

O'REILLY®

2-е издание

Apache Kafka



Потоковая
обработка
и анализ данных

Гвен Шапира, Тодд Палино
Раджини Сиварам, Крит Петти



SECOND EDITION

Kafka: The Definitive Guide

Real-Time Data and Stream Processing at Scale

*Gwen Shapira, Todd Palino,
Rajini Sivaram, and Krit Petty*

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Apache Kafka

Потоковая обработка
и анализ данных

2-е издание

Гвен Шапира, Тодд Палино
Раджини Сиварам, Крит Петти



Санкт-Петербург • Москва • Минск

2023

ББК 32.973.233
УДК 004.042
А79

Гвен Шапира, Тодд Палино, Раджини Сиварам, Крит Петти

А79 Apache Kafka. Потокковая обработка и анализ данных. 2-е изд. — СПб.: Питер, 2023. — 512 с.: ил. — (Серия «Бестселлеры O'Reilly»).
ISBN 978-5-4461-2288-2

При работе любого enterprise-приложения образуются данные: это файлы логов, метрики, информация об активности пользователей, исходящие сообщения и т. п. Правильные манипуляции над всеми этими данными не менее важны, чем сами данные. Если вы — архитектор, разработчик или выпускающий инженер, желающий решать подобные проблемы, но пока не знакомы с Apache Kafka, то именно отсюда узнаете, как работать с этой свободной потоковой платформой, позволяющей обрабатывать очереди данных в реальном времени.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.233
УДК 004.042

Права на издание получены по соглашению с O'Reilly. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1492043065 англ.

© Authorized Russian translation of the English edition of Kafka: The Definitive Guide, 2E ISBN 9781492043089 © 2022 Chen Shapira, Todd Palino, Rajini Sivaram and Krit Petty.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

ISBN 978-5-4461-2288-2

© Перевод на русский язык ООО «Прогресс книга», 2023

© Издание на русском языке, оформление ООО «Прогресс книга», 2023

© Серия «Бестселлеры O'Reilly», 2023

Краткое содержание

Отзывы о книге.....	17
Предисловие ко второму изданию.....	19
Предисловие к первому изданию.....	21
Введение.....	24
От издательства.....	29
Глава 1. Знакомьтесь: Kafka.....	30
Глава 2. Установка Kafka.....	52
Глава 3. Производители Kafka: запись сообщений в Kafka.....	83
Глава 4. Потребители Kafka: чтение данных из Kafka.....	115
Глава 5. Программное управление Apache Kafka.....	154
Глава 6. Внутреннее устройство Kafka.....	178
Глава 7. Надежная доставка данных.....	209
Глава 8. Семантика «точно один раз».....	231
Глава 9. Создание конвейеров данных.....	254
Глава 10. Зеркальное копирование между кластерами.....	285
Глава 11. Обеспечение безопасности Kafka.....	323
Глава 12. Администрирование Kafka.....	369
Глава 13. Мониторинг Kafka.....	405
Глава 14. Поточковая обработка.....	450
Приложение А. Установка Kafka в других операционных системах.....	495
Приложение Б. Дополнительные инструменты Kafka.....	501
Об авторах.....	509
Иллюстрация на обложке.....	511

Оглавление

Отзывы о книге.....	17
Предисловие ко второму изданию.....	19
Предисловие к первому изданию.....	21
Введение.....	24
Для кого предназначена эта книга.....	25
Условные обозначения.....	25
Использование примеров кода.....	26
Благодарности.....	27
От издательства.....	29
Глава 1. Знакомьтесь: Kafka.....	30
Обмен сообщениями по типу «публикация/подписка».....	30
С чего все начинается.....	31
Отдельные системы организации очередей.....	33
Открываем для себя систему Kafka.....	33
Сообщения и пакеты.....	36
Схемы.....	36
Топики и разделы.....	37
Производители и потребители.....	38
Брокеры и кластеры.....	39
Несколько кластеров.....	41
Почему Kafka?.....	43
Несколько производителей.....	43
Несколько потребителей.....	43
Сохранение информации на диске.....	43
Масштабируемость.....	44
Высокое быстродействие.....	44
Особенности платформы.....	44

Экосистема данных.....	45
Сценарии использования.....	46
История создания Kafka	48
Проблема LinkedIn	48
Рождение Kafka.....	49
Открытый исходный код	50
Коммерческое взаимодействие	51
Название	51
Приступаем к работе с Kafka.....	51
Глава 2. Установка Kafka	52
Настройка среды.....	52
Выбрать операционную систему.....	52
Установить Java.....	52
Установить ZooKeeper	53
Установка брокера Kafka.....	56
Настройка брокера.....	58
Основные параметры брокера	58
Настройки топиков по умолчанию.....	61
Выбор аппаратного обеспечения.....	67
Пропускная способность дисков	67
Емкость диска.....	68
Память.....	68
Передача данных по сети	69
CPU	69
Кafka в облачной среде	70
Microsoft Azure	70
Веб-сервисы Amazon Web Services.....	71
Настройка кластеров Kafka.....	71
Сколько должно быть брокеров	72
Конфигурация брокеров.....	74
Тонкая настройка операционной системы	74
Промышленная эксплуатация.....	78
Параметры сборки мусора.....	78
Планировка ЦОД.....	80
Размещение приложений на ZooKeeper	80
Резюме.....	82

Глава 3. Производители Kafka: запись сообщений в Kafka	83
Обзор производителя	84
Создание производителя Kafka	86
Отправка сообщения в Kafka	88
Синхронная отправка сообщения	89
Асинхронная отправка сообщения	90
Настройка производителей	91
client.id	91
acks	92
Время доставки сообщения	93
linger.ms	96
buffer.memory	96
compression.type	97
batch.size	97
max.in.flight.requests.per.connection	97
max.request.size	98
receive.buffer.bytes и send.buffer.bytes	98
enable.idempotence	99
Сериализаторы	99
Пользовательские сериализаторы	100
Сериализация с помощью Apache Avro	102
Использование записей Avro с Kafka	104
Разделы	107
Реализация пользовательской стратегии секционирования	108
Заголовки	110
Перехватчики	110
Квоты и регулирование запросов	112
Резюме	114
Глава 4. Потребители Kafka: чтение данных из Kafka	115
Принципы работы потребителей Kafka	115
Потребители и группы потребителей	115
Группы потребителей и переконфигурирование разделов	118
Статические участники группы	122
Создание потребителя Kafka	123
Подписка на топики	123

Цикл опроса	124
Потокобезопасность	126
Настройка потребителей	127
fetch.min.bytes	127
fetch.max.wait.ms	127
fetch.max.bytes	128
max.poll.records	128
max.partition.fetch.bytes	128
session.timeout.ms и heartbeat.interval.ms	128
max.poll.interval.ms	129
default.api.timeout.ms	130
request.timeout.ms	130
auto.offset.reset	130
enable.auto.commit	131
partition.assignment.strategy	131
client.id	132
client.rack	132
group.instance.id	133
receive.buffer.bytes и send.buffer.bytes	133
offsets.retention.minutes	133
Фиксация и смещения	134
Автоматическая фиксация	135
Фиксация текущего смещения	136
Асинхронная фиксация	137
Сочетание асинхронной и синхронной фиксации	139
Фиксация заданного смещения	140
Прослушивание на предмет перебалансировки	141
Получение записей с заданными смещениями	144
Выход из цикла	145
Десериализаторы	147
Пользовательские сериализаторы	148
Использование десериализации Avro в потребителе Kafka	150
Автономный потребитель: зачем и как использовать потребитель без группы	151
Резюме	153

Глава 5. Программное управление Apache Kafka	154
Обзор AdminClient	155
Асинхронный и в конечном итоге согласованный API	155
Опции	156
Плоская иерархия	156
Дополнительные примечания	156
Жизненный цикл AdminClient: создание, настройка и закрытие	157
client.dns.lookup	158
request.timeout.ms	159
Управление основными топиками	160
Управление конфигурацией	164
Управление группами потребителей	165
Изучение групп потребителей	166
Модификация групп потребителей	168
Метаданные кластера	170
Расширенные операции администратора	170
Добавление разделов в топик	170
Удаление записей из топика	171
Выборы лидера	172
Перераспределение реплик	173
Тестирование	174
Резюме	177
Глава 6. Внутреннее устройство Kafka	178
Членство в кластере	178
Контроллер	179
KRaft: новый контроллер Kafka на основе Raft	181
Репликация	183
Обработка запросов	186
Запросы от производителей	189
Запросы на извлечение	189
Другие запросы	194
Физическое хранилище	195
Многоуровневое хранилище	196
Распределение разделов	198
Управление файлами	200
Формат файлов	200

Индексы	203
Сжатие	204
Как происходит сжатие	204
Удаленные события	206
Когда выполняется сжатие топиков.....	207
Резюме	208
Глава 7. Надежная доставка данных.....	209
Гарантии надежности	210
Репликация.....	211
Настройка брокера.....	212
Коэффициент репликации	213
«Нечистый» выбор ведущей реплики	214
Минимальное число согласованных реплик.....	216
Поддержание синхронизации реплик	217
Долговременное хранение на диске.....	217
Использование производителей в надежной системе.....	218
Отправка подтверждений	219
Настройка повторов отправки производителями.....	220
Дополнительная обработка ошибок	221
Использование потребителей в надежной системе	221
Свойства конфигурации потребителей, важные для надежной обработки.....	222
Фиксация смещений в потребителях явным образом	224
Проверка надежности системы.....	226
Проверка конфигурации	226
Проверка приложений	228
Мониторинг надежности при промышленной эксплуатации	228
Резюме	230
Глава 8. Семантика «точно один раз».....	231
Идемпотентный производитель.....	232
Как работает идемпотентный производитель	232
Ограничения идемпотентного производителя.....	235
Как использовать идемпотентный производитель Kafka	236
Транзакции	237
Сценарии использования транзакций	237
Какие проблемы решают транзакции	238

Как транзакции гарантируют «точно один раз»	239
Какие проблемы не решаются транзакциями.....	242
Как использовать транзакции	245
Идентификаторы транзакций и ограждения	248
Как работают транзакции	250
Производительность транзакций.....	252
Резюме.....	253
Глава 9. Создание конвейеров данных	254
Соображения по поводу создания конвейеров данных.....	255
Своевременность	255
Надежность	256
Высокая/переменная нагрузка	257
Форматы данных.....	257
Преобразования	258
Безопасность	259
Обработка сбоев.....	260
Связывание и гибкость	261
Когда использовать Kafka Connect, а когда — клиенты-производители и клиенты-потребители.....	262
Kafka Connect.....	263
Запуск Kafka Connect.....	263
Пример коннектора: файловый источник и файловый приемник.....	266
Пример коннектора: из MySQL в Elasticsearch	269
Преобразования одиночных сообщений	276
Взглянем на Kafka Connect поближе.....	278
Альтернативы Kafka Connect.....	282
Фреймворки ввода и обработки данных для других хранилищ	282
ETL-утилиты на основе GUI.....	283
Фреймворки потоковой обработки.....	283
Резюме.....	283
Глава 10. Зеркальное копирование между кластерами	285
Сценарии зеркального копирования данных между кластерами	286
Мультикластерные архитектуры.....	287
Реалии взаимодействия между различными ЦОД.....	287
Архитектура с топологией типа «звезда»	289
Архитектура типа «активный — активный»	291

Архитектура типа «активный — резервный»	293
Эластичные кластеры	301
Утилита MirrorMaker (Apache Kafka)	302
Настройка MirrorMaker	304
Топология мультикластерной репликации	307
Обеспечение безопасности MirrorMaker	308
Развертывание MirrorMaker для промышленной эксплуатации	309
Тонкая настройка MirrorMaker	314
Другие программные решения для зеркального копирования между кластерами	317
uReplicator компании Uber	317
LinkedIn Brooklin	318
Решения Confluent для зеркального копирования между ЦОД	319
Резюме	322
Глава 11. Обеспечение безопасности Kafka	323
Блокировка Kafka	324
Протоколы безопасности	326
Аутентификация	328
SSL	329
SASL	334
Повторная аутентификация	347
Обновления системы безопасности без простоя	349
Шифрование	350
Сквозное шифрование	351
Авторизация	353
AclAuthorizer	354
Настройка авторизации	358
Вопросы безопасности	360
Аудит	361
Обеспечение безопасности ZooKeeper	362
SASL	362
SSL	363
Авторизация	364
Обеспечение безопасности платформы	364
Защита паролей	365
Резюме	367

Глава 12. Администрирование Kafka	369
Операции с топиками.....	369
Создание нового топика	370
Вывод списка всех топиков в кластере	371
Подробное описание топиков.....	372
Добавление разделов.....	373
Уменьшение количества разделов.....	374
Удаление топика.....	375
Группы потребителей.....	376
Вывод списка и описание групп	376
Удаление группы.....	377
Управление смещениями	378
Динамические изменения конфигурации	379
Переопределение значений настроек топиков по умолчанию	380
Переопределение настроек клиентов и пользователей по умолчанию ...	382
Переопределение настроек конфигурации брокера по умолчанию	383
Описание переопределений настроек	384
Удаление переопределений настроек	385
Производство и потребление	385
Консольный производитель.....	385
Консольный потребитель	388
Управление разделами.....	391
Выбор предпочтительной ведущей реплики	391
Изменение реплик раздела	393
Сброс на диск сегментов журнала.....	398
Проверка реплик.....	400
Другие утилиты	401
Небезопасные операции	402
Перенос контроллера кластера	402
Отмена удаления топиков	403
Удаление топиков вручную.....	403
Резюме.....	404
Глава 13. Мониторинг Kafka	405
Основы показателей	405
Как получить доступ к показателям.....	405
Какие показатели нам нужны.....	407
Контроль состояния приложения.....	409

Цели на уровне обслуживания	410
Определения уровня сервиса	410
Какие показатели являются хорошими индикаторами уровня обслуживания	411
Использование целей уровня обслуживания для оповещений	412
Показатели брокеров Kafka.....	414
Диагностика проблем с кластером	414
Искусство недореплицированных разделов	416
Показатели брокеров	422
Показатели топиков и разделов	432
Мониторинг JVM	435
Мониторинг ОС	436
Журналирование	438
Мониторинг клиентов.....	439
Показатели производителя	439
Показатели потребителей	442
Квоты	446
Мониторинг отставания.....	447
Сквозной мониторинг	448
Резюме	449
Глава 14. Потокковая обработка	450
Что такое потокковая обработка.....	452
Основные понятия потокковой обработки	455
Топология	455
Время.....	456
Состояние.....	458
Таблично-потокковый дуализм	459
Временные окна.....	461
Гарантии обработки	462
Паттерны проектирования потокковой обработки	462
Обработка событий по отдельности	463
Обработка с использованием локального состояния	463
Многоэтапная обработка/повторное разделение на разделы.....	465
Обработка с применением внешнего справочника: соединение потока данных с таблицей	467
Соединение таблицы с таблицей	468
Соединение потоков.....	470

Внеочередные события	471
Повторная обработка	472
Интерактивные запросы	473
Kafka Streams в примерах	474
Подсчет количества слов	474
Сводные показатели фондовой биржи	477
Обогащение потока событий перехода по ссылкам	480
Kafka Streams: обзор архитектуры	483
Построение топологии	483
Оптимизация топологии	484
Тестирование топологии	484
Масштабирование топологии	485
Как пережить отказ	489
Сценарии использования потоковой обработки	490
Как выбрать фреймворк потоковой обработки	492
Резюме	494
Приложение А. Установка Kafka в других операционных системах	495
Установка в Windows	495
Использование Windows Subsystem для Linux	495
Использование Java естественным образом	496
Установка в macOS	498
Использование Homebrew	499
Установка вручную	500
Приложение Б. Дополнительные инструменты Kafka	501
Комплексные платформы	501
Развертывание и управление кластером	503
Мониторинг и исследование данных	505
Клиентские библиотеки	506
Потоковая обработка	507
Об авторах	509
Иллюстрация на обложке	511

Отзывы о книге

«Apache Kafka. Поточковая обработка и анализ данных» содержит все, что вам нужно знать, чтобы получить максимальную отдачу от Kafka как в облаке, так и в локальной сети. Это та книга, которую обязательно должны прочесть как разработчики, так и специалисты по эксплуатации. Гвен, Тодд, Раджини и Крит собрали многолетнюю мудрость в одной лаконичной книге. Она нужна вам, если вы работаете с Kafka.

*Крис Риккомини (Chris Riccomini),
инженер-программист, консультант
по стартапам и соавтор книги Missing README*

Это исчерпывающее руководство по основам Kafka и его практическому применению.

*Сумант Тамбе (Sumant Tambe), старший
инженер-программист в LinkedIn*

Эта книга обязательна к прочтению любому разработчику или администратору Kafka. Прочитайте ее от корки до корки, чтобы узнать все детали, или держите под рукой для быстрого ознакомления с какой-либо темой. В любом случае ясность изложения материала и техническая точность здесь превосходны.

*Робин Моффатт (Robin Moffatt),
штатный консультант разработчиков
в Confluent*

Это основополагающая книга для всех инженеров, интересующихся Kafka. Она сыграла решающую роль в оказании помощи компании Robinhood при масштабировании, обновлении и настройке Kafka для поддержки быстрого роста пользователей.

*Джарен М. Гловер (Jaren M. Glover),
первый инженер Robinhood, инвестор-менеджер*

Обязательная книга для всех, кто работает с Apache Kafka: разработчиков или администраторов, новичков или экспертов, пользователей или штатных сотрудников.

*Маттиас Дж. Сакс (Matthias J. Sax),
инженер-программист компании Confluent
и член Apache Kafka PMC*

Отличное руководство для любой команды, серьезно использующей Apache Kafka в производстве, и инженеров, работающих над распределенными системами в целом. Эта книга выходит далеко за рамки обычного ознакомительного уровня и рассказывает о том, как на самом деле работает Kafka, как ее следует использовать и где кроются подводные камни. Для каждой замечательной функции Kafka авторы четко перечисляют предостережения, о которых можно услышать только от «ветеранов». Эту информацию нелегко найти в одном месте где-либо еще. Ясность и глубина объяснений таковы, что я бы даже рекомендовал эту книгу инженерам, которые не используют Kafka: изучение принципов, вариантов проектирования и эксплуатационных проблем поможет им принимать лучшие решения при создании других систем.

*Дмитрий Рябой (Dmitriy Ryaboy),
вице-президент по разработке
программного обеспечения в Zymergen*

Предисловие ко второму изданию

Первое издание книги «Apache Kafka. Потокковая обработка и анализ данных» вышло пять лет назад. В то время, по оценкам нашей компании, платформа Apache Kafka использовалась в 30 % компаний из списка Fortune 500. Сегодня с ней работают более 70 % компаний из этого списка. Она по-прежнему является одним из самых популярных проектов с открытым исходным кодом в мире и находится в центре огромной экосистемы.

Отчего возник такой ажиотаж? Я думаю, это связано с тем, что в нашей инфраструктуре для работы с данными образовался огромный пробел. Традиционно управление данными сводилось к хранению — файловым хранилищам и базам данных, которые обеспечивают безопасность наших данных и позволяют найти нужный фрагмент в нужное время. В эти системы было вложено огромное количество интеллектуальной энергии и коммерческих инвестиций. Но современная компания — это не просто программное обеспечение с одной базой данных. Современная компания — это невероятно сложная система, состоящая из сотен или даже тысяч пользовательских приложений, микросервисов, баз данных, слоев SaaS и аналитических платформ. И все чаще перед нами встает проблема, как объединить все это в одно целое и заставить работать вместе в режиме реального времени.

Эта проблема заключается не в управлении данными в состоянии покоя, а в управлении данными в движении. И в самом центре этого движения находится Apache Kafka, которая стала фактической основой для любой платформы для обработки потока данных.

На протяжении всего этого пути Kafka не оставалась статичной. То, что началось как простой журнал фиксации, развивалось: добавлялись коннекторы и возможности обработки потоков, изобреталась собственная архитектура. Сообщество не только улучшило существующие API, параметры конфигурации, метрики и инструменты для повышения удобства использования и надежности Kafka — мы также представили новый программный API администрирования, новое поколение глобальной репликации и DR с MirrorMaker 2.0, новый протокол консенсуса на основе Raft, который позволяет запускать Kafka в одном исполняемом файле, и настоящую эластичность с поддержкой многоуровневого хранения. Возможно, самое главное то, что мы сделали Kafka простым

в критически важных случаях использования на предприятии, добавив поддержку расширенных параметров безопасности — аутентификации, авторизации и шифрования.

По мере развития Kafka мы видим, как совершенствуются и сценарии использования. Когда было опубликовано первое издание, большинство установок Kafka все еще находились в традиционных локальных центрах обработки данных, где применялись традиционные сценарии развертывания. Наиболее популярными были ETL и обмен сообщениями, а использование потоковой обработки делало лишь первые шаги. Пять лет спустя большинство установок Kafka находятся в облаке, и многие из них работают на Kubernetes. ETL и обмен сообщениями по-прежнему популярны, но к ним добавились микросервисы, управляемые событиями, обработка потоков в реальном времени, технология «Интернет вещей» (IoT), конвейеры машинного обучения и сотни отраслевых вариантов применения и шаблонов, которые варьируются от обработки претензий в страховых компаниях до торговых систем в банках, помощи в управлении играми в реальном времени и персонализации в видеоиграх и потоковых сервисах.

Несмотря на то что Kafka распространяют на новые среды и находят новые способы ее использования, чтобы писать приложения, которые будут грамотно использовать Kafka и уверенно внедрять ее в производство, требуется привыкнуть к уникальному образу мышления Kafka. Эта книга охватывает все, что необходимо разработчикам и сервисам SRE для наилучшего применения Kafka, от базовых API и конфигураций до новейших передовых возможностей. В ней рассказывается не только о том, что и как можно делать с помощью Kafka, но и о том, чего делать не следует, а также об антипаттернах проектирования, которых следует избегать. Эта книга может стать надежным проводником в мир Kafka как для начинающих пользователей, так и для опытных практиков.

*Джей Крепс (Jay Kreps), соучредитель
и генеральный директор Confluent*

Предисловие к первому изданию

Сегодня платформу Apache Kafka используют в тысячах компаний, в том числе более чем в трети компаний из списка Fortune 500. Kafka входит в число самых быстрорастущих проектов с открытым исходным кодом и породила обширную экосистему. Она находится в самом эпицентре управления потоками данных и их обработки.

Откуда же появился проект Kafka? Почему он возник? И что это вообще такое?

Начало Kafka положила внутренняя инфраструктурная система, которую мы создавали в LinkedIn. Мы заметили простую вещь: в нашей архитектуре было множество баз данных и других систем, предназначенных для *хранения* данных, но ничего не было для обработки непрерывных *потоков* данных. Прежде чем создать Kafka, мы перепробовали всевозможные готовые решения, начиная от систем обмена сообщениями и заканчивая агрегированием журналов и ETL-утилитами, но ни одно из них не дало нам того, что мы хотели.

В конце концов решено было создать нужное решение с нуля. Идея состояла в том, чтобы не ставить во главу угла хранение больших объемов данных, как в реляционных базах данных, хранилищах пар «ключ/значение», поисковых индексах или кэшах, а рассматривать данные как непрерывно развивающийся и постоянно растущий поток данных и проектировать информационные системы — и, конечно, архитектуру данных — на этой основе.

Эта идея нашла себе даже более широкое применение, чем мы ожидали. Хотя первым назначением Kafka было обеспечение функционирования работающих в реальном масштабе времени приложений и потоков данных социальной сети, сейчас она лежит в основе самых передовых архитектур во всех отраслях промышленности. Крупные розничные торговцы пересматривают свои основные бизнес-процессы с точки зрения непрерывных потоков данных, автомобильные компании собирают и обрабатывают в режиме реального времени потоки данных, получаемые от подключенных к Интернету автомобилей, пересматривают свои фундаментальные процессы и системы с ориентацией на Kafka и банки.

Так что же это такое — Kafka? Чем Kafka отличается от хорошо знакомых вам систем, которые сейчас используются?

Мы рассматриваем Kafka как *потокową платформу* (streaming platform) — систему, которая дает возможность публикации потоков данных и подписки на них, их хранения и обработки. Именно для этого Apache Kafka и создавалась. Рассматривать данные с этой точки зрения может показаться непривычно, но эта абстракция предоставляет исключительно широкие возможности создания приложений и архитектур. Kafka часто сравнивают с несколькими существующими типами технологий: корпоративными системами обмена сообщениями, большими информационными системами вроде Hadoop, утилитами интеграции данных или ETL. Каждое из этих сравнений в чем-то обоснованно, но не вполне правомерно.

Kafka напоминает систему обмена сообщениями тем, что обеспечивает возможность публикации и подписки на потоки сообщений. В этом она похожа на такие продукты, как ActiveMQ, RabbitMQ, MQSeries компании IBM и др. Но, несмотря на это сходство, у Kafka есть несколько существенных отличий от традиционных систем обмена сообщениями. Вот три основных: во-первых, Kafka ведет себя как современная распределенная кластерная система, способная масштабироваться в пределах, достаточных для всех приложений даже самой крупной компании. Вместо запуска десятков отдельных брокеров сообщений, вручную подключенных к различным приложениям, Kafka предоставляет централизованную платформу, гибко масштабирующуюся для обработки всех потоков данных компании. Во-вторых, Kafka — это настоящая система хранения, созданная для хранения данных столько времени, сколько нужно. Это дает колоссальные преимущества при ее использовании в качестве связующего слоя, а благодаря реальным гарантиям доставки обеспечиваются репликация, целостность и хранение данных в течение любого промежутка времени. В-третьих, потоковая обработка весьма существенно повышает уровень абстракции. Системы обмена сообщениями чаще всего просто передают сообщения. Возможности потоковой обработки в Kafka позволяют на основе потоков данных динамически вычислять производные потоки и наборы данных, причем при гораздо меньшем количестве кода. Эти отличия ставят Kafka довольно обособленно, так что нет смысла рассматривать ее как просто еще одну очередь.

Kafka можно рассматривать так же как предназначенную для реального времени версию Hadoop, и это была одна из причин, побудивших нас создать ее. С помощью Hadoop можно хранить и периодически обрабатывать очень большие объемы файловых данных. С помощью Kafka можно хранить и непрерывно обрабатывать потоки данных также в очень больших масштабах. С технической точки зрения определенное сходство между ними, безусловно, есть, и многие рассматривают развивающуюся сферу потоковой обработки как надмножество пакетной обработки, подобной той, которая выполняется с помощью Hadoop и различных его слоев обработки. Делая такое сравнение, упускают из виду тот факт, что сценарии использования, возможные при непрерывной, с низким

значением задержки обработке, сильно отличаются от естественных сценариев для систем пакетной обработки. В то время как Hadoop и большие данные ориентированы на аналитические приложения, зачастую применяемые в области хранилищ данных, присущее Kafka низкое значение задержки позволяет использовать ее для тех базовых приложений, которые непосредственно обеспечивают функционирование бизнеса. Это вполне логично: события в бизнесе происходят непрерывно, и возможность сразу же реагировать на них значительно облегчает построение сервисов, непосредственно обеспечивающих работу бизнеса, а также влияет на качество обслуживания клиентов и т. д.

Еще одна категория, с которой сравнивают Kafka, — ETL и утилиты интеграции данных. В конце концов, эти утилиты занимаются перемещением данных, и Kafka делает то же самое. Это в некоторой степени справедливо, но мне кажется, что коренное отличие состоит в том, что Kafka перевернула эту задачу вверх ногами. Kafka — не просто утилита для извлечения данных из одной системы и добавления их в другую, это платформа, основанная на концепции потоков событий в режиме реального времени. Это значит, что она может не только связывать готовые приложения с информационными системами, но и обеспечивать функционирование пользовательских приложений, создаваемых на основе этих самых потоков данных. Мне представляется, что подобная архитектура, в основу которой положены потоки событий, — действительно важная вещь. В некотором смысле эти потоки данных — центральный аспект современной «цифровой» компании, ничуть не менее важный, чем денежные потоки, которые вы видите в финансовом отчете.

Именно объединение этих трех сфер — сведение всех потоков данных воедино во всех сценариях использования — делает идею потоковой платформы столь притягательной.

Однако это несколько отличается от традиционного представления, и создание приложений, ориентированных на работу с непрерывными потоками данных, требует существенной смены парадигмы мышления у разработчиков, пришедших из вселенной приложений в стиле «запрос — ответ» и реляционных баз данных. Эта книга — однозначно лучший способ выучить Kafka от внутреннего устройства до API, написанная теми, кто знаком с ней лучше всего. Я надеюсь, что вы насладитесь ее чтением не меньше, чем я!

*Джей Крепс,
соучредитель и CEO компании Confluent*

Введение

Величайший комплимент, который только могут сделать автору технической книги: «Я хотел бы, чтобы эта книга была у меня, когда я только начинал изучать описанную в ней тематику». Именно с целью создать такую книгу мы и начали писать. Мы вспомнили опыт, полученный при создании Kafka, запуске ее в промышленную эксплуатацию и поддержке множества компаний при разработке на ее основе архитектур программного обеспечения и управления их конвейерами данных, и спросили себя: «Чем по-настоящему полезным мы можем поделиться с нашими читателями, чтобы сделать из новичков экспертов?» Эта книга отражает нашу каждодневную работу: мы запускаем Apache Kafka и помогаем людям использовать ее наилучшим образом.

Мы включили в книгу то, что считаем необходимым для успешной работы Apache Kafka при промышленной эксплуатации и создании устойчивых к ошибкам высокопроизводительных приложений на ее основе. Уделили особое внимание популярным сценариям применения: шинам сообщений для событийно управляемых микросервисов, приложениям, обрабатывающим потоки, а также крупномасштабным конвейерам данных. Мы также постарались сделать книгу достаточно универсальной и всеобъемлющей, чтобы она оказалась полезной всем, кто применяет Kafka, вне зависимости от сценария использования или архитектуры. Мы охватили в ней практические вопросы, например установку и конфигурацию Kafka, использование API Kafka, и уделили определенное внимание принципам построения платформы и гарантиям ее надежности. Рассмотрели также несколько потрясающих нюансов архитектуры Kafka: протокол репликации, контроллер и слой хранения. Полагаем, что знание внутреннего устройства Kafka — не только интересное чтение для интересующихся распределенными системами. Оно чрезвычайно полезно для принятия взвешенных решений при развертывании Kafka в промышленной эксплуатации и проектировании использующих ее приложений. Чем лучше вы понимаете, как работает Kafka, тем более обоснованно можете принимать решения относительно множества компромиссов, связанных с проектированием.

Одна из проблем в разработке программного обеспечения заключается в том, что всегда существует несколько вариантов решения одной задачи. Такие платформы, как Apache Kafka, обеспечивают большую гибкость, что замечательно для специалистов, но усложняет обучение новичков. Зачастую Apache Kafka

показывает, *как* использовать ту или иную возможность, но не *почему* следует или не следует делать это. Мы старались разъяснить, какие в конкретном случае существуют варианты, компромиссы и то, когда стоит и не стоит использовать различные возможности Apache Kafka.

Для кого предназначена эта книга

«Apache Kafka. Потокковая обработка и анализ данных» написана для разработчиков, использующих в своей работе API Kafka, а также инженеров-технологов (именуемых также SRE, DevOps или системными администраторами), занимающихся установкой, конфигурацией, настройкой и мониторингом ее работы при промышленной эксплуатации. Мы не забыли также об архитекторах данных и инженерах-аналитиках — тех, кто отвечает за проектирование и создание всей инфраструктуры данных компании. Некоторые главы, в частности 3, 4 и 14, ориентированы на Java-разработчиков. Для их усвоения важно, чтобы читатель был знаком с основами языка программирования Java, включая такие вопросы, как обработка исключений и параллелизм. В других главах, особенно 2, 10, 12 и 13, предполагается, что у читателя есть опыт работы с Linux и он знаком с настройкой сети и систем хранения на Linux. В оставшейся части книги Kafka и архитектуры программного обеспечения обсуждаются в более общих чертах, поэтому каких-то специальных познаний от читателей не требуется.

Еще одна категория людей, которых может заинтересовать данная книга, — руководители и архитекторы, не работающие непосредственно с Kafka, но сотрудничающие с теми, кто работает с ней. Ничуть не менее важно, чтобы они понимали, каковы предоставляемые платформой гарантии и в чем могут заключаться компромиссы, на которые придется идти их подчиненным и сотрудникам при создании основанных на Kafka систем. Эта книга будет полезна тем руководителям, которые хотели бы обучить своих сотрудников работе с Kafka или убедиться, что команда разработчиков владеет нужной информацией.

Условные обозначения

В данной книге используются следующие типографские соглашения.

Курсив

Обозначает новые термины и важные понятия.

Рубленый шрифт

Им выделены URL и адреса электронной почты.

Моноширинный шрифт

Используется для листингов программ, а внутри абзацев — для имен и расширений файлов и ссылки на элементы программ, такие как переменные или имена функций, переменные окружения, операторы и ключевые слова.

Жирный моноширинный шрифт

Представляет собой команды или другой текст, который должен быть в точности набран пользователем.

Моноширинный курсив

Отмечает текст, который необходимо заменить пользовательскими значениями или значениями, определяемыми контекстом.



Данный рисунок означает совет или указание.



Этот рисунок означает общее примечание.



Данный рисунок указывает на предупреждение или предостережение.

Использование примеров кода

Если у вас возникли технические вопросы или проблемы с использованием примеров кода, пожалуйста, отправьте электронное письмо по адресу bookquestions@oreilly.com.

В общем случае все примеры кода из книги вы можете использовать в своих программах и в документации. Вам не нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить существенные части программного кода. Если вы разрабатываете программу и используете в ней несколько фрагментов кода из книги, вам не нужно обращаться за разрешением. Но для продажи или распространения примеров из книги вам потребуется разрешение от издательства O'Reilly. Вы можете отвечать на вопросы, цитируя данную книгу или примеры из нее, но для включения существенных объемов

программного кода из книги в документацию вашего продукта потребуется разрешение.

Мы рекомендуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN.

За получением разрешения на использование значительных объемов программного кода из книги обращайтесь по адресу permissions@oreilly.com.

Благодарности

Мы хотели бы поблагодарить множество людей, вложивших свой труд в Apache Kafka и ее экосистему. Без их труда этой книги не существовало бы. Особая благодарность Джею Крепсу (Jay Kreps), Ние Нархид (Neha Narkhede) и Чжану Рао (Jun Rao), а также их коллегам и руководству в компании LinkedIn за участие в создании Kafka и передаче ее в фонд программного обеспечения Apache (Apache Software Foundation).

Множество людей прислали свои замечания по черновикам книги, и мы ценим их знания и затраченное ими время. Это Апурва Мехта (Aparva Mehta), Арсений Ташоян (Arseniy Tashoyan), Дилан Скотт (Dylan Scott), Ивен Чеслак-Постава (Ewen Cheslack-Postava), Грант Хенке (Grant Henke), Ишмаэль Джума (Ismael Juma), Джеймс Чен (James Cheng), Джейсон Густафсон (Jason Gustafson), Джеф Холомен (Jeff Holoman), Джоэль Коши (Joel Koshy), Джонатан Сейдмэн (Jonathan Seidman), Чжан Рао (Jun Rao), Матиас Сакс (Matthias Sax), Майкл Нолл (Michael Noll), Паоло Кастанья (Paolo Castagna) и Джесси Андерсон (Jesse Anderson). Мы также хотели бы поблагодарить множество читателей, оставивших комментарии и отзывы на сайте обратной связи черновых версий книги.

Многие из рецензентов очень нам помогли и значительно повысили качество издания, так что вина за все оставшиеся ошибки лежит исключительно на нас.

Мы хотели бы поблагодарить редактора первого издания О'Reilly Шеннон Катт (Shannon Cutt) за терпение, поддержку и гораздо лучший, по сравнению с нашим, контроль ситуации. Редакторы второго издания, Джесс Хаберман (Jess Haberman) и Гэри О'Брайен (Gary O'Brien), здорово поддерживали нас, преодолевая глобальные трудности. Работа с издательством О'Reilly — замечательный опыт для любого автора: их поддержка, начиная с утилит и заканчивая автограф-сессиями, беспрецедентна. Мы благодарны всем, кто сделал выпуск этой книги возможным, и ценим то, что они захотели с нами работать.

Мы также благодарны своему руководству и коллегам за помощь и содействие, которые получили при написании этой книги.

Гвен также хотела бы поблагодарить своего супруга, Омера Шапиру (Omer Sharira), за терпение и поддержку на протяжении многих месяцев, потраченных на написание еще одной книги, кошек Люка и Лею — они такие милые, а также отца, Лиора Шапиру (Lior Sharira), за то что научил ее всегда хвататься за возможности, какими бы пугающими они ни были.

Тодд ничего не сделал бы без своей жены Марси и дочерей Беллы и Кайли, постоянно ободрявших его. Их поддержка, не ослабевавшая, несмотря на то, что написание книги потребовало дополнительного времени, и долгие часы пробежек для освежения мыслей помогали ему работать.

Раджини хотела бы поблагодарить своего мужа Манджунатха и сына Таруна за их неизменную поддержку и ободрение, за то, что они проводили выходные дни, просматривая начальные черновики, и всегда были рядом с ней.

Крит выражает свою любовь и благодарность жене Сесилии и детям Лукасу и Лизабет. Их любовь и поддержка превращают каждый день в радость, и без них он не смог бы заниматься своими увлечениями. Он также хочет поблагодарить свою маму Синди Петти, за то что она привила Криту желание всегда быть лучшей версией самого себя.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

ГЛАВА 1

Знакомьтесь: Kafka

Деятельность любого предприятия основана на данных. Мы получаем информацию, анализируем ее, выполняем над ней какие-либо действия и создаем новые данные в качестве результатов. Все приложения создают данные — сообщения журнала, показатели, информацию об операциях пользователей, исходящие сообщения или что-то еще. Каждый байт данных что-нибудь да значит — что-нибудь, определяющее дальнейшие действия. А чтобы понять, что именно, нам нужно переместить данные из места создания туда, где их можно проанализировать. Это мы каждый день наблюдаем на таких сайтах, как Amazon, где щелчки кнопкой мыши на интересующих нас товарах превращаются в предлагаемые нам же чуть позже рекомендации.

От скорости этого процесса зависят адаптивность и оперативность нашего предприятия. Чем меньше усилий мы тратим на перемещение данных, тем больше можем уделить внимания основной деятельности. Именно поэтому конвейер — ключевой компонент на ориентированном на работу с данными предприятии. Способ перемещения данных оказывается практически так же важен, как и сами данные.

