看的开销，只需要在一个 MBean 中便可以同时监控所有的运行状态。另外 MBean 的名字由 client ID 指定，完整格式是 `kafka.producer:type=producer-metrics,client-id=<CLIENT ID>`。

除了依赖上面的 MBean 从整体上监控 producer，Kafka 还允许用户指定节点序号和 topic，分别从 Kafka 节点和 topic 维度去查看相应的运行指标。它们的 MBean 格式分别是

```
kafka.producer:type=producer-node-metrics,client-id=<CLIENT_ID>,node-id=<BROKER ID>
```

和

```
kafka.producer:type=producer-topic-metrics,client-id=<CLIENT_ID>,topic=<TOPIC>
```

图 8.6 中比较重要的属性及其含义和实际使用方法建议如表 8.3 所示。

表 8.3 重要属性及其含义和实际使用方法建议

属 性 名	含 义	实际使用方法建议
waiting-threads	等待分配缓冲区的用户线程数。Java 版本 producer 端用户线程需要把消息放入内存缓冲区中，如果缓冲区空间紧张，则挂起用户线程	该值长时间为 0 是最佳状态。若持续大于 0 通常表明缓冲区空间不足，需要增加 <code>buffer.memory</code> 参数或减少用户线程数

续表

属 性 名	含 义	实际使用方法建议
batch-size-avg	每个分区发送的平均 batch 大小	用户可以使用该值与设定的 <code>batch.size</code> 做比较。若该值远远小于设定的 <code>batch.size</code> ，则可以考虑适当地增加延时（增大 <code>linger.ms</code> ）来让更多的消息装入 batch，再发送以提升吞吐量
record-queue-time-avg	消息 batch 在发送前缓存在内存中的平均等待时间	用户需要将该值与 <code>request.timeout.ms</code> 值进行比较。如果两者接近则说明 <code>producer</code> 容易出现请求超时的现象，从本质上说，这表明后台 <code>Sender</code> 线程无法快速地将缓存的数据发送出去，可以降低用户线程负载或调优 <code>Sender</code> 线程
record-error-rate	每秒发送消息出现错误的次数	很重要的一个 MBean。用户需要实时监控该值。理想情况下该值应该越接近 0 越好。如果该值长时间大于 0， <code>producer</code> 可能会丢弃待发送的消息。虽然用户可以配置 <code>retry</code> 次数，但一旦 <code>retry</code> 耗尽， <code>producer</code> 就会放弃发送这条消息
request-latency-avg	发送给 broker 端的 PRODUCE 请求的平均延时时间	生产环境中用户可以监控该 MBean 的值，一旦发现某个时间段 PRODUCE 请求持续性地超过了基准值，则说明 <code>producer</code> 性能变差，此时可以进一步地定位是网络问题还是 broker 端的问题

完整的 `producer` 端 JMX Mbean 参见官网链接 https://kafka.apache.org/documentation/#producer_monitoring。

8.4.2 consumer 端 JMX 监控

和 Java 版本 `producer` 类似，Java 版本 `consumer` 端 JMX 的指标也不是很多，但属性依然不少。和 `producer` 不同的是，`consumer` 的 JMX 大致分为如下几类。

- `fetcher-manager`：统计 `consumer` 底层 `fetcher`（消息获取器）的各种状态信息，MBean 名称是 `kafka.consumer:type=consumer-fetch-manager-metrics,client-id=<CLIENT_ID>`。
- `consumer`：统计 `consumer` 状态信息，MBean 名称是 `kafka.consumer:type=consumer-metrics,client-id=<CLIENT_ID>`。
- `coordinator`：统计 `consumer coordinator` 状态信息，MBean 名称是 `kafka.consumer:type=consumer-coordinator-metrics,client-id=<CLIENT_ID>`。

下面，我们分别对上面 3 类 MBean 展开，首先是 `Fetcher` 类。

如图 8.7 所示，`fetcher` JMX 属性包含了一些公共的属性以及 `topic` 级别的属性（如 `test-0.records-lag`），其中比较重要的如下。

- `fetch-latency-avg`：该属性与 `producer` 端的 `request-latency-avg` 含义类似，揭示了

FETCH 请求发送到 broker 端的延时。如果发现该值比较大，可能需要调优 consumer 端参数 `fetch.max.wait.ms` 和 `fetch.min.bytes`。

- `bytes-consumed-rate`: 该值揭示了每秒经由此 consumer 消费的字节数。在实际使用中，建议用户对此值做一个基准测试并确定业务稳定时的阈值。一旦之后发现该值严重偏离了阈值，则都需要仔细确认造成偏离的具体原因。

图 8.8 给出了 coordinator 相关的属性，其中比较重要的如下。

- `assigned-partitions`: 统计了该 consumer 被分配的分区数。可以利用此属性值来查看 consumer group 中所有成员的分区数是否被均匀地分配。
- `join-time-avg`: 统计该 consumer 加入 consumer group 的平均时间。
- `sync-time-avg`: 统计该 consumer 执行组同步操作的平均时间。
- `commit-latency-avg`: 统计该 consumer 提交位移的平均延时。

图 8.9 给出了与 consumer 直接相关的属性，其中比较重要的如下。

- `request-rate`: 统计该 consumer 平均每秒发送的 FETCH 请求数。
- `request-size-avg`: 统计该 consumer 平均每秒发送的 FETCH 请求字节数。

Name
<code>bytes-consumed-rate</code>
<code>fetch-latency-avg</code>
<code>fetch-latency-max</code>
<code>fetch-rate</code>
<code>fetch-size-avg</code>
<code>fetch-size-max</code>
<code>fetch-throttle-time-avg</code>
<code>fetch-throttle-time-max</code>
<code>records-consumed-rate</code>
<code>records-lag-max</code>
<code>records-per-request-avg</code>
<code>test-0.records-lag</code>
<code>test-0.records-lag-avg</code>
<code>test-0.records-lag-max</code>

图 8.7 fetcher 属性

Name
<code>assigned-partitions</code>
<code>commit-latency-avg</code>
<code>commit-latency-max</code>
<code>commit-rate</code>
<code>heartbeat-rate</code>
<code>heartbeat-response-time-max</code>
<code>join-rate</code>
<code>join-time-avg</code>
<code>join-time-max</code>
<code>last-heartbeat-seconds-ago</code>
<code>sync-rate</code>
<code>sync-time-avg</code>
<code>sync-time-max</code>

图 8.8 coordinator 属性

Name
<code>connection-close-rate</code>
<code>connection-count</code>
<code>connection-creation-rate</code>
<code>incoming-byte-rate</code>
<code>io-ratio</code>
<code>io-time-ns-avg</code>
<code>io-wait-ratio</code>
<code>io-wait-time-ns-avg</code>
<code>network-io-rate</code>
<code>outgoing-byte-rate</code>
<code>request-rate</code>
<code>request-size-avg</code>
<code>request-size-max</code>
<code>response-rate</code>
<code>select-rate</code>

图 8.9 consumer 属性

8.5 JVM 监控

Kafka 是标准的 JVM 系框架，必须运行在 JVM 之上。因此，除了监控 Kafka 提供的 JMX 监控指标之外，用户必须要实时监控 Kafka broker 运行环境中的 JVM，毕竟 JVM 运行状态以及性能将直接决定 broker 的表现。

谈到 JVM，有两个方面的表现一定要实时关注：进程状态和 GC 性能。

8.5.1 进程状态

JVM 默认提供了很多 JMX 指标，旨在实时提供关于应用程序的一些很有用的信息。有两个比较关键的指标值得用户对其进行日常的监控，分别是 `java.lang:type=OperatingSystem, MaxFileDescriptionCount` 和 `java.lang:type=OperatingSystem, OpenFileDescriptorCount`。前者定义了单个 JVM 进程允许打开的最大文件描述符（file descriptor, FD）个数，而后者统计了当前该 JVM 进程使用的文件描述符。

一旦发现后者值接近前者，那么就需要调整前者的最大值，通常情况下需要运行 `ulimit -n` 来指定更大的 FD 值。对于底层有很多分区数据的 broker 而言，这个值会增加得很快，因此一定要特别注意该 JMX 指标属性。

8.5.2 GC 性能

JVM 系框架最令人头疼的莫过于对 GC 性能的监控与调优了。本节将讨论如何监控 GC 性能。事实上，很多 JDK 工具都提供了对于 GC 性能的监控，比如 `jstat` 等。但很多用户的生产环境可能并没有安装 JDK，而只安装了 JRE，因此用户可以使用 JVM 默认提供的关于 GC 的 JMX 指标。

有两个比较重要的 JMX 指标 `java.lang:type=GarbageCollector,name=G1 Old Generation` 和 `java.lang:type=GarbageCollector,name=G1 Young Generation`。前者收集了 G1 垃圾收集器老年代的统计信息，包括收集次数、收集时长；而后者则表示 G1 新生代的统计信息，同样包括次数和时长。值得注意的是，这里的时长信息的单位是毫秒，统计了自 JVM 启动开始用于垃圾收集的时间。

这两个 JMX 指标还有一个 `LastGcInfo` 属性值，保存了最近一次 GC 的详情信息。该属性是一个复合属性，里面包含了详细的 GC 信息。比较重要的是 `duration` 值，它告诉用户上次 GC 花费了多长时间，其他信息还包括 `GcThreadCount`、`id`、`startTime` 和 `endTime` 等。

8.6 OS 监控

除了上面所说的 JVM 指标，用户在生产环境中还需要实时监测运行 Kafka broker 所在机器的操作系统状态，很多 OS 端的指标都是最能直接反映出系统瓶颈的指标。

绝大多数的操作系统本身已经提供了很健全的监控工具帮助收集各种 OS 指标。常见的操作系统指标包括：CPU 使用率、内存利用率、磁盘繁忙程度、磁盘使用率、磁盘 I/O 状况和网络利用率等。

若要监控 CPU 使用率，用户可以使用系统自带的 `top` 工具（Linux 平台），在其提供的各种指标中首先需要查看的就是系统负载情况（system load）。它直观地给出了当前系统运行的负载状态，如果 load 值长时间大于系统核数（CPU core），则通常说明系统负载已经过饱和，需要用户将该 broker 上的负载分散到其他机器上。常见的 `top` 性能指标还包括如下几种。

- `us`: 用于在用户态执行应用程序的时间百分比。
- `sy`: 用于运行内核代码的时间百分比。
- `id`: CPU 处于空闲状态的时间百分比。
- `wa`: 等待物理 I/O 的时间百分比。

通过以上指标，用户大致可以判断出当前 broker 所在机器的 CPU 负载情况，而 CPU 使用率通常可由 `us + sy` 或 `100 - id` 计算得出。如果该值长时间处于高位则需要结合 `JDK` 工具（比如 `jstack`）来定位消耗 CPU 的线程。当然，CPU 利用率处于高位不一定是坏事，毕竟我们总是希望应用可以充分地利用机器上的 CPU，因此在实际环境中用户需要监控等待队列（running queue）的值来定位 CPU 是否处于瓶颈。

等待队列中保存的都是等待 CPU 时间片的线程，故如果该值长时间大于 CPU 核数且 CPU 使用率处于高位，则表明系统 CPU 处于瓶颈。Linux 平台的用户可以使用 `vmstat` 命令查看等待队列的值，即 `vmstat` 输出的第 1 列。

对于内存监控而言，用户可简单地执行 `free` 命令来查看当前的内存使用情况。一般情况下，只需要确保内存没有被耗尽（特别是没有出现 swap 的情况）。

对于磁盘的监控则重点在磁盘的读/写使用率上。Linux 用户可以发起 `iostat -d -x -k 1 5` 这样的程序来监控 `%util` 列的值，如果该值长时间地保持在 95% 以上则表明磁盘 I/O 是系统瓶颈。另外，由于 Kafka 大量使用磁盘且当前不支持对于磁盘崩溃的故障转移（即一旦某块磁盘崩溃，broker 会立即被终止），磁盘空间的使用率也是一个必要的监控指标。

最后说说对于网络带宽的观测。Kafka broker 与 clients 以及 broker 之间都需要大量的 RPC 通信，所以 Kafka 需大量消耗网络带宽资源。用户一定要确保 broker 所在机器的入站/出站带宽都不要过于接近所允许的最大带宽，否则就会出现无法处理请求或请求超时的情况。Linux 平台下有各种各样的工具来检测带宽消耗情况，比较有名的工具包括 `nload` 和 `iftop`。如果用户一时无法安装这样的工具，单纯地监控 8.3 节中提到的 JMX 指标也是可以计算带宽消耗情况的。

8.7 主流监控框架

不论是作为消息队列还是流式处理框架，Apache Kafka 日益受到全世界各大公司的青睐。

不过令人遗憾的是，对于 Kafka 的监控始终是一个难题。当前，没有一款免费监控框架能够满足大部分的用户需求从而“一统江湖”。这些框架虽各具特色但都不够全面。本节将一一为读者介绍它们。到底选用哪一种框架，读者需要根据自身的情况自行确定。

8.7.1 JmxTool

严格来说，JmxTool 不是一个框架，而是 Kafka 社区默认提供的工具，用于实时查看 JMX 监控指标。如果用户一时找不到合适的工具来监测 JMX 指标，可以考虑使用该工具“临时救急”。

打开一个终端并进入 Kafka 安装目录，输入命令 `bin/kafka-run-class.sh kafka.tools.JmxTool`，便可以得到 JmxTool 工具的帮助信息，如表 8.4 所示。

表 8.4 JmxTool 工具参数及其含义

参 数	含 义
--attributes	指定 CSV 格式的属性。如果不指定该参数则默认读取所有属性
--date-format	设定时间格式，比如设置为 YYYY-MM-ss，则输出时间戳即为 2017-09-01 这样的格式
--jmx-url	指定要连接的 JMX 接口，默认格式是 service:jmx:rmi:///jndi/rmi://<JMX 端口>/jmxrmi
--object-name	指定 JMX 的 MBean 名称
--reporting-interval	设置实时监测时间间隔，默认是每 2 秒打印一次 JMX 指标值

假设我们要实时监控 8.3.1 节中提到的 JMX MBean——消息入站速率。我们想要每隔 5 秒输出一次该 MBean 过去 15 分钟的平均值，则可以执行下面的命令：

```

bin/kafka-run-class.sh kafka.tools.JmxTool --object-name kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec --jmx-url service:jmx:rmi:///jndi/rmi://:9997/jmxrmi --date-format "YYYY-MM-dd HH:mm:ss" --attributes FifteenMinuteRate --reporting-interval 5000

```

输出如图 8.10 所示。

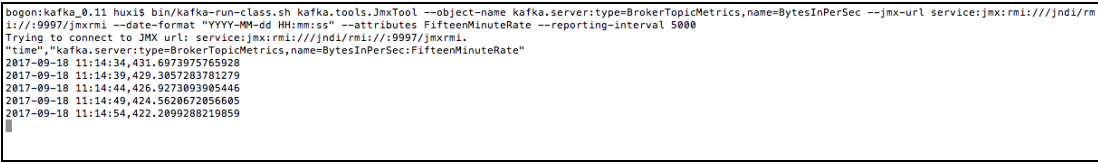


图 8.10 JmxTool 监控入站消息速率输出

8.7.2 kafka-manager

kafka-manager 是雅虎公司于 2015 年开源的一款 Kafka 监控框架，使用 Scala 语言编写，

用于管理和监控 Apache Kafka 集群。其官网地址是 <https://github.com/yahoo/kafka-manager>。

在笔者看来，kafka-manager 应该说是目前众多开源框架中的翘楚。无论是其界面展示内容的丰富程度，还是管理集群功能的全面性，kafka-manager 都是执牛耳者。

不过令人遗憾的是，该开源项目的实际维护者大概只有三四人，故很多 bug 或问题无法得到有效的处理，因此整个项目的进度十分缓慢，特别是无法追上 Apache Kafka 版本更迭的速度。比如 Kafka 在 0.9.0.0 版本推出了新版本 consumer（即 Java 版本 consumer），但 kafka-manager 对其的支持是相当缓慢的。为此，笔者还特意修改过 kafka-manager 的源码，令其能够监控新版本 consumer。不过好消息是，截止到笔者写稿时，kafka-manager 已然能够支持新版本 consumer 了。如果用户在使用 kafka-manager 时无法看到新版本 consumer group 的信息，那么就需要检查一下是否使用了过旧版本的 kafka-manager。

截止到笔者写稿时，kafka-manager 版本已进化到 1.3.3.13。用户若想要使用 kafka-manager，必须首先下载源码，然后手动编译成可执行文件。下面演示一下如何编译并安装 kafka-manager。

首先登录官网 <https://github.com/yahoo/kafka-manager> 下载 kafka-manager。本例中我们直接下载名为 kafka-manager-master 的 zip 文件即可。下载完毕之后解压缩 zip 文件到子目录 kafka-manager-master。

其次，使用 sbt 工具来构建 kafka-manager。sbt 是与 Maven、Gradle 类似的项目构建工具，多用于构建 Scala 语言编写的项目。实际上 Kafka 项目原先也是使用 sbt 构建的，不过目前改为使用 Gradle 了。sbt 的安装与配置请参考官网 <http://www.scala-sbt.org/1.x/docs/Setup.html>。本例将假设用户已经安装好 sbt 工具。一旦安装好 sbt，我们打开一个终端，进入 kafka-manager-master 目录下，输入以下命令即可开始构建 kafka-manager：

```
sbt clean dist
```

等待几分钟之后，构建完成，如图 8.11 所示。

构建完成之后，用户在 kafka-manager-master 目录的 target/universal 子目录下会发现一个名为 kafka-manager-1.3.3.13.zip 的文件，该文件即为构建后的 kafka-manager 可执行文件。

鉴于在构建过程中需要从国外 Maven 仓库下载很多 jar 包而国内网速糟糕，笔者特意将构建后的 1.3.3.13 版本的 kafka-manager 可执行文件共享到网盘上，地址是 <https://pan.baidu.com/s/1nuYBWRB>，读者可以直接下载构建后的文件。


```
[info] Updating {file:/Users/huxi/Downloads/kafka-manager-master/project/}kafka-manager-master-build...
[info] Resolving org.fusesource.jansi#jansi;1.4 ...
[info] Done updating.
Missing bintray credentials /Users/huxi/.bintray/.credentials. Some bintray features depend on this.
[info] Set current project to kafka-manager (in build file:/Users/huxi/Downloads/kafka-manager-master/)
[warn] Credentials file /Users/huxi/.bintray/.credentials does not exist
[success] Total time: 0 s, completed Sep 19, 2017 9:14:51 AM
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/Users/huxi/.ivy2/cache/org.slf4j/slf4j-nop/jars/slf4j-nop-1.7.7.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/Users/huxi/.ivy2/cache/org.slf4j/slf4j-simple/jars/slf4j-simple-1.7.12.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.helpers.NOPLoggerFactory]
[info] Packaging /Users/huxi/Downloads/kafka-manager-master/target/scala-2.11/kafka-manager_2.11-1.3.3.13-sources.jar ...
[info] Done packaging.
[warn] Credentials file /Users/huxi/.bintray/.credentials does not exist
[info] Updating {file:/Users/huxi/Downloads/kafka-manager-master/}root...
[info] Resolving jline#jline;2.12.1 ...
[info] Done updating.
[warn] There may be incompatibilities among your library dependencies.
[warn] Here are some of the libraries that were evicted:
[warn] * org.webjars:jquery:1.11.1 -> 2.1.4
[warn] Run 'evicted' to see detailed eviction warnings
[info] Wrote /Users/huxi/Downloads/kafka-manager-master/target/scala-2.11/kafka-manager_2.11-1.3.3.13.pom
[info] Compiling 125 Scala sources and 2 Java sources to /Users/huxi/Downloads/kafka-manager-master/target/scala-2.11/classes...
[info] Main Scala API documentation to /Users/huxi/Downloads/kafka-manager-master/target/scala-2.11/api...
[info] LESS compiling on 1 source(s)
[info] Packaging /Users/huxi/Downloads/kafka-manager-master/target/scala-2.11/kafka-manager_2.11-1.3.3.13-web-assets.jar ...
[info] Done packaging.
model contains 678 documentable templates
[info] Main Scala API documentation successful.
[info] Packaging /Users/huxi/Downloads/kafka-manager-master/target/scala-2.11/kafka-manager_2.11-1.3.3.13-javadoc.jar ...
[info] Done packaging.
[info] Packaging /Users/huxi/Downloads/kafka-manager-master/target/scala-2.11/kafka-manager_2.11-1.3.3.13.jar ...
[info] Done packaging.
[info] Packaging /Users/huxi/Downloads/kafka-manager-master/target/scala-2.11/kafka-manager_2.11-1.3.3.13-sans-externalized.jar ...
[info] Done packaging.
[info]
[info] Your package is ready in /Users/huxi/Downloads/kafka-manager-master/target/universal/kafka-manager-1.3.3.13.zip
[info]
[success] Total time: 73 s, completed Sep 19, 2017 9:16:04 AM
```

图 8.11 构建 kafka-manager

解压缩该文件后打开 conf 目录下的 application.conf 文件，用户需要修改 kafka-manager.zkhosts 的值，如下：

```
kafka-manager.zkhosts=localhost:2181
```

（本例使用测试 Kafka 环境演示，故这里配置 localhost）

保存修改之后，运行下列命令启动 kafka-manager：

```
bin/kafka-manager
```

（如果是 Windows 平台，运行 bin/kafka-manager.bat）

需要注意的是，默认构建出来的.sh 执行文件不具有“可执行”权限，故可以使用 bash bin/kafka-manager 运行或给该脚本赋予可执行权限，如 chmod u+x bin/kafka-manager。

如果一切正常，用户应该可以看到类似于[info] p.c.s.NettyServer - Listening for HTTP on /0.0.0.0:0.0.0.0:9000 这样的输出表明 kafka-manager 已经成功地在端口 9000 上启动。如果用户想要指定不同的端口，可以在启动时指定，如下：

```
bin/kafka-manager -Dhttp.port=8080
```

此时，打开浏览器输入 http://localhost:8080，我们可以看到 kafka-manager 的主界面，如图 8.12 所示。

初始界面中没有显示任何 Kafka 集群，因此用户需要首先添加一个 Kafka 集群。单击顶部菜单中的 Cluster→Add Cluster，输入 Kafka 集群信息，如图 8.13 所示。



图 8.12 kafka-manager 主界面

图 8.13 添加 Kafka 集群

图 8.13 中，我们输入了 Kafka 集群名称（可随意指定）、ZooKeeper 连接信息和 Kafka 版本。值得一提的是，目前最新版本的 kafka-manager 只支持到 Kafka 0.10.2.1 版本，不过根据笔者的测试，0.11.0.0 版本的 Kafka 也是可以管理的。另外笔者还勾选了“启用 JMX 轮询”复选框，这样 kafka-manager 就能实时获取 JMX 指标了。不过前提是在启动 Kafka broker 之前已经设置了 JMX_PORT 环境变量。

该界面下面还有很多配置信息，不过在本例中不做修改，直接使用默认值即可。单击保存之后，kafka-manager 显示添加集群成功，然后单击 Go to cluster view 进入集群管理界面，如图 8.14 所示。

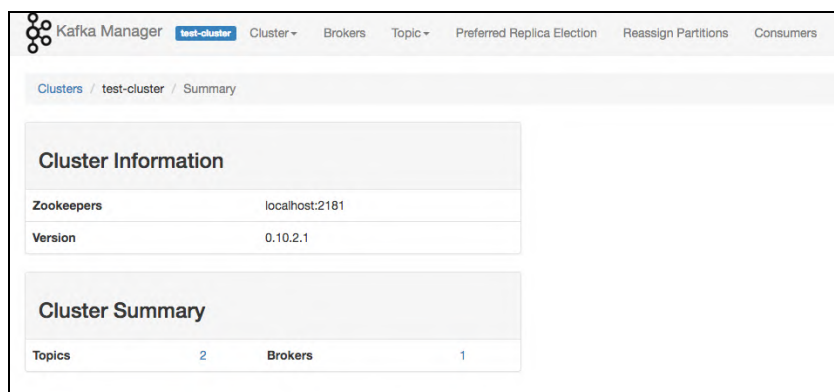


图 8.14 Kafka 集群管理界面

kafka-manager 提供的功能很多，顶部菜单中各个子菜单的功能如表 8.5 所示。

表 8.5 kafka-manager 菜单功能表

菜单项	功能说明
Cluster	新增 Kafka 集群、显示集群详情以及查询集群列表
Brokers	集群 broker 列表，附带基础的 JMX 指标监控（如消息出站/入站速率、每秒字节数等）
Topics	创建 topic、查询 topic 列表
Preferred Replica Election	执行 preferred leader 选举
Reassign Replica	执行副本重分配操作
Consumers	查询 consumer 信息以及 consumer group 详情（包括 LEO、Lag 信息等）

本节将重点演示一下 kafka-manager 对于 consumer 的监控，单击顶部菜单中的 consumers，进入 consumer 监控主界面，如图 8.15 所示。从图 8.15 中我们发现当前 Kafka 集群中共有两个 consumer group，类型分别是 ZK 和 KF，这表示一个 group 是老版本 consumer，即 ZooKeeper consumer（ZK）；另一个则是新版本 consumer（KF）。选择 KF 类型的 consumer 之后，我们能够查询到该 consumer group 当前的消费详情，如图 8.16 所示。



图 8.15 kafka-manager consumer 监控主界面

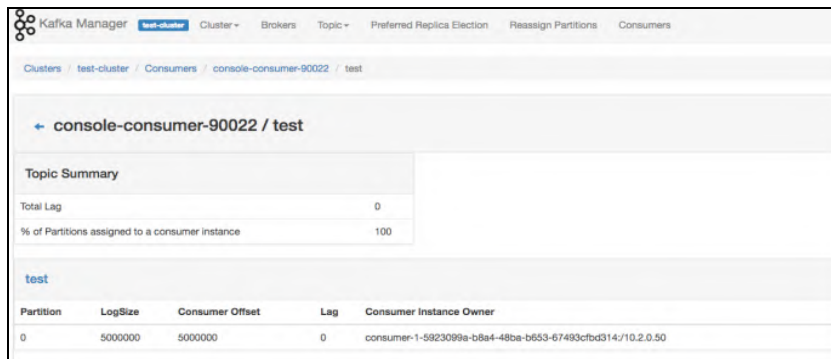


图 8.16 consumer group 详情

图 8.16 显示该 consumer group 当前的 consumer offset 是 5000000，而 Lag 是 0，这说明此

consumer group 没有任何消费滞后。

除了丰富的监控指标和集群状态展示外，kafka-manager 还允许用户执行很多管理工作，如创建 topic、preferred leader 选举以及分区重分配等。不过在实际的生产环境中这可能是一把双刃剑，因为这意味着任何能够访问 kafka-manager 的用户都能执行这些操作。对于运维人员来说，这往往是不能容忍的，特别是后面两种操作对生产环境的冲击极大，一定要谨慎使用。

因此很多 kafka-manager 用户都有这样的诉求：关闭这些管理操作，将 kafka-manager 只作为一个监控框架来使用。庆幸的是，kafka-manager 的确提供了这样的功能。用户打开 conf 目录下的 application.conf，修改 application.features 的值，删除其中的 “KMPreferredReplicaElectionFeature”、“KMReassignPartitionsFeature” 部分，然后重启 kafka-manager。这样再次进入 kafka-manager，用户会发现进入 Preferred Replica Election 和 Reassign Partitions 之后按钮已被隐去，再也无法执行相应的操作了。此时 kafka-manager 完全地变成了一个只读的监控框架（严格来说，用户依然可以在界面上创建 topic，不过它对生产环境的冲击远不如前两者来得猛烈）。

总结一下，kafka-manager 作为一款 Kafka 监控平台能够提供非常丰富的实时信息以及适当的管理功能，它是一款非常不错的免费框架。

8.7.3 Kafka Monitor

Kafka Monitor 是 LinkedIn 公司开源的一个免费框架，用于在 Kafka 集群上执行真实的 Kafka 系统测试并实时监控测试度量结果。该框架由 LinkedIn 的 Kafka 团队成员维护，维护的速度要比 kafka-manager 快一些。

如前所述，Kafka Monitor 是在集群上执行系统测试的，以便让用户对 Kafka 集群的性能有大致地了解。它会自动地创建一个监控 topic，然后分配分区并触发 preferred leader 选举以确保每个 broker 都至少是一个分区的 leader。这样做的好处在于可以全面地测试集群上所有 broker 的性能而无须用户手动干预。

下面我们来演示一下如何使用 Kafka Monitor。首先是安装 Kafka Monitor。安装 Kafka Monitor 需要用户已安装好 Gradle 构建工具（Gradle 安装方法参见 <https://gradle.org/install/>）。具体步骤如下：

```
$ git clone https://github.com/linkedin/kafka-monitor.git
$ cd kafka-monitor
$ ./gradlew jar
```

构建完成之后，用户需要打开 config 目录下的 kafka-monitor.properties 文件修改 broker 和 ZooKeeper 连接信息，如下：

```
"zookeeper.connect": "localhost:2181",
```

```
"bootstrap.servers": "localhost:9092",
```

之后，运行启动命令：

```
bin/kafka-monitor-start.sh config/kafka-monitor.properties
```

如果一切正常，此时用户可以看到控制台会阶段性地输出各种指标的值，如图 8.17 所示。同时，用户打开浏览器输入 <http://localhost:8000>，则可以看到 Kafka Monitor 的主界面，如图 8.18 所示。

```
[2017-09-19 10:50:48,403] INFO kmf:type=kafka-monitor:offline-runnable-count=0.0
kmf.services:name=single-cluster-monitor,type=produce-service:produce-availability-avg=1.0
kmf.services:name=single-cluster-monitor,type=consume-service:consume-availability-avg=0.0
kmf.services:name=single-cluster-monitor,type=produce-service:records-produced-total=56.0
kmf.services:name=single-cluster-monitor,type=consume-service:records-consumed-total=0.0
kmf.services:name=single-cluster-monitor,type=consume-service:records-lost-total=0.0
kmf.services:name=single-cluster-monitor,type=consume-service:records-duplicated-total=0.0
kmf.services:name=single-cluster-monitor,type=consume-service:records-delay-ms-avg=0.0
kmf.services:name=single-cluster-monitor,type=produce-service:records-produced-rate=0.9337690922430467
kmf.services:name=single-cluster-monitor,type=produce-service:produce-error-rate=0.0
kmf.services:name=single-cluster-monitor,type=consume-service:consume-error-rate=0.0
(com.linkedin.kmf.services.DefaultMetricsReporterService)
[2017-09-19 10:50:49,405] INFO kmf:type=kafka-monitor:offline-runnable-count=0.0
kmf.services:name=single-cluster-monitor,type=produce-service:produce-availability-avg=1.0
kmf.services:name=single-cluster-monitor,type=consume-service:consume-availability-avg=1.0
kmf.services:name=single-cluster-monitor,type=produce-service:records-produced-total=74.0
kmf.services:name=single-cluster-monitor,type=consume-service:records-consumed-total=4.0
kmf.services:name=single-cluster-monitor,type=consume-service:records-lost-total=0.0
kmf.services:name=single-cluster-monitor,type=consume-service:records-duplicated-total=0.0
kmf.services:name=single-cluster-monitor,type=consume-service:records-delay-ms-avg=10.25
kmf.services:name=single-cluster-monitor,type=produce-service:records-produced-rate=1.2338885831957715
kmf.services:name=single-cluster-monitor,type=produce-service:produce-error-rate=0.0
kmf.services:name=single-cluster-monitor,type=consume-service:consume-error-rate=0.0
(com.linkedin.kmf.services.DefaultMetricsReporterService)
[2017-09-19 10:50:50,401] INFO kmf:type=kafka-monitor:offline-runnable-count=0.0
kmf.services:name=single-cluster-monitor,type=produce-service:produce-availability-avg=1.0
kmf.services:name=single-cluster-monitor,type=consume-service:consume-availability-avg=1.0
kmf.services:name=single-cluster-monitor,type=produce-service:records-produced-total=92.0
kmf.services:name=single-cluster-monitor,type=consume-service:records-consumed-total=22.0
kmf.services:name=single-cluster-monitor,type=consume-service:records-lost-total=0.0
kmf.services:name=single-cluster-monitor,type=consume-service:records-duplicated-total=0.0
kmf.services:name=single-cluster-monitor,type=consume-service:records-delay-ms-avg=4.681818181818182
kmf.services:name=single-cluster-monitor,type=produce-service:records-produced-rate=1.534100383525966
kmf.services:name=single-cluster-monitor,type=produce-service:produce-error-rate=0.0
kmf.services:name=single-cluster-monitor,type=consume-service:consume-error-rate=0.0
(com.linkedin.kmf.services.DefaultMetricsReporterService)
```

图 8.17 Kafka Monitor 控制台输出



图 8.18 Kafka Monitor 主界面

Kafka Monitor 会在后台自动地创建一个测试 topic，并实时地监控该 topic 的运行状态。该框架默认给出了如下 8 个测试指标。

- `produce-availability-avg`: 统计该集群上所有分区响应 `PRODUCE` 请求的可用度。该值等于 1 是正常情况。
- `records-produced-rate`: 统计平均每秒发送分区的信息数。
- `records-lost-rate`: 统计平均每秒丢失的信息数。
- `records-duplicated-rate`: 统计平均每秒重新发送的信息数。
- `records-delay-ms-avg`: 统计从 `producer` 到 `consumer` 的端到端的平均消息延时，单位是毫秒。
- `records-delay-ms-99th`: 统计 99% 的消息端到端消息延时。
- `records-delay-ms-999th`: 统计 99.9% 的消息端到端消息延时。
- `records-delay-ms-max`: 统计消息端到端的最大延时。

当然，Kafka Monitor 提供的指标远不止这些，不过令人遗憾的是，官网未给出完整的指标列表。用户需要阅读源代码以发现所有的指标项，比如要查询 `records-produced-total` 指标，即统计目前已经发送的所有消息数，则可以执行下列命令：

```
curl localhost:8778/jolokia/read/kmf.services:type=produce-service,name=*/records-produced-total
```

结果返回一个 JSON 串，里面的 `value` 字段的值给出了当前发送的总消息数。注意，当前的端口 8778 被硬编码到代码中，故目前用户只能通过直接设置源代码并重新构建工程才能修改此值。

Kafka Monitor 作为 LinkedIn Kafka 团队维护的开源框架，从系统测试的角度给出了 Kafka 集群的性能评测结果。从这个意义上说，它不同于其他的监控框架。且该平台目前提供的 Web 界面功能十分简单，因此实际使用用户数并不是很多。不过鉴于其代码贡献者基本上都是 Kafka 社区团队成员，其框架的发展还是值得期待的。

8.7.4 Kafka Offset Monitor

和之前两个监控系统关注整体 Kafka 集群不同的是，Kafka Offset Monitor 关注 Kafka consumer 和位移的监控。其官网地址是 <https://github.com/quantifind/KafkaOffsetMonitor>。

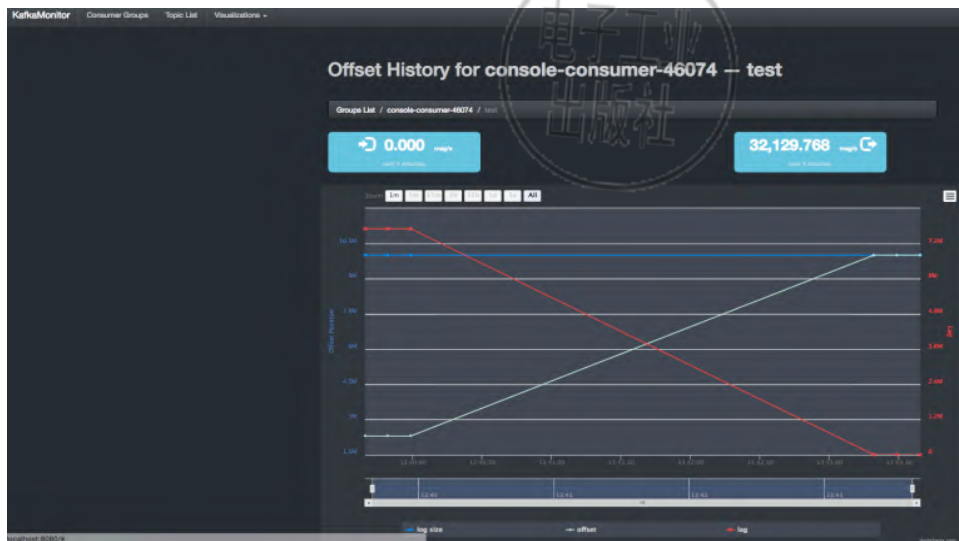
这是笔者用过的第一款监控软件，它提供了非常酷炫的图形化展示界面，但最大的弊病在于此项目已经将近 2 年未维护了，因此无法监控新版本 consumer。也就是说，它只能监控老版本 consumer 的位移，这一点要特别注意。

用户可以直接到官网下载编译好的 jar 包 <https://github.com/quantifind/KafkaOffsetMonitor/releases/download/v0.2.1/KafkaOffsetMonitor-assembly-0.2.1.jar>。为了节省时间，笔者已将其共享到网盘上，读者可以直接下载，地址是 <https://pan.baidu.com/s/1nuYBWRB>。

一旦下载完成，用户可以打开一个终端执行下列命令，启动 Kafka Offset Monitor 工具，如下：

```
java -cp KafkaOffsetMonitor-assembly-0.3.0.jar:kafka-offset-monitor-  
another-db-reporter.jar \  
    com.quantifind.kafka.offsetapp.OffsetGetterWeb \  
    --zk zk-server1,zk-server2 \  
    --port 8080 \  
    --refresh 10.seconds \  
    --retain 2.days
```

用户需要根据实际环境修改上面命令中的 zk 信息和端口信息。启动成功后，打开浏览器输入 <http://localhost:8080>，即可访问和查看当前 Kafka 集群中老版本 consumer group 的位移信息，如图 8.19 所示。



令人欣喜的是，该项目有一个同名的“姊妹”项目，地址是 <https://github.com/Morningstar/kafka-offset-monitor>。该项目的安装方法与 Kafka Offset Monitor 完全一样，而且还支持新版本 consumer 位移的查看。有兴趣的读者可以自行下载尝试。

8.7.5 CruiseControl

最后我们聊聊 LinkedIn 公司于 2017 年 8 月底开源的 Kafka 监控框架——CruiseControl，它主要用于管理超大规模的 Kafka 集群，提供的功能如下。

- 实时监测资源使用率（broker、topic、分区）。
- 提供多维度的重平衡策略。
 - 机架感知（rack awareness）。
 - 资源使用率再平衡（CPU、磁盘、网络 I/O）。
 - leader 副本流量分配。
 - 副本均匀分布。
- 异常检测以及实时告警。
- 开箱即用的运维操作。
 - 增加 broker 节点。
 - 去除 broker 节点。
 - 集群再平衡。

由于该框架刚刚开源，有很多功能尚在完善和演进中。打算使用 CruiseControl 的用户最好时刻关注该框架的发展，其官网地址是 <https://github.com/linkedin/cruise-control>。

下面我们来演示一下如何安装、配置和启动 CruiseControl，首先下载源码进行编译：

```
$ git clone https://github.com/linkedin/cruise-control.git
$ ./gradlew clean jar
```

构建成功后，我们可以在项目目录下的 `cruise-control-metrics-reporter/build/libs/` 下找到 `cruise-control-metrics-reporter.jar` 文件，然后将该文件拷贝到 broker 安装目录的 `/libs` 下。

之后在 broker 端配置文件 `server.properties` 中增加下列行，保存之后启动 broker：

```
metric.reporters=com.linkedin.kafka.cruisecontrol.metricsreporter.CruiseControlMetricsReporter
```

做完这些之后，我们打开 `cruisecontrol` 项目目录下的 `config/cruisecontrol.properties` 文件，修改里面的 `bootstrap.servers` 和 `zookeeper.connect` 值，然后执行以下命令：

```
$ ./gradlew jar copyDependantLibs
$ ./kafka-cruise-control-start.sh config/cruisecontrol.properties
```


如果一切正常，CruiseControl 应该已经成功地在端口 9090 上建立服务，用户此时可以访问 <http://localhost:9090> 上的各种 REST API。比如查看整个 Kafka 集群的负载情况，我们可以使用 curl (<http://localhost:9090/kafkacruisecontrol/load>)，执行结果如图 8.20 所示。

```
huxi:newenv huxi$ curl http://localhost:9090/kafkacruisecontrol/load
```

HOST	DISK(MB)	CPU(%)	LEADER_NW_IN(KB/s)	FOLLOWER_NW_IN(KB/s)	NW_OUT(KB/s)	PNW_OUT(KB/s)	REPLICAS
10.2.0.50,	-1990.388,	0.002,	0.038,	0.000,	0.036,	0.036,	116

HOST	BROKER	DISK(MB)	CPU(%)	LEADER_NW_IN(KB/s)	FOLLOWER_NW_IN(KB/s)	NW_OUT(KB/s)	PNW_OUT(KB/s)	REPLICAS
10.2.0.50,	0,	-1990.388,	0.002,	0.038,	0.000,	0.036,	0.036,	116

```
huxi:newenv huxi$
```

图 8.20 CruiseControl 查询集群负载

图 8.20 中列出了当前集群中的总分区数以及每个 broker 对应的系统负载情况。

遗憾的是，CruiseControl 目前所有的指标都是以 REST API 的方式提供的，没有对应的 Web 监控界面，用户需要自己的前端监控框架中调用这些 REST API。另外，由于该项目依然在积极开发中，后续很多功能和使用方法可能会有较大的变化，需要用户时时关注。

虽然 CruiseControl 是最新开源的框架，但其主“操刀手”是 Apache Kafka 的 committer，因此在框架质量和与 Kafka 的适配性上与其他开源框架相比都有着较大的优势。笔者也会时刻关注该框架后续的发展。

8.8 本章小结

本章我们详细探讨了监控 Kafka 集群的各个方面，包括 JMX 监控及其对应的 MBean 指标、JVM 监控、OS 监控，以及目前主流的 Kafka 监控框架。

结合这些知识，我们将在第 9 章中学习在生产环境中如何定位 Kafka 问题并调优 Kafka 集群。

第 9 章

调优 Kafka 集群

在第 8 章中，我们学习了监控 Apache Kafka 集群的方法以及目前各种主流的监控框架。本章将查看如何调优 Apache Kafka 集群。

学习本章，你将了解到以下内容。

- 集群调优目标确定。
- 集群调优参数设置。
- 性能调优方法。

9.1 引言

Apache Kafka 作为一个流式处理平台展现了很好的“开箱即用”特性：用户只需要在 clients 应用中指定 Kafka 集群的连接信息，剩下的所有事情都交由 Kafka 集群处理即可。这包括将负载自动地分发到不同的 broker 上、broker 默认使用 ZeroCopy 的传输优化特性来在 broker 与 clients 之间传输数据，以及当有 broker 出现崩溃时集群还能够自动地进行故障转移，将受影响分区的数据迁移到其他 broker 上。如果用户使用了消费者组（consumer group），Kafka 还提供了无须人工干预的组重平衡功能（group rebalance），从而确保所有 consumer 实例都能分配到数量大致相当的分区进行消费。所有的这一切对运维人员来说都是极大的福音，Kafka 能够显著降低人工操作的成本，从而将运维管理人员真正地从烦琐的日常操作中解放出来。