Первопричиной всякого спора ученых является нехватка данных. Постепенно мы приходим к согласию относительно того, какие данные нужны, получаем эти данные, и данные решают проблему. Или я оказываюсь прав, или вы, или мы оба ошибаемся. И можно двигаться дальше.

Нил Деграасс Тайсон (Neil deGrasse Tyson)

Обмен сообщениями по типу «публикация/подписка»

Прежде чем перейти к обсуждению нюансов Apache Kafka, важно понять концепцию обмена сообщениями по типу «публикация/подписка» и причину, по которой она является критически важным компонентом приложений, управляемых данными. *Обмен сообщениями по типу «публикация/подписка»* (publish/subscribe (pub/sub) messaging) — паттерн проектирования, отличающийся тем,

что отправитель (издатель) элемента данных (сообщения) не направляет его конкретному потребителю. Вместо этого он каким-то образом классифицирует сообщения, а потребитель (подписчик) подписывается на определенные их классы. В системы типа «публикация/подписка» для упрощения этих действий часто включают брокер — центральный пункт публикации сообщений.

С чего все начинается

Множество сценариев использования публикации/подписки начинается одинаково — с простой очереди сообщений или канала обмена ими между процессами. Например, вы создали приложение, которому необходимо отправлять куда-либо мониторинговую информацию, для чего приходится открывать прямое соединение между вашим приложением и приложением, отображающим показатели на инструментальной панели, и передавать последние через это соединение (рис. 1.1).

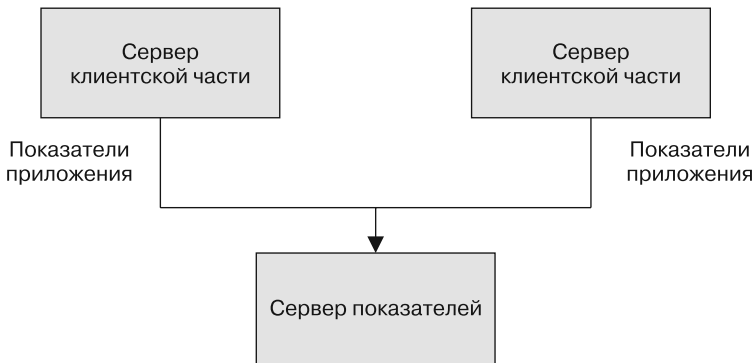


Рис. 1.1. Единый непосредственный издатель показателей

Это простое решение простой задачи, удобное для начала мониторинга. Но вскоре вам захочется анализировать показатели за больший период времени, а на инструментальной панели это не слишком удобно. Вы создадите новый сервис для получения показателей, их хранения и анализа. Для поддержки этого измените свое приложение так, чтобы оно могло записывать их в обе системы. К этому времени у вас появятся еще три генерирующих показатели приложения, каждое из которых будет точно так же подключаться к этим двум сервисам. Один из коллег предложит идею активных опросов сервисов для оповещения, так что вы добавите к каждому из приложений сервер, выдающий показатели по запросу. Вскоре у вас появятся дополнительные приложения, использующие эти серверы для получения отдельных показателей в различных целях. Архитектура станет напоминать рис. 1.2, возможно, соединениями, которые еще труднее отслеживать.

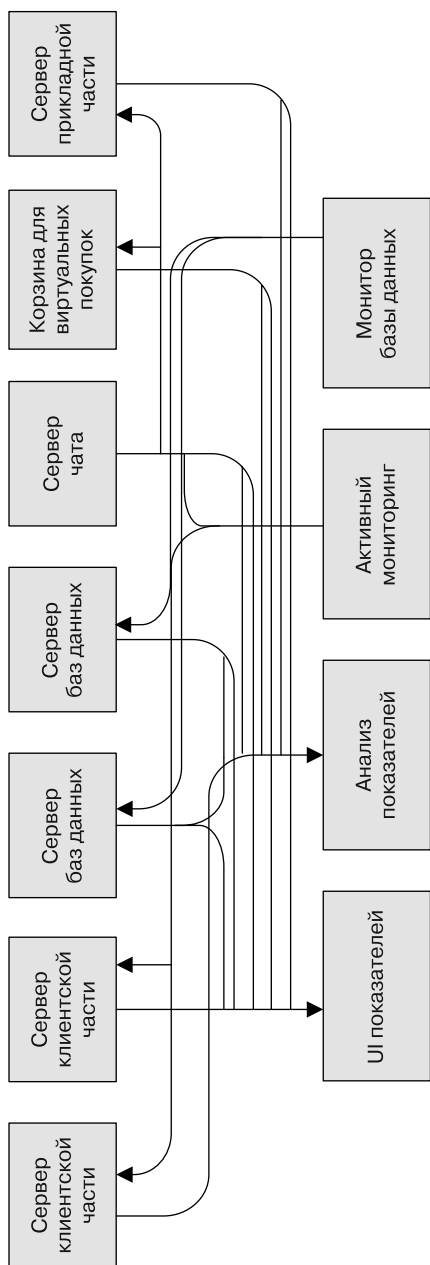


Рис. 1.2. Множество издателей показателей, использующих прямые соединения

Некоторая недоработка тут очевидна, так что вы решаете ее исправить. Создаете единое приложение, получающее показатели от всех имеющихся приложений и включающее сервер, — у него станут их запрашивать все системы, которым нужны эти показатели. Благодаря этому сложность архитектуры уменьшается (рис. 1.3). Поздравляем, вы создали систему обмена сообщениями по типу «публикация/подписка»!

Отдельные системы организации очередей

В то время как вы боролись с показателями, один из ваших коллег аналогичным образом трудился над сообщениями журнала. А еще один работал над отслеживанием действий пользователей на веб-сайте клиентской части и передачей этой информации разработчикам, занимающимся машинным обучением, параллельно с формированием отчетов для начальства. Вы все шли одним и тем же путем, создавая системы, разделяющие издателей информации и подписчиков на нее. Инфраструктура с тремя отдельными системами публикации/подписки показана на рис. 1.4.

Использовать ее намного лучше, чем прямые соединения (см. рис. 1.2), но возникает существенное дублирование. Компании приходится сопровождать несколько систем организации очередей, в каждой из которых имеются собственные ошибки и ограничения. А между тем вы знаете, что скоро появятся новые сценарии обмена сообщениями. Необходима единая централизованная система, поддерживающая публикацию обобщенных типов данных, которая могла бы развиваться по мере расширения вашего бизнеса.

Открываем для себя систему Kafka

Apache Kafka была разработана в качестве системы обмена сообщениями по принципу «публикация/подписка», предназначенной для решения описанной задачи. Ее часто называют распределенным журналом фиксации транзакций, а в последнее время — распределенной платформой потоковой обработки. Журнал фиксации файловой системы или базы данных предназначены для обеспечения долговременного хранения всех транзакций таким образом, чтобы можно было их воспроизвести с целью восстановления согласованного состояния системы. Аналогично данные в Kafka хранятся долго, упорядоченно, и их можно читать когда угодно. Кроме того, они могут распределяться по системе в качестве меры дополнительной защиты от сбоев, равно как и ради повышения производительности.

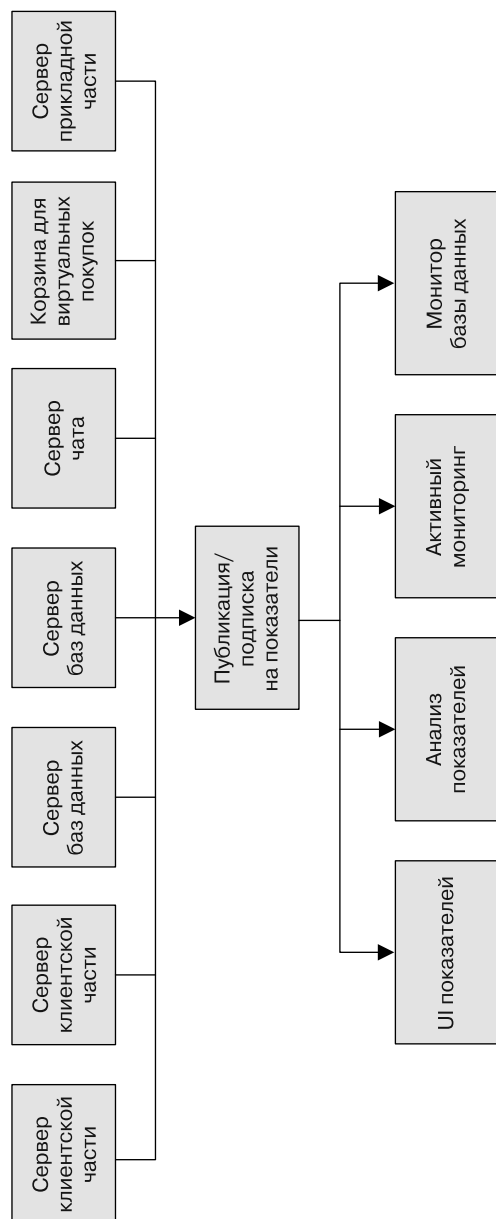
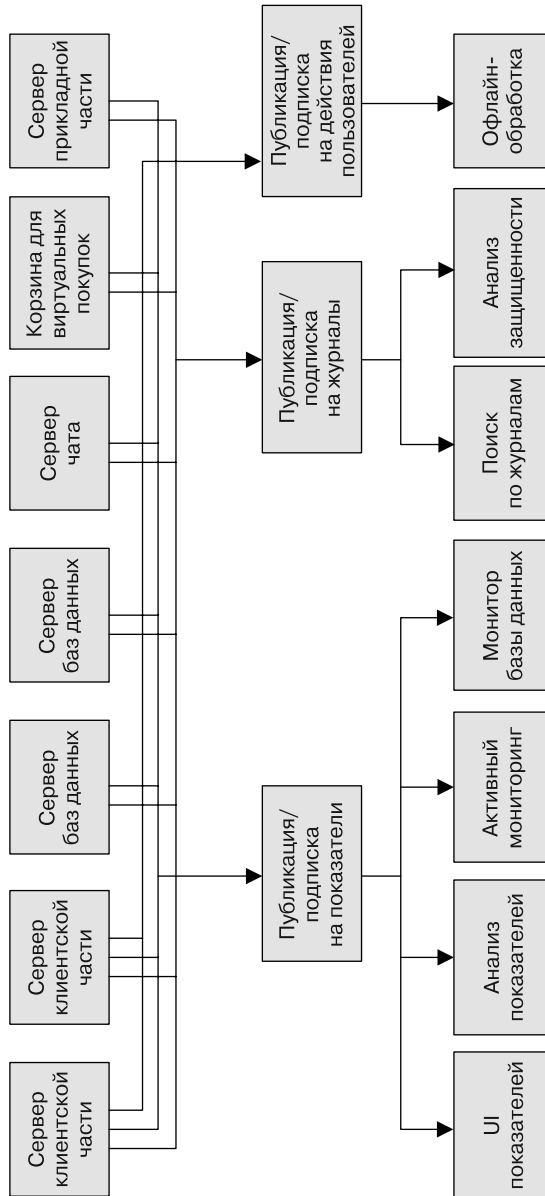


Рис. 1.3. Система публикации/подписки на показатели

**Рис. 1.4.** Несколько систем публикации/подписки

Сообщения и пакеты

Используемая в Kafka единица данных называется *сообщением* (message). Если ранее вы работали с базами данных, то можете рассматривать сообщение как аналог *строки* (row) или *записи* (record). С точки зрения Kafka сообщение представляет собой просто массив байтов, так что для нее содержащиеся в нем данные не имеют формата или какого-либо смысла. В сообщении может быть дополнительный фрагмент метаданных, называемый *ключом* (key). Он тоже представляет собой массив байтов и, как и сообщение, не несет для Kafka никакого смысла. Ключи используются при необходимости лучше управлять записью сообщений в разделы. Простейшая схема такова: генерация единообразного хеш-значения ключа с последующим выбором номера раздела для сообщения путем деления этого значения по модулю общего числа разделов в топике. Это гарантирует попадание сообщений с одним ключом в один раздел (при условии, что количество разделов не изменится).

Для большей эффективности сообщения в Kafka записываются пакетами. *Пакет* (batch) представляет собой просто набор сообщений, относящихся к одному топике и разделу. Передача каждого сообщения туда и обратно по сети привела бы к существенному перерасходу ресурсов, а объединение сообщений в пакет эту проблему уменьшает. Конечно, необходимо соблюдать баланс между временем задержки и пропускной способностью: чем больше пакеты, тем больше сообщений можно обрабатывать за единицу времени, но тем дольше распространяется отдельное сообщение. Пакеты обычно сжимаются, что позволяет передавать и хранить данные более эффективно за счет некоторого расхода вычислительных ресурсов. Мы обсудим ключи и пакеты более подробно в главе 3.

Схемы

Хотя сообщения для Kafka — всего лишь непрозрачные массивы байтов, рекомендуется накладывать на содержимое сообщений дополнительную структуру — схему, которая позволяла бы с легкостью их разбирать. Существует много вариантов задания *схемы* сообщений в зависимости от потребностей конкретного приложения. Упрощенные системы, например нотация объектов JavaScript (JavaScript Object Notation, JSON) и расширяемый язык разметки (Extensible Markup Language, XML), просты в использовании, их удобно читать человеку. Однако им не хватает таких свойств, как надежная работа с типами и совместимость разных версий схемы. Многим разработчикам Kafka нравится Apache Avro — фреймворк сериализации, изначально предназначенный для Hadoop. Avro обеспечивает компактный формат сериализации, схемы, отделенные от содержимого сообщений и не требующие генерации кода при изменении,

а также сильную типизацию данных и эволюцию схемы с прямой и обратной совместимостью.

Для Kafka важен единообразный формат данных, ведь он дает возможность разъединять код записи и чтения сообщений. При тесном сцеплении этих задач приходится модифицировать приложения-подписчики, чтобы они могли работать не только со старым, но и с новым форматом данных. Только после этого можно будет использовать новый формат в публикующих сообщения приложениях. Благодаря применению четко заданных схем и хранению их в общем репозитории сообщения в Kafka можно читать, не координируя действия. Мы рассмотрим схемы и сериализацию подробнее в главе 3.

Топики и разделы

Сообщения в Kafka распределяются по *топикам* (topics). Ближайшая аналогия — таблица базы данных или каталог файловой системы. Топики, в свою очередь, разбиваются на *разделы* (partitions). Если вернуться к описанию журнала фиксации, то раздел представляет собой отдельный журнал. Сообщения записываются в него путем добавления в конец, а читаются по порядку от начала к концу. Заметим: поскольку топик обычно состоит из нескольких разделов, не гарантируется упорядоченность сообщений в пределах всего топика — лишь в пределах отдельного раздела. На рис. 1.5 показан топик с четырьмя разделами, в конец каждого из которых добавляются сообщения. Благодаря разделам Kafka обеспечивает также избыточность и масштабируемость. Любой из разделов можно разместить на отдельном сервере, что означает возможность горизонтального масштабирования системы на несколько серверов для достижения производительности, далеко выходящей за пределы возможностей одного сервера. Кроме того, разделы могут быть реплицированы, так что на разных серверах будет храниться копия одного и того же раздела на случай выхода из строя одного сервера.



Рис. 1.5. Представление топика с несколькими разделами

При обсуждении данных, находящихся в таких системах, как Kafka, часто используется термин «*поток данных*» (stream). Чаще всего он рассматривается отдельным топиком независимо от числа разделов, представляющих собой единый поток данных, перемещающихся от производителей к потребителям. Чаще всего сообщения рассматривают подобным образом при обсуждении потоковой обработки, при которой фреймворки, в частности Kafka Streams, Apache Samza и Storm, работают с сообщениями в режиме реального времени. Принцип их действия подобен принципу работы офлайн-фреймворков, в частности Hadoop, предназначенных для операций с большими данными. Обзор темы потоковой обработки приведен в главе 14.

Производители и потребители

Пользователи Kafka делятся на два основных типа: производители и потребители. Существуют также продвинутые клиентские API — API Kafka Connect для интеграции данных и Kafka Streams для потоковой обработки. Продвинутые клиенты применяют производители и потребители в качестве строительных блоков, предоставляя на их основе функциональность более высокого уровня.

Производители (producers) генерируют новые сообщения. В других системах обмена сообщениями по типу «публикация/подписка» их называют *издателями* (publishers) или *авторами* (writers). Производители сообщений создают их для конкретного топика. По умолчанию производитель будет равномерно поставлять сообщения во все разделы топика. В некоторых случаях он направляет сообщение в конкретный раздел. Для этого обычно служат ключ сообщения и объект `Partitioner`, генерирующий хеш ключа и устанавливающий его соответствие с конкретным разделом. Это гарантирует запись всех сообщений с одинаковым ключом в один и тот же раздел. Производитель может также воспользоваться собственным объектом `Partitioner` со своими бизнес-правилами распределения сообщений по разделам. Более подробно поговорим о производителях в главе 3.

Потребители (consumers) читают сообщения. В других системах обмена сообщениями по типу «публикация/подписка» их называют *подписчиками* (subscribers) или *читателями* (readers). Потребитель подписывается на один топик или более и читает сообщения в порядке их создания в каждом разделе. Он отслеживает, какие сообщения он уже прочитал, запоминая смещение сообщений. *Смещение* (offset) — непрерывно возрастающее целочисленное значение — еще один элемент метаданных, который Kafka добавляет в каждое сообщение при его создании. Смещения сообщений в конкретном разделе не повторяются, а следующее сообщение имеет большее смещение (хотя и не обязательно монотонно большее).

Благодаря сохранению следующего возможного смещения для каждого раздела обычно в хранилище самой Kafka потребитель может приостанавливать и возобновлять свою работу, не забывая, в каком месте он читал.

Потребители работают в составе *групп потребителей* (consumer groups) — одного или нескольких потребителей, объединившихся для обработки топика. Организация в группы гарантирует чтение каждого раздела только одним членом группы. На рис. 1.6 представлены три потребителя, объединенные в одну группу для обработки топика. Два потребителя обрабатывают по одному разделу, а третий — два. Соответствие потребителя разделу иногда называют *владением* (ownership) раздела потребителем.

Таким образом, потребители получают возможность горизонтального масштабирования для чтения топиков с большим количеством сообщений. Кроме того, в случае сбоя отдельного потребителя оставшиеся члены группы переназначат потребляемые разделы так, чтобы взять на себя его задачу. Потребители и группы потребителей подробнее описываются в главе 4.



Рис. 1.6. Чтение топика группой потребителей

Брокеры и кластеры

Отдельный сервер Kafka называется *брокером* (broker). Он получает сообщения от производителей, присваивает им смещения и записывает сообщения в дисковое хранилище. Он также обслуживает потребители и отвечает на запросы выборки из разделов, возвращая опубликованные сообщения. В зависимости от конкретного аппаратного обеспечения и его производительности отдельный брокер может с легкостью обрабатывать тысячи разделов и миллионы сообщений в секунду.

Брокеры Kafka предназначены для работы в составе *кластера* (cluster). Один из брокеров кластера функционирует в качестве *контроллера* (cluster controller). Контроллер кластера выбирается автоматически из числа работающих членов кластера. Он отвечает за административные операции, включая распределение разделов по брокерам и мониторинг отказов последних. Каждый раздел принадлежит одному из брокеров кластера, который называется *ведущим* (leader). Реплицированный раздел, как видно на рис. 1.7, можно назначить дополнительным брокерам, которые называются *последователями* (followers) раздела. Репликация обеспечивает избыточность сообщений в разделе, так что в случае сбоя ведущего один из последователей сможет занять его место. Все производители должны соединяться с ведущим для публикации сообщений, но потребители могут получать сообщения либо от ведущего, либо от одного из последователей. Кластерные операции, включая репликацию разделов, подробно рассмотрены в главе 7.

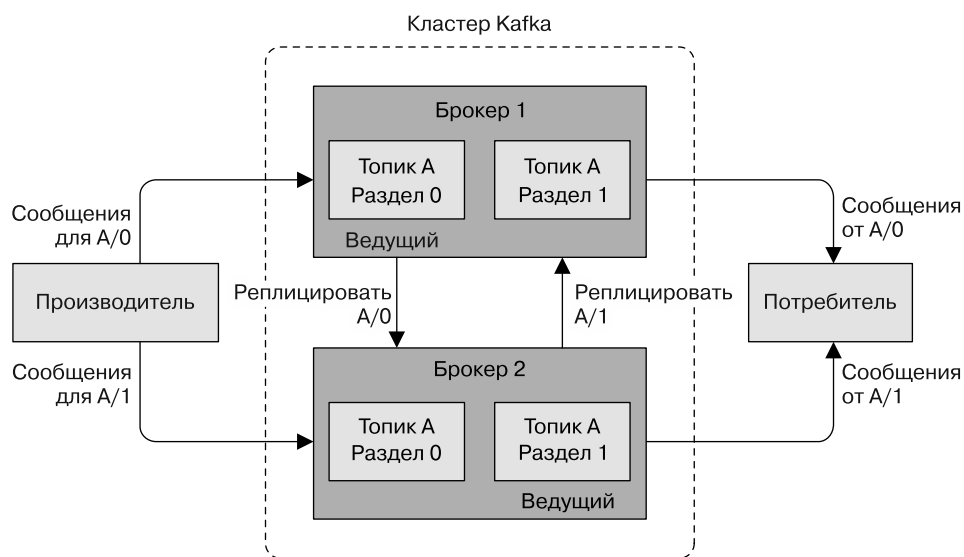


Рис. 1.7. Репликация разделов в кластере

Ключевая возможность Apache Kafka — *сохранение информации* (retention) в течение длительного времени. В настройки брокеров Kafka включается длительность хранения топиков по умолчанию — или в течение определенного промежутка времени (например, семь дней), или до достижения разделом определенного размера в байтах (например, 1 Гбайт). Превысившие эти пределы сообщения становятся недействительными и удаляются. Таким образом, на-

стройки сохранения определяют минимальное количество доступной в каждый момент информации. Можно задавать настройки сохранения и для отдельных топиков, чтобы сообщения хранились только до тех пор, пока они нужны. Например, топик для отслеживания действий пользователей можно хранить несколько дней, в то время как параметры приложений — лишь несколько часов. Можно также настроить для топиков вариант хранения *сжатых журналов* (log compacted). При этом Kafka будет хранить лишь последнее сообщение с конкретным ключом. Это может пригодиться для таких данных, как журналы изменений, в случае, когда нас интересует только последнее изменение.

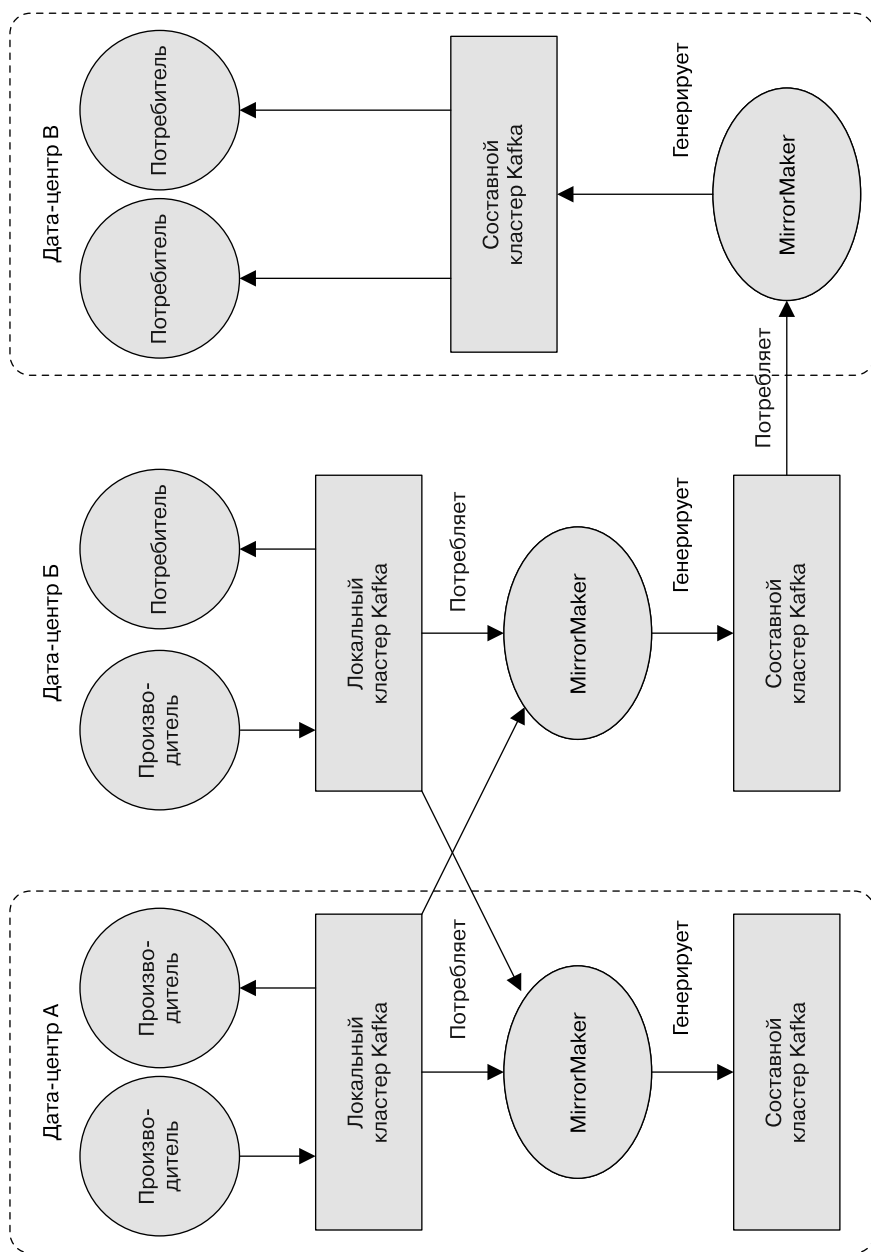
Несколько кластеров

По мере роста развертываемых систем Kafka может оказаться удобным наличие нескольких кластеров. Вот несколько причин этого.

- Разделение типов данных.
- Изоляция по требованиям безопасности.
- Несколько центров обработки данных (ЦОД) (восстановление в случае катаклизмов).

В ходе работы, в частности, с несколькими ЦОД часто выдвигается требование копирования сообщений между ними. Таким образом онлайн-приложения могут повсеместно получить доступ к информации о действиях пользователей. Например, если пользователь корректирует общедоступную информацию в своем профиле, изменения должны быть видны вне зависимости от ЦОД, в котором отображаются результаты поиска. Или данные мониторинга могут собираться с многих сайтов в одно место, где расположены системы анализа и оповещения. Механизмы репликации в кластерах Kafka предназначены только для работы внутри одного кластера, репликация между несколькими кластерами не осуществляется.

Проект Kafka включает утилиту *MirrorMaker* для репликации данных на другие кластеры. По существу, это просто потребитель и производитель Kafka, связанные воедино очередью. Данная утилита получает сообщения из одного кластера Kafka и публикует их в другом. На рис. 1.8 демонстрируется пример использующей MirrorMaker архитектуры, в которой сообщения из двух локальных кластеров агрегируются в составной кластер, который затем копируется в другие ЦОД. Пускай простота этого приложения не производит у вас ложного впечатления о его возможностях создавать сложные конвейеры данных, которые мы подробнее рассмотрим в главе 9.

**Рис. 1.8.** Архитектура с несколькими ЦОД

Почему Kafka?

Существует множество систем публикации сообщений и подписки на них. Чем же Apache Kafka лучше других?

Несколько производителей

Kafka может без каких-либо проблем работать с несколькими производителями вне зависимости от того, используют они один топик или несколько. Это делает платформу идеальной для агрегирования данных из множества клиентских систем и обеспечения их согласованности. Например, у сайта, выдающего пользователям контент посредством множества микросервисов, может существовать отдельный топик для сведений о просмотре страниц, в который все сервисы записывают данные в едином формате. Приложения-потребители затем будут получать единый поток данных просмотров страниц для всех приложений сайта, причем им не нужно будет согласовывать получение сообщений из нескольких топиков, по одному для каждого приложения.

Несколько потребителей

Помимо того что Kafka имеет несколько производителей, она спроектирована с учетом возможности для нескольких потребителей читать любой один поток сообщений, не мешая друг другу. Этим она отличается от множества других систем организации очередей, в которых сообщение, полученное одним клиентом, становится недоступным для других. Несколько потребителей Kafka могут работать в составе группы и совместно использовать поток данных, что гарантирует обработку любого конкретного сообщения этой группой лишь один раз.

Сохранение информации на диске

Kafka не только может работать с несколькими потребителями. Долговременное хранение означает, что потребители не обязательно должны работать в режиме реального времени. Сообщения записываются на диск и хранятся там в соответствии с настраиваемыми правилами, которые можно задавать для каждого топика по отдельности. Благодаря этому различные потоки сообщений будут храниться в течение разного времени в зависимости от запросов потребителя. Долговременное хранение также означает, что при отставании потребителя вследствие или медленной обработки, или резкого роста трафика опасности потери данных не возникнет. Еще оно означает возможность обслуживать потребители при коротком отключении приложений от сети, не беспокоясь о резервном копировании или вероятной потере сообщений в производителе.

Потребители можно останавливать, при этом сообщения будут сохраняться в Kafka. Это позволяет потребителям перезапускаться и продолжать обработку сообщений с того места, на котором они остановились, без потери данных.

Масштабируемость

Гибко масштабируемая Kafka позволяет обрабатывать любые объемы данных. Можно начать работу с одного брокера в качестве пробной версии, расширить систему до небольшого кластера из трех брокеров, предназначенного для разработки, а затем перейти к промышленной эксплуатации с большим кластером из десятков или даже сотен брокеров, растущим по мере увеличения объемов данных. При этом можно расширять систему во время работы кластера, что не повлияет на доступность системы в целом. Это означает также, что кластеру из множества брокеров не страшен сбой одного из них — обслуживание клиентов при этом не прервется. Кластеры, которые должны выдерживать одновременные отказы нескольких брокеров, можно настроить, увеличив коэффициенты репликации. Подробнее репликация обсуждается в главе 7.

Высокое быстродействие

Благодаря рассмотренным особенностям Apache Kafka как система обмена сообщениями по типу «публикация/подписка» отличается прекрасной производительностью при высокой нагрузке. Производители, потребители и брокеры можно масштабировать, чтобы иметь возможность легко обрабатывать очень большие потоки сообщений. Причем это можно делать параллельно с обеспечением менее чем секундной задержки сообщений по пути от производителя к потребителю.

Особенности платформы

В основной проект Apache Kafka добавлены некоторые функции потоковой платформы, которые могут значительно упростить для разработчиков выполнение обычных видов работ. Хотя эти функции не являются полноценными платформами, которые обычно включают структурированную среду выполнения, такую как YARN, они представлены в виде API и библиотек, обеспечивающих прочную основу для построения и гибкость в отношении того, где они могут быть запущены. Kafka Connect помогает извлечь данные из исходной системы данных и направить их в Kafka или извлечь данные из Kafka и направить их в исходную систему данных. Kafka Streams предоставляет библиотеку для простой разработки масштабируемых и отказоустойчивых приложений потоковой обработки. Connect описывается в главе 9, а Streams подробно рассматривается в главе 14.

Экосистема данных

В средах, создаваемых нами для обработки данных, задействуется множество приложений. Производители данных представляют собой приложения, которые создают данные или как-либо еще вводят их в систему. Выходные данные имеют форму показателей, отчетов и других результатов обработки. Мы создаем циклы, в которых одни компоненты читают данные из системы, преобразуют их с помощью данных от других производителей, после чего возвращают обратно в инфраструктуру данных для использования. Это происходит со множеством типов данных, у каждого из которых свои особенности в плане содержимого, размера и способа применения.

Apache Kafka — своего рода кровеносная система для экосистемы данных (рис. 1.9). Она обеспечивает перенос сообщений между разными членами инфраструктуры, предлагая единообразный интерфейс для всех клиентов. При наличии системы, предоставляющей схемы сообщений, производители и потребители больше не требуют сильного сцепления или прямого соединения. По мере возникновения и исчезновения бизнес-моделей можно добавлять и удалять компоненты, причем производителей не должно волновать то, какие приложения потребляют данные и сколько их.

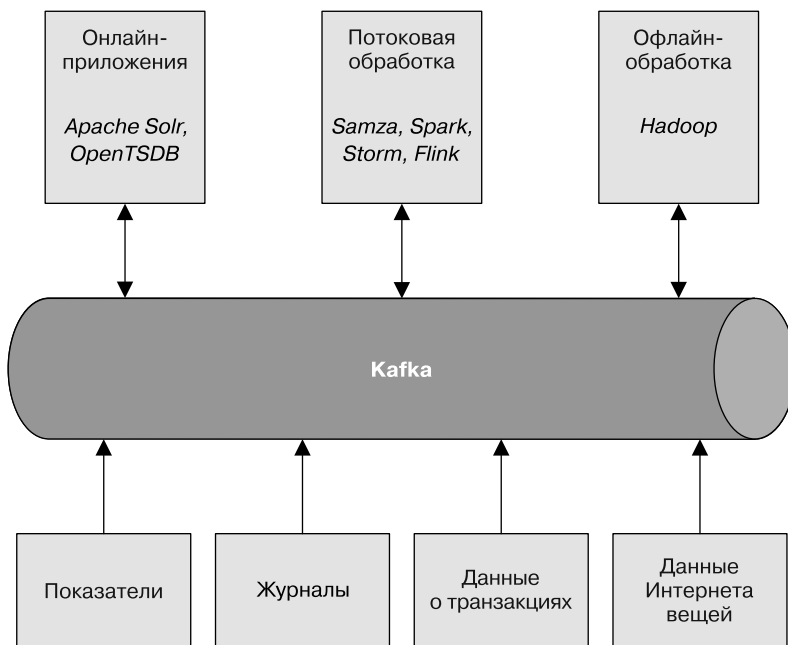


Рис. 1.9. Большая экосистема данных

Сценарии использования

Отслеживание действий пользователей

Первоначальный сценарий использования платформы Kafka, разработанный при ее создании в компании LinkedIn, состоял в отслеживании действий пользователей. Пользователи веб-сайтов взаимодействуют с приложениями клиентской части, генерирующими сообщения о предпринятых пользователями действиях. Это может быть пассивная информация, например, сведения о просмотрах страниц или отслеживание щелчков кнопкой мыши, или информация о более сложных действиях, например о добавлении пользователем данных в свой профиль. Сообщения публикуются в одном или нескольких топиках, потребителями которых становятся приложения в прикладной части. Эти приложения могут генерировать отчеты, служить производителем данных для систем машинного обучения, обновлять результаты поиска или выполнять другие операции, необходимые для повышения удобства применения.

Обмен сообщениями

Kafka задействуется также для обмена сообщениями, когда приложения должны отправлять пользователям уведомления, например сообщения электронной почты. Эти приложения могут создавать сообщения, не беспокоясь об их форматировании или фактической отправке. После этого одно-единственное приложение сможет читать все отправленные сообщения и обрабатывать их единообразно, включая:

- единообразное форматирование сообщений, называемое также декорированием;
- объединение нескольких сообщений в одно уведомление для отправки;
- учет предпочтений пользователя относительно способа получения сообщений.

Использование для этого единого приложения позволяет избежать дублирования функциональности в нескольких приложениях, а также дает возможность выполнять такие операции, как агрегирование, которые в противном случае были бы невозможны.

Показатели и журналирование

Kafka идеально подходит для сбора показателей и журналов приложения и системы. В случае реализации этого сценария особенно ярко проявляется вероятность наличия нескольких приложений, генерирующих однотипные со-

общения. Приложения регулярно публикуют в топиках Kafka показатели, потребителями которых становятся системы мониторинга и оповещения. Их также можно использовать в таких офлайн-системах, как Hadoop, для долгосрочного анализа, к примеру, прогноза роста. Журнальные сообщения можно публиковать аналогичным образом с маршрутизацией их на выделенные системы поиска по журналам, например, Elasticsearch или системы анализа безопасности. Еще одно достоинство Kafka заключается в том, что при необходимости изменения целевой системы (допустим, при наступлении времени обновления системы хранения журналов) не нужно менять приложения клиентской части или способ агрегирования.

Журнал фиксации

Поскольку в основе Kafka лежит понятие журнала фиксации, можно с легкостью публиковать в ней изменения базы данных и организовывать мониторинг приложениями этого потока данных с целью получения изменений сразу же после их выполнения. Поток журнала изменений можно использовать и для репликации изменений базы данных в удаленной системе или объединения изменений из нескольких систем в единое представление базы данных. Долгосрочное сохранение оказывается удобным для создания буфера для журнала изменений, поскольку позволяет повторное выполнение в случае сбоя приложений-потребителей. В качестве альтернативы можно воспользоваться топиками со сжатыми журналами для более длительного хранения за счет хранения лишь одного изменения для каждого ключа.

Потоковая обработка

Еще одна область потенциального применения Kafka — потоковая обработка. Хотя практически любой вид использования платформы можно рассматривать как потоковую обработку, этот термин обычно относится к приложениям с такой же функциональностью, как отображение/свертка в Hadoop. Hadoop обычно работает с агрегированием данных на длительном интервале времени — несколько часов или дней. Потоковая же обработка оперирует данными в режиме реального времени со скоростью генерации сообщений. Поточковые фреймворки позволяют пользователям писать маленькие приложения для работы с сообщениями Kafka, выполняя такие задачи, как расчет показателей, секционирование (разбиение на разделы) сообщений для повышения эффективности обработки другими приложениями и преобразование сообщений с использованием данных из нескольких производителей. Мы рассмотрим потоковую обработку в главе 14.

История создания Kafka

Платформа Kafka была создана для решения задачи организации конвейеров данных в компании LinkedIn. Она была нацелена на обеспечение высокопроизводительной системы обмена сообщениями, способной работать с множеством типов данных и выдавать в режиме реального времени очищенную и структурированную информацию о действиях пользователей и системных показателях.

Данные — истинный движитель всех наших начинаний.

*Джефф Вейнер (Jeff Weiner), бывший
генеральный директор LinkedIn*

Проблема LinkedIn

Как и в описанном в начале этой главы примере, в LinkedIn была система для сбора показателей — как системных, так и относящихся к приложениям, в которой применялись пользовательские средства сбора данных и утилиты с открытым исходным кодом для хранения и внутреннего представления данных. Помимо возможности фиксации обычных показателей, например коэффициента загрузки CPU и быстродействия приложения, в ней была продвинутая возможность отслеживания запросов, задействовавшая систему мониторинга и позволявшая анализировать прохождение запроса пользователя по внутренним приложениям. У системы мониторинга, однако, имелось немало недостатков, в частности сбор показателей на основе опросов, большие промежутки между значениями и то, что владельцам приложений невозможно было управлять своими показателями. Эта система была слабо автоматизированной, требовала вмешательства операторов для решения большинства простых задач, была неоднородной (одни и те же показатели по-разному назывались в разных подсистемах).