当然，这并不意味着在实际生产环境中运维人员不需要对集群进行任何调整和调优，毕竟使用默认参数配置搭建起来的 Kafka 集群通常只能满足基本的功能使用。对于有着各种非功能

性需求的场景而言，我们必须仔细地调优 Kafka 集群以满足这些需求目标。

在软件工程领域，非功能性需求是指依照某些条件判断系统运行情形或其特性，而不是针对系统特定行为的需求（该定义来自维基百科）。常见的非功能性需求包括（但不限于）如下这些。

- **性能（performance）**：最重要的非功能性需求之一。大多数生产环境对集群性能都有着严格的要求。不同的系统对于性能有着不同的诉求。比如对数据库系统来说，最重要的性能是请求的响应时间（response time）——用户总是希望一条查询或更新操作的整体响应时间越短越好；而对于 Kafka 而言，性能一般指的是吞吐量和延时两个方面。
 - **吞吐量（throughput/TPS）**：broker 或 clients 应用程序每秒能处理多少字节（或消息）。
 - **延时（latency）**：通常指代 producer 端发送消息到 broker 端持久化保存消息之间的时间间隔。该概念也用于统计端到端（end-to-end, E2E）的延时，比如 producer 端发送一条消息到 consumer 端消费这条消息的时间间隔。
- **可用性（availability）**：在某段时间内，系统或组件正常运行的概率或在时间上的比率。业界一般使用 N 个 9 来量化可用性，比如常见的“年度 4 个 9”即指系统可用性要达到 99.99%，即一年中系统宕机的时间不能超过 53 分钟（ $365 \times 24 \times 60 \times 0.01\% \approx 53$ ）。
- **持久性（durability）**：确保了已提交操作所产生的效果会被永久地保持，即使系统出现崩溃。对于 Kafka 来说，持久性通常意味着已提交的消息需要被持久化到 broker 底层的物理日志而不能发生丢失。

由于篇幅有限，上面仅仅罗列出了对 Kafka 非常重要的非功能性需求，本章也将对它们进行详细的展开来探讨如何从这些方面调优 Kafka 集群。

每个用户场景都有特定的调优需求，所以上面这些非功能性需求的目标也各不相同。在 Kafka 中调优这些目标通常都是通过调节各种 Kafka 参数来实现的。在实际中我们推荐用户深入了解一些重要的 Kafka 参数并结合不同的参数设置组合执行基准测试（benchmarking）以发现最适合用户场景的一组配置。在理想情况下，用户应该在正式将 Kafka 集群投入生产环境之前或准备大规模扩展 Kafka 集群之前完成这些尝试。

图 9.1 给出了一个调优 Kafka 集群的流程。它通常是一个迭代过程，需要用户不断地优化系统设置、参数配置甚至是应用架构，直到最终满足预期目标。

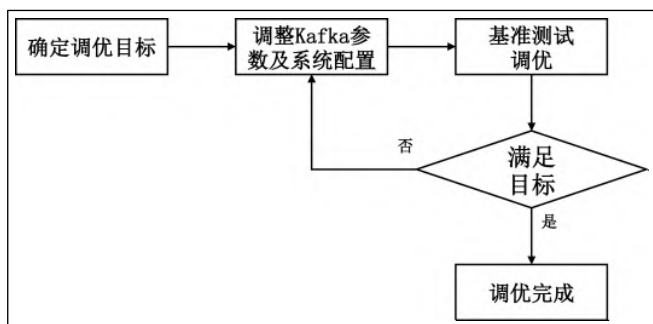


图 9.1 调优 Kafka 集群流程图

9.2 确定调优目标

实现非功能性需求目标的第一步就是要确定调优的目标。只有确定了目标才能明确调优的方向，进而做到有的放矢。笔者曾经接到过很多用户发来这样的问题：应该怎么优化我的 Kafka 集群？我怎么让 Kafka 跑得更快？为什么我的 producer 很慢？实际上，每个问题都对应了一个具体的优化目标。不能说这些用户提的问题是错的，但若想解决这些问题，我们一定要首先确定优化的目标才行。

如前所说，本章会从 4 个方面来考量调优目标：吞吐量、延时、持久性和可用性，如图 9.2 所示。上面那些问题实际上都可以对应到这 4 个方面。为了确定用户生产环境中的目标，用户需要结合业务仔细思考 Kafka 集群的使用场景与初衷。比如使用 Kafka 集群的目的是什么？是作为一个消息队列使用，还是作为数据存储，抑或是用作流式数据处理，更或是以上所有？



图 9.2 调优的 4 个方面

明确这个目标主要有两个重要的作用。第一，万物皆有度，世界上没有十全十美的事情。你不可能同时最大化上述 4 个目标。它们彼此之间可能是矛盾的，到底看重哪个方面实际上是一个权衡选择（trade-off）。举一个简单的例子，在性能调优时，吞吐量和延时就常常是相互制约的。讲到吞吐量和延时，笔者想要多花一些笔墨来讨论一下两者的区别。很多人都有这样的困惑：为什么我们要区分吞吐量和延时这两个指标？它们说的不是一回事吗？假设 Kafka

producer 每发送一条消息需要花费 2 毫秒（即延时是 2 毫秒），那么显然 producer 的吞吐量就应该是 500 条/秒，因为 1 秒可以发送 $1 / 0.002 = 500$ 条消息。因此，吞吐量和延时的关系似乎可以使用公式来表示： $TPS = 1000 / Latency$ （毫秒）。

其实，两者的关系远非上面公式表示的这么简单。我们依然以 Kafka producer 来举例，假设它仍然以 2 毫秒的延时来发送消息。如果每次只发送一条消息，那么 TPS 自然就是 500 条/秒。但如果 producer 不是每次发送一条消息，而是在发送前等待一段时间后统一发送一批消息。假设 producer 每次发送前会等待 8 毫秒，而 8 毫秒之后 producer 缓存了 1000 条消息，那此时总延时就累加到 10 毫秒（2 + 8），这时 TPS 等于 $1000 / 0.01 = 100000$ 条/秒。由此可见，虽然延时增加了 4 倍，但 TPS 却增加了将近 200 倍。

上面的场景解释了目前为什么批次化（batching）以及微批次化（micro-batching）流行的原因。实际环境中用户几乎总是愿意用增加较小延时的代价去换取 TPS 的显著提升。毕竟从 2 毫秒到 10 毫秒的延时增加通常是可以忍受的。值得一提的是，Kafka producer 也采取了这样的理念，如果读者回顾第 4 章就会发现，这里的 8 毫秒就是 producer 参数 `linger.ms` 所表达的含义。

有的读者可能会问，producer 等待 8 毫秒就能累积 1000 条消息吗？不是发送一条消息就需要 2 毫秒吗？这里需要解释一下，producer 累积消息一般仅仅是将消息发送到内存中的缓冲区，而发送消息却需要涉及网络 I/O 传输。内存操作和 I/O 操作的时间量级是不同的，前者通常是几百纳秒级别，而后者从毫秒到秒级别不等，故 producer 等待 8 毫秒积攒出的消息数远远多于同等时间内 producer 能够发送的消息数。

说了这么多其实只想强调上面 4 个调优目标通常是相互关联、相互制约的。当然，这种制约绝不是完全互斥，即提高一个目标一定会使其他目标降低。笔者想表明意思是，用户不能同时使这 4 个目标达到最优程度。这就是在调优前明确优化目标的第一个含义。

明确优化目标的第二个含义在于，用户需要明确调优的重点才能有针对性地调整不同的 Kafka 参数。截止到本章完稿时最新的 1.0.0 版本，Kafka 提供的各类参数已达几百个之多。只有我们明确要调优哪些方面才能确定适合的参数。比如如果要为集群中所有 topic 进行优化，那么就需要调整 broker 端的参数，而如果只是为某些 topic 进行优化，则需要调整 topic 级别的参数。

9.3 集群基础调优

配置合理的操作系统（OS）参数能够显著提升 Kafka 集群的性能、阻止错误条件的发生，而 OS 级错误几乎总是会降低系统性能，甚至影响其他非功能性需求指标。在 Kafka 中经常碰

到的操作系统级别错误可能包括如下几种。

- connection refused。
- too many open files。
- address in use: connect。

通过恰当的 OS 调优我们就可能提前预防这些错误的发生，从而降低问题修复的成本。本节我们将从以下几个方面分别探讨 OS 级别的调优。

9.3.1 禁止 atime 更新

由于 Kafka 大量使用物理磁盘进行消息持久化，故文件系统的选择是重要的调优步骤。对于 Linux 系统上的任何文件系统，Kafka 都推荐用户在挂载文件系统（mount）时设置 noatime 选项，即取消文件 atime（最新访问时间）属性的更新——禁掉 atime 更新避免了 inode 访问时间的写入操作，因此极大地减少了文件系统写操作数，从而提升了集群性能。Kafka 并没有使用 atime，因此禁掉它是安全的操作。用户可以使用 mount -o noatime 命令进行设置。

值得一提的是，Kafka 虽然没有使用 atime，但却使用了 mtime，即修改时间用于日志切分等操作。当然随着时间戳属性在 0.10.0.0 版本的引入，mtime 的使用场景也大大地减少了。

9.3.2 文件系统选择

选择哪种文件系统对于 Windows 和 Mac OS X 上的用户来说几乎不是问题。首先搭建在这两个平台上的 Kafka 生产环境少之又少；其次这两个平台上的文件系统类型选择也不多，几乎只能确定为 NTFS 和 HFS+。但对 Linux 平台上的用户而言，这个选择要困难得多。

Linux 平台当前有很多文件系统，最常见的当属 EXT4 和 XFS 了。EXT4 是 EXT 系列的最新一版，由 EXT3 演变提升而来。EXT4 已成为目前大部分 Linux 发行版的默认文件系统。鉴于 EXT4 是最标准的文件系统，故目前 EXT4 的适配性是最好的。绝大多数运行在 Linux 上的软件几乎都是基于 EXT4 构建和测试的，因此兼容性上 EXT4 要优于其他文件系统。

而作为高性能的 64 位日志文件系统（journaling file system），XFS 表现出了高性能、高伸缩性，因此特别适用于生产服务器，特别是大文件（30+ GB）操作。很多存储类的应用都适合选择 XFS 作为底层文件系统。目前 RHEL 7.0 已然将 XFS 作为默认的文件系统。

至于采用哪种文件系统实际上并没有统一的规定。事实上，根据笔者的经验，上面两种文件系统都能很好地与 Kafka 集群进行适配。只不过在使用时每种文件系统都有一些特定的配置。

对于使用 EXT4 的用户而言，Kafka 建议设置以下选项。

- 设置 `data=writeback`：默认是 `data=ordered`，即所有数据在其元数据被提交到日志（journal）前，必须要依次保存到文件系统中；而 `data=writeback` 则不要求维持写操作顺序。数据可能会在元数据提交之后才被写入文件系统。据称这是一个提升吞吐量的好方法，同时还维持了内部文件系统的完整性。不过该选项的不足在于，文件系统从崩溃恢复后过期数据可能出现在文件中。不过对不执行覆盖操作且默认提供最少一次处理语义的 Kafka 而言，这是可以忍受的。用户需要修改 `/etc/fstab` 和使用 `tune2fs` 命令来设置该选项。
- 禁掉记日志操作：日志化（journaling）是一个权衡（trade-off），它能极大地降低系统从崩溃中恢复的速度，但同时也引入了锁竞争导致写操作性能下降。对那些不在乎启动速度但却想要降低写操作延时的用户而言，禁止日志化是一个不错的选择。用户可执行 `tune2fs -O ^has_journal <device_name>` 来禁止 journaling。
- `commit=N_secs`：该选项设置每 N 秒同步一次数据和元数据，默认是 5 秒。如果该值设置得比较小，则可减少崩溃发生时带来的数据丢失；但若设置较大，则会提升整体吞吐量以及降低延时。鉴于 Kafka 已经在软件层面提供了冗余机制，故在实际生产环境中推荐用户设置一个较大的值，比如 1~2 分钟。
- `nobh`：只有当 `data=writeback` 时该值才生效，它将阻止缓存头部与数据页文件之间的关联，从而进一步提升吞吐量。设置方法为修改 `/etc/fstab` 中的 `mount` 属性，比如 `noatim,data=writeback,nobh,errors=remount-ro`。

对于 XFS 用户而言，推荐设置以下参数。

- `largeio`：该参数将影响 `stat` 调用返回的 I/O 大小。对于大数据量的磁盘写入操作而言，它能够提升一定的性能。`largeio` 是标准的 `mount` 属性，故可使用与 `nobh` 相同的方式设置。
- `nobarrier`：禁止使用数据块层的写屏障（write barrier）。大多数存储设备在底层都提供了基于电池的写缓存，故设置 `nobarrier` 可以禁掉阶段性的“写冲刷”操作，从而提高写性能。不过自 RHEL6 开始，`nobarrier` 已不被推荐，因为 `write barrier` 对系统的性能影响几乎可以忽略不计（大概 3%），而启用 `write barrier` 带来的收益要大于其负面影响。如果是 RHEL5 的用户，则可以考虑设置此 `mount` 选项。

9.3.3 设置 swapiness

笔者在这里不详细展开 Linux 的 swap 机制，只是讨论一下到底如何设置该值。一般 Linux 发行版会将该值默认设置为 60。很多教程和有经验的人都建议将该值设置为 0，即完全禁掉 swap 以提升内存使用率。

关于这一点，笔者有些不同意见。虽然 swap 开启时确实会拖慢机器的速度，但若 Kafka

“吃掉”了所有的物理内存，用户还可以通过 swap 来定位应用并及时处理。假设完全禁掉了 swap，当系统耗尽所有内存（out of memory, OOM）后，Linux 的 OOM killer 将会开启并根据一定法则选取一个进程杀掉（kill），这个过程对用户来说是不可见的，因此用户完全无法进行干预（比如在杀掉应用前保存状态等）。换句话说，一旦用户关闭了 swap，就意味着当 OOM 出现时有可能丢失一些进程的数据。因此我们禁掉 swap 的前提就是要确保你的机器永远不会出现 OOM，这对于生产环境上的 Kafka 集群而言，通常都是不能保证的。

笔者建议将 swap 限定在一个非常小的值，比如 1~10 之间。这样既预留了大部分的物理内存，同时也能保证 swap 机制可以帮助用户及时发现并处理 OOM。

临时修改 swappiness 可以使用 `sudo sysctl vm.swappiness=N` 来完成；若要永久生效，用户需要修改 `/etc/sysctl.conf` 文件增加 `vm.swappiness=N`，然后重启机器。

9.3.4 JVM 设置

对于比较新的 Kafka 版本（0.10.0.0 以后的版本）而言，笔者一律推荐使用 Java 8。事实上，Java 7 自 2015 年 4 月便已不再更新了，社区本已计划在 1.0.0 版本正式抛弃对 Java 7 的支持（最新的状态是要到 Kafka 2.0.0 才会摒弃对 Java 7 的支持），因此若从零搭建 Kafka 集群环境推荐用户使用 Java 8。

由于 Kafka 并未大量使用堆上内存（on-the-heap memory）而是使用堆外内存（off-the-heap memory），故不需要为 Kafka 设定太大的堆空间。生产环境中 6GB 通常是足够的，要知道以 LinkedIn 公司 1500+ 台的 Kafka 集群规模来说，其 JVM 设置中也就是 6GB 的堆大小。另外，由于是 Java 8，因此推荐使用 G1 垃圾收集器。下面给出一份典型的调优后 JVM 配置清单：

```
-Xmx6g -Xms6g -XX:MetaspaceSize=128m -XX:MaxMetaspaceSize=128m -XX:+UseG1GC
-XX:MaxGCPauseMillis= 20 -XX:InitiatingHeapOccupancyPercent= 35 -XX:
G1HeapRegionSize=16M -XX: MinMetaspaceFreeRatio=50 -XX:
MaxMetaspaceFreeRatio=85
```

对于使用 Java 7 的用户，可以参考下面的清单：

```
-Xms6g -Xmx6g -XX:PermSize=96m -XX:MaxPermSize=96m -XX:+UseG1GC -XX:
MaxGCPauseMillis=20 -XX:InitiatingHeapOccupancyPercent=35
```

9.3.5 其他调优

使用 Kafka 的用户有时候会碰到“too many files open”的错误，这就需要为 broker 所在机器调优最大文件部署符上限。调优可参考这样的公式：broker 上可能的最大分区数 ×（每个分

区平均数据量 / 平均的日志段大小 + 3)。这里的 3 是索引文件的个数。假设某个 broker 上未来要放置的最大分区数是 20，平均每个分区总的的数据量是 100GB（不考虑 follower 副本），每个日志段大小是 1GB，那么这台 broker 所在机器的最大文件部署符大小就大概是 $20 \times (100\text{GB} / 1\text{GB} + 3)$ ，即 2060。当然考虑到 broker 还会打开多个底层的 Socket 资源，实际一般将该值设置得很大，比如 100000。

在实际线上 Linux 环境中，如果单台 broker 上 topic 数过多，用户可能碰到 `java.lang.OutOfMemoryError: Map failed` 的严重错误。这是因为大量创建 topic 将极大地消耗操作系统内存，用于内存映射操作。在这种情况下，用户需要调整 `vm.max_map_count` 参数。具体方法可以使用命令 `/sbin/sysctl -w vm.max_map_count = N` 来设置。该参数默认值是 65536，可以考虑为线上环境设置更大的值，如 262144 甚至更大。

另外如果 broker 所在机器上有多块物理磁盘，那么通常推荐配置 Kafka 全部使用这些磁盘，即设置 broker 端参数 `log.dirs` 指定所有磁盘上的不同路径。这样 Kafka 可以同时读/写多块磁盘上的数据，从而提升系统吞吐量。需要注意的是，当前 Kafka 根据每个日志路径上分区数而非磁盘容量来做负载均衡，故在实际生产环境中容易出现磁盘 A 上有大量剩余空间但 Kafka 却将新增的分区日志放置到磁盘 B 的情形。用户需要实时监控各个路径上的分区数，尽量保证不要出现过度倾斜。一旦发生上述情况，用户可以执行 `bin/kafka-reassign-partitions.sh` 脚本，通过手工的分区迁移把占用空间多的分区移动到其他 broker 上来缓解这种不平衡性。

9.4 调优吞吐量

介绍了基础调优，下面来看看如何调优 Kafka 吞吐量。其实说到具体目标的调优，很多使用者都不知道如何开始。若要调优 TPS，producer、broker 和 consumer 都需要进行调整，以便让它们在相同的时间内传输更多的数据。

众所周知，Kafka 基本的并行单元就是分区。producer 在设计时就被要求能够同时向多个分区发送消息，这些消息也要能够被写入到多个 broker 中供多个 consumer 同时消费。因此通常来说，分区数越多 TPS 越高。那么这是否意味着我们每次创建 topic 时都要创建大量的分区呢？答案显然是否。这依然是一个权衡（trade-off）的问题。

过多的分区可能有哪些弊端呢？首先，server/clients 端将占用更多的内存。producer 默认使用缓冲区为每个分区缓存消息，一旦满足条件 producer 便会批量发出缓存的消息。看上去这是一个提升性能的设计，不过由于该参数是分区级别的，因此如果分区很多，这部分缓存的内存占用也会变大；而在 broker 端，每个 broker 内部都维护了很多分区级别的元数据，比如

controller、副本管理器、分区管理等。显然，分区数越多，缓存成本越大。

其次，每个分区在底层文件系统都有专属目录。该目录下除了 3 个索引文件之外还会保存日志段文件。通常生产环境下的日志段文件可能有多个，因此保守估计一个分区就可能要占用十几个甚至几十个文件句柄——当前 Kafka 一旦打开文件便不会显式关闭该文件，故文件句柄是不会被释放的。那么随着分区数的增加，系统文件句柄数也会相应地增长。

最后，每个分区通常都有若干个副本而副本保存在不同的 broker 上。当 leader 副本挂掉了，controller 会自动检测到，然后在 ZooKeeper 的帮助下选择新的 leader。虽然在大部分情况下 leader 选举只有很短的延时，但若分区数很多，当 broker 挂掉后，需要进行 leader 选举的分区数就会很多。当前 controller 是单线程处理事件的，所以 controller 只能一个一个地处理 leader 变更请求，可能会拉长整体系统恢复的时间。

基于以上 3 点，分区数的选择绝不是多多益善的。用户需要结合自身的实际环境基于吞吐量等指标进行一系列测试来确定需要多少分区数。有的用户总是抱怨说自身环境无法达到 Kafka 官网给出的性能结果。其实官网的基准测试对用户的实际意义并不大，因为不同的硬件、软件、负载情况必然会带来不同的测试结果。比如用户使用 1KB 大小的消息进行测试，最后发现吞吐量才 1MB/s，而官网说每秒能达到 10MB/s。这是因为官网使用的是 100 字节的消息体进行测试的，故根本没有可比性。

虽然没法给出统一的分区数，但用户基本上可以遵循下面的步骤来尝试确定分区数。

- 创建单分区的 topic，然后在实际生产机器上分别测试 producer 和 consumer 的 TPS，分别为 T_p 和 T_c 。
- 假设目标 TPS 是 T_t ，那么分区数大致可以确定为 $T_t / \max(T_p, T_c)$ 。

Kafka 提供了专门的脚本 kafka-producer-perf-test.sh 和 kafka-consumer-perf-test.sh 用于计算 T_p 和 T_c 。值得说明的是，测试 producer 的 TPS 通常是很容易的，毕竟逻辑非常简单，直接发送消息给 Kafka broker 即可；但测试 consumer 就与应用关系很大了，特别是与应用处理消息的逻辑有关。测试 consumer 时尽量使用真实的消息处理逻辑，这样测量的结果才能准确地反映线上环境。

在确定了分区数之后，我们分别从 producer、broker 和 consumer 这 3 个方面来讨论如何调优 TPS。

如前所述，producer 是批量发送消息的，它会将消息缓存起来后在一个发送请求中统一发送它们。若要优化 TPS，那么最重要的就是调优批量发送的性能参数：批次大小（batch size）和批次发送间隔，即 Java 版本 producer 参数 batch.size 和 linger.ms。通常情况下，增加这两个参数的值都会提升 producer 端的 TPS。更大的 batch size 可以令更多的消息封装进同一个请求，

故发送给 broker 端的总请求数会减少。此举既减少了 producer 的负载，也降低了 broker 端的 CPU 请求处理开销；而更大的 `linger.ms` 使 producer 等待更长的时间才发送消息，这样就能够缓存更多的消息填满 batch，从而提升了整体的 TPS。当然这样做的弊端在于消息的延时增加了，毕竟消息不是即时发送了。

就像前面说的，分区数的增加总要有个度，当增加到某个数值后由于锁竞争和内存占用过多等因素就会出现 TPS 的下降。在实际环境中，用户可以以 2 的倍数来逐步增加分区数进行测试，直至出现性能拐点。

除了上面这两个参数，producer 端的另一个参数 `compression.type` 也是调优 TPS 的重要手段之一。对消息进行压缩可以极大地减少网络传输量，降低网络 I/O 开销从而提升 TPS。由于压缩是针对 batch 做的，因此 batch 的效率也直接影响压缩率。这通常意味着 batch 中缓存的消息越多，压缩率越好。当前 Kafka 支持 GZIP、Snappy 和 LZ4，但由于目前一些固有配置等原因，Kafka + LZ4 组合的性能是最好的，因此推荐在那些 CPU 资源充足的环境中启用 producer 端压缩，即设置 `compression.type=lz4`。

当 producer 发送消息给 broker 时，消息被发送到对应分区 leader 副本所在的 broker 机器上。默认情况下，producer 会等待 leader broker 返回发送结果，这时才能知晓这条消息是否发送成功，consumer 端也只能消费那些已成功发送的消息。显然等待 leader 返回这件事情也会影响 producer 端 TPS：leader broker 返回的速度越快，producer 就能更快地发送下一条消息，因此 TPS 也就越高。producer 端参数 `acks` 控制了这种行为，默认值等于 1 表示 leader broker 把消息写入底层文件系统即返回，无须等待 follower 副本的应答。用户也可以将 `acks` 设置成 0，则表示 producer 端压根不需要 broker 端的响应即可开启下一条消息的发送。这种情况会提升 producer 的 TPS，当然是以牺牲消息持久化为代价的。

另外当设置 `acks` 不等于 0 时，一旦消息发送失败，producer 端会根据 `retries` 参数设置的值进行指定次数的重试。如果应用程序能够忍受偶发的消息丢失，那么可以将 `retries` 设置为 0，即不重试。这样 producer TPS 也会得到提升。只不过此时，一旦消息发送失败，producer 将不会尝试再次发送消息。应用端需要显式捕获 leader broker 发回的异常自行处理发送失败。

对于 Java 版本 producer 而言，它需要创建一定大小的缓冲区来缓存消息。当缓冲区被填满后，producer 立即进入阻塞状态直到有空闲内存被释放出来。这段阻塞等待时间最长不会超过 producer 端参数 `max.block.ms` 设置的值。一旦超过，producer 会抛出 `TimeoutException`，因此确保 producer 不会抛出此异常的关键在于设置缓冲区的大小，即 producer 端参数 `buffer.memory` 的值。该参数默认是 32MB，通常来说用户是不需要调整的，但若在实际应用中发现 producer 在高负载情况下经常抛出 `TimeoutException`，则可以考虑增加此参数的值。由于 Java 版本 producer 的主类 `KafkaProducer` 是线程安全的，因此很多用户在生产环境中可能使

用多线程共享一个 `KafkaProducer` 实例，所以缓冲区就更容易被快速填满，故在这种情况下增加 `buffer.memory` 就显得更加必要了。增加此值后，`producer` 阻塞的情况将得到缓解，从而比之前缓存更多分区的数据，因而整体上提升了 TPS。

同样地，对于 Java 版本 `consumer` 来说，用户可以调整 `leader` 副本所在 `broker` 每次返回的最小数据量来间接影响 TPS——这就是 `consumer` 端参数 `fetch.min.bytes` 的作用。该参数控制了 `leader` 副本每次返回 `consumer` 的最小数据字节数。通过增加该参数值，Kafka 会为每个 `FETCH` 请求的 `response` 填入更多的数据，从而减少了网络开销并提升了 TPS。当然它和 `producer` 端的 `batch` 类似，在提升 TPS 的同时也会增加 `consumer` 的延时，这是因为该参数增加后，`broker` 端必须花额外的时间积累更多的数据才发送 `response`。因此用户需要结合自身的调优目标有选择地调整。

另外，如果机器和资源充足，最好使用多个 `consumer` 实例共同消费多分区数据。令这些实例共享相同的 `group id`，构成 `consumer group` 并行化消费过程，能够显著地提升 `consumer` 端 TPS。在实际环境中，笔者推荐用户最好启动与待消费分区数相同的实例数，以保证每个实例都能分配一个具体的分区进行消费。

对于 `broker`，笔者推荐用户增加参数 `num.replica.fetchers` 的值。该值控制了 `broker` 端 `follower` 副本从 `leader` 副本处获取消息的最大线程数。默认值 1 表明 `follower` 副本只使用一个线程去实时拉取 `leader` 处的最新消息。对于设置了 `acks=all` 的 `producer` 而言，主要的延时可能都耽误在 `follower` 与 `leader` 同步的过程，故增加该值通常能够缩短同步的时间间隔，从而间接地提升 `producer` 端的 TPS。

最后谈一下 JVM 垃圾回收 (GC) 对 TPS 的影响。用户需要特别注意监控 GC 的停顿时间 (pause time)，确保不要出现经常性的长停顿。如果用户使用的是 CMS 垃圾收集器，确保不要频繁抛出 `concurrent mode failure` 的错误；如果用户使用的是 G1 垃圾收集器，则需要确保不出现 `Evacuation Failure`。这两种情况都将导致 GC 采用单线程进行一次 Full GC，而该 GC 过程是 `Stop-The-World` 的，Kafka 所有线程都将被暂停，因此会极大地伤害 TPS。事实上，GC 的性能对于老版本 `consumer` 的影响更加显著。老版本 `consumer` 极度依赖 ZooKeeper 会话来表征 `consumer` 存活情况，故若 GC 停顿时间过长，将导致 ZooKeeper 会话超期，Kafka 会立即对 `group` 进行 `rebalance`。若老版本 `consumer` 用户在实际生产环境中观测到 `consumer group` 经常 `rebalance`，则需要仔细检查一下是否是由 GC 导致的。笔者认为，新版本 `consumer` 在设计上弃用了对 ZooKeeper 的依赖，这在一定程度上也是为了规避这个负面影响。

好了，下面总结一下调优 TPS 的一些参数清单和要点。

broker 端

- 适当增加 num.replica.fetchers，但不要超过 CPU 核数。
- 调优 GC 避免经常性的 Full GC。

producer 端

- 适当增加 batch.size，比如 100~512KB。
- 适当增加 linger.ms，比如 10~100 毫秒。
- 设置 compression.type= lz4。
- acks=0 或 1。
- retries=0。
- 若多线程共享 producer 或分区数很多，增加 buffer.memory。

consumer 端

- 采用多 consumer 实例。
- 增加 fetch.min.bytes，比如 100000。

9.5 调优延时

针对不同的组件，延时（latency）的定义可能不同。对 producer 而言，延时主要是消息发送的延时，即 producer 发送 PRODUCE 请求到 broker 端返回请求 response 的时间间隔；对 consumer 而言，延时衡量了 consumer 发送 FETCH 请求到 broker 端返回请求 response 的时间间隔。还有一种延时定义表示的是集群的端到端延时（end-to-end latency），即 producer 端发送消息到 consumer 端“看到”这条消息的时间间隔。不论是哪种延时，它们调优的思想大致是相同的。

虽然在 9.4 节中我们谈到适度地增加分区数会提升 TPS，但大量分区的存在对于延时却是损害。分区数越多，broker 就需要越长的时间才能够实现 follower 与 leader 的同步。在同步完成之前，设置 acks=all 的 producer 不会认为该请求已完成，而 consumer 端更无法看到这些未提交成功的消息，因此这样既影响了 producer 端的延时也增加了 consumer 端的延时。若要调优延时，我们必须限制单台 broker 上的总分区数，缓解的办法有 3 种：①不要创建具有超多分区数的 topic；②适度地增加集群中 broker 数分散分区数；③和调优 TPS 类似，增加 num.replica.fetchers 参数提升 broker 端的 I/O 并行度。

和调优 TPS 相反的是，调优延时要求 producer 端尽量不要缓存消息，而是尽快地将消息发送出去。这就意味着最好将 linger.ms 参数设置成 0，不要让 producer 花费额外的时间去缓存

待发送的消息。

类似地，不要设置压缩类型。众所周知，压缩是用时间换空间的一种优化方式。为了减少网络 I/O 传输量，我们推荐启用消息压缩；但为了降低延时，我们推荐不要启用消息压缩。这样本用于压缩任务的 CPU 时钟周期被节省下来，producer 端能够更快地发送消息。当然这样做的负面影响是网络带宽增加了。总之在调优延时时，推荐设置 `compression.type=none`，即禁止消息压缩。

producer 端的 `acks` 参数也是优化延时的重要手段之一。leader broker 越快地发送 response，producer 端就能越快地发送下一批消息。该参数的默认值 1 实际上已经是一个非常不错的设置，但如果用户应用对于延时有着比较高的要求，但却能够容忍偶发的消息发送丢失，则可以考虑将 `acks` 设置成 0，在这种情况下 producer 压根不会理会 broker 端的 response，而是持续不断地发送消息，从而达成最低的延时。

在 consumer 端，用户需要调整 leader 副本返回的最小数据量来间接地影响 consumer 延时，即 `fetch.min.bytes` 参数值。对于延时来说，默认值 1 已经是一个很不错的选择，这样能够使 broker 尽快地返回数据，不花费额外的时间积累消费数据。

下面总结一下调优延时的一些参数清单。

broker 端

- 适度增加 `num.replica.fetchers`。
- 避免创建过多 topic 分区。

producer 端

- 设置 `linger.ms=0`。
- 设置 `compression.type=none`。
- 设置 `acks=1` 或 0。

consumer 端

- 设置 `fetch.min.bytes=1`。

9.6 调优持久性

顾名思义，持久性（durability）定义了 Kafka 集群中消息不容易丢失的程度。持久性越高表明 Kafka 越不会丢失消息。持久性通常由冗余来实现，而 Kafka 实现冗余的手段就是备份机制（replication）——它保证每条 Kafka 消息最终会保存在多台 broker 上。这样即使单个 broker

崩溃，数据依然是可用的。

对于有着高持久性需求的用户来说，保证 topic 数据不丢失最重要的就是设置 topic 的备份因子（replication factor，下称 rf）。参照 Hadoop 的三备份原则，用户可以设置 rf 为 3 以抵御因两台 broker 同时崩溃而可能造成的数据丢失。除了在创建 topic 时指定 rf，用户还需要注意自动创建 topic 的场景。Kafka 默认支持自动创建 topic，即当用户 producer 向一个不存在的 topic 发送消息时，Kafka 首先创建这个 topic，分区数和 rf 分别由 broker 端参数 num.partitions 和 default.replication.factor 指定。这两个参数的默认值都是 1，因此如果用户有自动创建 topic 的场景，推荐首先在 server.properties 文件中设置这两个参数的值，特别是设置 default.replication.factor。用户也可以设置 auto.create.topics.enable=false，从而禁止 Kafka 自动创建 topic。

若要达成高持久性，为用户创建 topic 提供充足的备份机制只是一个方面，Kafka 内部 topic 也要进行相应的配置。我们知道，新版本 consumer 已经把位移信息的存储从 ZooKeeper 转移到了内部 topic __consumer_offsets 中。虽然这个 topic 的 rf 也受到参数 default.replication.factor 的约束，但在 0.11.0.0 版本之前这种约束不是强制的。换句话说，在 0.11.0.0 版本之前，即使当前 broker 数小于 default.replication.factor 值，__consumer_offsets 也是被允许创建出来的，而且设置其 rf 值为当前 broker 数。__consumer_offsets 被自动创建的场景大致有 5 或 6 个，读者只需要记住最重要的一个：当集群中第一个 consumer 开始工作时，__consumer_offsets 会被创建。所以使用 0.11.0.0 之前版本的用户一定要在开始消费前确保已经正确设置了 default.replication.factor 并且保证当前 broker 数不小于该值，只有这样才能满足高持久性的要求。至于使用 Kafka 0.11.0.0 及以后版本的用户则不用担心此事，因为自该版本起，Kafka 会强制要求 __consumer_offsets 创建时必须满足 default.replication.factor 参数指定的副本数才能进行创建。

对于 producer 而言，高持久性与 acks 的设置息息相关。细心的读者会发现，acks 的设置对于调优 TPS 和延时都有一定的作用，但 acks 参数最核心的功能实际上是控制 producer 的持久性。毋庸置疑，若要达成最高的持久性必须设置 acks=all（或 acks=-1），即强制 leader 副本等待 ISR 中所有副本都响应了某条消息后发送 response 给 producer。ISR 副本全都响应消息写入意味着 ISR 中所有副本都已将消息写入底层日志，这样只要 ISR 中还有副本存活，这条消息就不会丢失。这虽然会增加 producer 端消息发送的延时，但却具有最高的持久性保障。

另一个提升持久性的参数是 producer 端的 retries。在 producer 发送失败后，producer 视错误情况而有选择性地自动重试发送消息。我们设想，如果网络临时“抖动”造成某条消息发送失败，那么一个比较大的 retries 值就能很好地规避这种瞬时问题。producer 通过重试机制自动地解决了网络问题，用户不需要手动解决——实际上，用户手动解决通常也是手动重试——因

此生产环境中推荐将该值设置为一个较大的值。当然，在设置之前，读者一定要明确设置 `retries` 的两个“副作用”。

首先，当 `producer` 重试时，待发送的消息可能已经发送成功。假设网络在消息被写入到 `broker` 底层日志与 `broker` 发送 `response` 给 `producer` 之间发生故障，那么 `response` 就无法顺利发回给 `producer`，于是 `producer` 端开始重试。最终结果是同一条消息被写入了两次，即重复消息（`duplicated message`）。出现这种问题后，`consumer` 应用需要自行处理“去重”逻辑，因此增加了整体系统的复杂度。好消息是自 0.11.0.0 版本开始，Kafka 提供了幂等性 `producer`，实现了精确一次的处理语义。幂等性 `producer` 保证同一条消息只会被 `broker` 写入一次，因此很好地解决了这个问题。启用幂等性 `producer` 的方法也十分简单，只需要设置 `producer` 端参数 `enable.idempotence=true`。

其次，因为重试发生的时机不是固定的，因此有可能出现这种场景：`producer` 依次发送消息 `m1` 和 `m2`，`m1` 写入失败但 `m2` 写入成功，当 `producer` 重试 `m1` 发送成功后，`m1` 就位于 `m2` 的后边，从而造成消息乱序。对于有顺序要求的应用来说，用户可以设置 `max.in.flight.requests.per.connection=1` 来规避这个问题。设置该参数的目的是保证 `producer` 在单个 `broker` 连接上在任意某个时刻只处理一个请求，若管道中存在未完成的请求，`producer` 不会发送新的请求，这样就能避免乱序问题。不过设置该参数为 1 很有可能会拉低 `producer` 的 TPS。

用户还需要设置某些 `broker` 端参数来保证持久性。首先就是 `unclean.leader.election.enable`。前面章节中说过，当 `broker` 出现崩溃时 `controller` 会自动检测出宕机的 `broker` 并为该 `broker` 上的所有分区重新选举 `leader`。选择的范围是从 ISR 副本集合中挑选，但若 ISR 副本所在 `broker` 全部宕机，Kafka 默认会从未非 ISR 副本中选举 `leader`。选择这些副本当 `leader` 可能造成数据丢失（毕竟这些副本尚未与 `leader` 同步，有些数据还没有写到日志中），但却维护了 Kafka 服务的高可用性。这个 `leader` 选举过程被称为 `unclean leader` 选举。是否允许 `unclean` 选举由 `unclean.leader.election.enable` 参数控制。在 0.11.0.0 版本之前其默认值一直是 `true`，表明了社区开发人员更加看重服务高可用性，但最近人们对数据的安全性和完整性有了更严格的要求，将该参数默认值变为 `false` 的呼声也越来越高，社区终于在 0.11.0.0 版本起正式将该参数默认值变更为 `false`。因此对于有高持久性需求的 0.11.0.0 之前版本的用户，推荐将该值设置为 `false`，避免因 `unclean leader` 选举而造成的数据丢失。

另一个重要的参数是 `min.insync.replicas`，它指定了若要成功写入某条消息则必须要等待响应完成的最少 ISR 副本数。虽然它是一个 `broker` 端的配置，但实际上该参数是与 `producer` 端 `acks` 参数配合使用的，而且是在 `acks=all` 时才会生效。也就是说如果 `producer` 端 `acks≠all`（即 `acks≠-1`），那么用户完全可以无视该参数。实际场景中通常推荐设置该值为副本因子-1：如

果 `replication-factor=3`，那么可以设置 `min.insync.replicas=2`，保证消息至少要被两个 broker 成功接收才算写入成功。

自 0.10.0.0 版本起，Kafka 仿照 Hadoop 为 broker 引入了机架属性信息（rack），由参数 `broker.rack` 设定。用户可以为每台 broker 指定机架信息，而 Kafka 集群在后台会收集所有的机架信息并在创建分区时根据这些信息将分区分散到不同的机架上。这样，即使整个机架机器全部宕机，数据持久性依然能够得到保障。这个功能对于大型数据中心或公有云上的 Kafka 集群来说至关重要，不仅有效地降低了维护成本，更是极大地提升了对抗机架整体宕机的风险应对。

broker 端调优持久性的另两个重要参数分别是 `log.flush.interval.ms` 和 `log.flush.interval.message`。事实上，它们是同一事物在不同纬度上的表现。前者指定了 Kafka 多长时间执行一次消息“落盘”，是时间维度上的参数；后者指定了 Kafka 写入多少条消息后执行一次消息“落盘”，是频率维度上的参数。默认情况下，`log.flush.interval.ms` 是空而将 `log.flush.interval.message` 设置为 `Long.MAX_VALUE`，这表示 Kafka 实际上不会自动地执行消息“冲刷”操作——事实上这就是 Kafka 开发人员设计的初衷，即把消息同步到磁盘的工作交由操作系统来完成，由 OS 控制页缓存数据到物理文件的同步，毕竟 OS 最擅长做这些事情。但是对于超过持久性需求的用户来说，如果不介意 TPS 下降，则可以设置这两个参数来强行让 Kafka 自己“冲刷”数据。举一个极端的例子，假设用户想要每条消息都及时同步到磁盘，那么可以显式设置 `log.flush.interval.message=1`。

在 consumer 端控制持久性的主要手段就是设置位移提交的方式。既然位移决定了 consumer 消费的进度，那么何时提交位移于持久性而言就显得非常重要了。首先，consumer 不能在处理消息前就提交位移，因为一旦 consumer 尚未处理消息便崩溃，而此时位移已经提交，那么 consumer 重启后无法消费这条消息，这条消息就丢失了。在实际应用中笔者几乎总是推荐用户关闭自动位移提交，即设置 `auto.commit.enable=false`。默认情况下，该参数是 `true`，即 consumer 在后台定期提交位移。另外，既然是手动提交，用户需要调用 `commitSync` 方法来提交位移，而不使用 `commitAsync` 方法。

下面总结一下调优持久性的参数清单和要点。

broker 端

- 设置 `unclean.leader.election.enable=false`（0.11.0.0 之前版本）。
- 设置 `auto.create.topics.enable=false`。
- 设置 `replication.factor = 3`，`min.insync.replicas = replication.factor - 1`。
- 设置 `default.replication.factor = 3`。
- 设置 `broker.rack` 属性分散分区数据到不同机架。

- 设置 `log.flush.interval.message` 和 `log.flush.interval.ms` 为一个较小的值。

producer 端

- 设置 `acks=all`。
- 设置 `retries` 为一个较大的值，比如 10~30。
- 设置 `max.in.flight.requests.per.connection=1`。
- 设置 `enable.idempotence=true` 启用幂等性。

consumer 端

- 设置 `auto.commit.enable=false`。
- 消息消费成功后调用 `commitSync` 提交位移。

9.7 调优可用性

所谓可用性（availability）反映的是 Kafka 集群应对崩溃的能力。无论 broker、producer 或 consumer 出现崩溃，Kafka 服务依然保持可用状态而不会因为 failure 就中断服务。调优可用性就是要让 Kafka 更快地从崩溃中恢复。下面我们分别从 topic、producer、broker 和 consumer 这 4 个方面来讨论如何调优可用性。