В то же время в LinkedIn существовала система, предназначенная для отслеживания информации о действиях пользователей. Серверы клиентской части периодически подключались к HTTP-сервису для публикации в нем пакетов сообщений (в формате XML). Эти пакеты затем передавались на платформы для автономной обработки данных, в ходе которой производились синтаксический разбор и объединение файлов. Эта система тоже имела немало недостатков. Форматирование XML было несогласованным, а синтаксический разбор — дорогостоящим в вычислительном отношении. Смена типа отслеживаемых действий пользователя требовала значительной слаженной работы клиентских частей и офлайн-обработки. К тому же в системе постоянно происходили сбои из-за

изменения схем. Отслеживание было основано на передаче пакетов каждый час, так что применять его в режиме реального времени было невозможно.

Системы мониторинга и отслеживания не могли использовать один и тот же сервис прикладной части. Сервис мониторинга был слишком неуклюжим, формат данных не подходил для отслеживания действий, а модель опросов для мониторинга была несовместима с моделью проталкивания для отслеживания. В то же время сервис отслеживания был недостаточно стабильным для того, чтобы применять его для показателей, а модель пакетной обработки не подходила для мониторинга и оповещения в режиме реального времени. Однако у данных мониторинга и отслеживания было много общих черт, а выявление взаимосвязей этой информации (например, влияния конкретных типов действий пользователя на производительность приложения) было крайне желательным. Уменьшение частоты конкретных видов действий пользователя могло указывать на проблемы с обслуживающим их приложением, но часовая задержка обработки пакетов с действиями пользователей означала, что реакция на подобные проблемы слишком медленная.

Прежде всего были тщательно изучены уже существующие готовые решения с открытым исходным кодом с целью нахождения новой системы, которая бы обеспечивала доступ к данным в режиме реального времени и масштабировалась настолько, чтобы справиться с требуемым объемом потока сообщений. Были созданы экспериментальные системы на основе брокера сообщений ActiveMQ, но на тот момент он не был способен обработать такой объем сообщений. К тому же, когда это решение работало так, как требовалось LinkedIn, оно было нестабильным, в ActiveMQ обнаружилось множество изъянов, приводивших к приостановке брокеров. В результате таких остановок возникали заторы в соединениях с клиентами и ограничивались возможности приложений по выдаче результатов запросов пользователям. Было принято решение перейти на свою инфраструктуру конвейеров данных.

Рождение Kafka

Команду разработчиков в LinkedIn возглавлял Джей Крепс (Jay Kreps), ведущий разработчик, ранее отвечавший за создание и выпуск распределенной системы хранения данных типа «ключ — значение» Voldemort с открытым исходным кодом. Первоначально в команду входили также Ния Нархид (Neha Narkhede), а позднее Чжан Рао (Jun Rao). Вместе они решили создать систему обмена сообщениями, которая отвечала бы требованиям как к мониторингу, так и к отслеживанию и которую в дальнейшем можно было бы масштабировать. Основные цели:

- разъединить производители и потребители с помощью модели проталкивания/извлечения;

- обеспечить сохраняемость сообщений в системе обмена сообщениями, чтобы можно было работать с несколькими потребителями;
- оптимизировать систему для обеспечения высокой пропускной способности по сообщениям;
- обеспечить горизонтальное масштабирование системы по мере роста потоков данных.

В результате была создана система публикации сообщений и подписки на них с типичными для систем обмена сообщениями интерфейсом и слоем хранения, более напоминающим систему агрегирования журналов. В сочетании с использованием Apache Avro для сериализации сообщений Kafka позволяла эффективно обрабатывать как показатели, так и информацию о действиях пользователей — миллиарды сообщений в день. Масштабируемость Kafka сыграла свою роль в том, что объем использования LinkedIn вырос до более чем 7 трлн сообщений и 5 Пбайт потребляемых данных ежедневно (по состоянию на февраль 2020 года).

Открытый исходный код

Kafka была выпущена в виде проекта с открытым исходным кодом на GitHub в конце 2010 года. По мере того как сообщество разработчиков ПО с открытым исходным кодом стало обращать на нее все больше внимания, было предложено (и предложение принято) внести ее в число проектов из инкубатора Apache Software Foundation (это произошло в июле 2011 года). В октябре 2012-го Apache Kafka была переведена из инкубатора и стала полноправным проектом. С этого времени сформировалось постоянное сообщество участников и коммитеров проекта Kafka вне компании LinkedIn, постоянно работавших над ней. В настоящее время Kafka используется в некоторых крупнейших каналах передачи данных в мире, включая Netflix, Uber и многие другие компании.

Широкое распространение Kafka создало здоровую экосистему вокруг основного проекта. В десятках стран мира существуют активные группы встреч, которые обеспечивают обсуждение и поддержку потоковой обработки на местах. Существует также множество проектов с открытым исходным кодом, связанных с Apache Kafka. LinkedIn продолжает поддерживать несколько из них, включая Cruise Control, Kafka Monitor и Burrow. В дополнение к своим коммерческим предложениям Confluent выпустила проекты, включая ksqlDB, реестр схем и прокси-сервер REST, под лицензией сообщества, которая не является строго открытым исходным кодом, поскольку имеет ограничения на использование. Некоторые из наиболее популярных проектов перечислены в приложении Б.

Коммерческое взаимодействие

Осенью 2014 года Джей Крепс, Ния Нархид и Чжан Рао покинули LinkedIn, чтобы основать компанию Confluent, которая занимается разработкой, корпоративной поддержкой и обучением для Apache Kafka. Они также присоединились к другим компаниям, таким как Heroku, для обеспечения облачных сервисов для Kafka. Confluent благодаря партнерству с Google предоставляет управляемые кластеры Kafka на облачной платформе Google Cloud Platform, а также аналогичные сервисы на Amazon Web Services и Azure. Одной из других важных инициатив компании Confluent является организация серии конференций Kafka Summit. Стартовав в 2016 году, они ежегодно проводятся в США и Лондоне, предоставляя множеству членов сообщества возможность собраться вместе и обменяться знаниями об Apache Kafka и связанных с ней проектах.

Название

Часто можно услышать вопрос: почему Kafka получила такое название и означает ли оно что-то конкретное в самом приложении? Джей Крепс рассказал следующее: «Мне показалось, что, раз уж Kafka — система, оптимизированная для записи, имеет смысл воспользоваться именем писателя. В колледже я посещал очень много литературных курсов, и мне нравился Франц Кафка. Кроме того, такое название для проекта с открытым исходным кодом звучит очень круто».

Так что, по сути, никакой особой связи тут нет.

Приступаем к работе с Kafka

Теперь, когда мы все знаем о платформе Kafka и ее истории, можно установить ее и создать собственный конвейер данных. В следующей главе обсудим установку и настройку Kafka. Затронем выбор подходящего для Kafka аппаратного обеспечения и некоторые нюансы, которые стоит учитывать при переходе к промышленной эксплуатации.

ГЛАВА 2

Установка Kafka

Эта глава описывает начало работы с брокером Apache Kafka, включая установку Apache ZooKeeper, применяемого платформой для хранения метаданных брокеров. Здесь также рассматриваются основные параметры конфигурации для развертываний Kafka и некоторые рекомендации по выбору аппаратного обеспечения, подходящего для работы брокеров. Наконец, мы расскажем, как установить несколько брокеров Kafka в виде единого кластера, и обсудим некоторые нюансы ее промышленной эксплуатации.

Настройка среды

Прежде чем начать использовать Kafka, следует настроить среду в соответствии с несколькими предварительными требованиями, чтобы обеспечить ее правильную работу. Следующие разделы помогут вам в этом.

Выбор операционную систему

Apache Kafka представляет собой Java-приложение, которое может работать на множестве операционных систем, в числе которых Windows, macOS, Linux и др., однако рекомендуемой операционной системой для общего использования является Linux. В этой главе мы сосредоточимся на установке Kafka в среде Linux. Linux — рекомендуемая операционная система и для развертывания Kafka общего назначения. Информацию по установке Kafka на Windows и macOS вы найдете в приложении А.

Установить Java

Прежде чем установить ZooKeeper или Kafka, необходимо установить и настроить среду Java. Kafka и ZooKeeper хорошо работают со всеми реализациями Java на базе OpenJDK, включая Oracle JDK. Последние версии Kafka поддер-

живают как Java 8, так и Java 11. Точная установленная версия может быть версией, предоставляемой вашей операционной системой или непосредственно загруженной из Интернета, например с сайта Oracle для версии Oracle (<https://www.oracle.com/java>). Хотя ZooKeeper и Kafka будут работать с Java Runtime Edition, при разработке утилит и приложений рекомендуется использовать полный Java Development Kit (JDK). Стоит установить последнюю выпущенную версию патча для вашей среды Java, так как более старые версии могут иметь уязвимости в системах безопасности. Приведенные далее шаги установки предполагают, что у вас в каталоге `/usr/java/jdk1.11.0.10` установлен JDK версии 11 обновления 10.

Установить ZooKeeper

Apache Kafka использует Apache ZooKeeper для хранения метаданных о кластере Kafka, а также подробностей о клиентах-потребителях (рис. 2.1). ZooKeeper — это централизованный сервис для хранения информации о конфигурации, присвоения имен, обеспечения распределенной синхронизации и предоставления группового обслуживания. В этой книге мы не будем подробно рассказывать о ZooKeeper, а ограничимся объяснениями только того, что необходимо для работы Kafka. Хотя ZooKeeper можно запустить и с помощью сценариев, включенных в дистрибутив Kafka, установка полной версии хранилища ZooKeeper из дистрибутива очень проста.

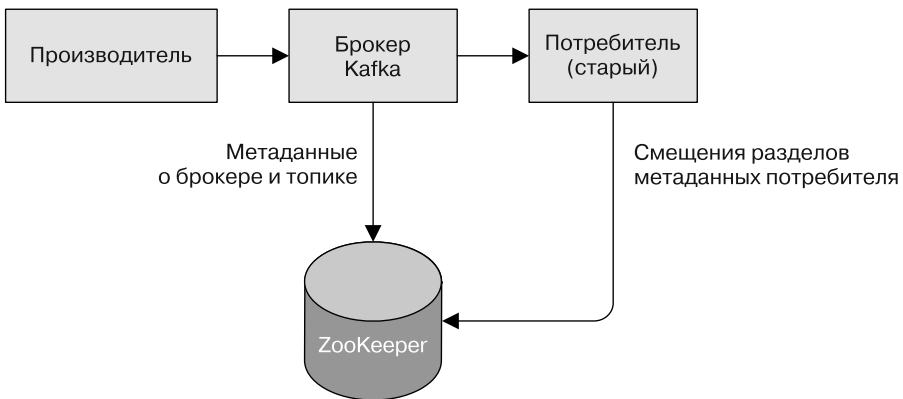


Рис. 2.1. Kafka и ZooKeeper

Kafka была тщательно протестирована со стабильной версией 3.5 хранилища ZooKeeper и регулярно обновляется до последней версии. В этой книге мы будем использовать сервис ZooKeeper 3.5.9, который можно скачать с сайта ZooKeeper по адресу <https://oreil.ly/iMzJR>.

Автономный сервер

ZooKeeper поставляется с базовым примером конфигурационного файла, который будет хорошо работать для большинства случаев использования в `/usr/local/zookeeper/config/zoo_sample.cfg`. Тем не менее в книге мы вручную создадим свой файл с некоторыми базовыми настройками для демонстрационных целей. Следующий пример демонстрирует установку ZooKeeper с базовыми настройками в каталог `/usr/local/zookeeper` с сохранением данных в каталоге `/var/lib/zookeeper`:

```
# tar -xzf zookeeper-3.4.6.tar.gz
# mv zookeeper-3.4.6 /usr/local/zookeeper
# mkdir -p /var/lib/zookeeper
# cat > /usr/local/zookeeper/conf/zoo.cfg << EOF
> tickTime=2000
> dataDir=/var/lib/zookeeper
> clientPort=2181
> EOF
# export JAVA_HOME=/usr/java/jdk1.11.0.10
# /usr/local/zookeeper/bin/zkServer.sh start
JMX enabled by default
Using config: /usr/local/zookeeper/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
#
```

Теперь можете проверить, что ZooKeeper, как полагается, работает в автономном режиме, подключившись к порту клиента и отправив четырехбуквенную команду `srvr`. Она вернет основную информацию о ZooKeeper с запущенного сервера:

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

Ансамбль ZooKeeper

ZooKeeper предназначен для работы в качестве кластера, который называется *ансамблем* (ensemble). Из-за особенностей алгоритма балансировки рекомендует-ся, чтобы ансамбль включал нечетное число серверов, например три, пять и т. д., поскольку ZooKeeper сможет отвечать на запросы, лишь когда функционирует большинство членов ансамбля (*кворум*). Это значит, что ансамбль из трех узлов может работать и при одном неработающем узле. Если в ансамбле пять узлов, таких может быть два.



Выбор размера ансамбля ZooKeeper

Рассмотрим вариант работы ZooKeeper в ансамбле из пяти узлов. Чтобы внести изменения в настройки ансамбля, включая настройки подкачки узлов, необходимо перезагрузить узлы по одному за раз. Если ансамбль не может функционировать при выходе из строя более чем одного узла одновременно, работы по обслуживанию становятся источником дополнительного риска. Кроме того, не рекомендуется запускать одновременно более семи узлов, поскольку производительность начнет страдать вследствие самой природы протокола консенсуса.

Если вы чувствуете, что пять или семь узлов не справляются с нагрузкой из-за слишком большого количества клиентских соединений, рассмотрите возможность добавления дополнительных узлов-наблюдателей для помощи в балансировке трафика только для чтения.

Чтобы настроить работу серверов ZooKeeper в ансамбле, у них должна быть единая конфигурация со списком всех серверов, а у каждого сервера в каталоге данных должен иметься файл `myid` с идентификатором этого сервера. Если хосты в ансамбле носят названия `zoo1.example.com`, `zoo2.example.com` и `zoo3.example.com`, то файл конфигурации может выглядеть приблизительно так:

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

В этой конфигурации `initLimit` представляет собой промежуток времени, на протяжении которого ведомые узлы могут подключаться к ведущему. Значение `syncLimit` ограничивает время, в течение которого происходит

отставание ведомых узлов от ведущего. Оба значения задаются в единицах `tickTime`, то есть `initLimit = 20 × 2000 мс = 40 с`. В конфигурации также перечисляются все серверы ансамбля. Они приводятся в формате `server.X=имя_хоста:одноранговый_порт:ведущий_порт` со следующими параметрами:

- `X` — идентификатор сервера. Обязан быть целым числом, но отсчет может вестись не от нуля и не быть последовательным;
- `имя_хоста` — имя хоста или IP-адрес сервера;
- `одноранговый_порт` — TCP-порт, через который серверы ансамбля взаимодействуют друг с другом;
- `ведущий_порт` — TCP-порт, через который выбирается ведущий узел.

Достаточно, чтобы клиенты могли подключаться к ансамблю через *клиентский_порт*, но участники ансамбля должны иметь возможность обмениваться сообщениями друг с другом по всем трем портам.

Помимо единого файла конфигурации у каждого сервера в каталоге `dataDir` должен быть файл `myid`. Он должен содержать идентификатор сервера, соответствующий приведенному в файле конфигурации. После завершения этих действий можно запустить серверы, и они будут взаимодействовать друг с другом в ансамбле.



Тестирование ансамбля ZooKeeper на одной машине

Можно протестировать и запустить ансамбль ZooKeeper на одной машине, указав все имена хостов в настройках как `localhost` и задав уникальные порты `peerPort` и `leaderPort` для каждого экземпляра. Кроме того, для каждого экземпляра нужно будет создать отдельный файл `zoo.cfg` с уникальными параметрами `dataDir` и `clientPort`. Это может быть полезно только в целях тестирования, но не рекомендуется для производственных систем.

Установка брокера Kafka

После завершения настройки Java и ZooKeeper можно приступить к установке Apache Kafka. Актуальную версию можно скачать с сайта Kafka по адресу <https://oreil.ly/xLopS>. На момент публикации книги это версия 2.8.0, работающая под управлением Scala 2.13.0. Примеры в книге приведены с использованием версии 2.7.0.

В следующем примере установим платформу Kafka в каталог `/usr/local/kafka`, настроив ее для использования запущенного ранее сервера ZooKeeper и сохранения сегментов журнала сообщений в каталоге `/tmp/kafka-logs`:

```
# tar -zxf kafka_2.13-2.7.0.tgz
# mv kafka_2.13-2.7.0 /usr/local/kafka
# mkdir /tmp/kafka-logs
# export JAVA_HOME=/usr/java/jdk1.11.0.10
# /usr/local/kafka/bin/kafka-server-start.sh -daemon
/usr/local/kafka/config/server.properties
#
```

После запуска брокера Kafka можно проверить его функционирование, выполнив с кластером какие-либо простые операции, включающие создание тестового топика, генерацию сообщений и их потребление.

Создание и проверка топика:

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

Генерация сообщений для топика `test` (нажмите Ctrl+C, чтобы остановить производитель в любой момент):

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

Потребление сообщений из топика `test`:

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



Устаревание подключений ZooKeeper в утилитах Kafka CLI

Если вы знакомы со старыми версиями утилит Kafka, то, возможно, привыкли использовать строку подключения `--zookeeper`. Этот вариант устарел почти для всех случаев. Сейчас лучше всего использовать более новую опцию `--bootstrap-server` и подключаться непосредственно к брокеру Kafka. Если работаете в кластере, можете указать `host:port` любого брокера в кластере.

Настройка брокера

Пример конфигурации брокера, поставляемый вместе с дистрибутивом Kafka, вполне подойдет для пробного запуска автономного сервера, но, скорее всего, его будет недостаточно для больших установок. Существует множество параметров конфигурации Kafka, регулирующих все аспекты установки и настройки. В большинстве опций можно оставить значения по умолчанию, поскольку они относятся к нюансам настройки брокера Kafka, не применяемым до тех пор, пока у вас не возникнет конкретный сценарий использования, требующий настройки этих параметров.

Основные параметры брокера

Существует несколько настроек параметров брокера Kafka, которые желательно обдумать при развертывании платформы в любой среде, кроме автономного брокера на отдельном сервере. Эти параметры относятся к основным настройкам брокера, и большинство из них нужно поменять, чтобы брокер мог работать в кластере с другими брокерами.

broker.id

У каждого брокера Kafka должен быть целочисленный идентификатор, задаваемый параметром `broker.id`. По умолчанию это значение равно 0, но может быть любым числом. Важно, чтобы оно было уникальным для каждого брокера в пределах одного кластера Kafka. Выбор числа может быть произвольным, причем при необходимости ради удобства сопровождения его можно переносить с одного брокера на другой. Тем не менее настоятельно рекомендуется, чтобы оно было как-то связано с хостом, тогда более прозрачным окажется соответствие идентификаторов брокеров хостам при сопровождении. Например, если у вас имена хостов содержат уникальные числа — `host1.example.com`, `host2.example.com` и т. д., числа 1 и 2 будут удачным выбором для значений, соответствующих `broker.id`.

Listeners

В старых версиях Kafka использовалась простая конфигурация портов. Ее все еще можно применять в качестве резервной копии для простых конфигураций, но это устаревшая конфигурация. Конфигурационный файл `example` запускает Kafka со слушателем `listener` на TCP-порту 9092. Новая конфигурация слушателей `listeners` представляет собой разделенный запятыми список URI, которые мы прослушиваем, с именами слушателей. Если имя слушателя не является общим протоколом безопасности, то необходимо настроить еще одну конфигурацию `listener.security.protocol.map`. Слушатель `listener` определяется как `<протокол>://<имя_хоста>:<порт>`. Пример конфигурации легального слушате-

ля — настройка `PLAIN TEXT://localhost:9092,SSL://:9091`. Указание имени хоста как `0.0.0.0` приведет к привязке ко всем интерфейсам. Если оставить имя хоста пустым, он будет привязан к интерфейсу по умолчанию. Имейте в виду, что при выборе порта с номером менее 1024 Kafka должна запускаться от имени пользователя `root`. Запуск Kafka от имени `root` не является рекомендуемой конфигурацией.

zookeeper.connect

Путь, который ZooKeeper использует для хранения метаданных брокеров, задается с помощью параметра конфигурации `zookeeper.connect`. В образце конфигурации ZooKeeper работает на порте 2181 на локальной хост-машине, что указывается как `localhost:2181`. Формат этого параметра — разделенный точками с запятой список строк вида `hostname:port/path`, включающий:

- `hostname` — имя хоста или IP-адрес сервера ZooKeeper;
- `port` — номер порта клиента для сервера;
- `/path` — необязательный путь ZooKeeper, используемый в качестве нового корневого (`chroot`) пути кластера Kafka. Если он не задан, применяется корневой путь.

Если заданного пути `chroot` (путь, назначенный в качестве корневого каталога для данного приложения) не существует, он будет создан при запуске брокера.



Зачем выбирать новый корневой путь

Выбор нового корневого пути для кластера Kafka обычно считается хорошей практикой. Это дает возможность использовать ансамбль ZooKeeper совместно с другими приложениями, включая другие кластеры Kafka, без каких-либо конфликтов. Стоит также задать в конфигурации несколько серверов ZooKeeper (частей одного ансамбля). Благодаря этому брокер Kafka сможет подключиться к другому участнику ансамбля ZooKeeper в случае отказа сервера.

log.dirs

Kafka сохраняет все сообщения на жесткий диск, и хранятся эти сегменты журналов в каталогах, задаваемых в настройке `log.dir`. Для нескольких каталогов предпочтительнее применять конфигурацию `log.dirs`. Если это значение не задано, по умолчанию будет установлен `log.dir`. `log.dirs` представляет собой разделенный запятыми список путей в локальной системе. Если задано несколько путей, брокер будет сохранять разделы в них, выбирая наименее используемые, сегменты журналов одного раздела будут сохраняться по одному пути. Отметим, что брокер поместит новый раздел в каталог, в котором в настоящий момент хранится меньше всего разделов, а не используется меньше всего дискового пространства, поэтому равномерное распределение данных по разделам не гарантируется.

num.recovery.threads.per.data.dir

Для обработки сегментов журналов Kafka использует настраиваемый пул потоков выполнения. В настоящий момент он применяется:

- при обычном запуске — для открытия сегментов журналов каждого из разделов;
- запуске после сбоя — для проверки и усечения сегментов журналов каждого из разделов;
- останове — для аккуратного закрытия сегментов журналов.

По умолчанию задействуется только один поток на каждый каталог журналов. Поскольку это происходит только при запуске и останове, имеет смысл использовать большее их количество, чтобы распараллелить операции. При восстановлении после некорректного останова в случае перезапуска брокера с большим числом разделов выгоды от применения такого подхода могут достичь нескольких часов! Помните, что значение этого параметра определяется из расчета на один каталог журналов из числа задаваемых с помощью `log.dirs`. То есть если значение параметра `num.recovery.threads.per.data.dir` равно 8, а в `log.dirs` указаны три пути, то общее число потоков — 24.

auto.create.topics.enable

В соответствии с конфигурацией Kafka по умолчанию брокер должен автоматически создавать топик, когда:

- производитель начинает писать в топик сообщения;
- потребитель начинает читать из топика сообщения;
- любой клиент запрашивает метаданные топика.

Во многих случаях такое поведение может оказаться нежелательным, особенно из-за того, что не существует возможности проверить по протоколу Kafka существование топика, не создав его. Если вы управляете созданием топиков явно, вручную или посредством системы инициализации, то можете установить для параметра `auto.create.topics.enable` значение `false`.

auto.leader.rebalance.enable

Для того чтобы кластер Kafka не стал несбалансированным из-за того, что ведущие реплики всех разделов сосредоточены на одном брокере, можно указать эту конфигурацию, чтобы обеспечить максимальную сбалансированность ведущих реплик. Она включает фоновый поток, который проверяет распределение ведущих реплик разделов через регулярные промежутки времени (этот временной интервал настраивается с помощью параметра `leader.imbalance.check.interval.seconds`).

Если дисбаланс ведущих реплик превышает другой параметр, `leader.imbalance.per.broker.percentage`, то запускается перебалансировка предпочтительных лидеров для разделов.

delete.topic.enable

В зависимости от вашей среды и рекомендаций по хранению данных вы можете захотеть заблокировать кластер, чтобы предотвратить произвольное удаление топиков. Запретить их удаление можно, установив для этого флага значение `false`.

Настройки топиков по умолчанию

Конфигурация сервера Kafka задает множество настроек по умолчанию для создаваемых топиков. Некоторые из этих параметров, включая число разделов и параметры сохранения сообщений, можно задавать для каждого топика отдельно с помощью инструментов администратора (рассматриваются в главе 12). Значения по умолчанию в конфигурации сервера следует устанавливать равными эталонным значениям, подходящим для большинства топиков кластера.



Индивидуальное переопределение значений для каждого топика

В старых версиях Kafka можно было переопределять значения описанных параметров конфигурации брокера отдельно для каждого топика с помощью параметров `log.retention.hours.per.topic`, `log.retention.bytes.per.topic` и `log.segment.bytes.per.topic`. Эти параметры больше не поддерживаются, и переопределять значения необходимо с помощью инструментов администратора.

num.partitions

Параметр `num.partitions` определяет, с каким количеством разделов создается новый топик, главным образом в случае, когда включено автоматическое создание топиков (это поведение по умолчанию). Значение этого параметра по умолчанию — 1. Имейте в виду, что количество разделов для топика можно лишь увеличивать, но не уменьшать. Это значит, что, если для него требуется меньше разделов, чем указано в `num.partitions`, придется аккуратно создать его вручную (это обсуждается в главе 12).

Как говорилось в главе 1, разделы представляют собой способ масштабирования топиков в кластере Kafka, поэтому важно, чтобы их было столько, сколько нужно для уравнивания нагрузки по сообщениям в масштабах всего кластера по мере добавления брокеров. Многие пользователи предпочитают, чтобы число разделов было равно числу брокеров в кластере или кратно ему. Это дает возможность равномерно распределять разделы по брокерам, что обеспечивает равномерное

распределение нагрузки по сообщениям. Например, топик с десятью разделами, работающий в кластере Kafka с десятью хостами с ведущими репликами, сбалансированными между всеми десятью хостами, будет иметь оптимальную пропускную способность. Однако это не обязательное требование, ведь вы можете выравнять нагрузку и другими способами, например наличием нескольких топиков.

КАК ВЫБРАТЬ КОЛИЧЕСТВО РАЗДЕЛОВ

Вот несколько факторов, которые следует учитывать при выборе количества разделов.

- Какой пропускной способности планируется достичь для топика? Например, планируете вы записывать 100 Кбайт/с или 1 Гбайт/с?
- Какая максимальная пропускная способность ожидается при потреблении сообщений из отдельного раздела? Раздел всегда будет полностью потребляться одним потребителем (даже если не используются группы потребителей, потребитель должен прочитать все сообщения в разделе). Если знать, что потребитель записывает данные в базу, не способную обрабатывать более 50 Мбайт/с по каждому записываемому в нее потоку, становится очевидным ограничение в 50 Мбайт/с при потреблении данных из раздела.
- Аналогичным образом можно оценить максимальную пропускную способность из расчета на производитель для одного раздела, но, поскольку быстродействие производителей обычно выше, чем быстродействие потребителей, этот шаг чаще всего можно пропустить.
- При отправке сообщений разделам по ключам добавление новых разделов может оказаться очень непростой задачей, так что желательно рассчитывать пропускную способность, исходя не из текущего объема использования, а из планируемого в будущем.
- Обдумайте число разделов, размещаемых на каждом из брокеров, а также доступные каждому брокеру объем дискового пространства и полосу пропускания сети.
- Старайтесь избегать завышенных оценок, ведь любой раздел расходует оперативную память и другие ресурсы на брокере и увеличивает время обновления метаданных и передачи руководства.
- Будете ли вы выполнять зеркальное копирование данных? Возможно, вам также потребуется учесть пропускную способность своей конфигурации зеркального копирования. Большие разделы могут стать серьезным недостатком во многих конфигурациях зеркального копирования.
- Если вы используете облачные сервисы, есть ли ограничение количества IOPS (операции ввода/вывода в секунду) на ваших виртуальных машинах или дисках? В зависимости от облачного сервиса и конфигурации виртуальных машин могут существовать жесткие ограничения на количество разрешенных IOPS, которые приведут к нарушению квот. Наличие слишком большого количества разделов может иметь побочный эффект увеличения количества IOPS из-за задействованного параллелизма.

С учетом всего этого ясно, что разделов должно быть много, но не слишком. Если у вас есть предварительные оценки целевой пропускной способности для топика и ожидаемой пропускной способности потребителей, можно полу-

читать требуемое число разделов делением целевой пропускной способности на ожидаемую пропускную способность потребителей. Так что, если необходимо читать из топика 1 Гбайт/с и записывать столько же и мы знаем, что каждый потребитель способен обрабатывать лишь 50 Мбайт/с, значит, требуется как минимум 20 разделов. Таким образом, из топика будут читать 20 потребителей, что в сумме даст 1 Гбайт/с.

Если же такой подробной информации у вас нет, то, по нашему опыту, ограничение размеров разделов на диске до 6 Гбайт сохраняемой информации в день часто дает удовлетворительные результаты. Начинать с малого и расширяться по мере необходимости легче, чем начинать слишком масштабно.

default.replication.factor

Если включено автоматическое создание топика, эта конфигурация задает коэффициент репликации для новых топиков. Стратегия репликации может варьироваться в зависимости от желаемой долговечности или доступности кластера и будет подробнее рассмотрена в последующих главах. Далее приведены краткие рекомендации, которые при использовании Kafka в кластере позволят избежать перебоев в работе из-за факторов, не зависящих от ее внутренних возможностей, например сбоев аппаратного оборудования.

Настоятельно рекомендуется устанавливать коэффициент репликации как минимум на 1 больше значения параметра `min.insync.replicas`. Для более отказоустойчивых настроек, если у вас довольно большие кластеры и достаточно оборудования, предпочтительно установить коэффициент репликации на 2 больше значения параметра `min.insync.replicas` (сокращенно RF++). RF++ позволит упростить обслуживание и предотвратить перебои в работе. Причина, лежащая в основе этой рекомендации, заключается в том, чтобы одновременно разрешить одно запланированное отключение в наборе реплик и одно незапланированное отключение. Для типичного кластера это означает наличие минимум трех точных копий каждого раздела. Примером может служить такая ситуация: когда во время развертывания или обновления Kafka или базовой операционной системы произойдет отключение сетевого коммутатора, отказ диска или возникнет другая незапланированная проблема, вы можете быть уверены, что дополнительная реплика все равно будет доступна. Подробнее об этом будет рассказано в главе 7.

log.retention.ms

Чаще всего продолжительность хранения сообщений в Kafka ограничивается по времени. Значение по умолчанию указано в файле конфигурации с помощью параметра `log.retention.hours` и равно 168 часам, или 1 неделе. Однако можно использовать и два других параметра — `log.retention.minutes` и `log.retention.ms`.

Эти три параметра контролируют одну и ту же цель — промежуток времени, по истечении которого сообщения удаляются, но рекомендуется использовать параметр `log.retention.ms`, ведь в случае указания нескольких параметров приоритет будет у наименьшей единицы измерения, так что всегда будет использоваться значение `log.retention.ms`.



Хранение информации в течение заданного промежутка времени и время последнего изменения

Хранение информации в течение заданного промежутка времени реализуется путем анализа времени последнего изменения (`mtime`) каждого из файлов сегментов журналов на диске. При обычных обстоятельствах оно соответствует времени закрытия сегмента журнала и отражает метку даты/времени последнего сообщения в файле. Однако при использовании инструментов администратора для переноса разделов между брокерами это время оказывается неточным и приводит к слишком длительному хранению информации для этих разделов. Более подробно мы обсудим этот вопрос в главе 12, когда будем рассматривать переносы разделов.

`log.retention.bytes`

Еще один способ ограничения срока действия сообщений — на основе общего размера (в байтах) сохраняемых сообщений. Значение задается с помощью параметра `log.retention.bytes` и применяется пораздельно. Это значит, что в случае топика из восьми разделов и равного 1 Гбайт значения `log.retention.bytes` максимальный объем сохраняемых для этого топика данных будет 8 Гбайт. Отметим, что объем сохранения зависит от отдельных разделов, а не от топика. Это значит, что в случае увеличения числа разделов для топика максимальный объем сохраняемых при использовании `log.retention.bytes` данных также возрастет. Если установить значение `-1`, это позволит сохранять данные бесконечно.



Настройка сохранения по размеру и времени

Если задать значения параметров `log.retention.bytes` и `log.retention.ms` (или другого параметра ограничения времени хранения данных), сообщения будут удаляться по достижении любого из этих пределов. Например, если значение `log.retention.ms` равно 86 400 000 (1 день), а `log.retention.bytes` — 1 000 000 000 (1 Гбайт), вполне могут удаляться сообщения младше 1 дня, если общий объем сообщений за день превысил 1 Гбайт. И наоборот, сообщения могут быть удалены через день, даже если общий объем сообщений раздела меньше 1 Гбайт. Для простоты рекомендуется выбирать либо хранение по размеру, либо хранение по времени, а не оба варианта, чтобы избежать неожиданностей и незапланированной потери данных, но для более продвинутых конфигураций можно использовать оба варианта.

log.segment.bytes

Упомянутые настройки сохранения журналов касаются сегментов журналов, а не отдельных сообщений. По мере генерации брокером Kafka сообщения добавляются в конец текущего сегмента журнала соответствующего раздела. По достижении сегментом журнала размера, задаваемого параметром `log.segment.bytes` и равного по умолчанию 1 Гбайт, этот сегмент закрывается и открывается новый. После закрытия сегмент журнала можно выводить из обращения. Чем меньше размер сегментов журнала, тем чаще приходится закрывать файлы и создавать новые, что снижает общую эффективность операций записи на диск.

Подбор размера сегментов журнала важен в случае, когда топики отличаются низкой частотой генерации сообщений. Например, если в топик поступает лишь 100 Мбайт сообщений в день, а для параметра `log.segment.bytes` установлено значение по умолчанию, для заполнения одного сегмента потребуется 10 дней. А поскольку сообщения нельзя объявить недействительными до тех пор, пока сегмент журнала не закрыт, то при значении 604 800 000 (1 неделя) параметра `log.retention.ms` к моменту вывода из обращения закрытого сегмента журнала могут скопиться сообщения за 17 дней. Это происходит потому, что при закрытии сегмента с накопившимися за 10 дней сообщениями его приходится хранить еще 7 дней, прежде чем можно будет вывести из обращения в соответствии с принятыми временными правилами, поскольку сегмент нельзя удалить до того, как окончится срок действия последнего сообщения в нем.



Извлечение смещений по метке даты/времени

Размер сегмента журнала влияет на извлечение смещений по метке даты/времени. При запросе смещений для раздела с конкретной меткой даты/времени Kafka ищет файл сегмента журнала, который был записан в этот момент. Для этого используется время создания и последнего изменения файла: выполняется поиск файла, который был бы создан до указанной метки даты/времени и последний раз менялся после нее. В ответе возвращается смещение начала этого сегмента журнала, являющееся также именем файла.

log.roll.ms

Другой способ управления закрытием сегментов журнала — с помощью параметра `log.roll.ms`, задающего отрезок времени, по истечении которого сегмент журнала закрывается. Как и параметры `log.retention.bytes` и `log.retention.ms`, параметры `log.segment.bytes` и `log.roll.ms` не являются взаимоисключающими. Kafka закрывает сегмент журнала после того, как или истекает промежуток времени, или достигается заданное ограничение по размеру, в зависимости от того, какое из этих событий произойдет первым. По умолчанию значение

параметра `log.roll.ms` не задано, в результате чего закрытие сегментов журналов обуславливается их размером.



Эффективность ввода/вывода на диск при использовании ограничений по времени на сегменты

Задавая ограничения по времени на сегменты журналов, важно учитывать, что произойдет с производительностью операций с жестким диском при одновременном закрытии нескольких сегментов. Это может произойти при наличии большого числа разделов, которые никогда не достигают пределов размера для сегментов журнала, поскольку отсчет времени начинается при запуске брокера и истечет оно для этих разделов небольшого объема тоже одновременно.

`min.insync.replicas`

При настройке кластера на долговечность данных установка параметра `min.insync.replicas` равным 2 гарантирует, что по крайней мере две реплики будут подхвачены и синхронизированы с производителем. Это используется в сочетании с настройкой конфигурации производителя на проверку всех запросов. Это гарантирует, что как минимум две реплики (лидер и еще одна) подтвердят запись, чтобы она была успешной. Так можно предотвратить потерю данных в сценариях, когда лидер подтверждает запись, затем происходит сбой и лидерство передается реплике, у которой нет успешной записи. Без этих настроек долговечности производитель считал бы, что он успешно и полностью произвел запись данных, а сообщение (-я) было бы сброшено и потеряно. Однако настройка на более высокую долговечность имеет побочный эффект в виде меньшей эффективности из-за дополнительных накладных расходов, поэтому кластерам с высокой пропускной способностью, которые могут допускать периодические потери сообщений, не рекомендуется изменять этот параметр с установленного по умолчанию значения 1. Более подробную информацию вы найдете в главе 7.

`message.max.bytes`

Брокер Kafka позволяет с помощью параметра `message.max.bytes` ограничивать максимальный размер генерируемых сообщений. Значение этого параметра по умолчанию 1 000 000 (1 Мбайт). Производитель, который попытается отправить сообщение большего размера, получит от брокера извещение об ошибке, а сообщение принято не будет. Как и в случае всех остальных размеров в байтах, указываемых в настройках брокера, речь идет о размере сжатого сообщения, так что производители могут отправлять сообщения, размер которых в несжатом виде гораздо больше, если их можно сжать до задаваемых параметром `message.max.bytes` пределов.

Увеличение допустимого размера сообщения серьезно влияет на производительность. Большой размер сообщений означает, что потоки брокера, обрабатывающие сетевые соединения и запросы, будут заниматься каждым запросом дольше. Большие сообщения также увеличивают объем записываемых на диск данных, что влияет на пропускную способность ввода/вывода. Другие решения для хранения данных, такие как хранилища больших двоичных объектов и/или многоуровневые хранилища, могут быть еще одним методом решения проблем с записью на большие диски, но они не будут рассматриваться в этой главе.