从 topic 层面来看，当 topic 分区数很多时，一旦分区 leader 副本所在的 broker 发生故障，就需要进行 leader 选举。虽然 leader 选举的时间一般都很短，大概只有几毫秒，但目前 controller 处理请求是单线程的，大量的 leader 选举请求（严格来说不应该称为 leader 选举请求，应该是 leader 选举之后 broker 需要执行的逻辑）只能逐一排队完成，而且在受影响分区 leader 选举之前无论 producer 还是 consumer 都无法工作，故 topic 分区数越多，恢复的时间就越慢。因此用户需要谨慎地结合自身硬件环境以及调优目标，根据之前所说原则确立 topic 的分区数。笔者曾经见过国外用户为 topic 创建了 2000 个分区，导致 Kafka 出现各种问题（比如 leader 选举慢，ZooKeeper 超时、Java 堆溢出等）。该用户只是单纯地认为分区数多多益善，但实则不然。在生产环境下万万不可“拍脑袋”式地决定 topic 分区数，一定要经过谨慎的测试来确定。

在 producer 端，与可用性相关的参数首推 `acks`。当 `acks` 设置为 `all` 时，broker 端参数 `min.insync.replicas` 的效果会影响 producer 端的可用性。该参数值设置得越大，Kafka 就会强制越多的 follower 副本同步写入日志。当出现 ISR 缩减到小于 `min.insync.replicas` 值时，producer 停止对特定分区发送消息，从而最终影响了服务的可用性。但如果设置一个很小的值（比如 1），那么 producer 对 ISR 缩减就有了更强的“免疫力”，即它能容忍更多的副本被“踢出”ISR。只要剩余的 ISR 副本数大于 `min.insync.replicas` 即可。此时 producer 可以一直正常工作，

不会出现服务中断的情况，因此实现了高可用性。

除了 `min.insync.replicas` 参数，broker 端影响高可用性最直接的表现方式就是 broker 崩溃。如前所述，崩溃之后 leader 选举有两种：①先从 ISR 中选；②视 `unclean.leader.election.enable` 值决定是否从非 ISR 中选。`unclean.leader.election.enable=true` 将允许 controller 从非 ISR 中选择 leader。这使得 Kafka 总是能够为受影响分区选择 leader，因而保证了服务的高可用性。这么做的前提是用户已明确知晓并承担因 unclean leader 选举而引起的数据丢失的风险。broker 端影响高可用性的另一个重要参数是 `num.recovery.threads.per.data.dir`。当 broker 从崩溃中重启恢复后，broker 首先会扫描并加载底层的分区数据执行清理和与其他 broker 的同步任务。这一过程被称为日志加载（log loading）或日志恢复（log recovery）。默认情况下，每个 broker 都只使用一个线程来做这些事情。打个比方，假设某个 broker 为 `log.dirs` 配置了 10 个日志目录，那么该 broker 使用单线程顺序扫描加载这些目录。显然，每个日志目录的加载任务是互不依赖的，非常适合多线程来做。因此 Kafka 的 broker 端 `num.recovery.threads.per.data.dir` 参数就是用于此目的的。适度增加该参数值可以显著缩短日志加载或日志恢复的时间，提升服务从崩溃中恢复的能力。在实际使用中，一般将该值配置为 broker 所在机器的物理磁盘数。

对 consumer 而言，高可用性主要是由基于组管理的 consumer group 来体现的。当 group 下某个或某些 consumer 实例“挂了”，group 的 coordinator 能够自动检测出这种崩溃并及时地开启 rebalance 将这些 consumer 实例的分区分配给其他存活实例。显然我们想要让 coordinator 能够快速检测出这些 failure。consumer 端参数 `session.timeout.ms` 就是用来做这件事情的。coordinator 保证检测出 failure 的时间最多不会超过 `session.timeout.ms` 指定的值，因此若要实现 consumer 端高可用，可以设置该参数为一个较低的值，比如 5~10 秒。

严格来说，上面的 failure 实际上涵盖了两种情况。第一种是崩溃，也就是真正意义上的 failure，表现为 consumer 实例所在机器“掉电”或外部被 KILL -9。coordinator 只能保证在 `session.timeout.ms` 段时间内检测出这种情况并开启新一轮 rebalance。第二种情况则是由软件或 consumer 应用自身的问题导致，比如消息处理时间过长从而导致超时。此时 consumer 将停止发送心跳，并显式告知 coordinator 它要离开该 group。针对这种情况，用户需要增加 `max.poll.interval.ms` 参数的值赋予 consumer 实例更多的时间来处理消息，从而维护 consumer 的高可用性。注意，`max.poll.interval.ms` 是在 Kafka 0.10.1.0 版本时引入的，使用之前版本的用户需要减少 `max.partition.fetch.bytes` 值来实现相同的目的。

下面总结一下调优可用性的一些参数清单。

broker 端

- 避免创建过多分区。

- 设置 `unclean.leader.election.enable=true`。
- 设置 `min.insync.replicas=1`。
- 设置 `num.recovery.threads.per.data.dir=broker` 端参数 `log.dirs` 中设置的目录数。

producer 端

- 设置 `acks=1`，若一定要设置为 `all`，则遵循上面 `broker` 端的 `min.insync.replicas` 配置。

consumer 端

- 设置 `session.timeout.ms` 为较低的值，比如 10000。
- （0.10.1.0 及之后版本）设置 `max.poll.interval.ms` 为比消息平均处理时间稍大的值。
- （0.10.1.0 之前版本）设置 `max.poll.records` 和 `max.partition.fetch.bytes` 减少 `consumer` 处理消息的总时长，避免频繁 `rebalance`。

9.8 本章小结

本章对 Kafka 集群的调优技术进行了整体性的介绍，让读者了解调优的目标以及调优的基本思路 and 具体方法。掌握了这些内容，将有助于读者在实际生产环境中对具体的性能问题进行系统性的分析。

在第 10 章中我们将介绍 Kafka 社区于 0.10.0.0 版本引入的两个具有里程碑意义的新组件 Kafka Connect 和 Kafka Streams。

第 10 章

Kafka Connect 与 Kafka Streams

终于来到最后一章了。在前面的章节中我们系统地学习了实际使用 Apache Kafka 的各个方面。事实上，到目前为止，读者应该已经能够独立地在生产环境中部署一套完整的 Kafka 集群。本章我们将关注 Apache Kafka 0.10.0.0 版本新引入的两个特性 Kafka Connect 和 Kafka Streams。

学习本章，你将了解到以下内容。

- Kafka Connect 简介及使用。
- 流式处理与 Kafka Streams。
- Kafka Streams 使用。

10.1 引言

Apache Kafka 0.9.0.0 和 0.10.0.0 版本分别引入了两个全新的组件 Kafka Connect 和 Kafka Streams。在详细介绍这两个新特性之前，笔者想多花一些笔墨来讲述一下 Kafka 引入它们的原因。随着 Apache Kafka 被越来越多的用户使用，Kafka 社区开发团队逐渐意识到大部分用户倾向于将 Kafka 作为消息队列来使用。事实上，这在当时就是最正确的用法。如果我们翻开 Kafka 0.9.0.0 版本的官方文档（<https://kafka.apache.org/090/documentation.html#introduction>），我们会发现官网对 Kafka 的定位是分布式的、分区化的、带备份机制的提交日志服务（distributed, partitioned and replicated commit log service），从本质上说它就是一个分布式消息队列。

既然是消息队列，必然存在上下游的处理子系统来处理消息。这就需要有一套系统负责处理消息在 Kafka 与其他系统间的搬进搬出，同时还额外需要一套系统来实现消息的业务处理逻辑。这两套系统就是我们常说的数据连接器（下称 connector）和数据处理系统。当前，在这

两个领域内分别有很多著名的开源框架：connector 中有名的框架包括 Apache Flume、Kettle¹、Facebook Scribe 等；而数据处理系统领域内的明星框架则更多，耳熟能详的如 Apache Hadoop、Apache Spark、Apache Storm 和 Apache Flink 等。

在实际生产环境中，Kafka 多与这些框架配合使用共同完成对业务数据的处理，因此 Kafka 社区开发人员开始思考：虽然 Kafka 本身承担着“消息中转站”的角色，但既然它已经保存了待处理的数据，为何不自己提供一套 connector 接口以及数据处理组件呢？这样用户便可只需要部署和维护一套系统就轻松实现上面三套系统才能实现的功能——这便是社区引入 Kafka Connect 和 Kafka Streams 的初衷。由此可以看出 Kafka 社区不仅要为用户提供一站式的业务消息处理能力，同时还有在消息处理特别是流式处理上一统江湖的“野心”。在这套解决方案中，Kafka Connect 负责 Kafka 与外部系统的数据集成，Kafka Streams 则是轻量级的流式处理组件。如果用 Shell 命令来展示 Kafka Connect、Kafka Streams 与 Kafka core 之间的关系，可以参见图 10.1。

如图 10.1 所示，Kafka connector 类似于 Shell 中重定向的作用（如<，>），负责将数据从外部系统转移到 Kafka 或从 Kafka 中转移到其他系统，Kafka core 则负担管道或通道（channel）的作用（如|），负责保存待处理或已处理过的数据；而 Kafka Streams 则执行真正的处理逻辑，如 grep 和 tr 命令一般。

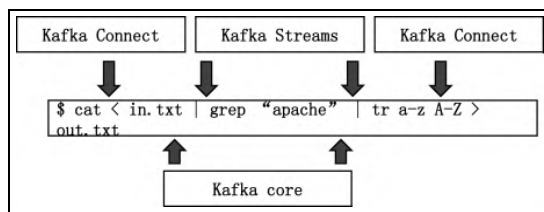


图 10.1 Kafka 组件关系

在简要介绍了引入 Kafka Connect 和 Kafka Streams 的背景之后，下面分别讲述两者的使用方法。

10.2 Kafka Connect

10.2.1 概要介绍

Kafka Connect 是一个高伸缩性、高可靠性的数据集成工具，用于在 Apache Kafka 与其他系

¹实际上 Kettle 是一个 ETL 工具，但笔者认为简单的 ETL 逻辑不属于消息处理系统的范畴。

统间进行数据搬运以及执行 ETL 操作，比如 Kafka Connect 能够将文件系统中某些文件的内容全部灌入 Kafka topic 中或者是把 Kafka topic 中的消息导出到外部的数据库系统，如图 10.2 所示。



图 10.2 Kafka Connect 架构图

如图 10.2 所示，Kafka Connect 主要由 source connector 和 sink connector 组成。事实上，几乎大部分的 ETL 框架都是由这两大类逻辑组件组成的，如 Apache Flume、Kettle 等。source connector 负责把输入数据从外部系统中导入到 Kafka 中，而 sink connector 则负责把输出数据导出到其他外部系统。

根据 Kafka Connect 官网的介绍，目前其主要的设计特点如下。

- 通用性：依托底层的 Kafka 核心系统封装了 connector 接口，方便开发、部署和管理。
- 兼具分布式（distributed）和单体式（standalone）两种模式：既可以以 standalone 单进程的方式运行，也可以扩展到多台机器成为分布式 ETL 系统。
- REST 接口：提供常见的 REST API 方便管理和操作，只适用于分布式模式。
- 自动位移管理：connector 自动管理位移，无须开发人员干预，降低开发成本。
- 集成性：方便与流/批处理系统对接。

显然，一个 ETL 框架或 connector 系统是否好用的主要标志之一就是，看 source connector 和 sink connector 的种类是否丰富。默认提供的 connector 越多，我们就能集成越多的外部系统，免去了用户自行开发的成本。不过令人遗憾的是，目前社区版本的 Kafka（即本书一直讨论的 Apache Kafka）只提供了与文件系统中文件的数据集成 connector，即 FileStreamSourceConnector 和 FileStreamSinkConnector，大部分与其他系统交互的 connector 在 GitHub 上都是由个人维护的，而非官方认证的。Confluent 公司（第 4 章中提到过）发行的 Kafka 版本则包含了较为丰富的 connector 种类，详见表 10.1。

表 10.1 Kafka 发行版支持的 connector

connector 名称	描 述	Apache Kafka	Confluent Kafka
File（source）	文件 source connector	✓	✓
File（sink）	文件 sink connector	✓	✓
HDFS（sink）	HDFS、Hadoop	✗	✓

续表

connector 名称	描 述	Apache Kafka	Confluent Kafka
JDBC (source)	JDBC、MySQL	✗	✓
JDBC (sink)	JDBC、MySQL	✗	✓
Elasticsearch (sink)	Elasticsearch	✗	✓

鉴于 Confluent 公司的 Kafka 发行版已超出本书讨论范围，读者只需要了解目前 Apache Kafka（社区版）只提供了 file connector。用户如果要使用其他 connector，要么到 GitHub 上面去搜寻有无现成的 connector，要么自行开发一个。后面我们会给出一个开发 connector 的例子。

值得注意的是，由于 Kafka Connect 还在不断地演进中，故读者需要时刻关注官网给出的最新动态。本章讨论的概念、设计以及使用教程很有可能会随着后续版本的演进而失效。官网地址参见 <https://kafka.apache.org/documentation/#connect>。

10.2.2 standalone Connect

前面说过，Kafka Connect 有两种运行模式 standalone 和 distributed。本节介绍如何在 standalone 模式下运行 Connect。

在 standalone 模式下所有的操作都是在一个进程中完成的。这种模式非常适合运行在测试或功能验证环境，抑或是必须是单线程才能完成的场景（比如收集日志文件）。由于是单进程，standalone 模式无法充分利用 Kafka 天然提供的负载均衡和高容错等特性。

下面我们在一个单节点的 Kafka 集群上运行 standalone 模式的 Kafka Connect，把输入文件 foo.txt 中的数据通过 Kafka 传输到输出文件 bar.txt 中。首先我们制作配置文件。Kafka Connect standalone 模式下通常有 3 类配置文件：connect 配置文件，若干 source connector 配置文件和若干 sink connector 配置文件。由于本例分别启动一个 source connector 读取 foo.txt 和一个 sink connector 写入 bar.txt，故 source 和 sink 配置文件都只有一个，所以总共有如下 3 个配置文件。

- connect-standalone.properties: connect standalone 模式下的配置文件。
- connect-file-source.properties: file source connector 配置文件。
- connect-file-sink.properties: file sink connector 配置文件。

我们首先来编辑 connect-standalone.properties 文件。实际上，Kafka 已经在 config 目录下为我们提供了一个该文件的模板。我们直接使用该模板并修改对应的字段即可，如下：

```
bootstrap.servers=localhost:9092
key.converter=org.apache.kafka.connect.json.JsonConverter
value.converter=org.apache.kafka.connect.json.JsonConverter
key.converter.schemas.enable=true
```



```
value.converter.schemas.enable=true  
offset.storage.file.filename=/tmp/connect.offsets
```

上面各个字段的含义分别如下。

- **bootstrap.servers**: 指定 Connect 要连接的 Kafka 集群主机名和端口号。本例使用 localhost:9092。
- **key/value.converter**: 设置 Kafka 消息 key/value 的格式转化类, 本例使用 JsonConverter, 即把每条 Kafka 消息转化成一个 JSON 格式。
- **key/value.converter.schemas.enable**: 设置是否需要把数据看成纯 JSON 字符串或者 JSON 格式的对象。本例设置为 true, 即把数据转换成 JSON 对象。
- **offset.storage.file.filename**: connector 会定期地将状态写入底层存储中。该参数设定了状态要被写入的底层存储文件的路径。本例使用/tmp/connect.offsets 保存 connector 的状态。

下面编辑 connect-file-source.properties, 它在 Kafka 的 config 目录下也有一份模板, 本例直接在该模板的基础上进行修改:

```
name=test-file-source  
connector.class=FileStreamSource  
tasks.max=1  
file=foo.txt  
topic=connect-file-test
```

上面各个字段的含义分别如下。

- **name**: 设置该 file source connector 的名称。
- **connector.class**: 设置 source connector 类的全限定名。有时候设置为类名也是可以的, Kafka Connect 可以在 classpath 中自动搜寻该类并加载。
- **tasks.max**: 每个 connector 下会创建若干个任务 (task) 执行 connector 逻辑以期望增加并行度, 但对于从单个文件读/写数据这样的操作, 任意时刻只能有一个 task 访问文件, 故这里设置最大任务数为 1。
- **file**: 输入文件全路径名。本例为 foo.txt, 即表示该文件位于 Kafka 目录下。实际使用时最好使用绝对路径。
- **topic**: 设置 source connector 把数据导入到 Kafka 的哪个 topic, 若该 topic 之前不存在, 则 source connector 会自动创建。最好提前手工创建出该 topic。本例使用 connect-file-test。

最后, 我们编辑 connect-file-sink.properties。同理, 直接修改位于 config 目录下的 connect-file-sink.properties 模板文件:

```
name= test-file-sink
connector.class=FileStreamSink
tasks.max=1
file=bar.txt
topics=connect-file-test
```

上面各个字段的含义分别如下。

- **name:** 设置该 sink connector 名称。
- **connector.class:** 设置 sink connector 类的全限定名。有时候设置为类名也是可以的，Kafka Connect 可以在 classpath 中自动搜寻该类并加载。
- **tasks.max:** 依然设置为 1，原理与 source connector 中配置设置相同。
- **file:** 输出文件全路径名。本例为 bar.txt，即表示该文件位于 Kafka 目录下。实际使用时最好使用绝对路径。
- **topic:** 设置 sink connector 导出 Kafka 中的哪个 topic 的数据。

编辑好所有的配置文件之后，首先启动 ZooKeeper 和 Kafka 环境（本例中我们以 0.11.0.0 版本为例进行说明。）：

```
$ cd /opt/kafka_2.12-0.11.0.0
$ bin/zookeeper-server-start.sh config/zookeeper.properties &
$ bin/kafka-server-start.sh -daemon config/server.properties
```

当成功启动 Kafka 后开始执行下列命令启动 standalone 模式的 Kafka Connect：

```
$ bin/connect-standalone.sh config/connect-standalone.properties \
config/connect-file-source.properties config/connect-file-sink.properties
```

启动之后，应该可以看到控制台不断地打印“Couldn't find file foo.txt for FileStreamSourceTask, sleeping to wait for it to be created”之类的日志。这是正常的，因为我们尚未创建输入文件 foo.txt。下面我们在 Kafka 的目录下创建 foo.txt 并写入一些文本行：

```
$ cd /opt/kafka_2.12-0.11.0.0
$ echo 'hello' >> ./foo.txt
$ echo 'kafka connect test example' >> ./foo.txt
$ echo 'this is a file connector test.' >> ./foo.txt
```

如果一切正常，可以看到在 Kafka 目录下生成了一个名为 bar.txt 的文件，我们查看该文件的内容：

```
$ cat ./bar.txt
hello
kafka connect test example
this is a file connector test.
```

可见，foo.txt 文件的内容已经成功地被 file connector 通过 Kafka 搬运到 bar.txt 文件中了。为了验证数据的确是通過 Kafka topic 进行转移的，我们读取一下 topic（connect-file-test）的数据，如：

```
$ bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --
topic connect-file-test --from-beginning
{"schema":{"type":"string","optional":false},"payload":"hello"}
{"schema":{"type":"string","optional":false},"payload":"kafka connect test
example"}
{"schema":{"type":"string","optional":false},"payload":"this is a file
connector test."}
```

看到了吧，这里的消息实际上都是 JSON 格式的对象。这就是上面参数 key/value.converter.schemas.enable=true 的缘故。

上面的例子只涉及 ETL 中的 E 和 L，即数据抽取（extract）与加载（load）。作为一个 ETL 框架，Kafka Connect 也支持相当程度的数据转换操作。下面演示在将文件数据导出到目标文件之前为每条消息增加一个 IP 字段。如果要插入 IP 静态字段，我们必须修改 source connector 的配置文件，增加以下这些行：

```
transforms=WrapMap,InsertHost
transforms.WrapMap.type=org.apache.kafka.connect.transforms.HoistField$V
alue
transforms.WrapMap.field=line
transforms.InsertHost.type=org.apache.kafka.connect.transforms.InsertFie
ld$Value
transforms.InsertHost.static.field=ip
transforms.InsertHost.static.value=com.connector.machine1
```

之后重启 Kafka Connect，然后写入 foo.txt 文件：

```
$ echo "this is a transformation test" >> ./foo.txt
```

查看 bar.txt 可以发现这条新增的数据：

```
$ tail -n 1 ./bar.txt
Struct{line=this is a transformation test,ip=com.connector.machine1}
```

显然，新增的数据被封装成一个结构体（Struct），并增加了 ip 字段。这就是上面 WrapMap 和 InsertHost 的作用。Kafka Connect 还提供了其他的转换操作，完整用法参见 https://kafka.apache.org/documentation/#connect_transforms。

10.2.3 distributed Connect

和 standalone 模式不同，distributed Connect 天然地结合了 Kafka 提供的负载均衡和故障转

移功能，能够自动地在多节点机器上平衡负载。用户可以增减机器来实现整体系统的高伸缩性。用户需要执行下列命令来启动 distributed 模式的 Connect，假设我们依然使用 Kafka config 目录下的配置文件模板：

```
$ bin/connect-distributed.sh config/connect-distributed.properties
```

读者可能会发现和 standalone 模式不同的是，在 distributed 模式中我们不需要指定 source 和 sink 的配置文件。distributed 模式中的 connector 只能通过 REST API 来创建和管理。

我们依然以 FileStreamSourceConnector/FileStreamSinkConnector 为例来演示如何在 distributed 模式下运行 Kafka Connect。上述命令启动成功后，我们可以执行以下命令来获取当前所有 connector：

```
$ curl http://localhost:8083/connectors
[]
```