Согласование настроек размеров сообщений

Размер сообщения, задаваемый на брокере Kafka, должен быть согласован с настройкой `fetch.message.max.bytes` на клиентах потребителей. Если это значение меньше, чем `message.max.bytes`, то потребители не смогут извлекать превышающие данный размер сообщения, вследствие чего потребитель может зависнуть и прекратить дальнейшую обработку. То же самое относится к параметру `replica.fetch.max.bytes` на брокерах при конфигурации кластера.

Выбор аппаратного обеспечения

Выбор подходящего аппаратного обеспечения для брокера Kafka — скорее искусство, чем наука. У самой платформы Kafka нет строгих требований к аппаратному обеспечению, она будет работать без проблем на большинстве систем. Но если говорить о производительности, то на нее могут влиять несколько факторов: емкость и пропускная способность дисков, оперативная память, сеть и CPU. При очень большом масштабировании Kafka могут возникнуть также ограничения на количество разделов, которые может обрабатывать один брокер, из-за объема обновляемых метаданных. Сначала нужно определиться с тем, какие типы производительности важнее всего для вашей системы, после чего можно будет выбрать оптимальную конфигурацию аппаратного обеспечения, соответствующую бюджету.

Пропускная способность дисков

Пропускная способность дисков брокера, которые используются для хранения сегментов журналов, самым непосредственным образом влияет на производительность клиентов-производителей. Сообщения Kafka должны фиксироваться в локальном хранилище, которое бы подтверждало их запись. Лишь после этого можно считать операцию отправки успешной. Это значит, что чем быстрее выполняются операции записи на диск, тем меньше будет задержка генерации сообщений.

Очевидное действие при возникновении проблем с пропускной способностью дисков — использовать жесткие диски с раскручивающимися пластинами (HDD) или твердотельные накопители (SSD). У SSD на порядки ниже время поиска/доступа и выше производительность. HDD же более экономичны, и у них более высокая относительная емкость. Производительность HDD можно улучшить за счет большего их числа в брокере, или используя несколько каталогов данных, или устанавливая диски в массив независимых дисков с избыточностью (redundant array of independent disks, RAID). На пропускную способность влияют и другие факторы, такие как технология изготовления жесткого диска (к примеру, SAS или SATA), а также характеристики контроллера жесткого диска. В целом наблюдения показывают, что жесткие диски обычно более полезны для кластеров с очень высокими потребностями в хранении данных, но к ним не так часто обращаются, в то время как твердотельные накопители оказываются лучшим вариантом при наличии очень большого количества клиентских подключений.

Емкость диска

Емкость — еще одна характеристика хранения. Требуемый объем дискового пространства определяется тем, сколько сообщений необходимо хранить одновременно. Если ожидается, что брокер будет получать 1 Тбайт трафика в день, то при семидневном хранении ему понадобится доступное для использования хранилище для сегментов журнала объемом минимум 7 Тбайт. Следует также учесть перерасход как минимум 10 % для других файлов, не считая буфера для возможных колебаний трафика или его роста с течением времени.

Емкость хранилища — один из факторов, которые следует учитывать при определении оптимального размера кластера Kafka и принятии решения о его расширении. Общий трафик кластера можно балансировать за счет нескольких разделов для каждого топика, что позволяет использовать дополнительные брокеры для наращивания доступной емкости в случаях, когда плотности данных на одного брокера недостаточно. Решение о том, сколько необходимо дискового пространства, определяется также выбранной для кластера стратегией репликации (подробнее обсуждается в главе 7).

Память

В обычном режиме функционирования потребитель Kafka читает из конца раздела, причем потребитель постоянно наверстывает упущенное и лишь незначительно отстает от производителей, если вообще отстает. При этом читаемые потребителем сообщения сохраняются оптимальным образом в страничном кэше

системы, благодаря чему операции чтения выполняются быстрее, чем если бы брокеру приходилось перечитывать их с диска. Следовательно, чем больший объем оперативной памяти доступен для страничного кэша, тем выше быстродействие клиентов-потребителей.

Для самой Kafka не требуется выделение для JVM большого объема оперативной памяти в куче. Даже брокер, который обрабатывает 150 000 сообщений в секунду при скорости передачи данных 200 Мбит/с, может работать с кучей 5 Гбайт. Остальная оперативная память системы будет применяться для страничного кэша и станет приносить Kafka пользу за счет возможности кэширования используемых сегментов журналов. Именно поэтому не рекомендуется располагать Kafka в системе, где уже работают другие важные приложения, так как ей придется делиться страничным кэшем, что снизит производительность потребителей Kafka.

Передача данных по сети

Максимальный объем трафика, который может обработать Kafka, определяется доступной пропускной способностью сети. Это может быть ключевым (наряду с объемом дискового хранилища) фактором выбора размера кластера. Затрудняет этот выбор присущий Kafka (из-за поддержки нескольких потребителей) дисбаланс между входящим и исходящим сетевым трафиком. Производитель может генерировать 1 Мбайт сообщений в секунду для заданного топика, но количество потребителей может оказаться каким угодно, привнося соответствующий множитель для исходящего трафика. Повышают требования к сети и другие операции, такие как репликация кластера (см. главу 7) и зеркальное копирование (обсуждается в главе 10). При интенсивном использовании сетевого интерфейса вполне возможно отставание репликации кластера, что вызовет неустойчивость его состояния. Чтобы сеть не стала основным определяющим фактором, рекомендуется применять сетевые адаптеры NICs (Network Interface Cards — сетевые интерфейсные карты) емкостью не менее 10 Гбайт. Старые машины с сетевыми адаптерами емкостью 1 Гбайт легко перегружаются, так что использовать их не рекомендуется.

CPU

Вычислительные мощности не так важны, как дисковое пространство и оперативная память, пока вы не начнете масштабировать Kafka в очень больших масштабах, но они тоже в некоторой степени влияют на общую производительность брокера. В идеале клиенты должны сжимать сообщения ради оптимизации использования сети и дискового пространства. Брокер Kafka, однако, должен

разархивировать все пакеты сообщений для проверки контрольных сумм отдельных сообщений и назначения смещений. После этого ему нужно снова сжать пакет сообщений для сохранения его на диске. Именно для этого Kafka требуется бóльшая часть вычислительных мощностей. Однако не следует рассматривать это как основной фактор при выборе аппаратного обеспечения, если только кластеры не становятся очень большими, с сотнями узлов и миллионами разделов в одном кластере. В этом случае выбор более производительного процессора может помочь уменьшить размер кластера.

Kafka в облачной среде

В последние годы Kafka чаще всего устанавливается в облачной вычислительной среде, например, в Microsoft Azure, Amazon's AWS или Google Cloud Platform. Существует множество вариантов установки Kafka в облаке и управления им с помощью таких поставщиков, как Confluent, или даже с помощью собственной программы Azure Kafka на HDInsight, но если вы планируете управлять собственными кластерами Kafka вручную, мы дадим вам несколько простых советов. В большинстве облачных сред у вас есть выбор из множества виртуальных вычислительных узлов, все с различными комбинациями CPU, операций ввода-вывода в секунду (IOPS), оперативной памяти и дискового пространства. Для выбора подходящей конфигурации виртуального узла нужно учесть в первую очередь факторы производительности Kafka.

Microsoft Azure

В Azure вы можете управлять дисками отдельно от виртуальной машины (VM), поэтому ваше решение о необходимости в хранении данных не обязательно должно быть связано с выбранным типом виртуальной машины. Тем не менее хорошая отправная точка для принятия решений — это необходимый объем хранения данных, после чего следует учесть требуемую эффективность производителей. В случае потребности в очень низкой задержке могут понадобиться виртуальные узлы с оптимизацией по операциям ввода/вывода, использующие SSD-накопитель премиального класса. В противном случае может оказаться достаточно управляемых вариантов хранилища, например управляемых дисков Azure (Azure Managed Disks) или хранилища больших двоичных объектов Azure (Azure Blob Storage).

На практике опыт использования Azure показывает, что стандартные экземпляры Standard D16s v3 — хороший выбор для небольших кластеров и они достаточно производительны для большинства сценариев использования.

Для удовлетворения потребностей в высокопроизводительном оборудовании и процессоре экземпляры D64s v4 обладают хорошей производительностью и могут масштабироваться для больших кластеров. Рекомендуется создавать кластер в наборе доступности Azure и сбалансировать разделы между домо-нами сбоев вычислений Azure, чтобы обеспечить доступность. После того как виртуальная машина выбрана, можно приступить к выбору типа хранилища. Настоятельно рекомендуется использовать управляемые диски Azure, а не эфемерные диски. Если виртуальная машина будет перемещена, вы рискуете потерять все данные на своем брокере Kafka. Управляемые диски HDD относительно недорогие, но не имеют четко определенных SLA от Microsoft по доступности. Твердотельные диски премиум-класса или конфигурации Ultra SSD стоят намного дороже, но работают намного быстрее и имеют хорошую поддержку с 99,99 % SLA от Microsoft. В качестве альтернативы можно использовать хранилище больших двоичных объектов Microsoft (Microsoft Blob Storage), если вы не так чувствительны к задержкам.

Веб-сервисы Amazon Web Services

В AWS, если требуется очень низкая задержка, могут потребоваться оптимизированные для операций ввода-вывода виртуальные узлы с локальным SSD-хранилищем. В противном случае может быть достаточно блочного хранилища, например Amazon Elastic Block Store.

Обычно в AWS выбирают экземпляры типа m4 или r3. Виртуальный узел типа m4 допускает более длительное хранение, но при меньшей пропускной способности записи на диск, поскольку основан на адаптивном блочном хранилище. Пропускная способность виртуального узла типа r3 намного выше благодаря использованию локальных SSD-дисков, но последние ограничивают доступный для хранения объем данных. Преимущества обоих этих вариантов сочетают существенно более дорогостоящие типы виртуальных узлов i2 и d2.

Настройка кластеров Kafka

Отдельный брокер Kafka хорошо подходит для локальной разработки или создания прототипов систем, но настроить несколько брокеров для совместной работы в виде кластера намного выгоднее (рис. 2.2). Основная выгода от этого — возможность масштабировать нагрузку на несколько серверов. Вторая по значимости — возможность использования репликации для защиты от потери данных вследствие отказов отдельных систем. Репликация также дает

возможность выполнить работы по обслуживанию Kafka или нижележащей системы с сохранением доступности для клиентов. В этом разделе мы рассмотрим только настройку базового кластера Kafka. Более подробную информацию о репликации данных и долговечности вы найдете в главе 7.

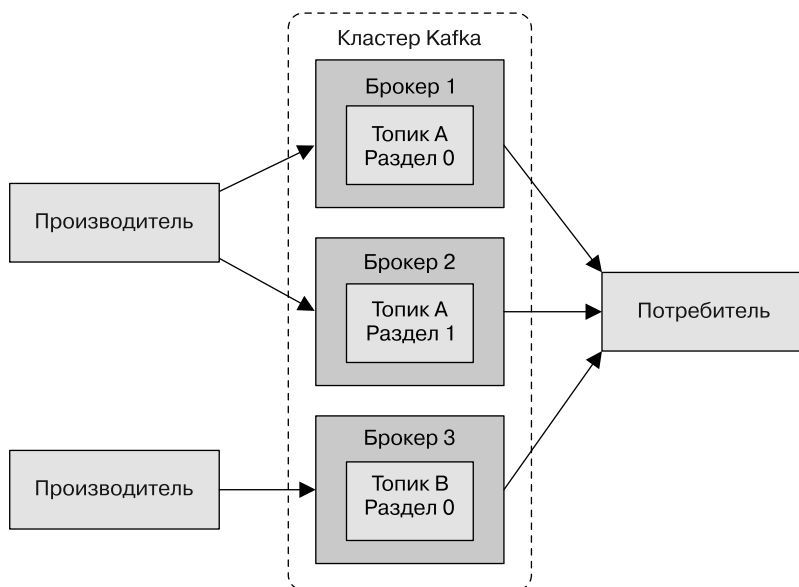


Рис. 2.2. Простой кластер Kafka

Сколько должно быть брокеров

Размер кластера Kafka определяется несколькими факторами. Как правило, размер вашего кластера зависит:

- от емкости диска;
- емкости реплик на одного брокера;
- мощности процессора;
- пропускной способности сети.

Первая характеристика — требующийся для хранения сообщений объем дискового пространства и объем доступного места на отдельном брокере. Если кластеру необходимо хранить 10 Тбайт данных, а отдельный брокер может хранить 2 Тбайт, то минимальный размер кластера — пять брокеров. Кроме того,

увеличение коэффициента репликации может повысить требования к хранилищу минимум на 100 % в зависимости от выбранной настройки коэффициента репликации (см. главу 7). Реплики в данном случае означают количество различных брокеров, в которые копируется один раздел. Это значит, что в случае использования репликации, настроенной на коэффициент 2, тот же кластер должен будет содержать как минимум десять брокеров.

Еще один фактор, который нужно учесть, — возможности кластера по обработке запросов. Это может проявляться в трех других узких местах, упомянутых ранее.

Если у вас кластер Kafka с десятью брокерами и более чем 1 млн реплик (то есть 500 000 разделов с коэффициентом репликации 2), каждый брокер принимает на себя примерно 100 000 реплик при равномерно сбалансированном сценарии. Это может привести к появлению узких мест в очередях производства, потребления и контроллеров. В прошлом официальные рекомендации заключались в том, чтобы иметь не более 4000 реплик разделов на брокер и не более 200 000 реплик разделов на кластер. Однако прогресс в области эффективности кластеров позволил масштабировать Kafka гораздо больше. В настоящее время в хорошо настроенной среде рекомендуется иметь не более 14 000 реплик разделов на брокер и 1 миллиона *реплик на кластер*.

Как упоминалось ранее в этой главе, центральный процессор обычно не является основным узким местом для большинства сценариев использования, но он может стать таковым при чрезмерном количестве клиентских соединений и запросов на брокере. Если следить за общим использованием ЦП в зависимости от количества уникальных клиентов и групп потребителей и расширять его в соответствии с этими потребностями, это поможет обеспечить более высокую общую производительность в больших кластерах. Говоря о пропускной способности сети, важно помнить о пропускной способности сетевых интерфейсов и о том, способны ли они справиться с клиентским трафиком при нескольких потребителях данных или колебаниями трафика на протяжении хранения данных, то есть в случае всплесков трафика в период пиковой нагрузки. Если сетевой интерфейс отдельного брокера используется на 80 % при пиковой нагрузке, а потребителя данных два, то они не смогут справиться с пиковым трафиком при менее чем двух брокерах. Если в кластере используется репликация, она играет роль дополнительного потребителя данных, который необходимо учесть. Вы можете захотеть увеличить количество брокеров в кластере, чтобы справиться с проблемами производительности, вызванными понижением пропускной способности дисков или объема доступной оперативной памяти.

Конфигурация брокеров

Есть только два требования к конфигурации брокеров, которые должны работать в составе одного кластера Kafka. Первое — в конфигурации всех брокеров должно быть одинаковое значение параметра `zookeeper.connect`. Он задает ансамбль ZooKeeper и путь хранения кластером метаданных. Второе — у каждого из брокеров кластера должно быть уникальное значение параметра `broker.id`. Если два брокера с одинаковым значением `broker.id` попытаются присоединиться к кластеру, то второй брокер запишет в журнал сообщение об ошибке и не запустится. Существуют и другие параметры конфигурации брокеров, используемые при работе кластера, а именно параметры для управления репликацией, описываемые в дальнейших главах.

Тонкая настройка операционной системы

Хотя в большинстве дистрибутивов Linux есть готовые конфигурации параметров конфигурации ядра, которые довольно хорошо подходят для большинства приложений, можно внести в них несколько изменений для повышения производительности брокера Kafka. В основном они относятся к подсистемам виртуальной памяти и сети, а также специфическим моментам, касающимся точки монтирования диска для сохранения сегментов журналов. Эти параметры обычно настраиваются в файле `/etc/sysctl.conf`, но лучше обратиться к документации конкретного дистрибутива Linux, чтобы выяснить все нюансы корректировки настроек ядра.

Виртуальная память

Обычно система виртуальной памяти Linux сама подстраивается под нагрузку системы. Но можно внести некоторые корректировки в работу как с областью подкачки, так и с «грязными» страницами памяти, чтобы лучше приспособить их к специфике нагрузки Kafka.

Как и для большинства приложений, особенно тех, где важна пропускная способность, стоит избегать подкачки практически любой ценой. Затраты, обусловленные подкачкой страниц памяти на диск, существенно влияют на все аспекты производительности Kafka. Кроме того, Kafka активно использует системный страничный кэш, и если подсистема виртуальной памяти выполняет подкачку на диск, то страничному кэшу выделяется недостаточное количество памяти.

Один из способов избежать подкачки — просто не выделять в настройках никакого места для нее. Подкачка — не обязательное требование, а, скорее,

страховка на случай какой-либо аварии в системе. Она может спасти от неожиданного прерывания системой выполнения процесса вследствие нехватки памяти. Поэтому рекомендуется делать значение параметра `vm.swappiness` очень маленьким, например 1. Этот параметр представляет собой вероятность (в процентах) того, что подсистема виртуальной машины будет использовать пространство подкачки вместо удаления страниц из страничного кэша. Предпочтительнее уменьшить объем памяти, доступной для кэша страниц, чем использовать любой объем памяти подкачки.



Почему бы не сделать параметр `swappiness` равным 0?

Ранее рекомендовалось всегда задавать параметр `vm.swappiness` равным 0. Смысл этого значения был таков: никогда не применять подкачку, разве что при нехватке памяти. Однако в версии 3.5-rc1 ядра Linux смысл поменялся, и это изменение было бэкпортировано во многие дистрибутивы, включая ядро Red Hat Enterprise Linux версии 2.6.32-303. Значение 0 при этом приобрело смысл «не использовать подкачку ни при каких обстоятельствах». Поэтому сейчас рекомендуется брать значение 1.

Корректировка того, что ядро системы делает с «грязными» страницами, которые должны быть сброшены на диск, также имеет смысл. Быстрота ответа Kafka производителям зависит от производительности дисковых операций ввода/вывода. Именно поэтому сегменты журналов обычно размещаются на быстрых дисках — или отдельных дисках с быстрым временем отклика (например, SSD), или дисковых подсистемах с большим объемом NVRAM для кэширования (например, RAID). В результате появляется возможность уменьшить число «грязных» страниц, по достижении которого запускается фоновый сброс их на диск. Для этого необходимо задать значение параметра `vm.dirty_background_ratio` меньшее, чем значение по умолчанию (равно 10). Оно означает долю всей памяти системы (в процентах), и во многих случаях его можно задать равным 5. Однако не следует делать его равным 0, поскольку в этом случае ядро начнет непрерывно сбрасывать страницы на диск и тем самым потеряет возможность буферизации дисковых операций записи при временных флуктуациях производительности нижележащих аппаратных компонентов.

Общее количество «грязных» страниц, при превышении которого ядро системы принудительно инициирует запуск синхронных операций по сбросу их на диск, можно повысить путем изменения параметра `vm.dirty_ratio` на значение, превышающее значение по умолчанию — 20 (тоже доля в процентах от всего объема памяти системы). Существует широкий диапазон возможных значений этого параметра, но наиболее разумные располагаются между 60 и 80. Изменение этого параметра несколько рискованно в смысле как объема не сброшенных на диск действий, так и вероятности возникновения длительных пауз

ввода/вывода в случае принудительного запуска синхронных операций сброса. При выборе более высоких значений параметра `vm.dirty_ratio` настойчиво рекомендуется использовать репликацию в кластере Kafka, чтобы защититься от системных сбоев.

При выборе значений этих параметров имеет смысл контролировать количество «грязных» страниц в ходе работы кластера Kafka под нагрузкой при промышленной эксплуатации или имитационном моделировании. Определить его можно с помощью просмотра файла `/proc/vmstat`:

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

Kafka использует файловые дескрипторы для сегментов журнала и открытых соединений. Если брокер имеет много разделов, то ему необходимо по меньшей мере $(\text{количество_разделов}) \times (\text{размер_раздела}/\text{размер_сегмента})$ для отслеживания всех сегментов журнала в дополнение к количеству соединений, которые создает брокер. Поэтому рекомендуется обновить значение параметра `vm.max_map_count` до очень большого числа на основе приведенного выше расчета. Изменение этого значения на 400 000 или 600 000 в зависимости от среды обычно дает положительный результат. Также рекомендуется установить значение параметра `vm.overcommit_memory` равным 0. Установка значения по умолчанию 0 указывает на то, что ядро определяет объем свободной памяти приложения. Если для свойства установлено значение, отличное от нуля, это может привести к тому, что операционная система будет захватывать слишком много памяти, лишая Kafka возможности работать оптимально. Это характерно для приложений с высокой скоростью поступления данных.

Диск

Если не считать выбора аппаратного обеспечения подсистемы жестких дисков, а также конфигурации RAID-массива в случае его использования, сильнее всего влияет на производительность применяемая для этих дисков файловая система. Существует множество разных файловых систем, но в качестве локальной файловой системы чаще всего задействуется Ext4 (fourth extended file system — четвертая расширенная файловая система) или XFS (Extents File System — файловая система на основе экстентов). Ext4 работает довольно хорошо, но требует потенциально небезопасных параметров тонкой настройки. Среди них установка более длительного интервала фиксации, чем значение по

умолчанию (5), с целью понижения частоты сброса на диск. В Ext4 появилось также отложенное выделение блоков, повышающее вероятность потери данных и повреждения файловой системы в случае системного отказа. В файловой системе XFS тоже используется алгоритм отложенного выделения, но более безопасный, чем в Ext4. И производительность XFS для типичной нагрузки Kafka выше, причем нет необходимости делать тонкую настройку сверх автоматической, выполняемой самой файловой системой. Она эффективнее также при пакетных операциях записи на диск, объединяемых для повышения пропускной способности при вводе/выводе.

Вне зависимости от файловой системы, выбранной в качестве точки монтирования для сегментов журналов, рекомендуется указывать параметр монтирования `noatime`. Метаданные файла содержат три метки даты/времени: время создания (`ctime`), время последнего изменения (`mtime`) и время последнего обращения к файлу (`atime`). По умолчанию значение атрибута `atime` обновляется при каждом чтении файла. Это значительно увеличивает число операций записи на диск. Атрибут `atime` обычно не слишком полезен, за исключением случая, когда приложению необходима информация о том, обращались ли к файлу после его последнего изменения (в этом случае можно применить параметр `realtime`). Kafka вообще не использует атрибут `atime`, так что можно спокойно его отключить. Установка параметра `noatime` для точки монтирования предотвращает обновления меток даты/времени, но не влияет на корректную обработку атрибутов `ctime` и `mtime`. Использование опции `largeio` также может помочь повысить эффективность работы Kafka при больших объемах записи на диск.

Передача данных по сети

Корректировка настроек по умолчанию сетевого стека Linux — обычное дело для любого приложения, генерирующего много сетевого трафика, так как ядро по умолчанию не приспособлено для высокоскоростной передачи больших объемов данных. На деле рекомендуемые для Kafka изменения не отличаются от изменений, рекомендуемых для большинства веб-серверов и других сетевых приложений. Вначале необходимо изменить объемы (по умолчанию и максимальный) памяти, выделяемой для буферов отправки и получения для каждого сокета. Это значительно увеличит производительность в случае передачи больших объемов данных. Соответствующие параметры для значений по умолчанию буферов отправки и получения каждого сокета называются `net.core.wmem_default` и `net.core.rmem_default` соответственно, а разумное их значение будет 2 097 152 (2 Мбайт). Имейте в виду, что максимальный размер не означает выделения для каждого буфера такого пространства, а лишь позволяет сделать это при необходимости.

Помимо настройки сокетов, необходимо отдельно задать размеры буферов отправки и получения для сокетов TCP с помощью параметров `net.ipv4.tcp_wmem` и `net.ipv4.tcp_rmem`. В них указываются три разделенных пробелами целых числа, определяющих минимальный размер, размер по умолчанию и максимальный размер соответственно. Пример этих параметров — `4096 65536 2048000` — означает, что минимальный размер буфера — 4 Кбайт, размер по умолчанию — 64 Кбайт, а максимальный — 2 Мбайт. Максимальный размер не может превышать значений, задаваемых для всех сокетов параметрами `net.core.wmem_max` и `net.core.rmem_max`. В зависимости от настоящей загрузки ваших брокеров Kafka может понадобиться увеличить максимальные значения для повышения степени буферизации сетевых соединений.

Существует еще несколько полезных сетевых параметров. Можно включить оконное масштабирование TCP, установив значение 1 параметра `net.ipv4.tcp_window_scaling`, что позволит клиентам эффективнее передавать данные и обеспечит возможность их буферизации на стороне брокера. Значение параметра `net.ipv4.tcp_max_syn_backlog` больше, чем принятое по умолчанию `1024`, позволяет увеличить число одновременных подключений. Значение параметра `net.core.netdev_max_backlog`, превышающее принятое по умолчанию `1000`, может помочь в случае всплесков сетевого трафика, особенно при скоростях сетевого подключения порядка гигабит, благодаря увеличению числа пакетов, помещаемых в очередь для последующей обработки ядром.

Промышленная эксплуатация

Когда наступит время перевести среду Kafka из режима тестирования в промышленную эксплуатацию, останется позаботиться лишь еще о нескольких моментах для настройки надежного сервиса обмена сообщениями.

Параметры сборки мусора

Тонкая настройка сборки мусора Java для приложения всегда была своеобразным искусством, требующим подробной информации об использовании памяти приложением и немалой толики наблюдений, проб и ошибок. К счастью, это изменилось после выхода Java 7 и появления сборщика мусора Garbage-First (G1GC). Хотя изначально G1GC считался нестабильным, в JDK8 и JDK11 наблюдалось заметное улучшение. Теперь для Kafka рекомендуется использовать G1GC в качестве сборщика мусора по умолчанию. G1GC умеет автоматически приспосабливаться к различным типам нагрузки и обеспечивать согласованность пауз на сборку мусора на протяжении всего жизненного цикла приложения. Он также с легкостью управляется с кучей большого размера, так как

разбивает ее на небольшие зоны, вместо того чтобы собирать мусор по всей куче при каждой паузе.

В обычном режиме работы для выполнения всего этого G1GC требуются минимальные настройки. Для корректировки его производительности используются два параметра.

- **MaxGCPauseMillis.** Задает желаемую длительность паузы на каждый цикл сборки мусора. Это не фиксированный максимум — при необходимости G1GC может превысить эту длительность. По умолчанию данное значение равно 200 мс. Это значит, что G1GC будет стараться планировать частоту циклов сборщика мусора, а также числа зон, обрабатываемых в каждом цикле, так, чтобы каждый цикл занимал примерно 200 мс.
- **InitiatingHeapOccupancyPercent.** Задаёт долю в процентах от общего размера кучи, до превышения которой G1GC не начинает цикл сборки мусора. Значение по умолчанию равно 45. Это значит, что G1GC не запустит цикл сборки мусора до того, как будет использоваться 45 % кучи, включая суммарное задействование зон как новых (eden), так и старых объектов.

Брокер Kafka весьма эффективно использует память из кучи и создает объекты, так что можно задавать более низкие значения этих параметров. Приведенные в данном разделе значения параметров сборки мусора признаны вполне подходящими для сервера с 64 Гбайт оперативной памяти, где Kafka работала с кучей 5 Гбайт. Этот брокер мог работать при значении 20 параметра `MaxGCPauseMillis`. А значение параметра `InitiatingHeap_OccupancyPercent` установлено 35, благодаря чему сборка мусора запускается несколько раньше, чем при значении по умолчанию.

Первоначально Kafka была выпущена до того, как сборщик G1GC оказался доступен и стал считаться стабильным. Поэтому Kafka по умолчанию использует параллельную маркировку и сборку мусора (mark-and-sweep) для обеспечения совместимости со всеми JVM. Новой лучшей практикой является использование G1GC для всего, что касается Java 1.8 и более поздних версий. Это можно легко изменить с помощью переменных среды. Используя команду `start` из предыдущей главы, измените ее следующим образом:

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

Планировка ЦОД

При использовании сред, ориентированных на тестирование и разработку, физическое расположение брокеров Kafka в ЦОД особого значения не имеет, так как частичная или полная недоступность кластера в течение коротких промежутков времени влияет на работу не сильно. Однако при промышленной эксплуатации простой в процессе выдачи данных обычно означает потерю денег или из-за невозможности обслуживания пользователей, или из-за невозможности получения телеметрии их действий. При этом возрастает важность применения репликации в кластере Kafka (см. главу 7), а также физического расположения брокеров в стойках в ЦОД. Предпочтительнее среда центра обработки данных, в которой существует концепция зон отказа. Если не позаботиться об этом до развертывания Kafka, могут потребоваться дорогостоящие работы по перемещению серверов.

Kafka может назначать новые разделы брокерам с учетом стоек, следя за тем, чтобы реплики для одного раздела не использовали одну стойку. Для этого необходимо правильно настроить конфигурацию стойки `broker.rack` для каждого брокера. Эта конфигурация может быть установлена на домен ошибок и в облачных средах по аналогичным причинам. Однако это относится только к вновь создаваемым разделам. Кластер Kafka не отслеживает наличие разделов, которые больше не поддерживаются в стойке (например, в результате переназначения разделов), и не исправляет эту ситуацию автоматически. Рекомендуется использовать инструменты, которые позволяют правильно сбалансировать кластер для поддержания осведомленности о стойках, такие как Cruise Control (см. приложение Б). Правильная настройка этого параметра поможет обеспечить постоянную осведомленность о стойке в долгосрочной перспективе.

В целом рекомендуемой практикой является установка каждого брокера Kafka в кластере в отдельной стойке или по крайней мере использование ими различных критических точек инфраструктурных сервисов, таких как питание и сеть. Обычно это означает, что серверы, на которых будут работать брокеры, должны иметь зарезервированное подключение к электросети (к двум разным цепям электропитания) и двойные сетевые коммутаторы со связанным интерфейсом на самих серверах для бесперебойной работы. Время от времени может понадобиться выполнить обслуживание аппаратной части стойки или шкафа с их отключением, например передвинуть сервер или заменить электропроводку.

Размещение приложений на ZooKeeper

Kafka использует ZooKeeper для хранения метаданных о брокерах, топиках и разделах. Запись в ZooKeeper выполняют только при изменении списков участников групп потребителей или изменениях в самом кластере Kafka. Обычно при этом объем трафика минимален, так что использование выделенного

ансамбля ZooKeeper для одного кластера Kafka необоснованно. На самом деле один ансамбль ZooKeeper часто применяется для нескольких кластеров Kafka (с задействованием нового корневого пути ZooKeeper для каждого кластера, как описывалось ранее в данной главе).



Потребители Kafka, инструментарий (tooling), ZooKeeper и вы

С течением времени зависимость от ZooKeeper уменьшается. В версии 2.8.0 Kafka предоставляет ранний доступ к Kafka без ZooKeeper, но он пока не готов к производству. Тем не менее мы все еще можем наблюдать уменьшение зависимости от ZooKeeper в версиях, предшествующих этой. Например, в более старых версиях Kafka потребители (в дополнение к брокерам) использовали ZooKeeper для непосредственного сохранения информации о составе групп потребителей и соответствующих топиках, а также для периодической фиксации смещений для каждого из обрабатываемых разделов (для обеспечения возможности восстановления после сбоев между потребителями в группе). В версии 0.9.0.0 интерфейс потребителей изменился, благодаря чему стало возможно делать это непосредственно с помощью брокеров Kafka. В каждом выпуске Kafka 2.x мы видим дополнительные шаги по удалению ZooKeeper из других необходимых путей Kafka. Инструменты администрирования теперь подключаются непосредственно к кластеру, и была отменена необходимость прямого подключения к ZooKeeper для таких операций, как создание топиков, динамические изменения конфигурации и т. д. Таким образом, многие инструменты командной строки, которые ранее использовали флаг `--zookeeper`, были обновлены для использования опции `--bootstrap-server`. С опциями `--zookeeper` по-прежнему можно работать, но они устарели и будут удалены в дальнейшем, когда Kafka больше не потребует подключения к ZooKeeper для создания топиков, управления ими или потребления из них.

Однако при работе потребителей и ZooKeeper с определенными настройками есть нюанс. Хотя использование ZooKeeper для таких целей устарело, для фиксации смещений потребители могут использовать или ZooKeeper, или Kafka, причем интервал между фиксациями можно настраивать. Если потребители применяют ZooKeeper для смещений, то каждый из них будет производить операцию записи ZooKeeper через заданное время для каждого потребляемого им раздела. Обычный промежуток времени для фиксации смещений — 1 мин, так как именно через такое время группа потребителей читает дублирующие сообщения в случае сбоя потребителя. Эти фиксации могут составлять существенную долю трафика ZooKeeper, особенно в кластере с множеством потребителей, так что их следует учитывать. Если ансамбль ZooKeeper не способен обрабатывать такой объем трафика, может потребоваться увеличить интервал фиксации. Однако рекомендуется, чтобы работающие с актуальными библиотеками Kafka потребители использовали Kafka для фиксации смещений и не зависели от ZooKeeper.

Если не считать использования одного ансамбля для нескольких кластеров Kafka, не рекомендуется делить ансамбль с другими приложениями, если этого можно избежать. Kafka весьма чувствительна к длительности задержки и времени ожидания ZooKeeper, и нарушение связи с ансамблем может вызвать непредсказуемое поведение брокеров. Вследствие этого несколько брокеров вполне могут одновременно отключиться в случае потери подключений к ZooKeeper, что приведет к отключению разделов. Это также создаст дополнительную нагрузку на диспетчер кластера, что может вызвать возникновение неочевидных ошибок через длительное время после нарушения связи, например, при попытке контролируемого останова брокера. Следует выносить в отдельные ансамбли другие приложения, создающие нагрузку на диспетчер кластера в результате активного использования или неправильного функционирования.

Резюме

В этой главе мы поговорили о том, как установить и запустить Apache Kafka. Обсудили, как выбрать подходящее аппаратное обеспечение для брокеров, и разобрались в специфических вопросах настроек для промышленной эксплуатации. Теперь, имея кластер Kafka, мы можем пройти по основным вопросам клиентских приложений Kafka. Следующие две главы будут посвящены созданию клиентов как для генерации сообщений для Kafka (глава 3), так и для их последующего потребления (глава 4).

Производители Kafka: запись сообщений в Kafka

Используется ли Kafka в качестве очереди, шины передачи сообщений или платформы хранения данных, в любом случае всегда создаются производитель, записывающий данные в Kafka, потребитель, читающий данные из нее, или приложение, выполняющее и то и другое.

Например, в системе обработки транзакций для кредитных карт всегда будет клиентское приложение, например интернет-магазин, отвечающее за немедленную отправку каждой из транзакций в Kafka сразу же после выполнения платежа. Еще одно приложение будет отвечать за немедленную проверку этой транзакции с помощью процессора правил и выяснение того, одобрена она или отклонена. Затем ответ «одобрить/отклонить» можно будет записать в Kafka и передать в интернет-магазин, инициировавший транзакцию. Третье приложение будет выполнять чтение как транзакции, так и информации о том, одобрена ли она, из Kafka и сохранять их в базе данных, чтобы аналитик мог позднее просмотреть принятые решения и, возможно, усовершенствовать процессор правил.

Apache Kafka поставляется со встроенным клиентским API, которым разработчики могут воспользоваться при создании взаимодействующих с Kafka приложений.

В этой главе мы научимся использовать производители Kafka, начав с обзора их архитектуры и компонентов. Мы покажем, как создавать объекты `KafkaProducer` и `ProducerRecord`, отправлять записи в Kafka и обрабатывать возвращаемые ею сообщения об ошибках. Далее приведем обзор наиболее важных настроек для управления поведением производителя. Завершим главу более детальным обзором использования различных методов секционирования и сериализаторов, а также обсудим написание ваших собственных сериализаторов и объектов `Partitioner`.

В главе 4 мы займемся клиентом-потребителем Kafka и чтением данных из нее.



Сторонние клиенты

Помимо встроенных клиентов, в Kafka имеется двоичный протокол передачи данных. Это значит, что приложения могут читать сообщения из Kafka или записывать сообщения в нее простой отправкой соответствующих последовательностей байтов на сетевой порт Kafka. Существует множество клиентов, реализующих протокол передачи данных Kafka на различных языках программирования, благодаря чему можно легко использовать ее не только в Java-приложениях, но и в таких языках программирования, как C++, Python, Go, и многих других. Эти клиенты не включены в проект Apache Kafka, но в «Википедии», в статье о проекте, приводится список клиентов для отличных от Java языков (<https://oreil.ly/9SbJr>). Рассмотрение протокола передачи данных и внешних клиентов выходит за рамки данной главы.

Обзор производителя

Существует множество причин, по которым приложению может понадобиться записывать сообщения в Kafka: фиксация действий пользователей для аудита или анализа, запись показателей, сохранение журнальных сообщений, запись поступающей от интеллектуальных устройств информации, асинхронное взаимодействие с другими приложениями, буферизация информации перед записью ее в базу данных и многое другое. Эти разнообразные сценарии применения означают также разнообразие требований: каждое ли сообщение критически важно, или потеря части сообщений допустима? Допустимо ли случайное дублирование сообщений? Нужно ли придерживаться каких-либо жестких требований относительно длительности задержки или пропускной способности?

В рассмотренном ранее примере обработки транзакций для кредитных карт критически важно было не терять ни одного сообщения и не допускать их дублирования. Длительность задержки должна быть низкой, но допустимы задержки до 500 мс, а пропускная способность должна быть очень высокой — предполагается обрабатывать миллионы сообщений в секунду.

Совсем другим сценарием использования было бы хранение информации о щелчках кнопкой мыши на сайте. В этом случае допустимы потеря части сообщений или небольшое количество повторяющихся сообщений. Длительность задержки может быть высокой, так как она никак не влияет на удобство работы пользователя. Другими словами, ничего страшного, если сообщение поступит в Kafka только через несколько секунд, главное, чтобы следующая страница загрузилась сразу же после щелчка пользователя на ссылке. Пропускная способность зависит от ожидаемого уровня пользовательской активности на веб-сайте.

Различие в требованиях влияет на применение API производителей для записи сообщений в Kafka и на используемые настройки.

Хотя API производителей очень простой, внутри производителя, «под капотом», при отправке данных происходит немного больше действий. Основные этапы отправки данных в Kafka демонстрирует рис. 3.1.

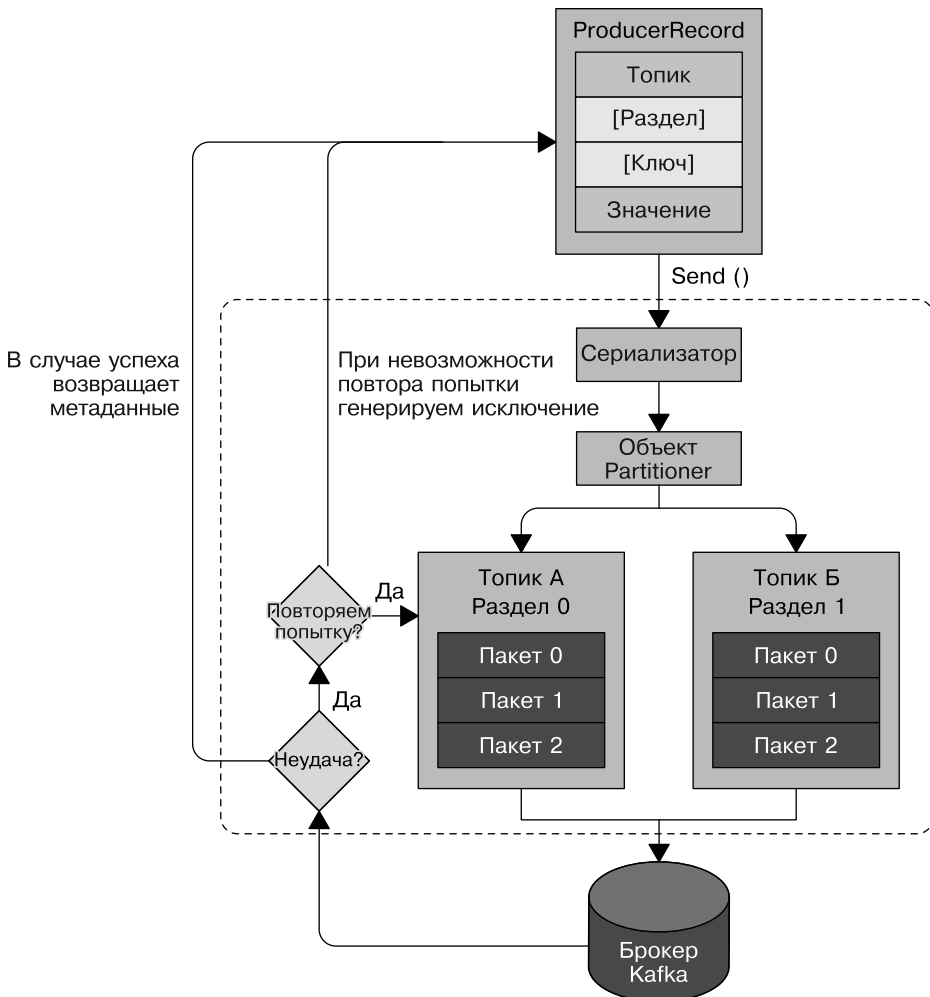


Рис. 3.1. Высокоуровневый обзор компонентов производителей Kafka

Для генерации сообщений для Kafka нам понадобится сначала создать объект **ProducerRecord**, включающий топик, в который мы собираемся отправить запись, и значение. При желании можно задать также ключ, раздел, временную

метку и/или набор заголовков. После отправки объекта `ProducerRecord` он прежде всего сериализует объекты ключа и значения в байтовые массивы для отправки по сети.

Далее, если мы не указали раздел явно, данные попадают в объект `Partitioner`. Объект `Partitioner` выбирает раздел, обычно в соответствии с ключом из `ProducerRecord`. Если раздел выбран, производитель будет знать, в какой топик и раздел должна попасть запись. После этого он помещает эту запись в пакет записей, предназначенных для отправки в соответствующие топик и раздел. За отправку пакетов записей соответствующему брокеру Kafka отвечает отдельный поток выполнения.

После получения сообщений брокер отправляет ответ. В случае успешной записи сообщений в Kafka будет возвращен объект `RecordMetadata`, содержащий топик, раздел и смещение записи в разделе. Если брокеру не удалось записать сообщения, он вернет сообщение об ошибке. При получении сообщения об ошибке производитель может попробовать отправить сообщение еще несколько раз, прежде чем оставит эти попытки и вернет ошибку.

Создание производителя Kafka

Первый шаг записи сообщений в Kafka — создание объекта производителя со свойствами, которые вы хотели бы передать производителю. У производителей Kafka есть три обязательных свойства:

- `bootstrap.servers` — список пар `host:port` брокеров, используемых производителем для первоначального соединения с кластером Kafka. Он не обязан включать все брокеры, поскольку производитель может получить дополнительную информацию после начального соединения. Но рекомендуется включить в него хотя бы два брокера, чтобы производитель мог подключиться к кластеру при сбое одного из них;
- `key.serializer` — имя класса, применяемого для сериализации ключей записей, генерируемых для отправки в Kafka. Брокеры Kafka ожидают, что в качестве ключей и значений сообщений задействуются байтовые массивы. Однако интерфейс производителя позволяет отправлять в качестве ключа и значения (посредством использования параметризованных типов) любой объект. Это сильно повышает удобочитаемость кода, но производителю при этом нужно знать, как преобразовать эти объекты в байтовые массивы. Значением свойства `key.serializer` должно быть имя класса, реализующего интерфейс `org.apache.kafka.common.serialization.Serializer`. С помощью этого класса производитель сериализует объект ключа в байтовый массив.

Пакет программ клиента Kafka включает классы `ByteArraySerializer` (почти ничего не делает), `StringSerializer`, `IntegerSerializer` и многие другие, так что при использовании обычных типов данных нет необходимости реализовывать свои сериализаторы. Задать значение свойства `key.serializer` необходимо, даже если вы планируете отправлять только значения, но можно применять тип `Void` для ключа и `VoidSerializer`;

- `value.serializer` — имя класса, используемого для сериализации значений записей, генерируемых для отправки в Kafka. Аналогично тому, как значение свойства `key.serializer` соответствует классу, с помощью которого производитель сериализует объект ключа сообщения в байтовый массив, значение свойства `value.serializer` должно быть равно имени класса, служащего для сериализации объекта значения сообщения.

Следующий фрагмент кода демонстрирует создание нового производителя с помощью задания лишь обязательных параметров и использования значений по умолчанию для всего остального:

```
Properties kafkaProps = new Properties(); ❶
kafkaProps.put("bootstrap.servers", "broker1:9092,broker2:9092");

kafkaProps.put("key.serializer",
    "org.apache.kafka.common.serialization.StringSerializer"); ❷
kafkaProps.put("value.serializer",
    "org.apache.kafka.common.serialization.StringSerializer");

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

- ❶ Начинаем с объекта `Properties`.
- ❷ Поскольку мы планируем использовать строки для ключа и значения сообщения, воспользуемся встроенным типом `StringSerializer`.
- ❸ Создаем новый производитель, задавая подходящие типы ключа и значения и передавая в конструктор объект `Properties`.

При таком простом интерфейсе понятно, что управляют поведением производителя, в основном задавая соответствующие параметры конфигурации. В документации Apache Kafka описаны все параметры конфигурации (<http://bit.ly/2sMu1c8>). Кроме того, мы рассмотрим самые важные из них далее в этой главе.

После создания экземпляра производителя можно приступить к отправке сообщений. Существует три основных метода отправки сообщений.

- *Сделать и забыть*. Отправляем сообщение на сервер, после чего особо не волнуемся, дошло оно или нет. Обычно оно доходит успешно, поскольку Kafka отличается высокой доступностью, а производитель повторяет отправку

сообщений автоматически. Однако в случае неустранимых ошибок или тайм-аута сообщения будут потеряны и приложение не получит никакой информации или исключений по этому поводу.

- *Синхронная отправка.* Технически производитель Kafka всегда является асинхронным — мы отправляем сообщения и метод `send()` возвращает объект-фьючерс (объект класса `Future`). Однако можно воспользоваться методом `get()` для организации ожидания в классе `Future` и выяснения того, успешно ли прошла отправка, перед тем как отправить следующую запись.
- *Асинхронная отправка.* Мы вызываем метод `send()`, которому передается функция обратного вызова, выполняемая при получении ответа от брокера Kafka.

В примерах в дальнейшем мы увидим, как отправлять сообщения с помощью данных методов и обрабатывать различные типы возникающих при этом ошибок.

Хотя все примеры в этой главе однопоточны, объект производителя может использоваться для отправки сообщений несколькими потоками.

Отправка сообщения в Kafka

Вот простейший способ отправки сообщения:

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

❶ Производитель получает на входе объекты `ProducerRecord`, так что начнем с создания такого объекта. У класса `ProducerRecord` есть несколько конструкторов, которые обсудим позднее. В данном случае мы имеем дело с конструктором, принимающим на входе строковое значение — название топика, в который отправляются данные, и отсылаемые в Kafka ключ и значение (тоже строки). Типы ключа и значения должны соответствовать нашим объектам `key serializer` и `value serializer`.

❷ Для отправки объекта типа `ProducerRecord` используем метод `send()` объекта `producer`. Как мы уже видели в схеме архитектуры производителя (см. рис. 3.1), сообщение помещается в буфер и отправляется брокеру в отдельном потоке. Метод `send()` возвращает объект класса `Future` языка Java (<http://bit.ly/2rG7Cg6>), включающий `RecordMetadata`, но поскольку мы просто игнорируем возвраща-

емое значение, то никак не можем узнать, успешно ли было отправлено сообщение. Такой способ отправки сообщений можно использовать, только если потерять сообщение вполне допустимо.

❸ Хотя ошибки, возможные при отправке сообщений брокерам Kafka или возникающие в самих брокерах, игнорируются, при появлении в производителе ошибки перед отправкой сообщения в Kafka вполне может быть сгенерировано исключение. К примеру, это может быть исключение `SerializationException` при неудачной сериализации сообщения, `BufferExhaustedException` или `TimeoutException` при переполнении буфера, `InterruptedException` при сбое отправляющего потока.

Синхронная отправка сообщения

Синхронная отправка сообщения проста, но все же позволяет производителю перехватывать исключения, когда Kafka отвечает на запрос на создание с ошибкой или когда повторные попытки отправки были исчерпаны. Основным компромиссом при этом является производительность. В зависимости от загруженности кластера Kafka брокерам может потребоваться от 2 мс до нескольких секунд, чтобы выдать ответ на запрос на создание. Если вы отправляете сообщения синхронно, поток отправителя будет тратить это время на ожидание и ничего больше не станет делать, даже не будет отправлять дополнительные сообщения. Это очень снижает производительность, поэтому синхронные отправки обычно не используются в производственных приложениях (но очень часто встречаются в примерах кода).

Вот простейший способ синхронной отправки сообщения:

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

❶ Используем метод `Future.get()` для ожидания ответа от Kafka. Этот метод генерирует исключение в случае неудачи отправки записи в Kafka. При отсутствии ошибок мы получим объект `RecordMetadata`, из которого можно узнать смещение, соответствующее записанному сообщению, и другие метаданные.

❷ Если перед отправкой или во время отправки записи в Kafka возникли ошибки, нас будет ожидать исключение. В этом случае просто выводим информацию о нем.

В классе `KafkaProducer` существует два типа ошибок. Первый тип — ошибки, которые можно исправить, отправив сообщение *повторно* (retriable). Например, так можно исправить ошибку соединения, поскольку через некоторое время оно способно восстановиться. Ошибка «отсутствует ведущий узел для раздела» может быть исправлена выбором нового ведущего узла для раздела, и метаданные клиента обновятся. Можно настроить `KafkaProducer` так, чтобы при таких ошибках отправка повторялась автоматически. Код приложения будет получать исключения подобного типа только тогда, когда лимит на повторы уже исчерпан, а ошибка все еще не исправлена. Некоторые ошибки невозможно исправить повторной отправкой сообщения. Такова, например, ошибка «сообщение слишком велико». В подобных случаях `KafkaProducer` не станет пытаться повторить отправку, а сразу же вернет исключение.

Асинхронная отправка сообщения

Пусть время прохождения сообщения между нашим приложением и кластером Kafka и обратно составляет 10 мс. Если мы будем ждать ответа после отправки каждого сообщения, отправка 100 сообщений займет около 1 с. Но если не ожидать ответов, отправка всех сообщений практически не займет времени. В большинстве случаев ответ и не требуется — Kafka возвращает топик, раздел и смещение записи, которые обычно не нужны отправляющему приложению. Однако нам нужно знать, удалась ли вообще отправка сообщения, чтобы можно было сгенерировать исключение, зафиксировать в журнале ошибку или, возможно, записать сообщение в файл ошибок для дальнейшего анализа.

Для асинхронной отправки сообщений с сохранением возможности обработки различных сценариев ошибок производители поддерживают добавление функции обратного вызова при отправке записи. Вот пример использования функции обратного вызова:

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

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

- ❶ Для использования функций обратного вызова нам понадобится класс, реализующий интерфейс `org.apache.kafka.clients.producer.Callback`, включающий одну-единственную функцию `onCompletion()`.
- ❷ Если Kafka вернет ошибку, в функцию `onCompletion()` попадет непустое исключение. В приведенном коде вся его обработка заключается в выводе информации, но в коде для промышленной эксплуатации функции обработки исключений, вероятно, будут более ошибкоустойчивыми.
- ❸ Записи остаются такими же, как и раньше.
- ❹ И мы передаем объект `Callback` при отправке записи.



Обратные вызовы выполняются в главном потоке производителя. Это гарантирует, что, когда мы отправляем два сообщения в один и тот же раздел одно за другим, их обратные вызовы будут выполняться в том же порядке, в котором мы их отправили. Но это означает также, что обратный вызов должен выполняться достаточно быстро, чтобы не задерживать производителя и не мешать отправке других сообщений. Не рекомендуется выполнять блокирующие операции внутри обратного вызова. Вместо этого следует использовать другой поток для одновременного выполнения любой блокирующей операции.

Настройка производителей

До сих пор мы практически не сталкивались с конфигурационными параметрами производителей — только с обязательным URI `bootstrap.servers` и сериализаторами.

У производителя есть множество параметров конфигурации, которые описаны в документации Apache Kafka (<https://oreil.ly/RkxSS>). Значения по умолчанию многих из них вполне разумны, так что нет смысла возиться с каждым параметром. Однако некоторые существенно влияют на использование памяти, производительность и надежность производителей. Рассмотрим их.

`client.id`

Параметр `client.id` является логическим идентификатором клиента и приложения, в котором он используется. Он может быть любой строкой и будет использоваться брокерами для идентификации сообщений, отправленных от клиента. Он применяется для ведения журналов и в метриках, а также для определения квот. Выбор подходящего имени клиента значительно облегчит

поиск и устранение неисправностей — в этом заключается разница между «Мы наблюдаем высокий уровень сбоев аутентификации с IP 104.27.155.134» и «Похоже, служба проверки заказов не может пройти аутентификацию, — не могли бы вы попросить Лауру взглянуть?».

acks

Параметр `acks` определяет, сколько реплик разделов должны получить запись, прежде чем производитель сможет считать запись успешной. Он существенно влияет на вероятность потери сообщений. По умолчанию Kafka отвечает, что запись была успешно записана, после того как лидер получил запись (ожидается, что в версии Apache Kafka 3.0 это значение по умолчанию будет изменено). Этот параметр оказывает значительное влияние на долговечность записанных сообщений, и в зависимости от вашего сценария использования значение по умолчанию может оказаться не лучшим выбором. В главе 7 подробно обсуждаются гарантии надежности Kafka, а пока давайте рассмотрим три допустимых значения параметра `acks`.

- **`acks=0`**. Производитель не будет ждать ответа от брокера, чтобы счесть отправку сообщения успешной. Это значит, что, если произойдет сбой и брокер не получит сообщение, производитель об этом не узнает и сообщение будет потеряно. Но поскольку производитель не ждет какого-либо ответа от сервера, то скорость отправки сообщений ограничивается лишь возможностями сети, так что эта настройка позволяет достичь очень высокой пропускной способности.
- **`acks=1`**. Производитель получает от брокера ответ об успешном получении сразу же, как только ведущая реплика получит сообщение. Если сообщение не удалось записать на ведущий узел (например, когда этот узел потерпел фатальный сбой, а новый еще не успели выбрать), производитель получит ответ об ошибке и может попытаться вновь отправить сообщение, предотвращая тем самым потенциальную потерю данных. Сообщение все равно может быть потеряно, если ведущий узел потерпел фатальный сбой и последние сообщения еще не были реплицированы новому лидеру.
- **`acks=all`**. Ответ от брокера об успешном получении сообщения приходит производителю после того, как оно дойдет до всех синхронизируемых реплик. Это самый безопасный режим, поскольку есть уверенность, что сообщение получено более чем одним брокером и не пропадет даже в случае аварийного сбоя (больше информации по этому вопросу вы найдете в главе 6). Однако в случае `acks=1` задержка будет еще выше, ведь придется ждать получения сообщения более чем одним брокером.



Вы увидите, что при более низкой и менее надежной конфигурации `acks` производитель сможет отправлять записи быстрее. Это означает, что вы жертвуете надежностью ради задержки производителя. Однако время сквозной задержки между конечными пунктами измеряется с момента создания записи до момента, когда она станет доступной для чтения потребителями, и оно одинаково для всех трех вариантов. Причина этого заключается в том, что для поддержания согласованности Kafka не позволяет потребителям читать записи до тех пор, пока они не будут записаны во все синхронизированные реплики. Поэтому, если вам важно время задержки между конечными пунктами, а не только задержка производителя, не нужно искать компромисс — вы получите одинаковое время сквозной задержки между конечными пунктами, если выберете наиболее надежный вариант.

Время доставки сообщения

У производителя есть несколько параметров конфигурации, которые взаимодействуют для управления одним из видов поведения, представляющих наибольший интерес для разработчиков: сколько времени потребуется, пока вызов функции `send()` не завершится успешно или с ошибкой. Это время, которое мы готовы потратить, пока Kafka не ответит успешно или пока мы не будем готовы сдаться и не признаем поражение.

Конфигурации и их поведение несколько раз изменялись на протяжении многих лет. Здесь мы опишем последнюю реализацию, представленную в Apache Kafka 2.1.

Начиная с Apache Kafka 2.1, мы разделяем время, затраченное на отправку `ProduceRecord`, на два временных интервала, которые обрабатываются отдельно.

- Время до возвращения асинхронного вызова функции `send()`. В течение этого интервала поток, вызвавший функцию `send()`, будет заблокирован.
- Время с момента успешного возврата асинхронного вызова `send()` до запуска обратного вызова (успешного или с ошибкой). Это то же самое, что и с момента, когда запись `ProduceRecord` была помещена в пакет для отправки, до момента, когда Kafka ответит успехом, ошибкой, неустранимым отказом или закончится время, отведенное на отправку.



Если вы используете функцию `send()` синхронно, поток отправки будет непрерывно блокироваться в течение обоих временных интервалов и вы не сможете определить, сколько времени было потрачено на каждый из них. Мы обсудим распространенный и рекомендуемый случай, когда функция `send()` используется асинхронно, с обратным вызовом.

Поток данных внутри производителя и то, как различные параметры конфигурации влияют друг на друга, можно обобщить на рис. 3.2¹.

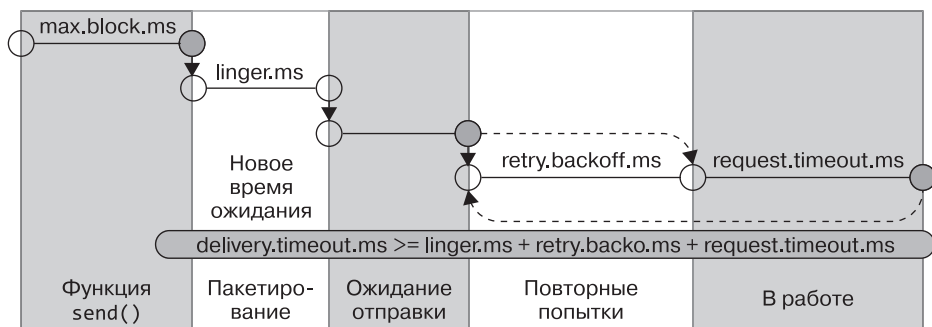


Рис. 3.2. Диаграмма последовательности распределения времени доставки внутри производителя Kafka

Мы рассмотрим различные параметры конфигурации, используемые для управления временем ожидания в этих двух интервалах, и то, как они взаимодействуют.

max.block.ms

Этот параметр контролирует, как долго производитель может блокировать при вызове функции `send()` и при явном запросе метаданных через `partitionsFor()`. Эти методы выполняют блокировку при переполнении буфера отправки производителя или недоступности метаданных. По истечении времени ожидания `max.block.ms` выдается исключение времени ожидания.

delivery.timeout.ms

Эта конфигурация ограничивает время, которое проходит с момента, когда запись готова к отправке (функция `send()` возвращается успешно, и запись помещается в пакет), до момента, пока брокер не ответит или клиент не откажется, включая время, затраченное на повторные попытки. Как показано на рис. 3.2, это время должно быть больше, чем `linger.ms` и `request.timeout.ms`. Если вы пытаетесь создать производитель с несоответствующей конфигурацией времени

¹ Изображение предоставлено Сумантом Тамбе (Sumant Tambe) для проекта Apache Kafka в соответствии с условиями лицензии ASLv2.

ожидания, то получите исключение. Сообщения могут быть успешно отправлены гораздо быстрее, чем `delivery.timeout.ms`, и обычно так и происходит.

Если производитель превысит `delivery.timeout.ms` при повторной попытке, обратный вызов будет выполнен с исключением, соответствующим ошибке, которую брокер вернул перед повторной попыткой. Если значение `delivery.timeout.ms` превышено в то время, когда пакет записей все еще ожидал отправки, обратный вызов будет выполнен с исключением тайм-аута.



Вы можете настроить время ожидания доставки на максимальное время ожидания отправки сообщения — обычно на несколько минут, а затем оставить количество повторных попыток по умолчанию (практически бесконечное). При такой конфигурации производитель будет повторять попытки до тех пор, пока у него есть время на это или пока он не добьется успеха. Это гораздо более разумный способ использования повторных попыток. Наш обычный процесс настройки повторных попыток таков: «В случае сбоя брокера обычно требуется 30 с для завершения выбора лидера, поэтому давайте на всякий случай будем повторять попытки в течение 120 с». Вместо того чтобы преобразовывать этот мысленный диалог в количество повторных попыток и время между ними, вы просто считаете `delivery.timeout.ms` равным 120.

request.timeout.ms

Этот параметр определяет, как долго производитель будет ждать ответа от сервера при отправке данных. Обратите внимание на то, что это время, потраченное на ожидание каждого запроса производителя перед отказом, оно не включает повторные попытки, время, потраченное перед отправкой, и т. д. Если время ожидания будет достигнуто без ответа, производитель либо повторит попытку отправки, либо завершит обратный вызов с исключением `TimeoutException`.

Повторные попытки и `retry.backoff.ms`

Когда производитель получает сообщение об ошибке от сервера, ошибка может быть временной — например, отсутствие лидера для раздела. В этом случае значение параметра `retries` будет определять, сколько раз производитель повторит попытку отправки сообщения, прежде чем отказаться от этого и уведомить клиента о проблеме. По умолчанию производитель будет ждать 100 мс между повторными попытками, но вы можете изменить это значение с помощью параметра `retry.backoff.ms`.

Мы не рекомендуем использовать эти параметры в текущей версии Kafka. Вместо этого проверьте, сколько времени потребуется для восстановления после сбоя брокера (то есть сколько времени пройдет, пока все разделы не получат новых лидеров), и установите значение параметра `delivery.timeout.ms` таким, чтобы общее количество времени, потраченного на повторные попытки, было больше, чем время, необходимое кластеру Kafka для восстановления после сбоя, — в противном случае производитель сдастся слишком быстро.

Не все ошибки будут повторно обработаны производителем. Некоторые из них не являются временными и не вызывают повторных попыток (например, ошибка «сообщение слишком большое»). В целом, поскольку производитель обрабатывает повторные попытки сам, нет смысла обрабатывать их в рамках вашей собственной логики приложения. Вы должны сосредоточить свои усилия на обработке ошибок, не допускающих повторных попыток, или случаев, когда повторные попытки были исчерпаны.



Если вы хотите полностью отключить повторные попытки, то единственный способ сделать это — установить значение `retries=0`.

linger.ms

Параметр `linger.ms` управляет длительностью времени ожидания дополнительных сообщений перед отправкой текущего пакета. `KafkaProducer` отправляет пакет сообщений, либо когда текущий пакет заполнен, либо когда достигнут предел `linger.ms`. По умолчанию производитель будет отправлять сообщения, как только появится поток отправителя, доступный для их отправки, даже если в пакете всего одно сообщение. Устанавливая `linger.ms` на значение выше 0, мы даем команду производителю подождать несколько миллисекунд, чтобы добавить дополнительные сообщения в пакет перед его отправкой брокерам. Это немного увеличивает задержку и значительно повышает пропускную способность — накладные расходы на одно сообщение намного ниже, а сжатие, если оно включено, намного лучше.

buffer.memory

Этот параметр задает объем памяти, используемой производителем для буферизации сообщений, ожидающих отправки брокерам. Если приложение отправляет сообщения быстрее, чем они могут быть доставлены серверу, у производителя может закончиться свободное место и дополнительные вызовы метода `send()` приведут к блокировке параметра `max.block.ms` и ожиданию, пока освободится

место, прежде чем сгенерировать исключение. Обратите внимание на то, что, в отличие от большинства исключений производителей, это время ожидания генерируется функцией `send()`, а не результирующим `Future`.

compression.type

По умолчанию сообщения отправляются в несжатом виде. Возможные значения этого параметра — `snappy`, `gzip`, `lz4` и `zstd`, при которых к данным применяются соответствующие алгоритмы сжатия перед отправкой брокерам. Алгоритм сжатия `snappy` был разработан компанией Google для обеспечения хорошей степени сжатия при низком перерасходе ресурсов CPU и высокой производительности. Его рекомендуется использовать в случаях, когда важны как производительность, так и пропускная способность сети. Алгоритм сжатия `gzip` обычно задействует больше ресурсов CPU и требует больше времени, но обеспечивает лучшую степень сжатия. Поэтому его рекомендуется применять в случаях, когда пропускная способность сети ограничена. Благодаря сжатию можно снизить нагрузку на сеть и хранилище, часто являющиеся узкими местами при отправке сообщений в Kafka.

batch.size

При отправлении в один раздел нескольких записей производитель соберет их в один пакет. Этот параметр определяет объем памяти в байтах (не число сообщений!) для каждого пакета. По заполнении пакета все входящие в него сообщения отправляются. Но это не значит, что производитель будет ждать наполнения пакета. Он может отправлять пакеты, заполненные наполовину, и даже пакеты, содержащие лишь одно сообщение. Следовательно, задание слишком большого размера пакета приведет не к задержкам отправки, а лишь к использованию большего количества памяти для пакетов. Слишком маленький размер пакета обусловит дополнительный расход памяти, поскольку производителю придется отправлять сообщения чаще.

max.in.flight.requests.per.connection

Управляет количеством пакетов сообщений, которые производитель может отправить серверу, не получая ответов. Высокие значения этого параметра приведут к повышенному использованию памяти, но одновременно и к увеличению пропускной способности. Вики-эксперименты Apache показывают (<https://oreil.ly/NZmJ0>), что в среде `singleDC` пропускная способность максимальна только при двух активных запросах, однако значение по умолчанию равно 5 и показывает аналогичную производительность.



Гарантии упорядочения

Apache Kafka сохраняет порядок сообщений внутри раздела. Это означает, что если сообщения отправляются производителем в определенном порядке, то в этом же порядке брокер запишет их в раздел и все потребители будут их читать. В некоторых случаях порядок очень важен. Существует большая разница между тем, чтобы внести 100 долларов на счет и позже снять их, и обратной ситуацией! Однако некоторые сценарии использования менее чувствительны к упорядочению.

Установка параметра `retries` в значение, отличное от нуля, и параметра `max.in.flight.requests.per.connection` в значение больше 1 означает, что, возможно, брокер, которому не удалось записать первую партию сообщений, успешно запишет вторую (которая уже была отправлена), а затем повторит попытку записи первой партии и добьется успеха, тем самым изменив порядок.

Поскольку мы хотим иметь как минимум два запроса в активном состоянии для повышения производительности и большое количество повторных попыток из соображений повышения надежности, лучшим решением будет установить `enable.idempotence=true`. Это гарантирует упорядочение сообщений при пяти запросах в активном состоянии, а также то, что при повторных попытках не будет дубликатов. В главе 8 подробно обсуждается идемпотентный производитель.

max.request.size

Этот параметр задает максимальный размер отправляемого производителем запроса. Он ограничивает как максимальный размер сообщения, так и число сообщений, отсылаемых в одном запросе. Например, при максимальном размере сообщения по умолчанию 1 Мбайт производитель может отправить максимум одно сообщение размером 1 Мбайт или скомпоновать в один запрос 1024 сообщения по 1 Кбайт каждое. Кроме того, у брокеров тоже есть ограничение максимального размера принимаемого сообщения (`message.max.bytes`). Обычно рекомендуется делать значения этих параметров одинаковыми, чтобы производитель не отправлял сообщения неприемлемого для брокера размера.

receive.buffer.bytes и send.buffer.bytes

Это размеры TCP-буферов отправки и получения, используемых сокетами при записи и чтении данных. Если значение этих параметров равно -1, будут использоваться значения по умолчанию операционной системы. Рекомендуется повышать их в случае, когда производители или потребители взаимодействуют с брокерами из другого ЦОД, поскольку подобные сетевые подключения обычно характеризуются более длительной задержкой и низкой пропускной способностью сети.

enable.idempotence

Начиная с версии 0.11, Kafka поддерживает семантику «ровно один раз». Ровно один раз — это довольно большая тема, и мы посвятим ей целую главу, но идемпотентный производитель — простая и очень полезная ее часть.

Предположим, вы настроили своего производителя на максимальную надежность `acks=all` и довольно большое значение параметра `delivery.timeout.ms`, чтобы обеспечить достаточное количество повторных попыток. Это гарантирует, что каждое сообщение будет записано в Kafka хотя бы один раз. В некоторых случаях это означает, что сообщения будут записываться в Kafka более одного раза. Например, представьте, что брокер получил от производителя запись, записал ее на локальный диск и эта запись была успешно реплицирована другим брокерам, но затем первый брокер вышел из строя, не успев отправить производителю ответ. Производитель будет ждать, пока не достигнет `request.timeout.ms`, а затем повторит попытку. Повторная попытка будет направлена новому лидеру, который уже имеет копию этой записи, поскольку предыдущая запись была успешно реплицирована. Теперь у вас есть дубликат записи.

Чтобы избежать этого, можете установить значение `enable.idempotence=true`. Когда включена функция идемпотентного производителя, производитель будет прикреплять порядковый номер к каждой отправляемой записи. Если брокер получит записи с тем же порядковым номером, что и у предыдущей, он отклонит вторую копию, а производитель получит безобидное исключение `DuplicateSequenceException`.



Включение функции идемпотентности требует, чтобы значение параметра `max.in.flight.requests.per.connection` было меньше или равно 5, значение параметра `retries` — больше 0, а `acks=all`. Если заданы несовместимые значения, будет выдано исключение `ConfigException`.

Сериализаторы

Как мы видели в предыдущих примерах, конфигурация производителя обязательно включает сериализаторы. Мы уже рассматривали применение сериализатора по умолчанию — `StringSerializer`. Kafka также включает сериализаторы для целых чисел, байтовых массивов `ByteArrays` и многие другие, но это охватывает далеко не все сценарии использования. В конце концов, вам понадобится сериализовывать более общие виды записей.

Начнем с написания пользовательского сериализатора, после чего покажем рекомендуемый альтернативный вариант — сериализатор `Avro`.

Пользовательские сериализаторы

Когда нужно отправить в Kafka объект более сложный, чем просто строка или целочисленное значение, можно или воспользоваться для создания записей универсальной библиотекой сериализации, например Avro, Thrift или Protobuf, или создать пользовательский сериализатор для уже используемых объектов. Мы настоятельно рекомендуем вам работать с универсальной библиотекой сериализации. Но чтобы разобраться, как действуют сериализаторы и почему лучше применять библиотеку сериализации, посмотрим, что требуется для написания собственного сериализатора.

Пусть вместо того, чтобы записывать только имя покупателя, вы создали простой класс `Customer`:

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

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

    public int getID() {
        return customerID;
    }

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

Теперь предположим, что вам нужно создать пользовательский сериализатор для этого класса. Он будет выглядеть примерно так:

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

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

public class CustomerSerializer implements Serializer<Customer> {

    @Override
    public void configure(Map configs, boolean isKey) {
        // нечего настраивать
    }

    @Override
    /**
```


Мы сериализуем объект `Customer` как:

4-байтное целое число, соответствующее `customerId`

4-байтное целое число, соответствующее длине `customerName` в байтах

в кодировке UTF-8 (0, если имя не заполнено)

N байт, соответствующих `customerName` в кодировке UTF-8

*/

```
public byte[] serialize(String topic, Customer data) {
    try {
        byte[] serializedName;
        int stringSize;
        if (data == null)
            return null;
        else {
            if (data.getName() != null) {
                serializedName = data.getName().getBytes("UTF-8");
                stringSize = serializedName.length;
            } else {
                serializedName = new byte[0];
                stringSize = 0;
            }
        }

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

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

@Override
public void close() {
    // нечего закрывать
}
}
```

Настройка производителя с использованием этого `CustomerSerializer` дает возможность определить тип `ProducerRecord<String, Customer>` и отправлять данные типа `Customer`, непосредственно передавая объекты `Customer` производителю. Приведенный пример очень прост, но из него можно понять, насколько ненадежен такой код. Если, например, у нас слишком много покупателей и понадобится поменять тип `customerId` на `Long` или добавить в тип `Customer` поле `startDate`, то вы столкнетесь с непростой проблемой поддержания совместимости между старым и новым форматами сообщения. Отладка проблем совместимости между различными версиями сериализаторов и десериализаторов — весьма непростая задача, ведь приходится сравнивать неформатированные байтовые массивы.

Что еще хуже, если нескольким группам разработчиков из одной компании понадобится записывать данные о покупателях в Kafka, им придется использовать одинаковые сериализаторы и менять код совершенно синхронно.

Поэтому мы рекомендуем использовать существующие сериализаторы и десериализаторы, например JSON, Apache Avro, Thrift или Protobuf. В следующем разделе расскажем про Apache Avro, а затем покажем, как сериализовать записи Avro и отправлять их в Kafka.

Сериализация с помощью Apache Avro

Apache Avro — независимый от языка программирования формат сериализации данных. Этот проект создан Дугом Каттингом (Doug Cutting) для обеспечения возможности использования данных совместно с большим количеством других людей.

Данные Avro описываются независимой от языка схемой. Она обычно выполняется в формате JSON, а сериализация производится в двоичные файлы, хотя сериализация в JSON тоже поддерживается. При записи и чтении файлов Avro предполагает наличие схемы, обычно во вложенном в файлы виде.

Одна из самых интересных возможностей Avro, благодаря которой он так хорошо подходит для использования в системах обмена сообщениями вроде Kafka, состоит в том, что при переходе записывающего сообщения приложения на новую, но совместимую схему читающее данные приложение может продолжать обработку сообщений без каких-либо изменений или обновлений.

Пусть исходная схема выглядела следующим образом:

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

❶ Поля `id` и `name` — обязательные, а поле `faxNumber` — необязательное, по умолчанию оно является неопределенным значением.

Допустим, что эта схема использовалась в течение нескольких месяцев и в таком формате было сгенерировано несколько терабайт данных. Теперь предположим, что в новой версии мы решили признать, что наступил XXI век, и вместо номера факса применим поле `email`.

Новая схема будет выглядеть вот так:

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

Теперь после обновления до новой версии в старых записях будет содержаться поле `faxNumber`, а в новых — `email`. Во многих организациях обновления выполняются медленно, на протяжении многих месяцев. Так что придется продумать обработку в Kafka всех событий еще не обновленными приложениями, использующими номера факса, и уже обновленными — с адресом электронной почты.

Выполняющее чтение приложение должно содержать вызовы таких методов, как `getName()`, `getId()` и `getFaxNumber()`. Наткнувшись на записанное по новой схеме сообщение, методы `getName()` и `getId()` продолжают работать без всяких изменений, но метод `getFaxNumber()` вернет `null`, поскольку сообщение не содержит номера факса.

Теперь предположим, что мы модифицировали читающее приложение и в нем вместо метода `getFaxNumber()` теперь есть метод `getEmail()`. Наткнувшись на записанное по старой схеме сообщение, метод `getEmail()` вернет `null`, поскольку в старых сообщениях нет адреса электронной почты.

Этот пример иллюстрирует выгоды использования Avro: хоть мы и поменяли схему сообщений без изменения всех читающих данные приложений, никаких исключений или серьезных ошибок не возникло, как не понадобилось и выполнять дорогостоящие обновления существующих данных.

Однако у этого сценария есть два нюанса.

- Используемая для записи данных схема и схема, которую ожидает читающее данные приложение, должны быть совместимыми. В документации Avro описаны правила совместимости (<http://bit.ly/2t9FmEb>).
- У десериализатора должен быть доступ к схеме, задействованной при записи данных, даже если она отличается от схемы, которую ожидает обращающееся к данным приложение. В файлах Avro схема для записи включается в сам файл, но для сообщений Kafka есть более удачный способ. Мы рассмотрим его далее.

Использование записей Avro с Kafka

В отличие от файлов Avro, при использовании которых хранение всей схемы в файле данных дает довольно умеренные накладные расходы, хранение всей схемы в каждой записи обычно более чем вдвое увеличивает размер последней. Однако в Avro при чтении записи необходима полная схема, так что нам нужно поместить ее куда-то в другое место. Для этого мы, придерживаясь распространенного архитектурного паттерна, воспользуемся *реестром схем* (*Schema Registry*). Реестр схем не включен в Kafka, но существует несколько его вариантов с открытым исходным кодом. Для нашего примера возьмем Confluent Schema Registry. Код Confluent Schema Registry можно найти на GitHub (<https://oreil.ly/htoZK>), его также можно установить в виде части платформы Confluent (<https://oreil.ly/n2V71>). Если вы решите использовать этот реестр схем, рекомендуем взглянуть в его документацию в Confluent (<https://oreil.ly/yFkTX>).