值得注意的是，distributed 模式下默认的 REST 端口是 8083，用户可以修改 connect-distributed.properties 文件中的 rest.port 属性来变更这一端口。如上可见，当前集群中没有创建任何的 connector。下面分别创建 file source connector 和 file sink connector，命令如下：

```
$ curl -i -X POST -H "Content-type:application/json" -H "Accept:application/json" -d '{"name":"test-file-source","config":{"connector.class":"FileStreamSource","tasks.max":"1","topic":"connect-file-test","file":"foo.txt"}}' http://localhost:8083/connectors
{"name":"test-file-source","config":{"connector.class":"FileStreamSource","tasks.max":"1","topic":"connect-file-test","file":"foo.txt","name":"test-file-source"},"tasks":[]}
```

```
$ curl -i -X POST -H "Content-type:application/json" -H "Accept:application/json" -d '{"name":"test-file-sink","config":{"connector.class":"FileStreamSink","tasks.max":"1","topics":"connect-file-test","file":"bar.txt"}}' http://localhost:8083/connectors
{"name":"test-file-sink","config":{"connector.class":"FileStreamSink","tasks.max":"1","topics":"connect-file-test","file":"bar.txt","name":"test-file-sink"},"tasks":[]}
```

本例中使用 curl 工具给 Kafka Connect 发送 POST 请求。当前 REST API 只支持 application/json 作为请求 (request) 和响应 (response) 的内容类型 (content type)，因此在发送 POST 请求时必须显式指定 HTTP 的 Accept 头部为 application/json，以设置 response 的 content type。另外，我们还需要设置 Content-Type 头部信息为 application/json，以指定 request 的 content type。在上面命令中我们只是把 standalone 模式下配置文件中的所有属性封装成 JSON 字符串传递给 curl 工具。注意，connector 的 name 字段和其他字段是分开的，即其他字

段首先要被封装到 `config` 下，然后和 `name` 一起做成 JSON 串。

下面再次获取当前所有 `connector` 以检查之前的两个 `connector` 是否已被创建出来：

```
$ curl http://localhost:8083/connectors
["test-file-source","test-file-sink"]
```

这次我们可以看到两个 `connector` 都已经被创建出来了。REST API 还提供了 `/connectors/{name}/config`，允许用户查询某个 `connector` 的具体配置信息，我们使用这个 endpoint 来查询 file sink `connector` 的信息：

```
$ curl http://localhost:8083/connectors/test-file-sink/config
{"connector.class":"FileStreamSink","file":"bar.txt","tasks.max":"1","name":"test-file-sink","topic":"connect-file-test"}
```

同时使用 `GET /connectors/{name}/status` 查询两个 `connector` 的运行状态：

```
$ curl http://localhost:8083/connectors/test-file-source/status
{"name":"test-file-source","connector":{"state":"RUNNING","worker_id":"10.2.0.28:8083"},"tasks":[{"state":"RUNNING","id":0,"worker_id":"10.2.0.28:8083"}]}
$ curl http://localhost:8083/connectors/test-file-sink/status
{"name":"test-file-sink","connector":{"state":"RUNNING","worker_id":"10.2.0.28:8083"},"tasks":[{"state":"RUNNING","id":0,"worker_id":"10.2.0.28:8083"}]}
```

目前两个 `connector` 都已正常工作。下面开始写入输入文件 `foo.txt`：

```
$ echo "hello" >> ./foo.txt
$ echo "kafka connect example" >> ./foo.txt
$ echo "distributed mode test" >> ./foo.txt
```

然后查询 `bar.txt` 验证数据是否已经被写入：

```
$ cat ./bar.txt
hello
kafka connect example
distributed mode test
```

做完这些之后，我们删除这两个 `connector` 把系统还原回初始状态。若要删除 `connector`，可以使用 REST API——`DELETE /connectors/{name}`，如下：

```
$ curl -i -X DELETE http://localhost:8083/connectors/test-file-source
HTTP/1.1 204 No Content
Date: Wed, 01 Nov 2017 01:14:59 GMT
Server: Jetty(9.2.15.v20160210)

$ curl -i -X DELETE http://localhost:8083/connectors/test-file-sink
```

```

HTTP/1.1 204 No Content
Date: Wed, 01 Nov 2017 01:15:08 GMT
Server: Jetty(9.2.15.v20160210)

curl http://localhost:8083/connectors
[]

```

Kafka Connect 当前为 distributed 模式，还提供了其他的 REST API，有兴趣的读者可以参考 https://kafka.apache.org/documentation/#connect_rest 来学习每个 API 具体的使用方法，这里就不一一赘述了。

10.2.4 开发 connector

如前所述，目前 Apache Kafka 提供的 connector 只有 file source/sink 一种，如果要与其他系统进行集成，只能使用 GitHub 上其他人开源的插件。除此之外，还有一个办法就是用户自己实现 connector。Kafka Connect 提供了一整套接口和标准供用户开发 connector。本节我们将简要探讨一下如何使用这套接口定义。

在开始开发 connector 之前，我们必须了解一下 Kafka Connect 和 connector 的底层设计和一些常用概念。首先所有的 connector 必须都继承 Connector 父类，而 Connector 又有两个子类 SourceConnector 和 SinkConnector，分别建模 source connector 和 sink connector。Connector 本身并不会直接执行数据搬移的逻辑，它只是负责进行一些配置处理（比如要拷贝哪些数据，从哪里拷贝数据等），同时 Connector 会在底层创建多个任务来执行真正的工作，这些任务以 Task 接口进行建模。同理，Task 也有两个实现类 SourceTask 和 SinkTask。所以通常来说，如果要实现自己的 connector，我们通常要实现这 4 个类的逻辑，即 SourceConnector、SinkConnector、SourceTask 和 SinkTask。

下面创建一个简单的 source connector。首先创建 SourceConnector：

```

public class FileStreamSourceConnector extends SourceConnector {
    private String filename;
    private String topic;
}

```

任何继承 Connector 的类都至少要实现 start、stop、taskClass 和 taskConfigs 方法，本例中的实现如下：

```

@Override
public Class<? extends Task> taskClass() {
    return FileStreamSourceTask.class;
}

@Override

```

```
public List<Map<String, String>> taskConfigs(int maxTasks) {
    ArrayList<Map<String, String>> configs = new ArrayList<>();
    // 添加一个输入流
    Map<String, String> config = new HashMap<>();
    if (filename != null)
        config.put(FILE_CONFIG, filename);
    config.put(TOPIC_CONFIG, topic);
    configs.add(config);
    return configs;
}
```

其中 `taskClass` 方法返回该 `connector` 底层创建的 `Task` 实现类，即后续我们要创建的 `FileStreamSourceTask` 类，而 `taskConfigs` 将该 `Task` 的所有属性加载到一个列表中。这里比较关键的属性包括读取文件的路径和要写入的 Kafka topic。`start` 和 `stop` 方法实现如下：

```
@Override
public void start(Map<String, String> props) {
    filename = props.get(FILE_CONFIG);
    topic = props.get(TOPIC_CONFIG);
}

@Override
public void stop() {
    // 在该方法中关闭 connector 用到的外部资源
}
```

`start` 方法中仅是从配置文件中读取关键的属性值，即 `filename` 和 `topic`。做完这些之后我们的 `FileStreamSourceConnector` 类就算开发完成了，下面开始创建 `Task` 类 `FileStreamSourceTask`。

如前所述，`SourceTask` 类必须要继承 `SourceTask`，如下：

```
public class FileStreamSourceTask extends SourceTask {
    String filename;
    InputStream stream;
    String topic;

    @Override
    public void start(Map<String, String> props) {
        filename = props.get(FileStreamSourceConnector.FILE_CONFIG);
        stream = openOrThrowError(filename);
        topic = props.get(FileStreamSourceConnector.TOPIC_CONFIG);
    }

    @Override
    public synchronized void stop() {
        stream.close();
    }
}
```

SourceTask 类至少要实现 3 个方法 start、stop 和 poll。下面首先来看看 start 和 stop。本例中 start 方法仅仅从 FileStreamSourceConnector 类中读取 file 和 topic 信息，然后创建一个输入流 InputStream 准备开始读取文件数据。注意这仅仅是为读取数据做准备而已，不会执行真正的操作。stop 方法的逻辑要简单得多，只需要关闭输入流即可。细心的读者可能会发现 stop 方法是同步方法，这是因为每个 SourceTask 实例都运行在专属的线程中。如果该线程一直阻塞，Kafka Connect 就要支持从另一个线程中直接中断该阻塞线程，故必须要保护线程共享的 InputStream。

SourceTask 类最关键的就是 poll 方法，它才是真正逻辑的实现者。本例中 poll 方法实现如下：

```
@Override
public List<SourceRecord> poll() throws InterruptedException {
    try {
        ArrayList<SourceRecord> records = new ArrayList<>();
        while (streamValid(stream) && records.isEmpty()) {
            LineAndOffset line = readToNextLine(stream);
            if (line != null) {
                Map<String, Object> sourcePartition = Collections.singletonMap(
                    "filename", filename);
                Map<String, Object> sourceOffset = Collections.singletonMap(
                    "position", streamOffset);
                records.add(new SourceRecord(sourcePartition, sourceOffset,
                    topic, Schema.STRING_SCHEMA, line));
            } else {
                Thread.sleep(1);
            }
        }
        return records;
    } catch (IOException e) {
        // 错误处理逻辑.....
    }
    return null;
}
```

虽然没有给出完整代码，但读者基本上可以看出该方法就是不停地读取文件中的每一行数据并封装成 SourceRecord 列表返回。当然本例的程序依然有很多不健壮的地方，比如错误处理以及没有批量处理数据等，但读者依然可以从上述代码中了解到 connector 开发的整体思路和方法。

完整的 file connector 代码可以参考 Kafka 源代码 <https://github.com/apache/kafka/tree/trunk/connect/file/src/main/java/org/apache/kafka/connect/file>。

10.3 Kafka Streams

Apache Kafka 一直以消息队列（message queue）或消息总线（message bus）自诩。在 0.10.0.0 版本之前官方对 Kafka 的定位一直是提交日志服务（commit log service）。事实上，Kafka 在消息队列方面的表现确实也呈突飞猛进的态势，这也是笔者撰写本书以及读者阅读本书的主要原因。Kafka 实现了高吞吐、高伸缩以及低延时的消息传输（message delivering）特性。不过，数据被传输的目的终归是为了对其进行处理从而实现其业务价值，但令人遗憾的是，0.10.0.0 之前版本的 Kafka 并未提供任何形式的数据处理语义，若要实现数据处理必须为 Kafka 接入上下游处理系统来实现。当今开源的数据处理框架有很多，比如 Apache Samza、Apache Storm、Apache Spark 和 Apache Flink 等。目前，Spark 凭借着定义良好的 API、活跃的社区等因素被认为是该领域的执牛耳者，不过 Flink 在设计上更加贴合流处理（streaming process）语义且提供了更便捷的 API，最近也大有赶超之势。另外值得一提的是，Google 捐献给社区的 Apache Beam。在意识到开源也可以掌控世界之后，错过了 Hadoop 这拨生态建设的 Google 终于决定开源自己的统一数据处理编程模型（unified programming model）。至于它能否“一统江湖”，我们拭目以待吧。另外，在流处理领域，待处理的元素有多种不同的称法，如消息、事件、记录等。由于它们均表示类似的含义，故本章将无差别地使用这 3 个名词。

既然数据都在 Kafka 上进行流转，Kafka 为何不能实现一套数据流处理语义框架？于是，在 0.10.0.0 版本 Kafka 正式推出了 Kafka Streams 组件。细心的读者会发现从 0.10.0.0 的官方文档开始，社区正式将 Kafka 定位为分布式流处理平台（distributed streaming platform）。它不只提供可靠的消息队列服务，同时还实现了一套完整的流处理语义库。用户可以使用该 API 库实现复杂的数据实时处理，不再依赖其他外部的处理框架，从而极大地降低了开发和维护成本。

在开始讨论 Kafka Streams 之前，我们首先来探讨一下流处理。

10.3.1 流处理

流处理（也可以称为流式处理，即 streaming process 或 stream processing）在当今的大数据领域越来越重要了，原因有如下 3 个。

- 企业或组织日益地期望获取及时数据，而使用流处理能够显著地降低延时。
- 企业或组织创建海量数据越来越容易，即使是创业小公司每天也能生产大量的业务数据，因此必须有一种处理语义或系统能够应对这种接近于无穷多的数据集（data set）。
- 实时处理数据有助于实现负载均衡，提升系统资源的整体利用率。

所谓的流处理是与批处理（batching process）是相对应的，它们是目前数据处理领域内的两大处理模式。对于批处理各位读者并不陌生，像 Apache Hadoop 或 Apache Pig 等框架通常就

属于该范畴，它指的是在无人工干预的情况下依次执行一系列任务或操作。严格来说，批处理是一种数据处理模式，它会在一批输入数据上执行操作任务。而流处理则不是这种处理模式。下面我们从以下两个方面来研究流处理。

1. 什么是流处理

从本质上说，流处理是一种处理模式或一类数据处理引擎，旨在处理无限多的数据集合。从广义上说，它既包括完全的流处理，也包括模拟流处理的微批次实现（micro-batch）。Spark Streaming 的设计理念就是这种 micro-batch 化。值得注意的是，流处理经常与以下的名词混淆。

- **无限数据集合（unbounded data set）**：该名词表示数据量一直增加，永无止境，因此有时被称为“流式数据”。注意它和流处理的区别：流处理或批处理表达的是处理数据的执行引擎，而非数据本身。另外，流处理与批处理的区别之一在于，它们处理的数据是否 unbounded。
- **无限数据处理（unbounded data processing）**：比起第一个名词，这个名词似乎与流处理的含义更加贴近一些，不过它依然是不准确的，因为批处理也能处理 unbounded data——只要能提供合理的切片机制，然后重复地执行批处理逻辑就好了，故流处理与 unbounded data processing 也是不同的。
- **低延时、结果近似的处理引擎**：实际上这是长期以来人们对流处理的一贯印象，仿佛流处理无法提供准确的计算结果，所以才出现了类似于 Lambda 架构（https://en.wikipedia.org/wiki/Lambda_architecture）这样的折中方案。事实上，随着越来越多的关于流处理实现精确一次处理语义论文的面市以及相应系统的出现，流处理无法得出准确结果的论断也已经过时了。诸如 Spark 和 Flink 这样的框架在满足特定条件的情况下都能实现准确的计算结果。

总之，流处理是处理无限数据集合的执行引擎，仅此而已。

2. 流处理能做什么

如前所述，人们长期以来一直以为流处理只能得出近似结果，因而在部署时引入了批处理与之协同工作，由流处理负责实时处理，而让批处理负责准确结果的计算，这就是所谓的 Lambda 架构——该架构最早由 Apache Storm 的原作者 Nathan Marz 提出，经实践证明在当时是非常成功的设计。从准确性这点来看，流处理系统的确不尽如人意，而批处理系统通常又很笨拙，因此两者的结合简直可谓是“鱼与熊掌的兼得”。令人遗憾的是，维护两套系统的成本是很高的——因为它们是两套独立的数据管道，同时它们产生的结果又需要执行合并操作。

后来，Kafka 原作者之一的 Jay Kreps 提出了可重演系统（replayable system）来解决重复

性的问题，并坚持使用一套管道（pipeline）来进行数据处理。如果把这个观点更进一步，我们可以认为定义良好的流系统提供的是批处理引擎所具功能的超集（superset）——事实上，去年火爆开源社区的 Apache Flink 就是这样的思想，Flink 中的 batch 引擎是作为 streaming 引擎的一个特例而实现的，这点和 Spark Streaming 正好相反，Spark Streaming 的 streaming 其实是借助于 micro-batch 的思想来实现的。如果不是目前执行效率上有一些差异，现在的 batch 系统根本没有存在的必要。在这点上要感谢 Flink 开发者为我们构建了一个“随时随地 streaming 化”的系统，即使是 batch 模式，Flink 底层也是使用 streaming 实现的。

若要彻底击败“老对手”批处理，流处理还需要提供两个方面的能力。

- 正确性（correctness）：流处理必须要能够得到正确的计算结果。从本质上说实现正确性可以归结为提供一致性存储。流处理需要定期地将应用状态持久化以维护系统崩溃时的一致性。Spark Streaming 在其实现论文中首次提出维护一致性的方案。时至今日越来越多的流处理框架都已能做到这点。
- 时间推导工具（time reasoning tool）：主要负责处理无限数据集合。当前海量数据的主要特点之一就是随机变化的数据倾斜（data skewing），即消息处理时间与消息创建时间的差值是随机分布的，而目前批处理系统以及很多早期的流处理系统都无法处理这种情况，因此一个“合格”的流处理框架必须能够正确地推导时间。

谈到时间，在数据处理系统中通常存在着两类时间概念。

- event time：消息或事件真实发生的时间，也被称为创建时间。
- processing time：消息或事件被观测到的时间，也被称为处理时间。

有的系统可能会做一些扩展，比如 Apache Flink 就提供了第三类时间概念 ingestion time，表示消息进入处理管道中的时间。ingestion time 和 processing time 的主要区别在于前者使用稳定的时间戳而不会受到后者因各系统墙钟导致的的不一致。Kafka Streams 也做了这样的区分。不过总的思路依然是适用的，即把时间大致分为创建时间和处理时间。

10.3.2 Kafka Streams 核心概念

Kafka Streams 是一个客户端 API 库（client library），依托于 Kafka core 提供的功能来处理和分析保存于 Kafka topic 之中的消息数据。它完全地根植于前面章节中提出的流处理理念，在 Kafka 事务和幂等性 producer 的帮助下，既实现了精确一次的处理语义，也提供了用于推导 event time 和 processing time 的工具。Kafka Streams 对于窗口化（windowing）提供了完整的支持，同时还实现了应用状态的持久化存储以及实时查询。

作为一个 client library，使用 Kafka Streams 的开销比起传统的全功能性数据处理框架来说

要轻量得多。用户能够在单机上编写 Streams 程序以执行小规模验证性测试，也可以简单地将该程序扩展到多机上轻松实现业务的横向扩展，底层的负载均衡和故障转移均有 Kafka Streams 自动承担，无须用户操心。

在 Kafka Streams 中，最核心的概念是“流”，即 stream。它表示了一个无限且持续更新的数据集。一个 stream 本质上就是一个有序、可重演且高容错的不可变消息集合，其中的每条消息都是键/值对的形式。Kafka Streams 应用就是使用 Kafka Streams API 库的应用程序，它定义并实现了数据被处理的方法，也称计算逻辑。实现的途径就是定义若干个数据加工拓扑（processor topology）。和几乎所有数据处理框架类似的是，Kafka Streams 中每个 topology 本质上就是一个有向无环图（DAG），该图上定义了处理节点（node）和连接节点的边（edge），如图 10.3 所示。

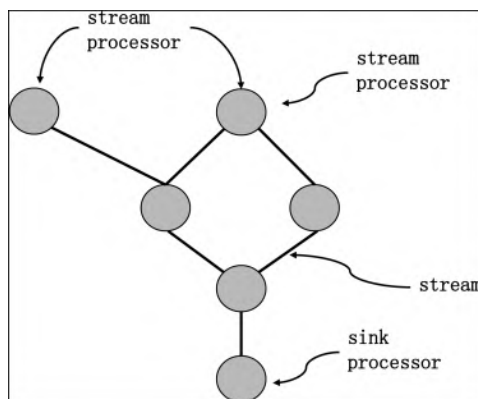


图 10.3 Kafka Streams 处理加工拓扑

如图 10.3 所示，一个 topology 通常由两类特殊的 processor 组成：source processor 和 sink processor。这里的 source processor 和 sink processor 与之前 Kafka Connect 中的 source connector 与 sink connector 含义是相同的，即 source processor 负责从 Kafka topic 中读取输入消息并将其传递到下游的处理节点上，source processor 无上游 processor，它是拓扑的起始节点；而 sink processor 则负责把处理结果发送到指定的 Kafka topic 中。sink processor 无下游 processor，它是拓扑的结束节点。一个拓扑中非 source 和 sink 的 processor 就是普通的 stream processor，它们执行消息的处理逻辑，如 map、filter、join 或 aggregation 等。

Kafka 提供了两种方式来定义这些 stream processor。

- Kafka Streams DSL¹：提供了最常见的数据转换操作。
- Processor API：低阶 API，用于帮助用户自定义 processor 节点以及节点之间的互连，同时还能直接管理底层的状态存储。

Kafka Streams 中一个拓扑的本质就是流处理逻辑的抽象表示。只有在运行时该拓扑才会被实例化并执行。

¹ DSL 是一类旨在应用于特定应用领域内的计算机语言。它与 GPL 是相对的。常见的 DSL 如 HTMP、SQL 等。

虽然 10.3.1 节中提到了流处理中的不同时间，但这里我们会详细展开一下 Kafka Streams 中各种时间的定义。反复提到时间概念是有意义的，因为流处理领域如何定义和推导消息/记录的时间至关重要，因为它处理的是无限数据集，故通常的做法是采用窗口化来对数据沿时间轴进行切片，切成一个一个的时间窗口。正确地把消息/记录划分到正确的窗口是流处理框架实现正确性的基础。在 Kafka Streams 中定义了如下 3 种时间。

- **event time**: 事件或消息真实发生的时间，即该事件在消息源中的创建时间。举一个例子，假设我们要统计某电商网站上付款按钮的点击量。显然，所有用户的点击行为以消息的形式汇聚到一起，构成了一个点击流（click stream），而每个用户点击发生的时间即为该消息的 event time。对于 Kafka 来说，基于 event time 计算时间窗口是很容易的，因为每条 Kafka 消息内都封装了时间戳字段，默认保存的就是 event time。
- **processing time**: 事件被流处理应用执行具体操作处理的时间。
- **ingestion time**: 指消息或记录被保存到 Kafka topic 中的时间。消息被产生出来到真正被写入 Kafka topic 中间通常有一段延时，因此 ingestion time 一般会晚于 event time。有意思的是，消息被保存到 topic 后可能未被处理，故此时该消息是没有 processing time 的，而这恰好是 ingestion time 和 processing time 的区别。

这 3 种时间各有优劣。对于 processing time 来说，当用户使用 processing time 作为消息时间戳时，Kafka Streams 使用的是 processor 节点所在机器的墙钟（wall clock）来记录消息时间。processing time 使用起来最简单，因为不需要用户操心时间戳，而且也无须任何流与机器之间的协调开销，故其性能和延时表现都是最好的。但是在分布式的异步环境中，processing time 可能无法实现正确性，因为它易受到消息进入流处理管道的速度以及消息在流处理管道中流转的速度的影响，比如消息的乱序等。

而对 event time 来说，它在进入管道之前已被嵌入消息体内。由于是真实发生的时间，所以无论消息何时到达或是否出现乱序，都不影响它被划分到正确的时间窗口，因此使用 event time 能够保证正确性。event time 的不足在于它引入了延时来应对延迟消息（即很晚才到达的消息）以及乱序消息。

ingestion time 则介于 event time 和 processing time 之间。比起 processing time，它的开销更大，但计算结果更正确；而相比 event time 而言，它又无法处理乱序消息或延迟消息。

Kafka Streams 默认提供了 TimestampExtractor 接口以及一组实现类来帮助用户指定到底使用哪类时间。用户可以自定义 TimestampExtractor 实现类，在 extract 方法中实现自己的时间定义逻辑。比如如果实现 extract 方法会 return System.currentTimeMillis();，那么即表示使用 processing time（因为直接使用了机器的墙钟）；而如果直接使用 extract 方法中消息体内的时间戳，则定义的是 event time 或 ingestion time。具体是哪一种时间取决于 broker 端参数

`log.message.timestamp.type` 或 `topic` 级别参数 `message.timestamp.type` 的值, `CreateTime` 代表 event time, `LogAppendTime` 代表 ingestion time。

介绍了 Kafka Streams 的时间定义, 下面来看看流处理领域内的状态 (state) 在 Kafka Streams 中的实现。

首先, 流处理可以是有状态的 (stateful)。一个 stateful 的流处理要求操作算子 (operator) 必须要记录之前输入的处理信息并依据这些信息处理后续的输入。与之相反, 在一个无状态 (stateless) 的流处理中它包含的 operator 只需要考虑当前输入数据, 不依赖于之前的输入数据。举一个简单的例子, 假设我们的消息格式是 `{"record_id": int, "record_payload": {"key": int, "value": string}}`, 若现在要把所有事件 payload 中的 value 单独提取出来发送到下游 operator, 那么这就是一个典型的 stateless 流处理。因为整个处理过程中我们仅仅需要提取消息体的部分数据即可, 没有任何依赖之前数据进行计算的需求。但若修改一下流处理逻辑, 统计所有 payload 中 key 比前一条记录大的记录。此时这个流处理就是 stateful 的, 因为它需要 operator 记录前一条记录的 payload key。

Kafka Streams 默认提供了很多这种 stateful 的 operator, 如 `aggregating`、`join` 等。我们会在 10.3.4 节中给出这些 operator 的用法示例。

既然 Kafka Streams 是支持 stateful 的, 那么状态到底被保存在哪里呢? 答案就是状态存储 (state stores)。Kafka Streams 使用 state stores 来保存状态数据并且允许用户查询这些状态。这对于实现 stateful 的流处理是必不可少的组件。Kafka Streams 中的每个子任务都会内置若干个 state stores 来保存和查询状态。这些存储要么是持久化的 KV 存储, 要么是在内存中的哈希映射表, 抑或是外部的数据存储。Kafka Streams 对本地化存储提供高容错性以及自动的错误恢复。

Kafka Streams 开放了一些接口允许用户通过方法、线程、processor 或外部应用程序直接查询状态——这个特性被称为交互式查询 (interactive query)。交互式查询只允许进行读操作, 而不允许直接修改底层存储数据。

如前所述, 流处理是能够实现正确性的, 而支持精确一次处理语义 (exactly-once semantic) 是实现正确性的前提。很多实际场景都不能容忍数据丢失和数据重复, 故 Kafka Streams 必然也要实现 exactly once。在 0.11.0.0 之前的版本中, Kafka 只提供“至少一次”的发送语义, 因此任何对接 Kafka 的数据处理引擎都必须自行实现业务去重逻辑, 否则将无法达成端到端的 exactly once 语义。事实上, 虽然很多处理框架都宣称它们支持 exactly once, 但是否能够实现端到端的 exactly once 还有待商榷。不过这一弊端在 0.11.0.0 版本中被彻底解决了, 自 0.11.0.0 版本开始, Kafka 正式支持幂等性 producer 和事务 (transaction)。得益于这些基础组件功能的

完善，Kafka Streams 也正式支持端到端的 exactly once 处理语义。它能保证从 Kafka 源 topic 中读取的每条消息都只被处理一次，然后发送到目标 topic 和（或）有状态 operator 中。比起其他流处理引擎，Kafka Streams 能够与底层的 Kafka 集群和存储系统天然集成，无论是 topic 消费进度位移的提交、状态的更新还是结果的写入，都能保证与底层的操作是原子性的。但是在其他引擎看来，Kafka 只是一个外部系统，因此保证这种原子性将是很困难的。Kafka Streams 中启用 exactly once 非常简单，只需要简单地将 streams 端参数 processing.guarantee 设置为 exactly_once。

10.3.3 Kafka Streams 与其他框架的异同

如其官网宣称的那样，Kafka Streams 是一个客户端库，适合构建那些输入/输出数据均保存于 Kafka 集群内的应用程序或微服务。在笔者看来这是它和其他处理引擎最大的两个不同点。

首先，官网强调它是一个库（library），而非一个全功能齐备（full-fledged）的处理框架。Kafka Streams 用户或操作人员不需要考虑底层的 processor 节点如何部署和管理，也不用提供高可用的解决方案。这些烦琐的事情全部都交由底层的 Kafka 集群来维护。但反观 Apache Storm、Spark 或 Flink 这样的框架，当用户使用这些框架开始处理数据时，必须首先确定框架的部署方式，比如是独立的集群部署还是采用集群管理框架（如 Memos、YARN 甚至是 Kubernetes）来托管部署等。此外，这些成熟框架通常都要显式定义统一的管理者角色来维护集群运行，虽然名称不同（如 Flink 中是 job manager，Storm 中是 nimbus 等），但职责是类似的。由于这类角色只允许有一个 active 的任务在工作，因此为其提供必要的高可用性也属框架的分内之事。而在 Kafka Streams 中则无这些功能。当然，这绝不是说 Kafka Streams 应用就不能与这些框架结合使用。用户当然可以直接在 Marathon UI（Mesos 上的一款用于容器编排的框架）上启动 Kafka Streams 程序。这里笔者只是想强调 Kafka Streams 将很多成熟流处理引擎该有的功能或特性转交由底层 Kafka 集群帮忙实现，故从本质上说它只是一个客户端库，而非一套功能齐备的框架，这是第一个不同。值得注意的是，本身是一个库不表示 Kafka Streams 就“矮他人一等”，这种设计上的差异只是不同使用场景的体现而已。

其次，Kafka Streams 擅长的是从 Kafka topic 中获取输入数据，然后对这些数据进行转换，最后发送到目标 topic 中。前面说过 Kafka Connect 提供的 source connector 种类非常有限。笔者写稿时最新版本是 1.0.0，依然只提供了物理文件到 Kafka 的转入/转出，可以说整个数据集成服务的生态依然在不断构建中，所以实际上 Kafka Streams 能够处理的输入数据源其实相当有限。目前它最“拿手”的还是从 Kafka topic 中获取数据进行处理。同样地，鉴于 sink connector 生态也不是十分健全，Kafka Streams 的处理结果比较适合发送到 Kafka 的 topic 中保存。

图 10.4 非常鲜明地给出了 Kafka Streams 与其他流处理框架的异同。

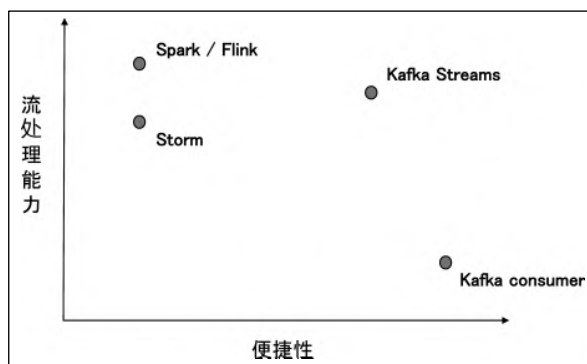


图 10.4 Kafka Streams 与其他流处理框架的异同

图 10.4 从便捷性和流处理能力两个维度评估了主流的处理引擎。Kafka Streams 的便捷性前面已经讲过了，就实现难易程度它确实要轻量得多，比起 Spark、Flink 或 Storm 这种全功能框架，部署和维护成本都要低得多。令人欣喜的是，在流处理能力上 Kafka Streams 也是很有竞争力的。这里的流处理能力主要从处理性能和转换丰富程度两个层面进行考量。从处理性能的层面来看，由于天然集成底层的 Kafka 消息队列，因而 Kafka Streams 具有很好的低延时表现，同时还兼具较高的吞吐量；而在操作算子的丰富程度上，Kafka Streams 也在不断地完善和补充常见的 operator。无论是其高阶流处理 DSL，还是低阶的 processor API，都已经具备了和 Flink、Spark Streaming 一较高下的能力。

总结一下，Kafka Streams 是一个轻量级的客户端流处理 API 库。它非常适用于输入/输出数据均来自 Kafka 集群的流处理场景。如果配合用户自定义的 connector，它也支持连接上下游外部系统的流处理应用或微服务的实现。

10.3.4 Word Count 实例

Word Count 对大数据框架类似于 Hello World 对编程语言。任何一款大数据框架，特别是数据处理框架，统计单词数都是展示其框架易用性、性能的绝佳例子。本节我们正式开始介绍 Kafka Streams 的应用开发以及如何使用它来实现 Word Count。

在运行任何 Kafka Streams 应用程序之前，我们至少要启动一个正常工作的 Kafka 集群。本例中我们启动一个单节点的 broker 集群，启动命令如下（所有配置均使用默认配置）：

```
# 启动 Kafka 自带的 ZooKeeper
$ bin/zookeeper-server-start.sh -daemon config/zookeeper.properties
# 启动 Kafka broker
```



```
$ bin/kafka-server-start.sh config/server.properties
```

成功启动 Kafka 之后，我们创建了两个 topic，一个是输入 topic，另一个是输出 topic。这两个 topic 的名字固定是 streams-plaintext-input 和 streams-plaintext-output，因为后续 Demo 程序的代码中硬编码了这两个名字。

```
# 创建输入 topic: streams-plaintext-input
$ bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic streams-plaintext-input
# 创建输出 topic: streams-plaintext-output
$ bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic streams-plaintext-output
```

topic 创建之后我们使用 kafka-topics.sh --describe 命令查询一下这两个 topic 的状态：

```
$ bin/kafka-topics.sh --zookeeper localhost:2181 --describe
Topic:streams-plaintext-input PartitionCount:1 ReplicationFactor:1 Configs:
  Topic: streams-plaintext-input Partition: 0 Leader: 0 Replicas:
0 Isr: 0
Topic:streams-plaintext-output PartitionCount:1 ReplicationFactor:1 Configs:
  Topic: streams-plaintext-output Partition: 0 Leader: 0 Replicas:
0 Isr: 0
```

准备好了这些之后我们就可以运行 Kafka Streams 的 Word Count 程序了。Kafka 自带了一个 Demo 程序供用户使用，本例直接使用这个 Demo 来演示：

```
$ bin/kafka-run-class.sh org.apache.kafka.streams.examples.wordcount.WordCountDemo
```

上面的 WordCountDemo 会固定地读取名为 streams-plaintext-input 的 topic，为读取的每条消息执行 Word Count 程序的转换计算逻辑，然后持续地把处理结果固定写入 streams-wordcount-output 中。当运行上述命令时，用户不会看到任何输出，这是正常的。下面，我们使用 Kafka 自带的 console producer 来生产一些输入数据供 Word Count 程序消费。新开启一个终端，运行以下命令：

```
$ bin/kafka-console-producer.sh --broker-list localhost:9092 --topic streams-plaintext-input
# 暂时不要发送消息
```

现在再开启一个终端，运行 console consumer 脚本来验证 Word Count 程序的计算结果：

```
$ bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 \
  --topic streams-wordcount-output \
  --from-beginning \
  --formatter kafka.tools.DefaultMessageFormatter \
  --property print.key=true \
```

```

--property print.value=true \
--property
key.deserializer=org.apache.kafka.common.serialization.StringDeserializer \
--property
value.deserializer=org.apache.kafka.common.serialization.LongDeserializer
[2017-11-10 10:02:01,924] WARN [Consumer clientId=consumer-1, groupId=
console-consumer-41325] Error while fetching metadata with correlation id 2 :
{streams-wordcount-output=LEADER_NOT_AVAILABLE}
(org.apache.kafka.clients.NetworkClient)

```

上面的 WARN 警告信息是正常的，这是因为当 clients 首次向 broker 发送请求获取该 topic 数据时，很可能尚未有该 topic 的元数据信息，故 broker 向 clients 返回的响应中会带上 LEADER_NOT_AVAILABLE 异常，表明 clients 应该主动更新元数据。更新元数据后该警告就不会再次出现了。下面我们在 console producer 的终端输入一些输入消息：

```

$ bin/kafka-console-producer.sh --broker-list localhost:9092 --topic
streams-plaintext-input
>hello world
>kafka streams example
>kafka topics
>streams word count

```

如果一切正常，我们在 console consumer 的终端中应该能够看到这样的输出：

```

$ bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --
topic streams-wordcount-output --from-beginning --formatter
kafka.tools.DefaultMessageFormatter --property print.key=true --property
print.value=true --property key.deserializer=org.apache.kafka.
common.serialization.StringDeserializer --property value.deserializer=
org.apache.kafka.common.serialization.LongDeserializer
[2017-11-10 10:02:01,924] WARN [Consumer clientId=consumer-1, groupId=
console-consumer-41325] Error while fetching metadata with correlation id 2 :
{streams-wordcount-output=LEADER_NOT_AVAILABLE}
(org.apache.kafka.clients.NetworkClient)
hello      1
world      1
kafka      1
streams    1
example    1
kafka      2
topics     1
streams    2
word 1
count      1

```

再次生产一些新的消息，如下：

```
>new message  
>new record
```

再次回到 console consumer 终端，发现新的单词已经被统计出来：

```
hello      1  
world      1  
kafka      1  
streams    1  
example    1  
kafka      2  
topics     1  
streams    2  
word       1  
count      1  
new        1  
message    1  
new        2  
record     1
```

可见，Word Count 的 Demo 程序正确地统计了输入源的所有单词。如果要停止 WordCountDemo 程序，只需要在对应终端上按下 Ctrl + C 组合键即可。

10.3.5 Kafka Streams 应用开发

若要开发 Kafka Streams 应用，用户需要遵循一定的流程。第一件要做的事情就是，必须定义一个拓扑（topology）来封装具体的流处理实现逻辑。如前所述，Kafka Streams 提供了两套 API 用于定义 topology，即高阶 Streams DSL 和低阶 processor API。

首先，我们来讨论高阶 Streams DSL 的使用。一个典型的高阶 Streams DSL 开发应用流程如图 10.5 所示。流程大致分为如下 4 步。

（1）构建或定义 stream 配置：Kafka Streams 应用本质上都是使用 Kafka producer/consumer 实现的，故用户必须显式设置最基础的参数。对于 Kafka Streams 应用而言，定义以下参数的值是必需的。完整的参数列表请参见 <https://kafka.apache.org/documentation/#streamsconfigs>。

- application.id: 定义了此流处理应用的 ID，在整个 Kafka 集群中该 ID 必须是唯一的。
- bootstrap.servers: 定义了 Kafka 集群的连接信息。任何客户端应用都必须设置此参数。
- state.dir: Kafka Streams 状态保存的路径。虽然不设置该值也是可以的，但默认把状态放在/tmp 下终归不是一个长久的办法，因此在生产环境中推荐用户显式设置此参数。

- `default.key.serde/default.value.serde`：分别设置消息 key/value 的序列化器（包括 `serializer` 和 `deserializer`）。

（2）创建拓扑并生成 `stream`：这一步主要是使用 `StreamsBuilder` 类来创建对应的 `stream`。

（3）实现流处理逻辑：使用 `Streams DSL` 提供的各种转换 API 实现复杂的消息处理逻辑。

（4）启动流处理应用：上一步仅仅是告知 `Kafka Streams` 消息处理的逻辑，只有显式地开启了流处理应用才会正式对消息进行处理。

我们依然以 `Word Count` 为例进行说明。下面就遵循图 10.5 的这 4 个步骤完整地实现一个 `Word Count` 程序。第一步，我们构建 `Streams` 程序的参数，如下面的代码所示：

```
Properties props = new Properties();
props.put(StreamsConfig.APPLICATION_ID_CONFIG, "streams-wordcount");
props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass().getName());
props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.String().getClass().getName());
props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
```

我们使用 `streams-wordcount` 作为此 `Streams` 应用的 ID，由于连接的是本地的 `Kafka` 测试环境，故将 `bootstrap.servers` 设置为 `localhost:9092`。key 和 value 的序列化器都使用 `StringSerde`（实际上 `serializer` 使用的是 `StringSerializer`，`deserializer` 使用的是 `StringDeserializer`）。最后 `AUTO_OFFSET_RESET_CONFIG` 被设置为 `earliest`，使得每次运行程序时都能保证从头消费一次消息。

下面来构建拓扑和 `stream`，如下面的代码所示：

```
StreamsBuilder builder = new StreamsBuilder();
KStream<String, String> source = builder.stream("streams-plaintext-input");
```

这一步很简单，我们仅仅实例化一个 `StreamsBuilder` 对象，然后调用其 `stream` 方法生成了一个 `KStream` 实例。注意 `stream` 方法的参数是 `streams-plaintext-input`。细心的读者会发现它和 10.3.4 节中 `Word Count` 实例的输入 topic 名字相同。没错，这里我们依然设置输入 topic 的名字是 `streams-plaintext-input`，即指定待统计的单词是来自哪个 topic 的消息数据。

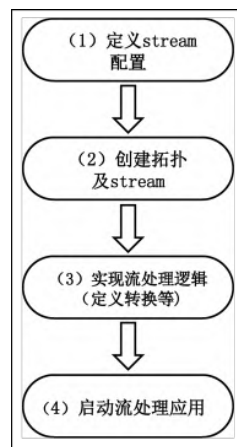


图 10.5 高阶 `Streams DSL` 开发流程

拿到 KStreams 对象实例后，我们开始实现具体的字数统计逻辑。由于是有状态（stateful）的流处理——还记得 stateful 的含义吧？我们需要使用 KTable 类。两者的区别稍后会讨论，首先来看看具体的实现代码：

```
KTable<String, Long> counts = source
    .flatMapValues(new ValueMapper1<String, Iterable<String>>() {
        @Override
        public Iterable<String> apply(String value) {
            return Arrays.asList(value.toLowerCase(Locale.getDefault()).
split(" "));
        }
    })
    .groupByKey(new KeyValueMapper<String, String, String>() {
        @Override
        public String apply(String key, String value) {
            return value;
        }
    })
    .count();

// 设置消息 key 的序列化类是 StringSerializer, value 的序列化类是 LongSerializer
counts.toStream().to("streams-wordcount-output", Produced.with
(Serdes.String(), Serdes.Long()));
```

上面的代码中比较关键的部分在于 `source.flatMapValues(...)`。KStream<String, String>本质上是一个消息 stream，而本例中我们对消息的 key 不感兴趣。相反地，程序只需要考虑每条消息的 value，故这里调用 `source.flatMapValues` 表示处理的是消息的 value。具体对 value 执行的操作由 ValueMapper 匿名实现类来定义，在 apply 方法中我们仅仅将消息按照空格进行 split 并封装成 List 返回。举一个例子来说明，假设消息的 value 是“hello Kafka Streams”，那么 flatMapValues 之后，整个 stream 会包含单词列表的 stream，即 stream 中包含的列表为[hello, Kafka, Streams]。因此我们说这一步实现了单词的分割。

拿到这些单词之后，我们就可以开始对其进行计数统计了，不过在计数之前，必须对其进行分组，即把相同单词归为一组。这就是 groupBy 操作的逻辑。groupBy 接收一个 KeyValueMapper 实现类，该实现类的 apply 方法返回的值将作为新的 key。由于本例只需要把单词作为 key，故这里 apply 方法直接返回第二个参数 value 即可。

groupBy 返回一个 KGroupedStream 类，这里的 KGroupedStream 和普通的 KStream 是有区别的。前者表示的是根据某些字段进行分组后的 stream，因此可以执行一些专属的聚合操作；

¹ 这种匿名类是 Java 7 的写法，Java 8 可以使用 Lambda 表达式。

而后者只是简单的消息 stream，无 key 的语义。

既然是进行计数操作，我们只需在 `KGroupedStream` 上调用其 `count` 方法来统计单词个数。`count` 方法会根据分组 key 对消息（也就是一个个具体的单词）进行计数，并封装成 `KTable` 返回。代码的最后部分调用 `counts.toStream.to` 将结果写入输出 topic——`streams-wordcount-output`，以供后续状态查询或结果保存之用。