Суть этого приема состоит в хранении в реестре всех используемых для записи данных в Kafka схем. В этом случае можно хранить в отправляемой в Kafka записи только идентификатор схемы. Потребители могут в дальнейшем извлечь запись из реестра схем по идентификатору и десериализовать данные. Самое главное, что вся работа — сохранение схемы в реестре и извлечение ее при необходимости — выполняется в сериализаторах и десериализаторах. Код производителя отправляемых в Kafka сообщений просто использует сериализатор Avro так же, как использовал бы любой другой сериализатор. Этот процесс показан на рис. 3.3.

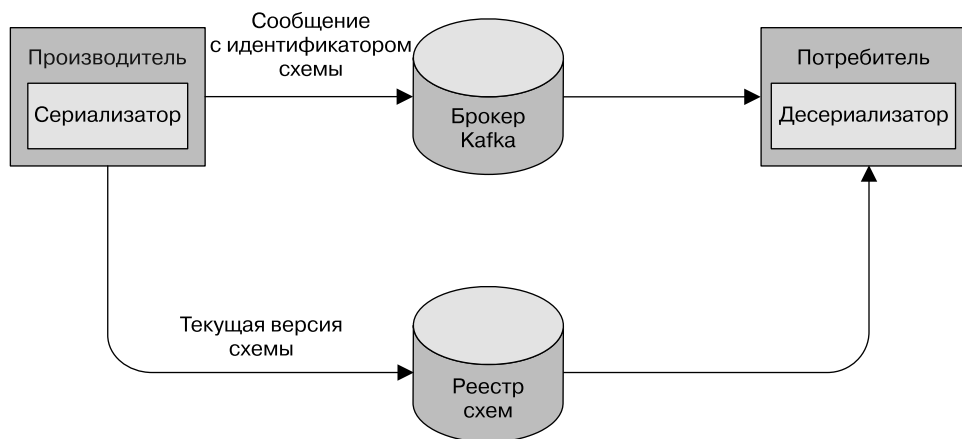


Рис. 3.3. Блок-схема сериализации и десериализации записей Avro

Вот пример отправки сгенерированных Avro объектов в Kafka (см. документацию Avro по адресу <https://oreil.ly/klcjK>, чтобы получить информацию о том, как генерировать объекты из схем Avro):

```

Properties props = new Properties();

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

String topic = "customerContacts";

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

// Генерация новых событий продолжается вплоть до нажатия Ctrl+C
while (true) {
    Customer customer = CustomerGenerator.getNext(); ❹
    System.out.println("Generated customer " +
        customer.toString());
    ProducerRecord<String, Customer> record =
        new ProducerRecord<>(topic, customer.getName(), customer); ❺
    producer.send(record); ❻
}

```

❶ Для сериализации объектов с помощью Avro мы используем класс `KafkaAvroSerializer`. Обратите внимание на то, что `KafkaAvroSerializer` может работать с простыми типами данных, именно поэтому в дальнейшем объект `String` используется в качестве ключа записи, а объект `Customer` — в качестве значения.

❷ `schema.registry.url` — это конфигурация сериализатора Avro, которая будет передана сериализатору производителем. Она указывает на место хранения схем.

❸ `Customer` — сгенерированный объект. Мы сообщаем производителю, что значение наших записей будет представлять собой объект `Customer`.

❹ Класс `Customer` — это не обычный класс Java (простой старый объект Java, или POJO), а, скорее, специализированный объект Avro, созданный на основе схемы с помощью генерации кода Avro. Сериализатор Avro может сериализовать только объекты Avro, но не POJO. Генерация классов Avro может быть выполнена либо с помощью `avro-tools.jar`, либо с помощью подключаемого плагина Avro Maven, оба — часть Apache Avro. Подробности о том, как генерировать классы Avro, смотрите в руководстве по началу работы с Apache Avro (Java) (<https://oreil.ly/sHGEE>).

❺ Мы также создаем экземпляр класса `ProducerRecord` с объектом `Customer` в качестве типа значения и передаем объект `Customer` при создании новой записи.

❻ Вот и все. Мы отправили запись с объектом `Customer`, а `KafkaAvroSerializer` позаботится обо всем остальном.

Авго позволяет также задействовать общие объекты Авго, которые используются в качестве карт «ключ/значение», вместо сгенерированных объектов Авго с получателями и установщиками, соответствующими схеме, с помощью которой они сгенерированы. Чтобы применять типовые объекты Авго, вам просто нужно предоставить схему:

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

String schemaString =
    "{ \"namespace\": \"customerManagement.avro\",
      \"type\": \"record\", \" + ❸
      \"name\": \"Customer\", \" +
      \"fields\": [ \" +
        \"{ \"name\": \"id\", \"type\": \"int\" }, \" +
        \"{ \"name\": \"name\", \"type\": \"string\" }, \" +
        \"{ \"name\": \"email\", \"type\": [\"null\", \"string\"],
          \"default\": \"null\" } \" +
      \" ] }";

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

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

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

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

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

❶ Мы по-прежнему используем тот же класс `KafkaAvroSerializer`.

❷ И передаем URI того же реестра схем.

❸ Но теперь нам приходится указывать схему Авго, поскольку ее уже не предоставляет сгенерированный Авго объект.

❹ Тип объекта теперь `GenericRecord`. Мы инициализируем его своей схемой и предназначенными для записи данными.

❺ Значение `ProducerRecord` представляет собой просто объект `GenericRecord`, содержащий схему и данные. Сериализатор будет знать, как получить из этой записи схему данных, сохранить ее в реестре схем и сериализовать данные из объекта.

Разделы

В предыдущих примерах создаваемые нами объекты `ProducerRecord` включали название топика, ключ и значение. Сообщения Kafka представляют собой пары «ключ/значение», и хотя можно создавать объекты `ProducerRecord` только с топиком и значением, с неопределенным значением по умолчанию для ключа, большинство приложений отправляют записи с ключами. Ключи служат двум целям: они представляют собой дополнительную информацию, сохраняемую вместе с сообщением, и на их основе обычно определяется, в какой раздел топика записывать сообщение (ключи играют важную роль также в сжатых топиках — мы обсудим их в главе 6). Все сообщения с одинаковым ключом попадут в один раздел. Это значит, что, если каждый процесс читает лишь часть разделов топика (подробнее об этом — в главе 4), все записи для конкретного ключа будут читать один и тот же процесс. Для создания записи типа «ключ/значение» нужно просто создать объект `ProducerRecord`, вот так:

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

При создании сообщений с неопределенным значением ключа можно просто его не указывать:

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

❶ В этом примере ключ равен `null`.

Если ключ равен `null` и используется метод секционирования по умолчанию, запись будет отправлена в один из доступных разделов топика случайным образом. Для балансировки сообщений по разделам при этом будет применяться циклический алгоритм (round-robin). Начиная с версии Apache Kafka 2.4, циклический алгоритм, используемый в разделителе по умолчанию при работе с нулевыми ключами, является «липким». Это означает, что он будет заполнять пакет сообщений, отправленных в один раздел, прежде чем переключиться на следующий раздел. Это позволяет отправлять то же количество сообщений

в Kafka за меньшее количество запросов, что уменьшает задержку и снижает загрузку процессора брокера.

Если же ключ присутствует и используется метод секционирования по умолчанию, Kafka вычислит хеш-значение ключа с помощью собственного алгоритма хеширования, так что хеш-значения не изменятся при обновлении Java, и отправит сообщение в конкретный раздел на основе полученного результата. А поскольку важно, чтобы ключи всегда соответствовали одним и тем же разделам, для вычисления соответствия используются все разделы топика, а не только доступные. Это значит, что, если конкретный раздел недоступен на момент записи в него данных, будет возвращена ошибка. Это происходит довольно редко, как вы увидите в главе 7, когда мы будем обсуждать репликацию и доступность Kafka.

В дополнение к разделителю по умолчанию клиенты Apache Kafka предоставляют также `RoundRobinPartitioner` и `UniformStickyPartitioner`. Они обеспечивают случайное назначение разделов и «липкое» случайное назначение разделов соответственно, даже если сообщения имеют ключи. Это полезно в тех случаях, когда ключи важны для потребляющего приложения (например, существуют приложения ETL, которые используют ключ из записей Kafka в качестве первичного ключа при загрузке данных из Kafka в реляционную базу данных), но рабочая нагрузка может быть асимметричной, поэтому один ключ может иметь непропорционально большую рабочую нагрузку. Использование `UniformStickyPartitioner` приведет к равномерному распределению рабочей нагрузки по всем разделам.

Когда применяется разделитель по умолчанию, соответствие ключей разделам остается согласованным лишь до тех пор, пока число разделов в топике не меняется. Так что пока это число постоянно, вы можете быть уверены, например, что относящиеся к пользователю 045189 записи всегда будут записываться в раздел 34. Эта особенность открывает дорогу для всех видов оптимизации при чтении данных из разделов. Но, как только вы добавите в топик новые разделы, такое поведение больше нельзя будет гарантировать: старые записи останутся в разделе 34, а новые могут оказаться записанными в другой раздел. Если ключи секционирования важны, простейшим решением будет создавать топики с достаточным числом разделов. (В блоге Confluent содержатся предложения о том, как выбрать количество разделов (<https://oreil.ly/ortRk>) и никогда не добавлять новые разделы.)

Реализация пользовательской стратегии секционирования

До сих пор мы обсуждали особенности метода секционирования по умолчанию, используемого чаще всего. Однако Kafka не ограничивает вас лишь хеш-разделами, и иногда появляются веские причины секционировать данные иначе. Например, представьте, что вы B2B-поставщик, а ваш крупнейший по-

купатель — компания Banapa, производящая карманные устройства. Допустим, что более 10 % ваших ежедневных транзакций приходится на этого покупателя. При использовании хеш-секционирования, принятого по умолчанию, записи Banapa будут распределяться в тот же раздел, что и записи других заказчиков, в результате чего один из разделов окажется намного больше других. Это приведет к исчерпанию места на серверах, замедлению обработки и т. д. На самом деле лучше выделить для покупателя Banapa отдельный раздел, после чего применить хеш-секционирование для распределения остальных заказчиков по всем остальным разделам.

Вот пример пользовательского объекта `Partitioner`:

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

public class BananaPartitioner implements Partitioner {

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

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

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

        if (((String) key).equals("Banana"))
            return numPartitions-1; // Banana всегда попадает в последний раздел
        // Другие записи распределяются по разделам путем хеширования
        return Math.abs(Utils.murmur2(keyBytes)) % (numPartitions - 1);
    }

    public void close() {}
}
```

❶ Интерфейс объекта секционирования включает методы `configure`, `partition` и `close`. Здесь мы реализовали только метод `partition`, хотя следовало бы передавать имя нашего особого заказчика через метод `configure`, а не зашивать его в код метода `partition`.

❷ Мы ожидаем только строковые ключи, так что в противном случае генерируем исключение.

Заголовки

Записи могут, помимо ключа и значения, включать в себя и заголовки. Заголовки записей дают вам возможность добавлять некоторые метаданные о записи Kafka, не добавляя никакой дополнительной информации к паре «ключ/значение» самой записи. Заголовки часто используются для указания источника данных в записи, а также для маршрутизации или отслеживания сообщений на основе информации заголовка без необходимости анализа самого сообщения (возможно, сообщение зашифровано и маршрутизатор не имеет прав доступа к данным).

Заголовки реализуются в виде упорядоченной коллекции пар «ключ/значение». Ключи всегда являются строками, а значениями могут быть любые сериализованные объекты — точно так же, как и значения сообщений.

Вот небольшой пример, показывающий, как добавить заголовки к записи `ProducerRecord`:

```
ProducerRecord<String, String> record =  
    new ProducerRecord<>("CustomerCountry", "Precision Products", "France");  
  
record.headers().add("privacy-level", "YOLO".getBytes(StandardCharsets.UTF_8));
```

Перехватчики

Бывают случаи, когда вы хотите изменить поведение своего клиентского приложения Kafka, не изменяя его код, возможно, потому, что хотите добавить идентичное поведение во все приложения в организации. Или, может, у вас нет доступа к исходному коду.

Перехватчик `ProducerInterceptor` в Kafka включает в себя два ключевых метода.

- `ProducerRecord<K, V> onSend(ProducerRecord<K, V> record)`. Этот метод будет вызван до того, как созданная запись будет отправлена в Kafka, — фактически до того, как она будет сериализована. Переопределяя этот метод, вы можете получить информацию об отправленной записи и даже изменить ее. Просто убедитесь, что этот метод возвращает корректную запись `ProducerRecord`. Запись, которую возвращает этот метод, будет сериализована и отправлена в Kafka.
- `void onAcknowledgement(RecordMetadata metadata, Exception exception)`. Этот метод будет вызван, если и когда Kafka ответит подтверждением на отправку. Метод не позволяет изменить ответ Kafka, но вы можете перехватить информацию об ответе.

Общие сценарии использования перехватчиков производителей включают сбор информации для мониторинга и отслеживания, дополнение сообщения стандартными заголовками, особенно для отслеживания его происхождения, и редактирование конфиденциальной информации.

Вот пример очень простого производителя-перехватчика. Он просто подсчитывает отправленные сообщения и полученные подтверждения в течение определенных временных интервалов:

```
public class CountingProducerInterceptor implements ProducerInterceptor {

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

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

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

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

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

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

❶ `ProducerInterceptor` — это настраиваемый интерфейс. Вы можете переопределить метод `configure` и выполнить настройку перед вызовом любого другого метода. Этот метод получает всю конфигурацию производителя, и вы можете получить доступ к любому параметру конфигурации. В данном случае мы добавили собственную конфигурацию, на которую ссылаемся здесь.

❷ Когда запись отправлена, мы увеличиваем счетчик записей и возвращаем запись, не изменяя ее.

❸ Когда Kafka отвечает подтверждением получения, мы увеличиваем счетчик подтверждений и не должны ничего возвращать.

❹ Этот метод вызывается, когда производитель закрывается, давая нам возможность очистить состояние перехватчика. В данном случае мы закрываем созданный нами поток. Если вы открывали дескрипторы файлов, подключения к удаленным хранилищам данных или что-то подобное, это место, где можно все закрыть и избежать утечек.

Как мы упоминали ранее, перехватчики производителей могут применяться без каких-либо изменений в коде клиента. Чтобы использовать предыдущий перехватчик с `kafka-console-producer` — примером приложения, поставляемого с Apache Kafka, выполните следующие три простых шага.

1. Добавьте свой `jar` в путь к классу:

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

2. Создайте конфигурационный файл, включающий:

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

3. Запустите приложение как обычно, но не забудьте включить в него конфигурацию, которую создали на предыдущем шаге:

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

Квоты и регулирование запросов

Брокеры Kafka имеют возможность ограничивать скорость создания и потребления сообщений. Это делается с помощью механизма квотирования. В Kafka есть три типа квот: на производство, на потребление и на запрос. Квоты на производство и потребление ограничивают скорость, с которой клиенты могут отправлять и получать данные, измеряемую в байтах в секунду. Квоты запросов ограничивают процент времени, которое брокер тратит на обработку клиентских запросов.

Квоты можно применить ко всем клиентам путем установки квоты по умолчанию, к конкретным идентификаторам клиентов, конкретным пользователям или к обоим. Квоты для конкретных пользователей имеют смысл только в тех кластерах, где настроена безопасность и клиенты проходят аутентификацию.

Установленные по умолчанию квоты на производство и потребление, которые применяются ко всем клиентам, являются частью конфигурационного файла

брокера Kafka. Например, чтобы ограничить каждого производителя отправкой в среднем не более 2 Мбайт/с, добавьте в файл конфигурации брокера следующую настройку: `quota.producer.default=2M`.

Хотя это и не рекомендуется, вы также можете настроить определенные квоты для определенных клиентов, которые переопределяют квоты по умолчанию в файле конфигурации брокера. Чтобы разрешить клиенту А производить 4 Мбайт/с, а клиенту В — 10 Мбайт/с, можете использовать следующее: `quota.producer.override="clientA:4M,clientB:10M"`.

Квоты, указанные в конфигурационном файле Kafka, статичны, и вы можете изменить их только изменением конфигурации и последующим перезапуском всех брокеров. Поскольку новые клиенты могут появиться в любой момент, это очень неудобно. Поэтому обычным методом применения квот к конкретным клиентам является динамическая конфигурация, которую можно задать с помощью `kafka-config.sh` или API `AdminClient`.

Рассмотрим несколько примеров:

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

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

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

❶ Ограничение клиента С (идентифицированного по идентификатору клиента) на выдачу только 1024 байт/с.

❷ Ограничение пользователя 1 (идентифицированного аутентифицированным участником) на выдачу только 1024 байт/с и потребление только 2048 байт/с.

❸ Ограничение на потребление только 2048 байт/с всех пользователей, за исключением имеющих более конкретные настройки. Это способ динамического изменения квоты по умолчанию.

Когда клиент выберет свою квоту, брокер начнет регулировать запросы клиента, чтобы предотвратить превышение квоты. Это означает, что он будет задерживать ответы на запросы клиента. У большинства клиентов это автоматически уменьшит частоту запросов (поскольку количество активных запросов ограничено) и снизит трафик до уровня, разрешенного квотой. Чтобы защититься от ненадлежащего поведения клиентов, посылающих дополнительные запросы во время процесса регулирования, брокер отключает также канал связи с клиентом на время, необходимое для достижения соответствия квоте.

Поведение регулирования отображается клиентам через `produce-throttle-time-avg`, `produce-throttle-time-max`, `fetch-throttle-time-avg` и `fetch-throttle-time-max` — среднее и максимальное время задержки запроса на производство и запроса на выборку из-за регулирования соответственно. Обратите внимание на то, что это время может представлять собой регулирование из-за квот на производство и потребление пропускной способности, квот на время запроса или и того и другого. Другие типы клиентских запросов могут быть ограничены только из-за квот времени запроса, и они будут отображаться с помощью аналогичных показателей.



Если вы используете `async Producer.send()` и продолжаете отправлять сообщения со скоростью, превышающей скорость, которую допускает брокер (из-за квот или просто из-за устаревших возможностей), сообщения сначала будут поставлены в очередь в памяти клиента. Если скорость отправки и дальше будет превышать скорость приема сообщений, то в итоге у клиента закончится буферное пространство для хранения избыточных сообщений и он заблокирует следующий вызов `Producer.send()`. Если задержка времени ожидания недостаточна для того, чтобы позволить брокеру догнать производителя и освободить место в буфере, то `Producer.send()` выдаст исключение `TimeoutException`. В качестве альтернативы некоторые записи, уже помещенные в пакеты, будут ждать дольше, чем `delivery.time out.ms`, и истекнут, что приведет к вызову обратного вызова `send()` с исключением `TimeoutException`. Поэтому важно планировать и контролировать, чтобы гарантировать, что пропускная способность брокера со временем будет соответствовать скорости, с которой производители отправляют данные.

Резюме

Мы начали эту главу с простого примера производителя — всего десять строк кода, отправляющих события в Kafka. Расширили его за счет добавления обработки ошибок и опытов с синхронной и асинхронной отправкой. Затем изучили наиболее важные конфигурационные параметры производителя и выяснили, как они влияют на его поведение. Мы обсудили сериализаторы, которые служат для управления форматом записываемых в Kafka событий. Мы подробно рассмотрели `Avro` — один из многих способов сериализации событий, часто применяемый вместе с Kafka. В завершение главы обсудили секционирование в Kafka и привели пример продвинутой методики пользовательского секционирования.

Теперь, разобравшись с записью событий в Kafka, в главе 4 рассмотрим все, что касается чтения событий из нее.

Потребители Kafka: чтение данных из Kafka

Приложения, читающие данные из Kafka, используют объект `KafkaConsumer` для подписки на ее топики и получения из них сообщений. Чтение данных из Kafka несколько отличается от чтения данных из других систем обмена сообщениями: некоторые принципы работы и идеи здесь весьма оригинальны. Чтобы научиться использовать API потребителей, необходимо сначала разобраться с этими принципами. Мы начнем с пояснений по поводу важнейших из них, а затем рассмотрим примеры, демонстрирующие различные способы применения API потребителей для реализации приложений с разнообразными требованиями.

Принципы работы потребителей Kafka

Чтобы научиться читать данные из Kafka, нужно сначала разобраться с концепциями потребителей и групп потребителей. Мы рассмотрим эти понятия в следующих разделах.

Потребители и группы потребителей

Представьте себе приложение, читающее данные из топика Kafka, проверяющее их и записывающее результаты в другое хранилище данных. Это приложение должно будет создать объект-потребитель, подписаться на соответствующий топик и приступить к получению сообщений, их проверке и записи результатов. До поры до времени такая схема будет работать, но что, если производители записывают сообщения в топик быстрее, чем приложение может их проверить? Если чтение и обработка данных ограничиваются одним потребителем, приложение, не способное справиться с темпом поступления сообщений, будет все больше и больше отставать. Понятно, что в такой ситуации нужно масштабировать получение сообщений из топиков. Необходимо, чтобы несколько потребителей

могли делить между собой данные и читать из одного топика, подобно тому как несколько производителей могут писать в один топик.

Потребители Kafka обычно состоят в *группе потребителей*. Если несколько потребителей подписаны на один топик и относятся к одной группе, все они будут получать сообщения из различных подмножеств разделов группы.

Рассмотрим топик T1 с четырьмя разделами. Предположим, что мы создали нового потребителя C1 — единственного потребителя в группе G1 — и подписали его на топик T1. C1 будет получать все сообщения из всех четырех разделов топика (рис. 4.1).

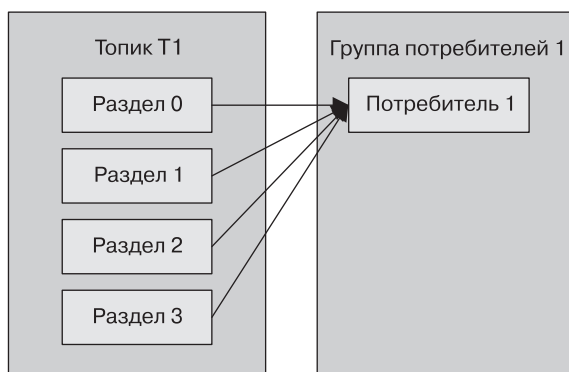


Рис. 4.1. Одна группа потребителей с четырьмя разделами

Если мы добавим в группу G1 еще один потребитель, C2, то каждый потребитель будет получать сообщения только из двух разделов. Например, сообщения из разделов 0 и 2 попадут к C1, а из разделов 1 и 3 — к C2 (рис. 4.2).

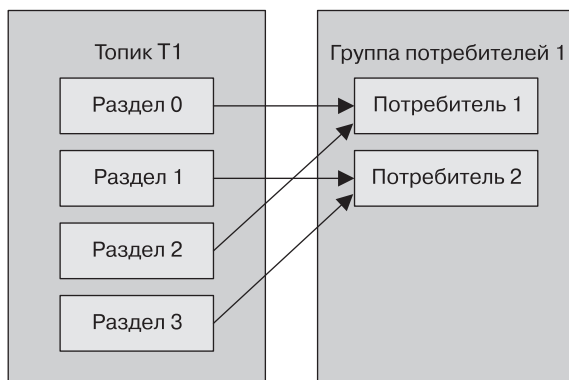


Рис. 4.2. Четыре раздела разбиты по двум потребителям в группе

Если бы в группе G1 было четыре потребителя, то каждый из них читал бы сообщения из своего раздела (рис. 4.3).

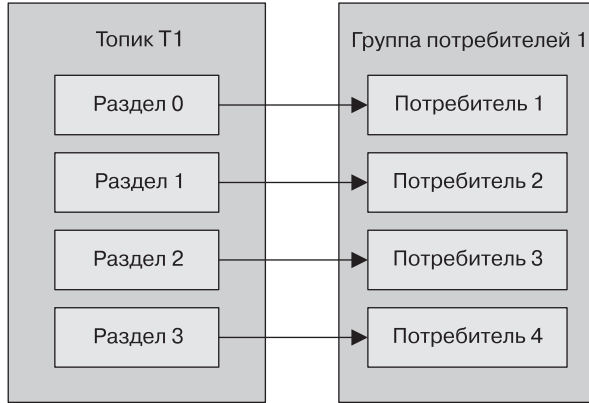


Рис. 4.3. Четыре потребителя в группе, по одному разделу на каждого

Если же в одной группе с одним топиком будет больше потребителей, чем разделов, часть потребителей будут простаивать и вообще не получать сообщений (рис. 4.4).

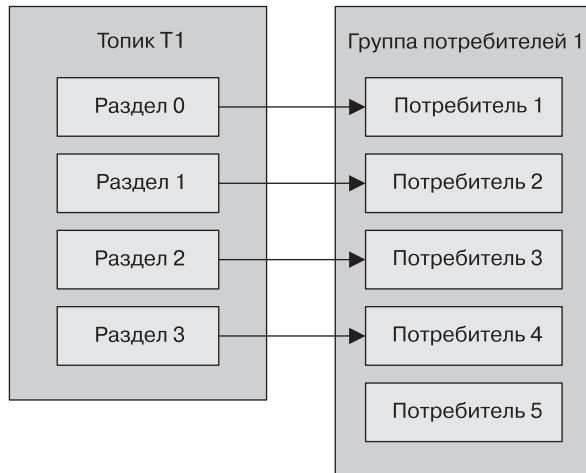


Рис. 4.4. Потребителей в группе больше, чем разделов, поэтому часть из них простаивает

Основной способ масштабирования получения данных из Kafka — добавление новых потребителей в группу. Потребители Kafka часто выполняют операции с высоким значением задержки, например запись данных в базу или занимающие

много времени вычисления с ними. В этих случаях отдельный потребитель неизбежно будет отставать от темпов поступления данных в топик, и разделение нагрузки путем добавления новых потребителей, каждый из которых отвечает лишь за часть разделов и сообщений, — основной метод масштабирования. Поэтому имеет смысл создавать топики с большим числом разделов, ведь это дает возможность добавлять новые потребители при возрастании нагрузки. Помните, что нет смысла добавлять столько потребителей, чтобы их стало больше, чем разделов в топике, — часть из них будет просто простаивать. В главе 2 мы приводили соображения по поводу выбора числа разделов в топике.

Помимо добавления потребителей для масштабирования отдельного приложения, широко распространено чтение несколькими приложениями данных из одного топика. На самом деле одной из главных задач создания Kafka было обеспечение возможности использования записанных в топике Kafka данных во множестве сценариев в организации. В подобном случае хотелось бы, чтобы каждое из приложений получило все данные, а не только их подмножество. А чтобы приложение получило все данные из топика, у него должна быть своя группа потребителей. В отличие от многих традиционных систем обмена сообщениями, Kafka масштабируется до очень больших количеств потребителей и их групп без снижения производительности.

Если мы в предыдущем примере добавим новую группу G2 с одним потребителем, этот потребитель прочитает все сообщения из топика T1 вне зависимости от группы G1. В G2 может быть свыше одного потребителя, подобно тому как было в G1, но G2 все равно получит все сообщения вне зависимости от других групп потребителей (рис. 4.5).

Резюмируем: для каждого приложения, которому нужны все сообщения из одного топика или из нескольких, создается новая группа потребителей. В существующую группу их добавляют при необходимости масштабирования чтения и обработки сообщений из топиков, так что до каждого дополнительного потребителя в группе доходит только подмножество сообщений.

Группы потребителей и перебалансировка разделов

Как мы видели в предыдущем разделе, потребители в группе делят между собой разделы топиков, на которые подписаны. Добавленный в группу потребитель начинает получать сообщения из разделов, за которые ранее отвечал другой потребитель. То же самое происходит, если потребитель останавливается или аварийно завершает работу: он покидает группу, а разделы, за которые он ранее отвечал, обрабатываются одним из оставшихся потребителей. Переназначение разделов потребителям происходит также при изменении топиков, которые читает группа (например, при добавлении администратором новых разделов).

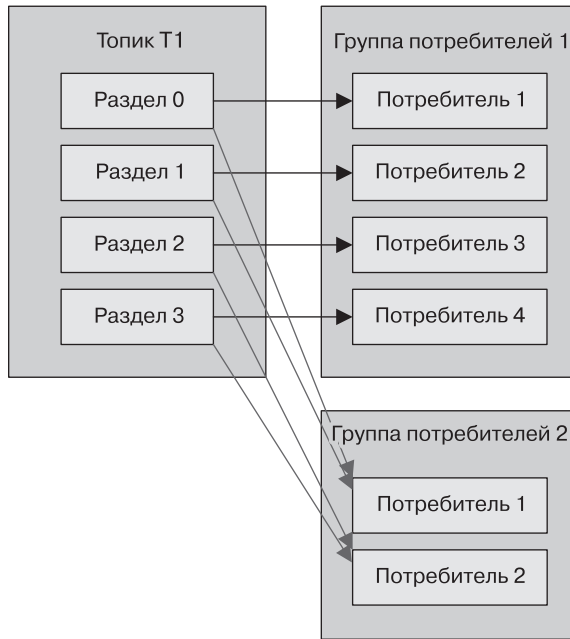


Рис. 4.5. Добавление новой группы потребителей, обе группы получают все сообщения

Передача раздела от одного потребителя другому называется перебалансировкой (rebalance). Перебалансировка важна, потому что обеспечивает группе потребителей масштабируемость и высокую доступность, позволяя легко и безопасно добавлять и удалять потребителей, но при обычных обстоятельствах она нежелательна.

Существует два типа перебалансировки в зависимости от стратегии назначения разделов, которую использует группа потребителей¹.

- *Безотлагательная перебалансировка.* В этом случае все потребители прекращают потребление, отказываются от своих прав владения всеми разделами, снова присоединяются к группе потребителей и получают совершенно новое назначение разделов. По сути, это короткое окно недоступности для всей группы потребителей. Длина окна зависит от размера группы потребителей, а также от нескольких параметров конфигурации. На рис. 4.6 показано, каким образом безотлагательная перебалансировка имеет две различные фазы: во-первых, все потребители отказываются от назначения разделов, а во-вторых, после того, как все они завершают это и снова присоединяются к группе, они получают новые назначения разделов и могут возобновить потребление.

¹ Диаграммы Софи Бли-Голдман (Sophie Blee-Goldman) из ее статьи в блоге за май 2020 года «От жаждущего к умному в перебалансировках Apache Kafka» (<https://oreil.ly/fZzac>).

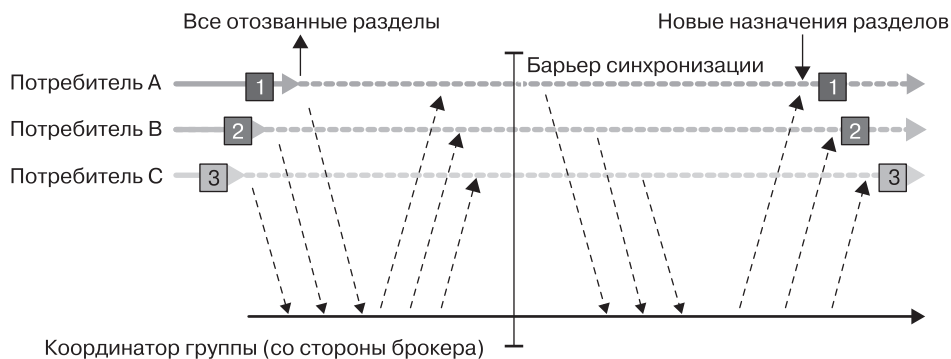


Рис. 4.6. Безотлагательная перебалансировка отзывает все разделы, приостанавливает потребление и переназначает их

- *Совместная перебалансировка.* Эта перебалансировка, называемая также инкрементной перебалансировкой, обычно включает в себя переназначение лишь небольшого подмножества разделов от одного потребителя к другому и позволяет потребителям продолжать обработку записей из всех разделов, которые не были переназначены. Это достигается путем перебалансировки в два или более этапа.

Сначала лидер группы потребителей сообщает всем потребителям, что они потеряют право владения подмножеством своих разделов, затем потребители прекращают потребление из этих разделов и отказываются от своего права владения ими. На втором этапе лидер назначает эти осиротевшие разделы их новым владельцам. Этот инкрементный подход может занять несколько итераций, пока не будет достигнуто стабильное распределение разделов, но он позволяет избежать полной недоступности, когда «останавливается весь мир», которая возникает при безотлагательном подходе. Это особенно важно для больших групп потребителей, где перебалансировка может занять значительное количество времени. На рис. 4.7 показано, что совместная перебалансировка является инкрементной и в ней участвует лишь некое подмножество потребителей и разделов.

Потребители поддерживают членство в группе и принадлежность разделов за счет отправки назначенному *координатором группы* брокеру Kafka (для разных групп потребителей это могут быть разные брокеры) периодических *контрольных сигналов* (heartbeats). Их посылает фоновый поток потребителя, и до тех пор, пока потребитель регулярно отправляет контрольные сигналы, он считается активным.

Если потребитель на длительное время прекращает отправку контрольных сигналов, время его сеанса истекает, координатор группы признает его неработающим и инициирует перебалансировку. В случае аварийного сбоя потребителя

и прекращения им обработки сообщений координатору группы хватит нескольких секунд без контрольных сигналов, чтобы признать его неработающим и инициировать перебалансировку. На протяжении этого времени никакие сообщения из относящихся к данному потребителю разделов обрабатываться не будут. Если же потребитель завершает работу штатным образом, он извещает об этом координатора группы, и тот инициирует перебалансировку немедленно, сокращая тем самым перерыв в обработке. Далее в этой главе мы обсудим параметры конфигурации, управляющие частотой отправки контрольных сигналов и длительностью времени ожидания сеанса, а также те, которые могут быть использованы для тонкой настройки поведения потребителя.

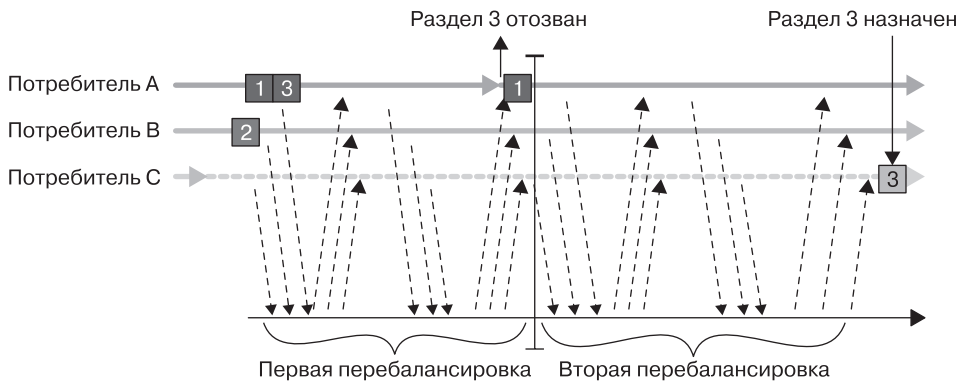


Рис. 4.7. Совместная перебалансировка приостанавливает потребление лишь для подмножества разделов, которые будут переназначены



Как происходит распределение разделов по потребителям

Когда потребитель хочет присоединиться к группе, он отправляет координатору группы запрос `JoinGroup`. Первый присоединившийся к группе потребитель становится ведущим потребителем группы. Он получает от координатора группы список всех потребителей группы, которые недавно отправляли контрольные сигналы, а значит, функционируют нормально, и отвечает за назначение потребителям подмножеств разделов. Для определения того, за какие разделы какой потребитель должен отвечать, используется реализация класса `PartitionAssignor`.

В Kafka есть несколько стратегий назначения разделов, которые мы подробнее обсудим в посвященном настройке разделе. После распределения разделов лидер группы потребителей отправляет список назначений координатору группы, который пересылает эту информацию потребителям. Каждый потребитель знает только о назначенных ему разделах. Единственный клиентский процесс, обладающий полным списком потребителей группы и назначенных им разделов, — ведущий группы. Эта процедура повторяется при каждой перебалансировке.

Статические участники группы

По умолчанию идентификация потребителя как члена потребительской группы является временной. Когда потребитель покидает потребительскую группу, разделы, которые были ему назначены, аннулируются, а когда он снова присоединяется, с помощью протокола перебалансировки ему назначается новый идентификатор участника и новый набор разделов.

Все это верно, если только вы не настроили потребителя с уникальным идентификатором `group.instance.id`, что делает его статическим участником группы. Когда потребитель впервые присоединяется к группе потребителей в качестве статического члена, обычно ему назначается набор разделов в соответствии со стратегией назначения разделов, которую использует группа. Однако, когда этот потребитель выключается, он не покидает группу автоматически — он остается ее членом до тех пор, пока его сессия не завершится. Когда потребитель снова присоединяется к группе, он распознается со своей статической идентификацией и ему переназначаются те же разделы, которые он занимал ранее, без запуска перебалансировки. Координатору группы, который кэширует назначение для каждого члена группы, не нужно запускать перебалансировку, а можно просто отправить кэш-назначение вновь присоединившемуся статическому члену.

Если два потребителя присоединятся к одной группе с одинаковым идентификатором `group.instance.id`, второй потребитель получит ошибку, говорящую о том, что потребитель с таким идентификатором уже существует.

Статическое членство в группе полезно, когда ваше приложение поддерживает локальное состояние или кэш, который заполняется разделами, назначенными каждому потребителю. Если повторное создание этого кэша занимает много времени, вы не захотите, чтобы этот процесс происходил при каждом перезапуске потребителя. В то же время важно помнить, что разделы, принадлежащие каждому потребителю, не будут переназначаться при его перезапуске. В течение определенного времени ни один потребитель не будет потреблять сообщения из этих разделов, а когда потребитель наконец перезапустится, он будет отставать от последних сообщений в этих разделах. Вы должны быть уверены, что потребитель, которому принадлежат эти разделы, сможет наверстать отставание после перезапуска.

Важно отметить, что статические члены групп потребителей не покидают группу проактивно при выключении, и определение того, когда они «действительно ушли», зависит от конфигурации `session.timeout.ms`. Вам желательно установить его значение достаточно высоким, чтобы не вызывать перебалансировку при простом перезапуске приложения, но достаточно низким, чтобы обеспечить автоматическое переназначение разделов потребителей при более длительном времени простоя и таким образом избежать больших пробелов в обработке этих разделов.

Создание потребителя Kafka

Первый шаг к получению записей — создание экземпляра класса `KafkaConsumer`. Создание экземпляра класса `KafkaConsumer` очень похоже на создание экземпляра `KafkaProducer` — необходимо просто создать экземпляр Java-класса `Properties`, содержащий свойства, которые вы хотели бы передать потребителю. Далее в этой главе мы подробнее обсудим все свойства. Для начала нам понадобятся лишь три обязательных: `bootstrap.servers`, `key.deserializer` и `value.deserializer`.