这就是一个简单的 Word Count 流处理逻辑的实现。读者可以发现，由于 Kafka Streams 封装了很好的 API，用户只需要按照处理语义调用相应的转换算子。这极大地简化了流处理程序的开发。

在开始最后一步的实现之前，我们讨论一下 `KStream` 和 `KTable` 的区别。在上面的代码解释中多次出现 `KStream` 或 `KTable` 字眼，实际上，它们都是流的表现形式，只是在语义表达上有所不同。`KStream` 中的消息彼此没有任何关联，无论它们的 key 相同与否，消息之间是独立无关的；而 `KTable` 则不同，key 相同的消息会产生覆盖效果。一般认为后面出现的消息是对前面消息的状态覆盖，因此处理方式也是不同的，比如同样是求和操作，`KStream` 对同一个 key 的所有消息求和，而 `KTable` 只会取每个 key 的最后一条消息。对于单词计数结果而言，我们显然希望每次计算之后结果是最新的，也就是说新计算的结果要覆盖之前的结果。举一个简单的例子来说明，假设我们的单词流是：

```
hello -> 1
Kafka -> 2
Streams -> 3
```

当单词“Kafka”再次出现后，单词计数应该是：

```
hello -> 1
Kafka -> 3
Streams -> 3
```

对 `KStream` 而言，求和的处理结果是 `hello -> 2, Kafka -> 5, Streams -> 6`；而对 `KTable` 而言，求和的处理结果是 `hello -> 1, Kafka -> 3, Streams -> 3`。显然，对于计数，使用 `KTable` 得到的处理结果是准确的。

好了，写好上面 3 步逻辑之后，下面实现最后一步的代码，开启流应用：

```
streams.start();
```

很简单，对吧？`start` 方法会开启底层的所有线程从而执行真正的消息处理逻辑。下面给出完整的代码：

```
public class WordCountDemo {
```

```

        public static void main(String[] args) throws Exception {
            Properties props = new Properties();
            props.put(StreamsConfig.APPLICATION_ID_CONFIG, "streams-wordcount");
            props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
            props.put(StreamsConfig.CACHE_MAX_BYTES_BUFFERING_CONFIG, 0);
            props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.
String().getClass().getName());
            props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.
String().getClass().getName());
            props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");

            StreamsBuilder builder = new StreamsBuilder();

            KStream<String, String> source = builder.stream("streams-plaintext-
input");

            KTable<String, Long> counts = source
                .flatMapValues(new ValueMapper<String, Iterable<String>>() {
                    @Override
                    public Iterable<String> apply(String value) {
                        return Arrays.asList(value.toLowerCase(Locale.getDefault()).
split(" "));
                    }
                })
                .groupBy(new KeyValueMapper<String, String, String>() {
                    @Override
                    public String apply(String key, String value) {
                        return value;
                    }
                })
                .count();

            counts.toStream().to("streams-wordcount-output", Produced.with
(Serdes.String(), Serdes.Long()));

            final KafkaStreams streams = new KafkaStreams(builder.build(),
props);
            final CountDownLatch latch = new CountDownLatch(1);

            Runtime.getRuntime().addShutdownHook(new Thread("streams-wordcount-
shutdown-hook") {
                @Override
                public void run() {
                    streams.close();
                }
            });
        }
    }
}

```

```

        latch.countDown();
    }
});
try {
    streams.start();
    latch.await();
} catch (Throwable e) {
    System.exit(1);
}
System.exit(0);
}
}

```

总结一下，使用 Streams DSL 构建 Kafka Streams 程序简单易用，用户只需要关注流处理转换逻辑的实现部分，通常不需要关心底层的状态存储等细节。

不过，有些时候高阶 DSL 提供的语义无法满足用户定制化的需求，所以 Kafka Streams 还提供了第二套 API 供用户实现较为复杂的逻辑。这就是 processor API，也被称为低阶 API。下面演示如何使用 processor API 构建 Word Count 程序。图 10.6 给出了典型的 processor API 开发流程。有些步骤与高阶 Streams DSL 的流程是相同的，故这里不再赘述。我们关心的是其中的（2）、（3）、（4）步，即添加 source/sink 以及 processor 逻辑实现。

熟悉 Apache Storm 的读者可知这套 processor API 基本上与 Storm 的开发流程是一致的。在开发任何一个拓扑前，我们必须显式地定义 source（数据从哪里来）、sink（数据到哪里去）以及中间的转换逻辑（processor）。只不过后者在 Storm 中的专属名称是 Bolt，在 Kafka

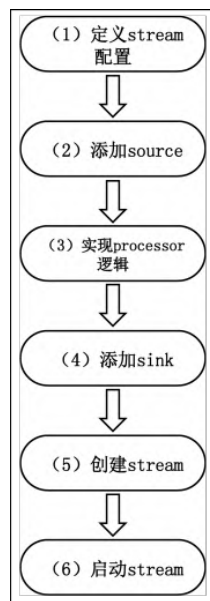


图 10.6 低阶 processor API 开发流程

Streams 中它被称为 processor 而已。讲到这里，笔者最想说的是实际上这些流处理框架在设计上都遵循类似的处理流程，读者领会了其中的一个框架往往能触类旁通，迅速地掌握其他框架的设计理念。为了验证笔者所言不虚，读者可以自行比较 Apache Flink 与 Storm、Kafka Streams。你会惊讶地发现，Flink 同样提供了一套低阶的流处理操作，只不过换了名字，叫 Process Function。

言归正传，当前在 Kafka Streams 中指定 source 和 sink 实际上只是指定了底层的 Kafka topic，也就是说它只与 Kafka 集群的 topic 数据进行交互。在本章中我们反复强调 Kafka

Streams 只是一个客户端库，故和其他流处理框架提供开箱即用的 connector 不同的是，与外部系统集成这件事情它是不管的，这部分的工作交由 Kafka Connect 组件处理，而 Kafka Connect 实际上也是首先把数据搬移到 Kafka topic 中后再交由 Kafka Streams 处理。这再次证明了 Kafka Connect 和 Kafka Streams 与底层的 Kafka 集群是深度绑定的。具体到本例，代码如下：

```
// 指定 source topic 名称为 streams-plaintext-input
builder.addSource("Source", "streams-plaintext-input");
// 指定 sink topic 名称为 streams-wordcount-processor-output
builder.addSink("Sink", "streams-wordcount-processor-output", "Process");
```

在上面的代码中设置 source 的名字为 Source，sink 的名称为 Sink。注意这两个名称，后面的 processor 中需要使用这些名称进行关联。

现在我们使用 processor API 实现单词计数逻辑，代码如下：

```
builder.addProcessor("Process", new MyProcessorSupplier(), "Source");
```

在上面的代码中我们创建了一个名为 Process 的 processor，其核心代码是在自定义类 MyProcessorSupplier 中实现的。下面看看该类的具体实现：

```
private static class MyProcessorSupplier implements ProcessorSupplier<
String, String> {

    @Override
    public Processor<String, String> get() {
        return new Processor<String, String>() {
            private ProcessorContext context;
            private KeyValueStore<String, Integer> kvStore;

            @Override
            @SuppressWarnings("unchecked")
            public void init(final ProcessorContext context) {
                this.context = context;
                this.context.schedule(1000, PunctuationType.STREAM_TIME,
new Punctuator() {

                    @Override
                    public void punctuate(long timestamp) {
                        try (KeyValueIterator<String, Integer> iter =
kvStore.all()) {
                            System.out.println(timestamp);
                            while (iter.hasNext()) {
                                KeyValue<String, Integer> entry = iter.
next();
                                System.out.println "[" + entry.key + ",
" + entry.value + " ]");
                            }
                        }
                    }
                });
            }
        };
    }
}
```

```

        context.forward(entry.key, entry.value.
toString());
    }
}

});
this.kvStore = (KeyValueStore<String, Integer>) context.
getStateStore("Counts");
}

@Override
public void process(String dummy, String line) {
    String[] words = line.toLowerCase(Locale.getDefault()).
split(" ");

    for (String word : words) {
        Integer oldValue = this.kvStore.get(word);

        if (oldValue == null) {
            this.kvStore.put(word, 1);
        } else {
            this.kvStore.put(word, oldValue + 1);
        }
    }

    context.commit();
}
};
}
}

```

创建 `processor` 通常由实现 `ProcessSupplier` 接口开始。该接口只有一个方法 `get()`。此方法返回一个 `processor` 对象。本例中我们创建 `MyProcessorSupplier` 类实现 `ProcessSupplier` 接口的 `get` 方法，以此来创建具体的 `processor` 实例。

由于是低阶 API，故用户需要自己处理状态保存的事宜，因此在创建的 `processor` 实例中使用 `KeyValueStore` 作为状态存储。`processor` 也是一个接口，其中比较重要的方法包括 `init` 和 `process`。

在 `init` 中创建一个名为 `Counts` 的 KV 存储并调度一个额外的线程每隔 1 秒打印存储中保存的内容，而在 `process` 方法中执行真正的单词分割-分组-计数逻辑。另外还必须手动实现计算结果的状态保存，如下：

```

for (String word : words) {
    Integer oldValue = this.kvStore.get(word);

```

```
        if (oldValue == null) {
            this.kvStore.put(word, 1);
        } else {
            this.kvStore.put(word, oldValue + 1);
        }
    }
}
```

对于每个单词，首先查询在存储中是否保存了该单词的计数结果，如果是则更新计数器+1，否则设置初始计数为1。

定义好 **processor** 之后，还需要指定这个 **processor** 处理结果保存到哪里。本例中调用以下代码指定使用内存空间作为 **KV** 存储，如下：

```
builder.addStateStore(Stores.keyValueStoreBuilder(
    Stores.inMemoryKeyValueStore("Counts"),
    Serdes.String(),
    Serdes.Integer()),
    "Process");
```

做完这些之后，采用与高阶 **Stream DSL** 同样的流程创建 **stream** 并启动 **stream** 即可。完整的 **processor API** 版的代码如下：

```
public class WordCountProcessorDemo {

    private static class MyProcessorSupplier implements ProcessorSupplier<
String, String> {

        @Override
        public Processor<String, String> get() {
            return new Processor<String, String>() {
                private ProcessorContext context;
                private KeyValueStore<String, Integer> kvStore;

                @Override
                @SuppressWarnings("unchecked")
                public void init(final ProcessorContext context) {
                    this.context = context;
                    this.context.schedule(1000, PunctuationType.STREAM_TIME,
new Punctuator() {

                        @Override
                        public void punctuate(long timestamp) {
                            try (KeyValueIterator<String, Integer> iter
= kvStore.all()) {

                                while (iter.hasNext()) {
                                    KeyValue<String, Integer> entry =
```

```

iter.next();

                                System.out.println "[" + entry.key +
", " + entry.value + "]);

                                context.forward(entry.key, entry.value.
toString());
                                }
                                }
                                }
                                });
                                this.kvStore = (KeyValueStore<String, Integer>) context.
getStateStore("Counts");
                                }

                                @Override
                                public void process(String dummy, String line) {
                                    String[] words = line.toLowerCase(Locale.getDefault()).
split(" ");

                                    for (String word : words) {
                                        Integer oldValue = this.kvStore.get(word);
                                        this.kvStore.put(word, oldValue == null ? 1 :
oldValue + 1);
                                    }
                                    context.commit();
                                }

                                @Override
                                @Deprecated
                                public void punctuate(long timestamp) {}

                                @Override
                                public void close() {}
                            };
                        }
                    }

    public static void main(String[] args) throws Exception {
        Properties props = new Properties();
        props.put(StreamsConfig.APPLICATION_ID_CONFIG, "streams-wordcount-
processor");
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(StreamsConfig.CACHE_MAX_BYTES_BUFFERING_CONFIG, 0);
    }
}

```

```

        props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.
String().getClass());
        props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.
String().getClass());

        // 重设位移为 earliest
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");

        Topology builder = new Topology();

        builder.addSource("Source", "streams-plaintext-input");

        builder.addProcessor("Process", new MyProcessorSupplier(), "Source");
        builder.addStateStore(Stores.keyValueStoreBuilder(
            Stores.inMemoryKeyValueStore("Counts"), Serdes.String(),
Serdes.Integer()), "Process");

        builder.addSink("Sink", "streams-wordcount-processor-output", "Process");

        final KafkaStreams streams = new KafkaStreams(builder, props);
        final CountDownLatch latch = new CountDownLatch(1);

        // 增加 JVM 关闭钩子响应 Ctrl+C
        Runtime.getRuntime().addShutdownHook(new Thread("streams-wordcount-
shutdown-hook") {
            @Override
            public void run() {
                streams.close();
                latch.countDown();
            }
        });

        try {
            streams.start();
            latch.await();
        } catch (Throwable e) {
            System.exit(1);
        }
        System.exit(0);
    }
}

```

10.3.6 Kafka Streams 状态查询

除了进行实时的流处理，Kafka Streams 还提供了一些方法供用户实时查询处理结果或状

态。有了这种查询能力，用户甚至不必为查询而单独将结果持久化到外部存储中（如数据库等）。还是以 Word Count 程序为例，若 Kafka Streams 没有提供 API 让我们实时查询单词计数的结果，我们就必须将结果实时保存到外部持久化存储上，比如数据库的某张表中。然后使用 SQL 或脚本来查询当前的计数结果。反之，如果直接使用 Kafka Streams 提供的状态查询方法，那么通常情况下就没有必要将结果存入数据库了，因此在部署上简化了软件栈维护的开销。

Kafka Streams 的状态查询既可以是对本地状态的查询，也允许用户在应用程序外查询。当然默认情况下 Kafka Streams 应用是未启用查询的，需要用户显式地调用相应的 API 来开启。

查询本地状态指的是查询本地运行的 Kafka Streams 应用实例的状态存储。用户能够在应用程序中直接使用 API 查询该状态存储上的数据。查询状态是只读的，故用户无法修改状态。

查询远程状态存储指的是查询整个 Kafka Streams 应用程序的存储。当涉及整个应用状态时，我们需要把各种本地状态联合在一起共同构成一个整体状态。如果要查询远端的应用实例状态，我们需要一种 RPC 机制，用来在网络上传输这些实例的状态信息。目前 RPC 机制需要用户自行提供。

下面我们依然以 Word Count 为例，分别介绍如何调用相关 API 查询本地状态和远程状态。

如果一个 Kafka Streams 应用运行有多个实例，那么每个实例专属的状态信息对该实例而言就是本地状态。根据 Kafka Streams 的定义，查询本地状态只会获取特定实例上的数据。我们可以使用 `KafkaStreams.store` 方法查询本地状态。如果依然以 Word Count 为例，下面的代码是统计单词计数的核心逻辑：

```
StreamsConfig config = ...;
StreamsBuilder builder = ...;
KStream<String, String> textLines = ...;

KGroupedStream<String, String> groupedByWord = textLines
    .flatMapValues(value -> Arrays.asList(value.toLowerCase().split("\\W+")))
    .groupBy((key, word) -> word, Serialized.with(stringSerde, stringSerde));

groupedByWord.count("CountsKeyValueStore");

KafkaStreams streams = new KafkaStreams(builder.build(), config);
streams.start();
```

和 10.3.4 节中的代码不同的是，这次我们在调用 `count` 方法时传入了一个字符串“CountsKeyValueStore”。这个名字即本地存储的名字，而该 KV 存储保存了输入 topic——word-count-input 的单词计数统计数据。当启动上面的代码所在的 Kafka Streams 程序时，就能够访问名为 CountsKeyValueStore 的 KV 存储，查询方法如下：

```
ReadOnlyKeyValueStore<String, Long> keyValueStore =
    streams.store("CountsKeyValueStore",
QueryableStoreTypes.keyValueStore());

System.out.println("count for hello:" + keyValueStore.get("hello"));

KeyValueIterator<String, Long> range = keyValueStore.range("all", "streams");
while (range.hasNext()) {
    KeyValue<String, Long> next = range.next();
    System.out.println("count for " + next.key + ": " + value);
}
range.close();

KeyValueIterator<String, Long> range = keyValueStore.all();
while (range.hasNext()) {
    KeyValue<String, Long> next = range.next();
    System.out.println("count for " + next.key + ": " + value);
}
range.close();
```

首先，我们通过调用 `KafkaStreams.store` 构建了一个只读的 KV 存储对象 `ReadOnlyKeyValueStore`，然后使用该对象的 `get` 方法获取单词 `hello` 出现的次数。接下来，继续通过该对象的 `range` 方法遍历当前存储中`[all, streams]`内的所有单词计数情况。最后通过 `all` 方法遍历当前存储中所有单词的计数。值得注意的是，不论是范围遍历还是全局遍历，都不要忘记关闭迭代器，以免造成内存泄漏。

虽然 `Kafka Streams` 默认提供了一些常用的状态存储以及查询 API，但如果用户实现了自定义的状态存储，那么应该如何操作呢？事实上，`Kafka Streams` 提供了一些接口供用户实现。

- 首先，自定义状态存储必须实现 `StateStore` 接口。
- 其次，用户必须实现针对该存储的各种操作。
- 再次，推荐用户最好实现一套只读的操作以保证状态不会被修改。
- 最后，用户需要实现 `StoreSupplier` 接口创建具体的状态存储。

如果是查询远程状态存储，用户通常需要遵循以下步骤。

(1) 自行引入 RPC 机制到 `Kafka Streams` 应用中：用户需要自行在应用中内置 RPC 服务并暴露出可供其他应用连接和访问的 `endpoint`。

(2) 设置 `Kafka Streams` 端参数 `application.server=RPC 服务 endpoint`（如 `localhost:4000`），并确保每个应用实例都有专属的 RPC 服务。

(3) 通过 RPC 服务达成远程实例的服务发现并调用专属 API 获取状态存储数据。

下面详细说说上面最后两步的具体实现。假设用户在应用实例中内置了一个 RPC 服务，endpoint 信息是 localhost:4000。下面代码给出了如何为该实例设置 RPC 服务：

```
Properties props = new Properties();
String rpcEndpoint = "localhost:4000";
props.put(StreamsConfig.APPLICATION_SERVER_CONFIG, rpcEndpoint);
//.....其他参数设置.....

StreamsConfig config = new StreamsConfig(props);
StreamsBuilder builder = new StreamsBuilder();

KStream<String, String> textLines = builder.stream("word-count-input",
Consumed.with(stringSerde, stringSerde));

KGroupedStream<String, String> groupedByWord = textLines
    .flatMapValues(value -> Arrays.asList(value.toLowerCase().split("\\W+")))
    .groupBy((key, word) -> word, Serialized.with(stringSerde, stringSerde));

groupedByWord.count(Materialized.<String, Long, KeyValueStore<Bytes, byte[]>>as
("word-count"));

KafkaStreams streams = new KafkaStreams(builder.build(), streamsConfiguration);
streams.start();

MyRPCService rpcService = ...;
rpcService.listenAt(rpcEndpoint);
```

上面的代码中比较关键的地方有两点：第一是设置 `application.server` 参数，即 `StreamsConfig.APPLICATION_SERVER_CONFIG`，这对于使用 RPC 服务是必要的一步；第二便是在启动 stream 之后还要再启动 RPC 服务。

做好这些之后，我们就可以使用另一个程序来发现并获取这个远程 Streams 实例的服务及数据了。Kafka Streams 提供了 `StreamsMetadata` 类来表征应用实例的元数据信息，例如 RPC endpoint 和本地状态存储等。下面这段代码可用于获取远端应用 Word Count 实例的状态信息：

```
KafkaStreams streams = ...;
Collection<StreamsMetadata> wordCountHosts = streams.allMetadataForStore
("word-count");

HttpClient http = ...;

StreamsMetadata metadata = streams.metadataForKey("word-count", "alice",
Serdes.String().serializer());
```



```
Long result = http.getLong("http://" + metadata.host() + ":" + metadata.port() + "/word-count/alice");

Optional<Long> result = streams.allMetadataForStore("word-count")
    .stream()
    .map(streamsMetadata -> {
        String url = "http://" + streamsMetadata.host() + ":" + streamsMetadata.port() + "/word-count/alice";
        return http.getLong(url);}).filter(s -> s != null).findFirst();
```

上面的代码中使用 `streams.allMetadataForStore("word-count")` 即可获得 Word Count 应用所有运行实例的元数据信息。之后我们使用 `metadataForKey` 来获取单词 `alice` 的总计数。当然还有另一种方式来实现，即代码中最后一段表述的那样：遍历所有应用实例的状态信息直至找到包含 `alice` 单词计数信息的数据并返回。总之，通过 RPC 我们能够发现并获取所有应用实例的状态信息。

10.4 本章小结

本章主要介绍了 Kafka 0.10.0.0 版本引入的两个新组件 Kafka Connect 和 Kafka Streams，并分别介绍了它们的设计思想、功能定位以及常用的使用方法。这两个新组件的引入使得 Kafka 的定位发生了根本性的变化。Kafka 不再定义自己为消息引擎，如今它更像一个流处理平台，既能承接与外部系统的数据集成服务，也能支持上下游的实时数据处理。这极大地扩展了 Kafka 的使用场景，降低了用户搭建企业级流数据处理系统的成本与开销。

当然，由于这两个组件是新引入的且还在不断演进中，本章中介绍的方法和使用场景可能会随着后续版本的变化而变化，对于这两个组件今后的发展，让我们拭目以待吧。

Apache Kafka实战

本书是涵盖Apache Kafka各方面的具有实践指导意义的工具书和参考书。作者结合典型的使用场景，对Kafka整个技术体系进行了较为全面的讲解，以便读者能够举一反三，直接应用于实践。同时，本书还对Kafka的设计原理及其流式处理组件进行了较深入的探讨，并给出了翔实的案例。

ISBN: 978-7-121-33776-5

书价: 89.00