Свойство `bootstrap.servers` представляет собой строку подключения к кластеру Kafka. Оно используется так же, как и в `KafkaProducer` (за подробностями можете обратиться к главе 3). Два других свойства, `key.deserializer` и `value.deserializer`, схожи с сериализаторами для производителей, но вместо классов, преобразующих Java-объекты в байтовые массивы, необходимо задать классы, преобразующие байтовые массивы в Java-объекты.

Есть и четвертое свойство, `group.id`, которое, строго говоря, не является обязательным, но очень широко используется. Оно задает группу потребителей, к которой относится экземпляр `KafkaConsumer`. Хотя можно создавать и потребители, не принадлежащие ни к одной группе, в большей части данной главы будем предполагать, что все потребители состоят в группах.

Следующий фрагмент кода демонстрирует создание объекта `KafkaConsumer`:

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

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

Большая часть этого кода вам уже знакома, если вы читали главу 3, посвященную созданию производителей. Мы считаем, что как ключ, так и значение читаемых нами записей представляют собой объекты `String`. Единственное новое свойство тут `group.id` — название группы, к которой принадлежит потребитель.

Подписка на топики

Следующий шаг после создания потребителя — подписка его на один топик или несколько. Метод `subscribe()` всего лишь требует передачи в качестве параметра списка топиков, так что использовать его довольно просто:

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

❶ Просто создаем список, содержащий один элемент — название топика `customerCountries`.

Можно также вызвать метод `subscribe()` с регулярным выражением в качестве параметра. Это выражение может соответствовать нескольким названиям топиков, так что при создании нового топика с подходящим под это регулярное выражение названием практически тотчас же будет выполнена перебалансировка, а потребители начнут получать данные из нового топика. Это удобно для приложений, которым требуется получать данные из нескольких топиков и обрабатывать данные, содержащиеся в них. Подписка на несколько топиков с помощью регулярного выражения чаще всего используется в приложениях, реплицирующих данные между Kafka и другой системой или приложениями для обработки потоков.

Например, для подписки на все топики `test` можно выполнить следующий вызов:

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



Если ваш кластер Kafka имеет большое количество разделов, возможно, 30 000 или более, вы должны знать, что фильтрация топиков для подписки осуществляется на стороне клиента. Это означает, что, когда вы подписываетесь на подмножество топиков с помощью регулярного выражения, а не с помощью явного списка, потребитель будет запрашивать список всех топиков и их разделов у брокера через регулярные промежутки времени. Затем клиент будет использовать этот список для обнаружения новых топиков, которые он должен включить в свою подписку, и подписываться на них. Когда список топиков велик и есть много потребителей, размер списка топиков и разделов значителен, а подписка с помощью регулярного выражения приводит к серьезным накладным расходам на брокера, клиента и сеть. Бывают случаи, когда пропускная способность, используемая метаданными топика, больше, чем пропускная способность для отправки данных. Это также означает, что для подписки с помощью регулярного выражения клиенту необходимы разрешения на описание всех топиков в кластере, то есть полное разрешение на описание всего кластера.

Цикл опроса

В самой основе API потребителей лежит простой цикл опроса сервера. Основной код потребителя выглядит следующим образом:

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

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



```

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

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

```

❶ Разумеется, это бесконечный цикл. Потребители обычно представляют собой работающие в течение длительного времени приложения, непрерывно опрашивающие Kafka на предмет дополнительных данных. Далее в этой главе мы покажем, как можно аккуратно выйти из цикла и закрыть потребитель.

❷ Это важнейшая строка кода в данной главе. Как акулы должны непрерывно двигаться, чтобы не погибнуть, потребители должны опрашивать Kafka, иначе их сочтут неработающими, а разделы, откуда они получают данные, будут переданы другим потребителям группы. Передаваемый нами в метод `poll()` параметр представляет собой длительность ожидания и определяет, сколько времени будет длиться блокировка в случае недоступности данных в буфере потребителя. Если этот параметр равен 0 или если уже имеются записи, возврат из метода `poll()` произойдет немедленно, в противном случае он будет ожидать в течение указанного числа миллисекунд.

❸ Метод `poll()` возвращает список записей, каждая из которых содержит топик и раздел, из которого она поступила, смещение записи в разделе и, конечно, ключ и значение записи. Обычно по списку проходят в цикле, и записи обрабатываются по отдельности.

❹ Обработка обычно заканчивается записью результата в хранилище данных или обновлением сохраненной записи. Цель состоит в ведении текущего списка покупателей из каждого округа, так что мы обновляем хеш-таблицу и выводим результат в виде JSON. В более реалистичном примере результаты обновлений сохранялись бы в хранилище данных.

Цикл `poll` не только получает данные. При первом вызове метода `poll()` для нового потребителя он отвечает за поиск координатора группы, присоединение потребителя к группе и назначение ему разделов. Перебалансировка в случае ее запуска также выполняется в цикле опроса, включая связанные обратные вызовы. Это означает, что почти все, что может пойти не так с потребителем или

в обратных вызовах, используемых в его слушателях, скорее всего, проявится как исключение, создаваемое `poll()`.

Помните, что, если `poll()` не вызывается дольше, чем значение `max.poll.interval.ms`, потребитель станет считаться мертвым и будет вычеркнут из группы потребителей, поэтому избегайте действий, которые могут блокировать на непредсказуемые интервалы времени внутри цикла опроса.

Потокобезопасность

Несколько потребителей, относящихся к одной группе, не могут работать в одном потоке, и несколько потоков не могут безопасно использовать один и тот же потребитель. Железное правило: один потребитель на один поток. Чтобы запустить несколько потребителей в одной группе в одном приложении, вам придется запускать каждый из них в отдельном потоке. Полезно обернуть логику потребителя в его собственный объект, а затем использовать `Java ExecutorService` для запуска нескольких потоков, каждый с собственным потребителем. В блоге Confluent есть учебное пособие (<https://oreil.ly/8Y0Ve>), которое показывает, как это сделать.



В более старых версиях Kafka полная сигнатура метода была `poll(long)`, теперь же она устарела и новым API является `poll(Duration)`. В дополнение к изменению типа аргумента семантика того, как метод блокирует, незначительно изменилась. Исходный метод, `poll(long)`, будет блокироваться столько времени, сколько потребуется для получения необходимых метаданных из Kafka, даже если это больше, чем длительность тайм-аута. Новый метод, `poll(Duration)`, будет придерживаться ограничений по тайм-ауту и не станет ждать метаданных. Если у вас есть существующий код пользователя, который применяет `poll(0)` в качестве метода, заставляющего Kafka получать метаданные без потребления записей (довольно распространенный метод взлома), вы не можете просто изменить его на `poll(Duration.ofMillis(0))` и ожидать такого же поведения. Вам придется придумать новый способ достижения своих целей. Часто решением является размещение логики в методе `rebalanceListener.onPartitionAssignment()`, который гарантированно будет вызван после того, как у вас окажутся метаданные для назначенных разделов, но до того, как начнут поступать записи. Другое решение было описано Джесси Андерсоном (Jesse Anderson) в его блоге в статье «У Kafka появился абсолютно новый Poll» (<https://oreil.ly/zN6ek>).

Другой подход заключается в том, чтобы один потребитель заполнял очередь событий, а несколько рабочих потоков выполняли работу из нее. Пример можно увидеть в статье в блоге Игоря Бузатовича (Igor Buzatović) (<https://oreil.ly/uMzj1>).

Настройка потребителей

До сих пор мы сосредотачивались на изучении API потребителей, но рассмотрели лишь несколько параметров настройки — обязательные параметры `bootstrap.servers`, `group.id`, `key.deserializer` и `value.deserializer`. Все настройки потребителей описаны в документации Apache Kafka (<https://oreil.ly/Y00GI>). Значения по умолчанию большинства параметров вполне разумны и не требуют изменения, но некоторые могут серьезно повлиять на производительность и доступность потребителей. Рассмотрим наиболее важные из них.

`fetch.min.bytes`

Это свойство позволяет потребителю задавать минимальный объем данных, получаемых от брокера при извлечении записей, — по умолчанию 1 байт. Если брокеру поступает запрос на записи от потребителя, но новые записи оказываются на несколько байтов меньше, чем значение `fetch.min.bytes`, брокер будет ждать до тех пор, пока не появятся новые сообщения, прежде чем отправлять записи потребителю. Это снижает нагрузку как на потребитель, так и на брокер, ведь им приходится обрабатывать меньше перемещаемых туда и обратно сообщений при небольшом объеме новых действий в топиках или в часы пониженной активности. При слишком активном использовании CPU при небольшом количестве доступных данных или для снижения нагрузки на брокеры при большом числе потребителей лучше повысить значение этого параметра по сравнению с принятым по умолчанию. Однако имейте в виду, что рост этого значения может увеличить задержку для случаев с низкой пропускной способностью.

`fetch.max.wait.ms`

Задавая параметр `fetch.min.bytes`, вы сообщаете Kafka о необходимости подождать до того момента, когда у него будет достаточно данных для отправки, прежде чем отвечать потребителю. Параметр `fetch.max.wait.ms` позволяет управлять тем, сколько именно ждать. По умолчанию Kafka ждет 500 мс. Это приводит к дополнительной задержке до 500 мс при отсутствии достаточного объема поступающих в топик Kafka данных. Если нужно ограничить потенциальную задержку (обычно из-за соглашений об уровне предоставления услуг, определяющих максимальную задержку приложения), можно задать меньшее значение параметра `fetch.max.wait.ms`. Если установить для `fetch.max.wait.ms` 100 мс, а для `fetch.min.bytes` — 1 Мбайт, Kafka отправит данные в ответ на запрос потребителя, или когда объем возвращаемых данных достигнет 1 Мбайт, или по истечении 100 мс, в зависимости от того, что произойдет первым.

fetch.max.bytes

Это свойство позволяет указать максимальное количество байтов, которые Kafka будет возвращать каждый раз, когда потребитель опрашивает брокер (по умолчанию 50 Мбайт). Оно предназначено для ограничения размера памяти, которую потребитель будет использовать для хранения данных, возвращенных с сервера, независимо от того, сколько разделов или сообщений было возвращено. Обратите внимание на то, что записи отправляются клиенту пакетами и, если первый пакет записей, который должен отправить брокер, превышает этот размер, он будет отправлен, а ограничение — проигнорировано. Это гарантирует, что потребитель сможет продолжить работу. Стоит отметить, что существует соответствующая конфигурация брокера, которая позволяет администратору Kafka ограничить также максимальный размер выборки. Эта конфигурация брокера может быть полезна, так как запросы на большие объемы данных могут привести к большим чтениям с диска и длительным отправкам по сети, что способно вызывать конфликты и увеличить нагрузку на брокер.

max.poll.records

Это свойство определяет максимальное количество записей, которые возвращает один вызов функции `poll()`. Используйте его для управления количеством данных (но не размером данных), которые ваше приложение должно обработать за одну итерацию цикла опроса.

max.partition.fetch.bytes

Это свойство определяет максимальное число байтов, возвращаемых сервером из расчета на один раздел (по умолчанию 1 Мбайт). При возврате методом `KafkaConsumer.poll()` объекта `ConsumerRecords` объект записи будет занимать не более `max.partition.fetch.bytes` на каждый назначенный потребителю раздел. Обратите внимание на то, что управление использованием памяти с помощью этой конфигурации может быть довольно сложным, поскольку вы не можете контролировать, сколько разделов будет включено в ответ брокера. Поэтому мы настоятельно рекомендуем использовать свойство `fetch.max.bytes`, если у вас нет особых причин пытаться обрабатывать одинаковые объемы данных из каждого раздела.

session.timeout.ms и **heartbeat.interval.ms**

По умолчанию потребитель может находиться вне связи с брокерами и продолжать считаться работающим не более 10 с. Если потребитель не отправляет контрольный сигнал координатору группы в течение промежутка времени, боль-

шего, чем определено параметром `session.timeout.ms`, он считается отказавшим и координатор группы инициирует перебалансировку группы потребителей с назначением разделов отказавшего потребителя другим потребителям группы. Этот параметр тесно связан с параметром `heartbeat.interval.ms`, который задает частоту отправки потребителем Kafka контрольных сигналов координатору группы, в то время как параметр `session.timeout.ms` задает время, в течение которого потребитель может обходиться без отправки контрольного сигнала. Следовательно, эти два параметра чаще всего меняют одновременно — значение `heartbeat.interval.ms` должно быть меньше значения `session.timeout.ms` (обычно составляет треть от него). Так, если значение `heartbeat.interval.ms` составляет 3 с, то `session.timeout.ms` следует задать равным 1 с. Значение `session.timeout.ms`, меньшее, чем задано по умолчанию, позволяет группам потребителей быстрее обнаруживать отказы потребителей и восстанавливаться после них. В то же время оно может стать причиной нежелательной перебалансировки. Задание более высокого значения `session.timeout.ms` снизит вероятность случайной перебалансировки, но и для обнаружения настоящего сбоя в этом случае потребуется больше времени.

max.poll.interval.ms

Это свойство позволяет вам установить продолжительность времени, в течение которого потребитель может обходиться без опроса, прежде чем будет признан мертвым. Как упоминалось ранее, контрольные сигналы и время ожидания сеансов являются основным механизмом, с помощью которого Kafka обнаруживает мертвых потребителей и удаляет их разделы. Однако мы также упоминали, что контрольные сигналы отправляются фоновым потоком. Существует вероятность того, что основной поток, потребляющий данные из Kafka, находится в состоянии взаимоблокировки (*deadlock*), но фоновый поток продолжает посылать контрольные сигналы. Это означает, что записи из разделов, принадлежащих этому потребителю, не обрабатываются. Самый простой способ узнать, продолжает ли потребитель обрабатывать записи, — это проверить, запрашивает ли он новые записи. Однако интервалы между запросами новых записей трудно предсказать, они зависят от объема доступных данных, типа обработки, выполняемой потребителем, а иногда и от задержки дополнительных служб. В приложениях, которым необходимо выполнять трудоемкую обработку каждой возвращаемой записи, свойство `max.poll.records` используется для ограничения объема возвращаемых данных и, следовательно, для ограничения продолжительности времени до того, как приложение станет доступно для повторного запроса `poll()`. Даже если `max.poll.records` определено, интервал между вызовами функции `poll()` трудно предсказать, поэтому свойство `max.poll.interval.ms` используется как отказоустойчивый и запасной вариант. Интервал должен быть достаточно большим, чтобы его очень редко достигал

здоровый потребитель, но и в то же время достаточно низким, чтобы избежать значительного воздействия со стороны зависшего потребителя. Значение по умолчанию составляет 5 мин. Когда истечет время ожидания, фоновый поток отправит запрос «покинуть группу», чтобы сообщить брокеру, что потребитель мертв и группа должна восстановить баланс, а затем прекратит отсылку контрольных сигналов.

default.api.timeout.ms

Это тайм-аут, который будет применяться почти ко всем вызовам API, выполняемым потребителем, если вы не указали явный тайм-аут при вызове API. По умолчанию это значение равно 1 мин, и, поскольку оно больше, чем тайм-аут по умолчанию, оно будет включать повторную попытку, когда это необходимо. Заметным исключением из API, использующих это значение по умолчанию, является метод `poll()`, который всегда требует явного тайм-аута.

request.timeout.ms

Это максимальное количество времени, в течение которого потребитель будет ожидать ответа от брокера. Если брокер не отвечает в течение этого времени, клиент будет считать, что он вообще не отвечает, закроет соединение и попытается подключиться снова. По умолчанию в этой конфигурации установлено значение 30 с, и рекомендуется не уменьшать его. Важно предоставить брокеру достаточно времени для обработки запроса, прежде чем отказаться, — повторная отправка запросов уже перегруженному брокеру мало что даст, а процесс разъединения и повторного соединения лишь увеличит накладные расходы.

auto.offset.reset

Этот параметр управляет поведением потребителя при начале чтения раздела, для которого у него зафиксированное смещение отсутствует или стало некорректным (например, вследствие слишком продолжительного бездействия потребителя, приведшего к удалению записи с этим смещением с брокера). Значение по умолчанию — `latest` («самое позднее»). Это значит, что в отсутствие корректного смещения потребитель начинает читать самые свежие записи (сделанные после начала его работы). Альтернативное значение — `earliest` («самое раннее»), при котором в отсутствие корректного смещения потребитель читает все данные из раздела с начала. Установка параметру `auto.offset.reset` значения `none` приведет к возникновению исключения при попытке использования с недопустимым смещением.

enable.auto.commit

Данный параметр определяет, будет ли потребитель фиксировать смещения автоматически, и по умолчанию равен `true`. Если вы предпочитаете контролировать, когда фиксируются смещения, установите для него значение `false`. Это нужно для того, чтобы уменьшить степень дублирования и избежать пропущенных данных. При значении `true` этого параметра имеет смысл задать также частоту фиксации смещений с помощью параметра `auto.commit.interval.ms`. Далее в этой главе мы более подробно обсудим различные варианты фиксации смещений.

partition.assignment.strategy

Мы уже знаем, что разделы распределяются по потребителям в группе. Класс `PartitionAssignor` определяет (при заданных потребителях и топиках, на которые они подписаны), какие разделы к какому потребителю будут относиться. По умолчанию в Kafka есть следующие стратегии распределения.

- *Диапазонная (Range)*. Каждому потребителю присваиваются последовательные подмножества разделов из топиков, на которые он подписан. Так что, если потребители C1 и C2 подписаны на топики T1 и T2, оба по три раздела, то потребителю C1 будут назначены разделы 0 и 1 из топиков T1 и T2, а C2 — раздел 2 из топиков T1 и T2. Поскольку в каждом из топиков нечетное число разделов, а их распределение по потребителям выполняется для каждого топика отдельно, у первого потребителя окажется больше разделов, чем у второго. Подобное происходит всегда, когда используется диапазонная стратегия распределения, а число потребителей не делится нацело на число разделов в каждом топике.
- *Циклическая (RoundRobin)*. Все разделы от всех подписанных топиков распределяются по потребителям последовательно, один за другим. Если бы описанные ранее потребители C1 и C2 использовали циклическое распределение, C1 были бы назначены разделы 0 и 2 из топика T1 и раздел 1 из топика T2, а C2 — раздел 1 из топика T1 и разделы 0 и 2 из топика T2. Когда все потребители подписаны на одни и те же топики (очень распространенный сценарий), циклическое распределение дает одинаковое количество разделов у всех потребителей или в крайнем случае различие в один раздел.
- *«Липкая» (Sticky)*. «Липкий» распределитель преследует две цели: первая состоит в том, чтобы получить максимально сбалансированное назначение, а вторая — в том, чтобы в случае перебалансировки оставить как можно больше назначений на месте, минимизируя накладные расходы, связанные с перемещением назначений разделов от одного потребителя

к другому. В общем случае, когда все потребители подписаны на один и тот же топик, начальное назначение от «липкого» распределителя будет таким же сбалансированным, как и от циклического распределителя. Последующие назначения будут такими же сбалансированными, но уменьшат количество перемещений разделов. В случаях, когда потребители в одной группе подписаны на разные топики, распределение, достигнутое «липким» распределителем, будет более сбалансированным, чем у циклического распределителя.

- *Совместная «липкая» (Cooperative Sticky).* Эта стратегия назначения идентична стратегии назначения «липкого» распределителя, но поддерживает совместные перебалансировки, при которых потребители могут продолжать потреблять из разделов, которые не были переназначены. Подробнее о совместной перебалансировке читайте в подразделе «Группы потребителей и перебалансировка разделов» ранее в этой главе. Если вы переходите с версии старше 2.3, то для включения совместной стратегии «липкого» назначения вам необходимо будет следовать определенному пути обновления, поэтому обратите особое внимание на руководство по обновлению (<https://oreil.ly/kMI6>).

Параметр `partition.assignment.strategy` позволяет выбирать стратегию распределения разделов. Стратегия по умолчанию — `org.apache.kafka.clients.consumer.RangeAssignor`, реализующая описанную ранее диапазонную стратегию. Ее можно заменить стратегией `org.apache.kafka.clients.consumer.RoundRobinAssignor`, `org.apache.kafka.clients.consumer.StickyAssignor` или `org.apache.kafka.clients.consumer.CooperativeStickyAssignor`. В качестве более продвинутого решения можете реализовать собственную стратегию распределения, при этом параметр `partition.assignment.strategy` должен указывать на имя вашего класса.

client.id

Значение этого параметра может быть любой строкой. В дальнейшем его будут использовать брокеры для идентификации отправленных клиентом запросов, таких как запросы на получение. Применяется при журналировании и для показателей, а также при задании квот.

client.rack

По умолчанию потребители получают сообщения от ведущей реплики каждого раздела. Однако, когда кластер охватывает несколько центров обработки данных или несколько зон доступности облака, есть преимущества как в производительности, так и в стоимости получения сообщений от реплики, расположенной

в той же зоне, что и потребитель. Чтобы включить выборку из ближайшей реплики, вам необходимо задать настройку `client.rack` и определить зону, в которой находится клиент. Затем вы можете настроить брокеры таким образом, чтобы они заменили стандартный класс `replica.selector.class` на `org.apache.kafka.common.replica.RackAwareReplicaSelector`.

Вы также можете реализовать собственный класс `replica.selector.class` с пользовательской логикой для выбора наилучшей реплики для потребления на основе метаданных клиента и метаданных раздела.

group.instance.id

Это может быть любая уникальная строка, которая используется для предоставления потребителю статического членства в группе.

receive.buffer.bytes и send.buffer.bytes

Это размеры TCP-буферов отправки и получения, применяемых сокетами при записи и чтении данных. Если значение этих параметров равно `-1`, будут использоваться значения по умолчанию операционной системы. Рекомендуется повышать их в случае, когда производители или потребители взаимодействуют с брокерами из другого ЦОД, поскольку подобные сетевые подключения обычно характеризуются более длительной задержкой и низкой пропускной способностью сети.

offsets.retention.minutes

Это настройка брокера, но о ней важно знать из-за ее влияния на поведение потребителей. До тех пор пока в группе потребителей есть активные члены (то есть участники, которые активно поддерживают членство в группе, посылая контрольные сигналы), последнее смещение, зафиксированное группой для каждого раздела, будет храниться в Kafka, чтобы его можно было получить в случае переназначения или перезапуска. Однако, как только группа станет пустой, Kafka будет сохранять зафиксированные смещения только в течение срока, установленного данной настройкой, — семь дней по умолчанию. После удаления смещений, если группа снова станет активной, она будет вести себя как совершенно новая группа потребителей, не помня ничего из того, что она потребляла в прошлом. Обратите внимание на то, что это поведение менялось несколько раз, поэтому, если вы используете версии старше 2.1.0, проверьте документацию своей версии на предмет ожидаемого поведения.

Фиксация и смещения

Метод `poll()` при вызове возвращает записанные в Kafka данные, еще не прочитанные потребителями из нашей группы. Это означает возможность отслеживать, какие записи были прочитаны потребителями данной группы. Как уже обсуждалось, одна из отличительных черт Kafka — отсутствие отслеживания подтверждений от потребителей, подобно тому как это делают многие JMS-системы организации очередей. Вместо этого потребители могут использовать Kafka для отслеживания их позиции (смещения) в каждом из разделов.

Мы будем называть действие по обновлению текущей позиции потребителя в разделах *фиксацией смещения* (`offset commit`). В отличие от традиционных очередей сообщений Kafka не фиксирует записи по отдельности. Вместо этого потребители фиксируют последнее сообщение, которое они успешно обработали из раздела, и неявно предполагают, что все сообщения до последнего также были успешно обработаны.

Каким образом потребители фиксируют смещение? Они отправляют сообщение в Kafka, которое обновляет специальный топик `__consumer_offsets`, содержащий смещение для каждого из разделов. Это ни на что не влияет до тех пор, пока все потребители работают нормально. Однако в случае аварийного сбоя потребителя или присоединения к группе нового потребителя это *инициирует переконфигурирование*. После нее каждому из потребителей может быть назначен набор разделов, отличный от обрабатываемого им ранее. Чтобы знать, с какого места продолжать работу, потребитель должен прочитать последнее зафиксированное смещение для каждого из разделов и продолжить оттуда.

Если зафиксированное смещение меньше смещения последнего обработанного клиентом сообщения, то расположенные между ними сообщения будут обработаны дважды (рис. 4.8).



Рис. 4.8. Повторно обрабатываемые события

Если зафиксированное смещение превышает смещение последнего фактически обработанного клиентом события, то расположенные в этом промежутке сообщения будут пропущены группой потребителей (рис. 4.9).

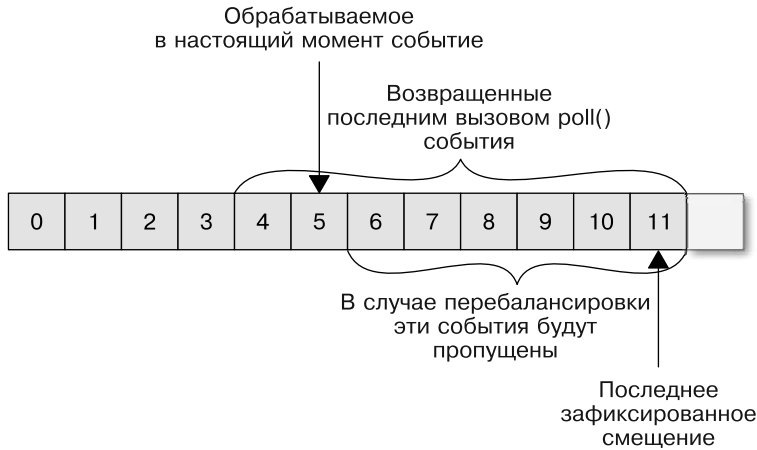


Рис. 4.9. Пропущенные события между смещениями

Очевидно, что организация смещений существенно влияет на клиентское приложение. API `KafkaConsumer` предоставляет множество способов фиксации смещений.



Какое смещение фиксируется

При фиксации смещений автоматически или без указания предполагаемых смещений по умолчанию фиксируется смещение после последнего смещения, которое было возвращено функцией `poll()`. Это важно иметь в виду, когда вы пытаетесь вручную зафиксировать определенные смещения или просите зафиксировать определенные смещения. Однако будет утомительно многократно читать: «Зафиксируйте смещение, которое на единицу больше, чем последнее смещение, полученное клиентом от `poll()`», и в 99 % случаев оно не имеет значения. Поэтому мы станем писать: «Зафиксировать последнее смещение», когда будем ссылаться на поведение по умолчанию, и если вам нужно вручную управлять смещениями, пожалуйста, помните об этом.

Автоматическая фиксация

Простейший способ фиксации смещений — делегировать эту задачу потребителю. При значении `true` параметра `enable.auto.commit` потребитель каждые 5 с будет автоматически фиксировать самое последнее смещение, возвращенное клиенту методом `poll()`. Пятисекундный интервал — значение по умолчанию,

которое можно изменить заданием параметра `auto.commit.interval.ms`. В основе автоматической фиксации лежит цикл опроса. При каждом опросе потребитель проверяет, не пора ли выполнить фиксацию, и, если да, фиксирует возвращенные при последнем опросе смещения.

Прежде чем воспользоваться этой удобной возможностью, нужно четко представить себе последствия.

Учтите, что по умолчанию автоматическая фиксация происходит каждые 5 с. Предположим, после последней фиксации прошло 3 с и наш потребитель вышел из строя. После перебалансировки выжившие потребители начнут потреблять разделы, которые ранее принадлежали вышедшему из строя брокеру. Но они будут получать данные с последнего зафиксированного смещения. В этом случае «возраст» смещения составляет 3 с, так что все поступившие в течение этих 3 с события будут обработаны два раза. Можно настроить интервал фиксации так, чтобы она выполнялась чаще, и уменьшить окно, в пределах которого записи дублируются. Однако полностью устранить дублирование невозможно.

При включенной автофиксации, когда наступит время для фиксации смещения, следующий вызов метода `poll()` будет фиксировать последнее смещение, возвращенное предыдущим вызовом. Этот метод не знает, какие события были обработаны, так что очень важно всегда обрабатывать все возвращенные методом `poll()` события до того, как вызывать `poll()` снова (как и `poll()`, метод `close()` тоже автоматически фиксирует смещения). Обычно это не проблема, но будьте внимательны при обработке исключений или досрочном выходе из цикла опроса.

Автоматическая фиксация удобна, но она не позволяет разработчику управлять так, чтобы избежать дублирования сообщений.

Фиксация текущего смещения

Большинство разработчиков стараются жестко контролировать моменты фиксации смещений как для исключения вероятности пропуска сообщений, так и для уменьшения количества их дублирования при перебалансировке. В API потребителей есть возможность фиксировать текущее смещение в нужный разработчику приложения момент вместо фиксации по таймеру.

При задании параметра `enable.auto.commit=false` смещения будут фиксироваться только тогда, когда приложение явно потребует этого. Простейший и наиболее надежный API фиксации — `commitSync()`. Он фиксирует последнее возвращенное методом `poll()` смещение и сразу же после этого завершает выполнение процедуры, генерируя исключение в случае сбоя фиксации.

Важно помнить, что `commitSync()` зафиксирует последнее возвращенное методом `poll()` смещение, так что, вызывая `commitSync()` до завершения обработки

всех записей в наборе, вы в случае сбоя приложения рискуете пропустить сообщения, которые были зафиксированы, но не обработаны. Если приложение выходит из строя, когда все еще обрабатывает записи в наборе, все сообщения с начала самого недавнего пакета и до момента начала перебалансировки будут обработаны дважды — это может как быть, так и не быть предпочтительнее, чем пропущенные сообщения.

Далее показано, как использовать `commitSync()` для фиксации смещений после завершения обработки последнего пакета сообщений:

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

while (true) {
    ConsumerRecords<String, String> records = consumer.poll(timeout);
    for (ConsumerRecord<String, String> record : records) {
        System.out.printf("topic = %s, partition = %s, offset = %d, customer = %s, country = %s\n",
            record.topic(), record.partition(),
            record.offset(), record.key(), record.value()); ❶
    }
    try {
        consumer.commitSync(); ❷
    } catch (CommitFailedException e) {
        log.error("commit failed", e) ❸
    }
}
```

❶ Допустим, обработка записи состоит в выводе ее содержимого. Вероятно, реальное приложение будет делать с записями намного больше — модифицировать, расширять, агрегировать и отображать их на инструментальной панели или оповещать пользователей о важных событиях. Решать, когда обработка записи завершена, следует в зависимости от конкретного сценария применения.

❷ После завершения обработки всех записей текущего пакета вызываем `commitSync()` для фиксации последнего смещения, прежде чем выполнять опрос для получения дополнительных сообщений.

❸ Метод `commitSync()` повторяет фиксацию до тех пор, пока не возникнет непоправимая ошибка, которую можно разве что записать в журнал.

Асинхронная фиксация

Один из недостатков фиксации вручную — то, что приложение оказывается заблокировано, пока брокер не ответит на запрос фиксации. Это ограничивает пропускную способность приложения. Повысить ее можно за счет снижения частоты фиксации, но тем самым мы повысили бы число потенциальных дубликатов, которые могут возникать при перебалансировке.

Другой вариант — использование API асинхронной фиксации. Вместо того чтобы ждать ответа брокера на запрос фиксации, просто отправляем запрос и продолжаем работу:

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

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

❶ Фиксируем последнее смещение и продолжаем работу.

Недостаток этого подхода в том, что, в то время как `commitSync()` будет повторять попытку фиксации до тех пор, пока она не завершится успешно или не возникнет ошибка, которую нельзя исправить путем повтора, `commitAsync()` повторять попытку не станет. Причина такого поведения состоит в том, что на момент получения `commitAsync()` ответа от сервера уже может быть успешно выполнена более поздняя фиксация. Представьте себе, что мы отправили запрос на фиксацию смещения 2000. Из-за временных проблем со связью брокер не получил этого запроса и, следовательно, не ответил. Тем временем мы обработали другой пакет и успешно зафиксировали смещение 3000. Если теперь `commitAsync()` попытается выполнить неудавшуюся предыдущую фиксацию смещения, она может зафиксировать смещение 2000 *после* обработки и фиксации смещения 3000. В случае перебалансировки это приведет к дополнительным дубликатам.

Мы упомянули эту проблему и важность правильного порядка фиксаций, поскольку `commitAsync()` позволяет также передать функцию обратного вызова, применяемую при ответе брокера. Обратные вызовы часто используют для журналирования ошибок фиксаций или их подсчета в виде показателей, но, чтобы воспользоваться обратным вызовом для повторения попытки, нужно учитывать проблему с порядком фиксаций:

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

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

```

        record.topic(), record.partition(), record.offset(),
        record.key(), record.value());
    }
    consumer.commitAsync(new OffsetCommitCallback() {
        public void onComplete(Map<TopicPartition,
            OffsetAndMetadata> offsets, Exception e) {
            if (e != null)
                log.error("Commit failed for offsets {}", offsets, e);
        }
    }); ❶
}

```

❶ Отправляем запрос на фиксацию и продолжаем работу, но в случае сбоя фиксации записываем в журнал информацию о сбое и соответствующие смещения.



Повтор асинхронной фиксации

Простой способ обеспечить правильный порядок при асинхронных повторях — использовать монотонно возрастающий порядковый номер. Увеличивайте порядковый номер при каждой фиксации и вставьте его во время фиксации в обратный вызов `commitAsync`. Когда будете готовы отправить повторный запрос, проверьте, равен ли порядковый номер фиксации в обратном вызове переменной экземпляра. Если да, то более поздняя фиксация не выполнялась и можно спокойно пробовать отправить запрос еще раз. Если же значение переменной экземпляра больше, не нужно пробовать повторно отправлять запрос, потому что уже был сделан более поздний запрос на фиксацию.

Сочетание асинхронной и синхронной фиксации

При обычных обстоятельствах случайные сбои при фиксации (без повторных запросов) — незначительная помеха, ведь если проблема временная, то следующая фиксация будет выполнена успешно. Но если мы знаем, что речь идет о последней фиксации перед закрытием потребителя или перебалансировкой, то лучше позаботиться, чтобы она точно оказалась успешной.

Поэтому часто непосредственно перед остановом комбинируют `commitAsync()` с `commitSync()`. Вот таким образом (мы обсудим фиксацию перед перебалансировкой в разделе о прослушивании на предмет перебалансировки):

```

Duration timeout = Duration.ofMillis(100);

try {
    while (!closing) {
        ConsumerRecords<String, String> records = consumer.poll(timeout);
        for (ConsumerRecord<String, String> record : records) {

```

```

        System.out.printf("topic = %s, partition = %s, offset = %d,
            customer = %s, country = %s\n",
            record.topic(), record.partition(),
            record.offset(), record.key(), record.value());
    }
    consumer.commitAsync(); ❶
}
consumer.commitSync(); ❷
} catch (Exception e) {
    log.error("Unexpected error", e);
} finally {
    consumer.close();
}

```

❶ Пока все нормально, мы используем `commitAsync()`. Это быстрее, и если одна фиксация пройдет неудачно, то следующая сыграет роль повторной попытки.

❷ Но при закрытии никакой следующей фиксации не будет. Поэтому нужно вызвать метод `commitSync()`, который станет повторять попытки вплоть до успешного выполнения или невозможного сбоя.

Фиксация заданного смещения

Фиксация последнего смещения происходит только при завершении обработки пакетов. Но что делать, если требуется выполнять ее чаще? Что делать, если метод `poll()` вернул огромный пакет и необходимо зафиксировать смещения в ходе его обработки, чтобы не пришлось обрабатывать все эти строки снова в случае перебалансировки? Просто вызвать метод `commitAsync()` или `commitSync()` нельзя, ведь они зафиксируют последнее возвращенное смещение, которое вы еще не обработали.

К счастью, API потребителей предоставляет возможность вызывать методы `commitAsync()` или `commitSync()`, передавая им ассоциативный словарь разделов и смещений, которые нужно зафиксировать. Если идет обработка пакета записей и смещение последнего полученного вами из раздела 3 в топике «покупатели» сообщения равно 5000, то можете вызвать метод `commitSync()` для фиксации смещения 5001 для раздела 3 в топике «покупатели». А поскольку потребитель может отвечать более чем за один раздел, придется отслеживать смещения во всех его разделах, что приведет к дополнительному усложнению кода.

Вот так выглядит фиксация заданных смещений:

```

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

```



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

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

❶ Этот ассоциативный словарь будем использовать для отслеживания смещений вручную.

❷ Напомним: `printf` здесь лишь заглушка для реальной обработки получаемых записей.

❸ После чтения каждой записи обновляем ассоциативный словарь смещений, указывая смещение следующего намеченного для обработки сообщения. Зафиксированное смещение всегда должно быть смещением следующего сообщения, которое будет прочитано вашим приложением. Именно с этого места мы начнем чтение в следующий раз.

❹ Здесь мы решили фиксировать текущие смещения через каждые 1000 записей. В своем приложении можете выполнять фиксацию по времени или, возможно, на основе содержимого записей.

❺ Мы решили вызвать здесь метод `commitAsync()` (без обратного вызова, поэтому второй параметр равен `null`), но `commitSync()` тоже прекрасно подошел бы. Конечно, при фиксации конкретных смещений необходимо выполнять всю показанную в предыдущих разделах обработку ошибок.

Прослушивание на предмет перебалансировки

Как упоминалось в предыдущем разделе, посвященном фиксации смещений, потребителю необходимо выполнить определенную чистку перед завершением выполнения, а также перед перебалансировкой разделов.

Если известно, что раздел вот-вот перестанет принадлежать данному потребителю, то желательно зафиксировать смещения последних обработанных событий. Возможно, придется также закрыть дескрипторы файлов, соединения с базой данных и т. п.

API потребителей позволяет вашему коду работать во время смены (добавления/удаления) принадлежности разделов потребителю. Для этого необходимо передать объект `ConsumerRebalanceListener` при вызове обсуждавшегося ранее метода `subscribe()`. У класса `ConsumerRebalanceListener` есть три доступных для реализации метода.

- `Public void onPartitionsAssigned(Collection<TopicPartition>partitions)` вызывается после переназначения разделов потребителю, но до того, как он начнет получать сообщения. Здесь вы подготавливаете или загружаете любое состояние, которое хотите использовать с разделом, ищете правильные смещения, если это необходимо, и т. д. Любая подготовка, выполненная здесь, должна гарантированно вернуться в течение `max.poll.timeout.ms`, чтобы потребитель мог успешно присоединиться к группе.
- `public void onPartitionsRevoked(Collection<TopicPartition> partitions)` вызывается, когда потребитель должен отказаться от разделов, которыми он ранее владел, либо в результате перебалансировки, либо при закрытии потребителя. В общем случае, когда используется алгоритм безотлагательной перебалансировки, этот метод вызывается до начала перебалансировки и после того, как потребитель перестал получать сообщения. Если используется совместный алгоритм перебалансировки, этот метод вызывается в конце перебалансировки только с тем подмножеством разделов, от которых потребитель должен отказаться. Именно здесь вам необходимо зафиксировать смещения, чтобы тот потребитель, который получит этот раздел следующим, знал, с чего начать.
- `public void onPartitionsLost(Collection partitions)` вызывается только при использовании совместного алгоритма перебалансировки и только в исключительных случаях, когда разделы были назначены другим потребителям без предварительного отзыва алгоритмом перебалансировки (в обычных случаях будет вызвана функция `onPartitionsRevoked()`). Здесь вы очищаете все состояния или ресурсы, которые использовались с этими разделами. Обратите внимание: это требуется делать аккуратно — новый владелец разделов уже может сохранить собственное состояние и вам нужно будет избежать конфликтов. А если вы не реализуете этот метод, вместо него будет вызван `onPartitionsRevoked()`.



Если вы используете алгоритм совместной переконфигурации, обратите внимание на следующее.

- Метод `onPartitionsAssigned()` будет вызываться при каждой переконфигурации как способ уведомления потребителя о том, что она произошла. Однако, если нет новых разделов, назначенных пользователю, он будет вызван с пустой коллекцией.
- Метод `onPartitionsRevoked()` будет вызываться в обычных условиях переконфигурации, но только в том случае, если потребитель отказался от права владения разделами. Он не будет вызываться с пустой коллекцией.
- Метод `onPartitionsLost()` будет вызываться в исключительных условиях переконфигурации, и к моменту его вызова у разделов в коллекции уже будут новые владельцы.

Если вы реализовали все три метода, то вам гарантируется, что во время обычной переконфигурации функция `onPartitionsAssigned()` будет вызвана новым владельцем разделов, которые будут переназначены только после того, как предыдущий владелец выполнит функцию `onPartitionsRevoked()` и откажется от своего права владения.

Этот пример показывает, как использовать метод `onPartitionsRevoked()` для фиксации смещений перед сменой принадлежности раздела:

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

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

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

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

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

```

        record.topic(), record.partition(), record.offset(),
        record.key(), record.value());
    currentOffsets.put(
        new TopicPartition(record.topic(), record.partition()),
        new OffsetAndMetadata(record.offset()+1, null));
    }
    consumer.commitAsync(currentOffsets, null);
}
} catch (WakeupException e) {
    // Игнорируем, поскольку закрываемся
} catch (Exception e) {
    log.error("Unexpected error", e);
} finally {
    try {
        consumer.commitSync(currentOffsets);
    } finally {
        consumer.close();
        System.out.println("Closed consumer and we are done");
    }
}
}

```

- ❶ Начинаем с реализации класса `ConsumerRebalanceListener`.
- ❷ В этом примере при назначении нового раздела не требуется ничего делать, мы просто начинаем получать сообщения.
- ❸ Но когда потребитель вот-вот потеряет раздел из-за перебалансировки, необходимо зафиксировать смещения. Мы фиксируем смещения для всех разделов, а не только тех, которые потеряем, — раз смещения относятся к уже обработанным событиям, никакого вреда это не принесет. И мы используем метод `commitSync()` для гарантии фиксации смещений до перебалансировки.
- ❹ Самое главное — передаем объект `ConsumerRebalanceListener` в метод `subscribe()` для вызова потребителем.

Получение записей с заданными смещениями

До сих пор мы использовали метод `poll()`, чтобы запустить получение сообщений с последнего зафиксированного смещения в каждом из разделов и дальнейшую обработку всех сообщений по очереди. Однако иногда необходимо начать чтение с другого смещения. Kafka предлагает множество методов, которые приводят к тому, что следующий метод `poll()` начинает потреблять данные с другим смещением.

Если вы хотели бы начать чтение всех сообщений с начала раздела или пропустить все разделы до конца и получать только новые сообщения, можно применить

специализированные API `seekToBeginning(Collection<Topic Partition> tp)` и `seekToEnd(Collection<Topic Partition> tp)`.

API Kafka дает возможность переходить к конкретному смещению. Ее можно использовать для множества различных целей: например, приложение, чувствительное ко времени, может пропустить вперед несколько записей, если оно отстает, или потребитель, записывающий данные в файл, может быть возвращен назад к определенному моменту времени, чтобы восстановить данные в случае потери файла.

Вот краткий пример того, как установить текущее смещение для всех разделов на записи, сделанные в определенный момент времени:

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

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

❶ Мы создаем карту из всех разделов, назначенных этому потребителю (с помощью метода `consumer.assignment()`), до метки времени, к которой мы хотим вернуть потребителя.

❷ Затем получаем смещения, которые были актуальны в этих временных метках. Этот метод отправляет запрос брокеру, где индекс временной метки используется для возврата соответствующих смещений.

❸ Наконец, мы сбрасываем смещение для каждого раздела на смещение, которое было возвращено на предыдущем шаге.

Выход из цикла

Ранее в этой главе при обсуждении цикла опроса мы советовали не волноваться по поводу выполнения опроса в бесконечном цикле, потому что скоро расскажем, как аккуратно выйти из этого цикла. Что ж, поговорим об этом.

Когда вы решите выключить потребитель и захотите немедленно выйти, не смотря на то что потребитель может ожидать длинный цикл метода `poll()`, вам понадобится еще один поток выполнения для вызова метода `consumer.wakeup()`.

Если цикл опроса выполняется в главном потоке, это можно сделать из потока `ShutdownHook`. Отметим, что `consumer.wakeup()` — единственный метод потребителя, который можно безопасно вызывать из другого потока. Вызов метода `wakeup()` приведет к завершению выполнения метода `poll()` с генерацией исключения `WakeUpException`. А если `consumer.wakeup()` был вызван в момент, когда поток не ожидает опроса, то исключение будет вызвано при вызове метода `poll()` во время следующей итерации. Обработать исключение `WakeUpException` не требуется, но перед завершением выполнения потока нужно вызвать `consumer.close()`. Закрытие потребителя приведет при необходимости к фиксации смещений и отправке координатору группы сообщения о том, что потребитель покидает группу. Координатор группы сразу же инициирует переконфигурирование, и вам не придется ждать истечения времени сеанса для назначения разделов закрываемого потребителя другому потребителю из данной группы.

Если потребитель работает в главном потоке приложения, то код завершения его выполнения выглядит следующим образом (пример немного сокращен, полный вариант можно найти на GitHub (<http://bit.ly/2u47e9A>)):

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

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

try {
    // Выполняем цикл вплоть до нажатия Ctrl+C,
    // об очистке при завершении выполнения
    // позаботится ShutdownHook
    while (true) {
        ConsumerRecords<String, String> records =
            movingAvg.consumer.poll(timeout);
        System.out.println(System.currentTimeMillis() +
            "-- waiting for data...");
        for (ConsumerRecord<String, String> record : records) {
            System.out.printf("offset = %d, key = %s, value = %s\n",
                record.offset(), record.key(), record.value());
        }
        for (TopicPartition tp: consumer.assignment())
            System.out.println("Committing offset at position:" +
                consumer.position(tp));
    }
}
```

```
        movingAvg.consumer.commitSync();
    }
} catch (WakeupException e) {
    // Игнорируем ❸
} finally {
    consumer.close(); ❹
    System.out.println("Closed consumer and we are done");
}
```

❶ `ShutdownHook` работает в отдельном потоке, так что единственное, что можно сделать безопасно, — это вызвать `wakeup` для выхода из цикла `poll()`.

❷ Особенно длительное время ожидания опроса. Если цикл опроса довольно короткий и вы не против немного подождать перед выходом, не нужно вызывать `wakeup` — достаточно просто проверять атомарное логическое значение на каждой итерации. Длительное время ожидания опроса полезно при использовании топиков с низкой пропускной способностью — таким образом клиент использует меньше ресурсов процессора для постоянного заикливания, пока брокер не получает новых данных для возврата.

❸ В результате вызова `wakeup` из другого потока `poll` генерирует исключение `WakeupException`. Его лучше перехватить, чтобы не произошло непредвиденного завершения выполнения приложения, но ничего делать с ним не требуется.

❹ Перед завершением выполнения потребителя аккуратно закрываем его.

Десериализаторы

Как обсуждалось в предыдущей главе, для производителей Kafka требуются *сериализаторы* для преобразования объектов в отправляемые в нее байтовые массивы. А для потребителей Kafka необходимы *десериализаторы* для преобразования полученных из нее байтовых массивов в объекты Java. В предыдущих примерах мы просто считали, что ключ и значение всех сообщений — строки, и оставили в настройках потребителей сериализатор по умолчанию — `StringDeserializer`.

В главе 3, посвященной производителям Kafka, мы наблюдали сериализацию пользовательских типов данных и использование Avro и объектов `AvroSerializer` для генерации объектов Avro на основе описания схемы и последующей их сериализации при отправке сообщений в Kafka. Теперь изучим создание пользовательских десериализаторов для ваших собственных объектов, а также использование десериализаторов Avro.

Вполне очевидно, что используемый для отправки событий в Kafka сериализатор должен соответствовать десериализатору, применяемому при их получении

оттуда. Ничего хорошего из сериализации с помощью `IntSerializer` с последующей десериализацией посредством `StringDeserializer` не выйдет. Это значит, что, как разработчик, вы должны отслеживать, какие сериализаторы использовались для записи в каждый из топиков, и гарантировать, что топики содержат только такие данные, которые понятны используемым вами десериализаторам. Как раз в этом и состоит одно из преимуществ сериализации и десериализации с помощью `Avro` и реестра схем — `AvroSerializer` гарантирует, что все записываемые в конкретный топик данные совместимы со схемой топика, а значит, их можно будет десериализовать с помощью соответствующего десериализатора и схемы. Любые несовместимости — на стороне производителя или потребителя — легко поддаются перехвату с выводом соответствующего сообщения об ошибке, так что нет нужды в отладке байтовых массивов в поисках ошибок сериализации.

Начнем с небольшого примера написания пользовательского десериализатора, хотя этот вариант применяется реже прочих, после чего перейдем к примеру использования `Avro` для десериализации ключей и значений.

Пользовательские сериализаторы

Возьмем тот же пользовательский объект, который мы сериализовали в главе 3, и напишем для него десериализатор:

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

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

    public int getID() {
        return customerID;
    }

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

Пользовательский десериализатор выглядит следующим образом:

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

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



```
public class CustomerDeserializer implements Deserializer<Customer> { ❶

    @Override
    public void configure(Map configs, boolean isKey) {
        // настраивать нечего
    }

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

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

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

            byte[] nameBytes = new byte[nameSize];
            buffer.get(nameBytes);
            name = new String(nameBytes, 'UTF-8');

            return new Customer(id, name); ❷

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

    @Override
    public void close() {
        // закрывать нечего
    }
}
```

❶ Потребителю требуется также реализация класса `Customer`, причем как класс, так и сериализатор должны совпадать в приложении-производителе и приложении-потребителе. В большой компании со множеством потребителей и производителей, совместно работающих с данными, это представляет собой непростую задачу.

❷ Мы просто меняем логику сериализатора на противоположную ей — извлекаем идентификатор и имя покупателя из байтового массива и применяем их для формирования нужного объекта.

Использующий этот десериализатор код потребителя будет выглядеть примерно так:

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

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

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

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

Важно отметить, что реализовывать пользовательские сериализаторы и десериализаторы не рекомендуется. Такое решение приводит к сильному сцеплению производителей и потребителей, ненадежно и чревато возникновением ошибок. Лучше использовать стандартный формат сообщений, например JSON, Thrift, Protobuf или Avro. Сейчас мы рассмотрим использование десериализаторов Avro в потребителе Kafka. Основные сведения о библиотеке Apache Avro, ее схемах и их совместимости приведены в главе 3.

Использование десериализации Avro в потребителе Kafka

Предположим, что мы используем показанный в главе 3 класс `Customer`. Чтобы получать такие объекты из Kafka, необходимо реализовать примерно такое приложение-потребитель:

```
Duration timeout = Duration.ofMillis(100);
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("group.id", "CountryCounter");
props.put("key.serializer",
```

```
"org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.serializer",
    "io.confluent.kafka.serializers.KafkaAvroDeserializer"); ❶
props.put("specific.avro.reader", "true");
props.put("schema.registry.url", schemaUrl); ❷
String topic = "customerContacts"

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

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

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

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

❶ Для десериализации сообщений Avro используем класс `KafkaAvroDeserializer`.

❷ `schema.registry.url` — параметр, указывающий на место хранения схем. С его помощью потребитель может использовать зарегистрированную производителем схему для десериализации сообщения.

❸ Указываем сгенерированный класс `Customer` в качестве типа значения записи.

❹ `record.value()` представляет собой экземпляр класса `Customer`, и его можно использовать соответствующим образом.

Автономный потребитель: зачем и как использовать потребитель без группы

До сих пор мы обсуждали группы потребителей, в которых их членам автоматически назначаются разделы и которые автоматически подвергаются перебалансировке при добавлении или удалении потребителей. Обычно такое поведение — именно то, что требуется, но в некоторых случаях хочется чего-то более простого. Иногда у вас заведомо один потребитель, которому нужно всегда читать данные из всех разделов топика или из его конкретного раздела. В этом случае оснований для организации группы потребителей или перебалансировки нет, достаточно просто назначить потребителю соответствующие топик и/или разделы, получать сообщения и периодически

фиксировать смещения (хотя вам все равно нужно настроить параметр `group.id` для фиксации смещений, без вызова подписки потребитель не присоединится ни к одной группе).

Если вы точно знаете, какие разделы должен читать потребитель, то не *подписываетесь* на топик, а просто *назначаете* себе несколько разделов. Пользователь может или подписываться на топики и состоять в группе потребителей, или назначать себе разделы, но не то и другое одновременно.

Вот пример, в котором потребитель назначает себе все разделы конкретного топика и получает из них сообщения:

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

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

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

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

❶ Начинаем с запроса у кластера доступных в данном топике разделов. Если вы собираетесь получать только данные из конкретного раздела, то можете эту часть пропустить.

❷ Выяснив, какие разделы нам нужны, вызываем метод `assign()` и передаем ему их список.

Если не считать отсутствия перебалансировок и необходимости вручную искать разделы, все остальное происходит как обычно. Не забывайте, что при добавлении в топик новых разделов потребитель об этом уведомлен не будет. Об этом вам придется позаботиться самим, периодически обращаясь к `consumer.partitionsFor()` или просто перезапуская приложение при добавлении разделов.

Резюме

Мы начали главу с подробного описания групп потребителей Kafka и обеспечиваемых ими возможностей разделения работы по чтению событий из топиков между несколькими потребителями. После изложения теории привели практический пример потребителя, подписывающегося на топик и непрерывно читающего события. Затем сделали обзор важнейших параметров конфигурации потребителей и их влияния на поведение последних. Значительную часть главы мы посвятили обсуждению смещений и их отслеживания потребителями. Понимание механизма фиксации смещений потребителями чрезвычайно важно для написания надежных потребителей, так что мы не пожалели времени на объяснение различных способов сделать это. Затем обсудили дополнительные части API потребителей, перебалансировку и закрытие потребителя.

В завершение главы мы остановились на десериализаторах, применяемых потребителями для преобразования хранимых в Kafka байтовых массивов в Java-объекты, доступные для обработки приложениями. Довольно подробно обсудили десериализаторы Avro, поскольку они чаще всего используются с Kafka, хотя это лишь один из доступных типов десериализаторов.

ГЛАВА 5

Программное управление Apache Kafka

Для управления Kafka существует множество инструментов с интерфейсом командной строки (Command Line Interface, CLI) и с графическим интерфейсом пользователя (GUI), которые мы обсудим в главе 9. Но бывает, что вам нужно выполнить некоторые административные команды из своего клиентского приложения. Создание новых тем по запросу на основе пользовательского ввода или данных — это особенно широко распространенный сценарий использования: приложения Интернета вещей (IoT) часто получают события от пользовательских устройств и записывают события в топики в зависимости от типа устройства. Если производитель выпускает устройство нового типа, то либо необходимо вспомнить, что с помощью какого-то процесса нужно создать топик, либо приложение может динамически создавать новый топик, если оно получает события с нераспознанным типом устройства. Вторая альтернатива имеет свои недостатки, но отсутствие зависимости от дополнительного процесса для создания топиков является привлекательной функцией в правильных сценариях.

Apache Kafka в версии 0.11 добавил AdminClient, чтобы предоставить программный API для административных функций, которые ранее выполнялись в командной строке: составления списков, создания и удаления топиков, описания кластера, управления списками контроля доступа (ACL) и изменения конфигурации.

Вот один из примеров. Ваше приложение будет создавать события для определенного топика. Это означает, что перед созданием первого события топик должен существовать. До того как Apache Kafka добавил AdminClient, существовало несколько вариантов, ни один из которых не был особенно удобен для пользователей: вы могли перехватить исключение `UNKNOWN_TOPIC_OR_PARTITION` из метода `producer.send()` и сообщить пользователю, что ему нужно создать топик, или надеяться, что кластер Kafka, в который вы пишете, поддерживает автоматическое создание топиков, или положиться на внутренние API и бороть-

ся с последствиями отсутствия гарантий совместимости. Теперь, когда Apache Kafka предоставляет AdminClient, есть гораздо лучшее решение: используйте AdminClient, чтобы проверить, существует ли топик, и если не существует, создайте его в тот же момент.

В этой главе мы сначала сделаем обзор AdminClient, а затем перейдем к подробному описанию его использования в ваших приложениях. Сосредоточимся на наиболее часто применяемых функциях — управлении топиками, группами потребителей и конфигурацией сущностей.

Обзор AdminClient

Начиная работать с Kafka AdminClient, важно знать основные принципы его проектирования. Когда вы поймете, как был разработан AdminClient и как его следует использовать, специфика каждого метода станет гораздо более интуитивно понятной.

Асинхронный и в конечном итоге согласованный API

Возможно, самое важное, что нужно понять об AdminClient Kafka, — это то, что он является асинхронным. Каждый метод возвращает управление сразу после передачи запроса контроллеру кластера, а также возвращает один или несколько объектов Future. Объекты Future — это результат асинхронных операций, у них есть методы для проверки состояния асинхронной операции, ее отмены, ожидания ее завершения и выполнения функций после ее завершения. AdminClient Kafka оборачивает объекты Future в объекты Result, которые предоставляют методы для ожидания завершения операции и вспомогательные методы для обычных последующих операций. Например, метод `Kafka AdminClient.createTopics` возвращает объект `CreateTopicsResult`, который позволяет дождаться создания всех топиков, проверить состояние каждого из них в отдельности и получить конфигурацию конкретного топика после его создания.

Поскольку распространение метаданных от контроллера к брокерам в Kafka происходит асинхронно, Futures, возвращаемые API AdminClient, считаются завершенными, когда состояние контроллера полностью обновлено. На этом этапе не все брокеры могут быть осведомлены о новом состоянии, поэтому запрос метода `listTopics` может быть обработан брокером, который не осведомлен о новом состоянии и не будет содержать топик, созданный совсем недавно. Это свойство также называется конечной согласованностью: в конечном итоге каждый брокер будет знать о каждом топике, но мы не можем предугадать, когда именно это произойдет.

Опции

Каждый метод в `AdminClient` принимает в качестве аргумента объект `Options`, специфичный для данного метода. Например, метод `listTopics` принимает в качестве аргумента объект `ListTopicsOptions`, а метод `describeCluster` — объект `DescribeClusterOptions`. Эти объекты содержат различные настройки того, как запрос будет обрабатываться брокером. Единственная настройка, которая есть у всех методов `AdminClient`, — `timeoutMs`: она определяет, как долго клиент будет ждать ответа от кластера, прежде чем выдавать исключение `TimeoutException`. Это ограничивает время, в течение которого ваше приложение может быть заблокировано работой `AdminClient`. Другие опции задают, должен ли метод `listTopics` также возвращать внутренние топик и должен ли метод `describeCluster` возвращать то, какие операции клиент уполномочен выполнять на кластере.

Плоская иерархия

Все операции администратора, поддерживаемые протоколом Apache Kafka, реализуются непосредственно в `KafkaAdminClient`. Здесь нет иерархии объектов или пространств имен. Это немного спорно, поскольку интерфейс может быть довольно большим и, возможно, немного громоздким, но главное преимущество заключается в том, что, если вы хотите узнать, как программно выполнить любую операцию администратора в Kafka, у вас есть только один JavaDoc для поиска, а автозаполнение в интегрированной среде разработки (IDE) будет весьма удобным. Вам не придется задаваться вопросом, не пропустили ли вы нужное место для поиска. Если этого нет в `AdminClient`, значит, оно еще не реализовано (но мы приветствуем дополнения и сотрудничество!).



Если вы заинтересованы в том, чтобы внести свой вклад в развитие Apache Kafka, ознакомьтесь с нашим руководством «Как внести свой вклад» (<https://oreil.ly/8zFsJ>). Начните с небольших, не вызывающих споров исправлений ошибок и улучшений, прежде чем приступать к внесению более значительных изменений архитектуры или протокола. Также приветствуются дополнения, не связанные с кодом, такие как отчеты об ошибках, улучшение документации, ответы на вопросы и сообщения в блоге.

Дополнительные примечания

Все операции, которые изменяют состояние кластера, — создание, удаление и изменение — обрабатываются контроллером. Операции, считывающие список состояний кластера и описание, могут обрабатываться любым брокером и направляются к наименее загруженному из них (на основании того, что известно

клиенту). Это не должно повлиять на вас как на пользователя API, но это может быть полезно знать в случае неожиданного поведения, если вы заметили, что некоторые операции проходят успешно, а другие — нет, или если пытаетесь выяснить, почему операция занимает слишком много времени.

На момент написания этой главы (незадолго до ожидаемого выпуска Apache Kafka 2.5) большинство административных операций можно было выполнить либо через AdminClient, либо напрямую, изменяя метаданные кластера в ZooKeeper. Мы настоятельно рекомендуем вам никогда не использовать ZooKeeper напрямую, и, если вам это необходимо, сообщите об этом как об ошибке в Apache Kafka. Причина заключается в том, что в ближайшем будущем сообщество Apache Kafka удалит зависимость от ZooKeeper и каждое приложение, задействующее ZooKeeper непосредственно для административных операций, необходимо будет изменить. В то же время API AdminClient останется точно таким же, просто с другой реализацией внутри кластера Kafka.

Жизненный цикл AdminClient: создание, настройка и закрытие

Чтобы использовать AdminClient от Kafka, первое, что вам нужно сделать, — создать экземпляр класса AdminClient. Это довольно просто:

```
Properties props = new Properties();
props.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
AdminClient admin = AdminClient.create(props);
// TODO: Do something useful with AdminClient
admin.close(Duration.ofSeconds(30));
```

Статический метод `create` принимает в качестве аргумента объект `Properties` с параметром. Единственным обязательным параметром является URI для вашего кластера — список брокеров для подключения, разделенных запятыми. Как обычно, в производственных средах требуется указать не менее трех брокеров на случай, если один из них в данный момент недоступен. О том, как настроить безопасное и аутентифицированное соединение, мы поговорим в главе 11.

Если вы запустили AdminClient, то со временем захотите его закрыть. Важно помнить, что, когда вы вызываете метод `close`, некоторые операции AdminClient могут еще находиться в процессе выполнения. Поэтому метод `close` принимает параметр тайм-аута. После вызова этого метода вы не сможете вызывать другие методы и посылать новые запросы, но клиент будет ждать ответов до

окончания тайм-аута. По истечении тайм-аута клиент прервет все текущие операции с исключением тайм-аута и освободит все ресурсы. Вызов метода `close` без тайм-аута подразумевает, что клиент будет ждать столько, сколько потребуется для завершения всех текущих операций.

Вы, вероятно, помните из глав 3 и 4, что `KafkaProducer` и `Kafka Consumer` имеют довольно много важных параметров конфигурации. Хорошая новость в том, что `AdminClient` намного проще и в нем не так много параметров для настройки. Обо всех параметрах конфигурации можно прочитать в документации по Kafka (<https://oreil.ly/0kjKE>). Важные, на наш взгляд, параметры конфигурации описаны в следующих разделах.

client.dns.lookup

Эта конфигурация была введена в выпуске Apache Kafka 2.1.0.

По умолчанию Kafka проверяет, разрешает и создает соединения на основе имени хоста, указанного в конфигурации сервера начальной загрузки (`bootstrap server`) (и позже в именах, возвращаемых брокерами, как указано в конфигурации `advertised.listeners`). Эта простая модель работает большую часть времени, но не охватывает два важных сценария использования: использование псевдонимов DNS, особенно в конфигурации начальной загрузки, и использование одного DNS, который сопоставляется с несколькими IP-адресами. Они звучат похоже, но немного отличаются друг от друга. Давайте рассмотрим эти взаимоисключающие сценарии немного подробнее.

Использование псевдонима DNS

Предположим, у вас есть несколько брокеров со следующим соглашением об именовании: `broker1.hostname.com`, `broker2.hostname.com` и т. д. Вместо того чтобы указывать их все в конфигурации сервера начальной загрузки, которая может стать сложной в обслуживании, можно создать единый псевдоним DNS, который будет сопоставляться со всеми ними. Вы будете использовать `all-brokers.hostname.com` для начальной загрузки, поскольку неважно, какой брокер получает начальное соединение от клиентов. Все это очень удобно, за исключением случаев, когда вы используете SASL для аутентификации. В этом случае клиент будет пытаться аутентифицироваться по адресу `all-brokers.hostname.com`, но основным именем сервера будет `broker2.hostname.com`. Если имена не совпадут, SASL откажет в аутентификации (сертификат брокера может быть атакой «злоумышленник посередине») и соединение завершится ошибкой.

В этом сценарии лучше использовать `client.dns.lookup=resolve_canonical_bootstrap_servers_only`. При такой конфигурации клиент «израсходует» псевдоним DNS, и результат окажется таким же, как если бы вы включили все имена брокеров, к которым подключается псевдоним DNS, в качестве брокеров в исходный список начальной загрузки.

DNS-имя с несколькими IP-адресами

В современных сетевых архитектурах принято размещать все брокеры за прокси-сервером или балансировщиком нагрузки. Особенно часто это встречается при использовании Kubernetes, где балансировщики нагрузки необходимы, чтобы разрешить подключения из-за пределов кластера Kubernetes. В таких случаях вы не захотите, чтобы балансировщики нагрузки становились единой точкой отказа. Поэтому очень часто `broker1.hostname.com` указывает на список IP-адресов, все из которых разрешаются для балансировщиков нагрузки, и все они направляют трафик на один и тот же брокер. Эти IP-адреса также могут меняться со временем. По умолчанию клиент Kafka будет пытаться подключиться к первому IP, который разрешается именем хоста. Это означает, что, если данный IP станет недоступным, клиент не сможет подключиться, даже если брокер будет полностью доступен. Поэтому настоятельно рекомендуется использовать `client.dns.lookup=use_all_dns_ips`, чтобы клиент не упустил преимущества высокодоступного уровня балансировки нагрузки.

request.timeout.ms

Этот параметр ограничивает время, которое ваше приложение может потратить на ожидание ответа AdminClient. Сюда входит время, затраченное на повторную попытку, если клиент получает ошибку, допускающую повторную попытку.

Значение по умолчанию составляет 120 с, что довольно долго, но для некоторых операций AdminClient, особенно команд управления группами потребителей, может потребоваться некоторое время для ответа. Как мы уже говорили в разделе «Обзор AdminClient» ранее в этой главе, каждый метод AdminClient принимает объект `Options`, который может содержать значение тайм-аута, применимое конкретно к этому вызову. Если операция AdminClient находится на критическом пути для вашего приложения, вы можете использовать меньшее значение тайм-аута и обрабатывать отсутствие своевременного ответа от Kafka другим способом. Распространенным примером является то, что при первом запуске сервисы пытаются проверить существование определенных топиков, но если Kafka требует более 30 с для ответа, вы можете продолжить запуск сервера и проверить существование топиков позже или полностью пропустить проверку.

Управление основными топиками

Теперь, когда `AdminClient` создан и настроен, пришло время посмотреть, что мы можем с ним делать. Наиболее распространенным сценарием использования `AdminClient` для Kafka является управление топиками. Сюда входит получение списка, описание, создание и удаление топиков.

Давайте начнем с перечисления всех топиков в кластере:

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

Обратите внимание: метод `admin.listTopics()` возвращает объект `ListTopicsResult`, который является тонкой оберткой над коллекцией `Futures`. Обратите также внимание на то, что функция `topics.name()` возвращает `Future` набора `name`. Когда мы вызываем функцию `get()` для этого `Future`, выполняющий поток будет ждать, пока сервер не ответит с набором имен топиков или мы получим исключение по тайм-ауту. Получив список, мы выполним итерацию по нему, чтобы вывести все названия топиков.

Теперь попробуем нечто более амбициозное: проверить, существует ли топик, и создать его, если его нет. Один из способов проверить, существует ли конкретный топик, — это получить список всех топиков и проверить, есть ли в нем нужный вам. На большом кластере это может быть неэффективно. Кроме того, иногда требуется не только проверить, существует ли топик, но и убедиться, что он имеет нужное количество разделов и реплик. Например, `Kafka Connect` и `Confluent Schema Registry` используют топик Kafka для хранения конфигурации. При запуске они проверяют, существует ли топик конфигурации, имеет ли он точно один раздел, чтобы гарантировать, что изменения конфигурации будут поступать в строгом порядке, имеет ли он три реплики, чтобы гарантировать доступность, и является ли топик сжатым, чтобы старая конфигурация сохранялась в течение неограниченного времени:

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

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

    if (topicDescription.partitions().size() != NUM_PARTITIONS) { ❸
        System.out.println("Topic has wrong number of partitions. Exiting.");
        System.exit(-1);
    }
} catch (ExecutionException e) { ❹
```

```
// exit early for almost all exceptions
if (! (e.getCause() instanceof UnknownTopicOrPartitionException)) {
    e.printStackTrace();
    throw e;
}

// if we are here, topic doesn't exist
System.out.println("Topic " + TOPIC_NAME +
    " does not exist. Going to create it now");
// Note that number of partitions and replicas is optional. If they are
// not specified, the defaults configured on the Kafka brokers will be used
CreateTopicsResult newTopic = admin.createTopics(Collections.singletonList(
    new NewTopic(TOPIC_NAME, NUM_PARTITIONS, REP_FACTOR))); ❸

// Check that the topic was created correctly:
if (newTopic.numPartitions(TOPIC_NAME).get() != NUM_PARTITIONS) { ❹
    System.out.println("Topic has wrong number of partitions.");
    System.exit(-1);
}
}
```

❶ Чтобы проверить, существует ли топик с правильной конфигурацией, вызываем `describeTopics()` со списком имен топиков, которые хотим проверить. Она возвращает объект `DescribeTopicResult`, который содержит карту имен топиков в описаниях `Future`.

❷ Мы уже видели, что, если дождаться завершения `Future`, то с помощью `get()` можно получить желаемый результат, в данном случае `TopicDescription`. Но есть также вероятность того, что сервер не сможет правильно выполнить запрос: если топик не существует, сервер не сможет ответить с его описанием. В этом случае сервер отправит ответ с ошибкой, а `Future` завершится выдачей исключения `ExecutionException`. Фактическая ошибка, отправленная сервером, будет причиной исключения. Поскольку мы хотим обработать случай, когда топик не существует, обрабатываем эти исключения.

❸ Если топик существует, `Future` завершается возвратом `TopicDescription`, который содержит список всех разделов топика, а для каждого раздела, в котором брокер является лидером, — список реплик и список синхронизированных реплик. Обратите внимание на то, что сюда не входит конфигурация топика. Мы обсудим конфигурацию позже в этой главе.

❹ Обратите внимание: все объекты результатов `AdminClient` выдают `ExecutionException`, когда Kafka отвечает с ошибкой. Это происходит потому, что результаты `AdminClient` обернуты объектами `Future`, а те обертываются исключения. Вам всегда нужно исследовать причину `ExecutionException`, чтобы получить ошибку, которую вернула Kafka.

❷ Если топик не существует, мы создаем новый. При этом можно указать только название и использовать значения по умолчанию для всех деталей. Можно также указать количество разделов, количество реплик и конфигурацию.

❸ Наконец, нам нужно дождаться возврата создания топика и, возможно, проверить результат. В данном примере проверяем количество разделов. Поскольку при создании топика мы указали количество разделов, то уверены, что оно правильное. Проверка результата более распространена, если при создании топика вы полагались на параметры по умолчанию брокера. Обратите внимание: поскольку мы снова вызываем `get()` для проверки результатов `CreateTopic`, этот метод может выдать исключение. Исключение `TopicExists` часто встречается в данном сценарии, и нам нужно его обработать (возможно, описав топик, чтобы проверить правильность конфигурации).

Теперь, когда у нас есть топик, давайте удалим его:

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

// Check that it is gone. Note that due to the async nature of deletes,
// it is possible that at this point the topic still exists
try {
    topicDescription = demoTopic.values().get(TOPIC_NAME).get();
    System.out.println("Topic " + TOPIC_NAME + " is still around");
} catch (ExecutionException e) {
    System.out.println("Topic " + TOPIC_NAME + " is gone");
}
```

На этом этапе код должен быть довольно хорошо знакомым. Мы вызываем метод `deleteTopics` со списком имен топиков для удаления и используем `get()`, чтобы дождаться завершения процесса.



Несмотря на простоту кода, необходимо помнить, что в Kafka удаление топиков является окончательным — здесь нет корзины или мусорного бака, которые помогут вам спасти удаленный топик, и нет никаких проверок, подтверждающих, что топик пуст и вы действительно хотели его удалить. Удаление неправильного топика может привести к безвозвратной потере данных, поэтому используйте этот метод с особой осторожностью.

Во всех примерах до сих пор использовался блокирующий вызов `get()` для `Future`, возвращаемого различными методами `AdminClient`. В большинстве случаев это все, что вам нужно, — операции администрирования выполняются редко, и ожидание до тех пор, пока операция не завершится успешно или не истечет тайм-аут, обычно является приемлемым. Есть одно исключение — если вы пишете на сервер, который, как ожидается, будет обрабатывать большое количество административных запросов. В этом случае вы не хотите блокировать

потоки сервера в ожидании ответа Kafka. Вы хотите продолжать принимать запросы от пользователей и отправлять их в Kafka, а когда она ответит, отправить ответ клиенту. В таких сценариях универсальность `KafkaFuture` становится весьма полезной. Вот простой пример.

```
vertx.createHttpServer().requestHandler(request -> { ❶
    String topic = request.getParam("topic"); ❷
    String timeout = request.getParam("timeout");
    int timeoutMs = NumberUtils.toInt(timeout, 1000);
    DescribeTopicsResult demoTopic = admin.describeTopics( ❸
        Collections.singletonList(topic),
        new DescribeTopicsOptions().timeoutMs(timeoutMs));

    demoTopic.values().get(topic).whenComplete( ❹
        new KafkaFuture.BiConsumer<TopicDescription, Throwable>() {
            @Override
            public void accept(final TopicDescription topicDescription,
                               final Throwable throwable) {
                if (throwable != null) {
                    request.response().end("Error trying to describe topic "
                        + topic + " due to " + throwable.getMessage()); ❺
                } else {
                    request.response().end(topicDescription.toString()); ❻
                }
            }
        })
    });
}).listen(8080);
```

❶ Мы используем `Vert.x` для создания простого сервера HTTP. Каждый раз, когда этот сервер получает запрос, он вызывает `requestHandler`, который мы здесь определяем.

❷ Запрос включает в себя имя топика в качестве параметра, и мы отвечаем описанием этого топика.

❸ Мы вызываем `AdminClient.describeTopics` как обычно и получаем в ответ обернутое `Future`.

❹ Вместо того чтобы использовать блокирующий вызов `get()`, мы создаем функцию, которая будет вызвана, когда `Future` завершится.

❺ Если `Future` завершается с исключением, мы отправляем ошибку HTTP-клиенту.

❻ Если `Future` завершается успешно, мы отвечаем клиенту описанием топика.

Ключевым моментом здесь является то, что мы не ждем ответа от Kafka. `DescribeTopicResult` отправит ответ HTTP-клиенту, когда придет ответ от Kafka. Тем временем сервер HTTP может продолжать обрабатывать другие запросы.

Вы можете проверить это поведение, используя `SIGSTOP` для приостановки Kafka (не пытайтесь сделать это в рабочей среде!) и отправив на `Vert.x` два HTTP-запроса: один с долгим тайм-аутом, другой — с коротким. Даже если вы отправили второй запрос после первого, он ответит раньше благодаря меньшему тайм-ауту и не заблокируется после первого запроса.

Управление конфигурацией

Управление конфигурацией осуществляется путем описания и обновления коллекций `ConfigResource`. Ресурсами конфигурации могут быть брокеры, регистраторы брокеров и топик. Проверка и изменение конфигурации брокеров и журналов брокеров обычно выполняются с помощью таких инструментов, как `kafka-config.sh`, или других инструментов управления Kafka, но проверка и обновление конфигурации топиков из приложений, которые их используют, — довольно распространенное явление.

Например, многие приложения полагаются на сжатые топик для корректной работы. Логично, что периодически (на всякий случай чаще, чем период хранения по умолчанию) эти приложения будут проверять, что топик действительно сжат, и предпринимать действия по исправлению его конфигурации, если это не так.

Вот пример того, как это делается:

```
ConfigResource configResource =
    new ConfigResource(ConfigResource.Type.TOPIC, TOPIC_NAME); ❶
DescribeConfigsResult configsResult =
    admin.describeConfigs(Collections.singleton(configResource));
Config configs = configsResult.all().get().get(configResource);

// print nondefault configs
configs.entries().stream().filter(
    entry -> !entry.isDefault()).forEach(System.out::println); ❷

// Check if topic is compacted
ConfigEntry compaction = new ConfigEntry(TopicConfig.CLEANUP_POLICY_CONFIG,
    TopicConfig.CLEANUP_POLICY_COMPACT);
if (!configs.entries().contains(compaction)) {
    // if topic is not compacted, compact it
    Collection<AlterConfigOp> configOp = new ArrayList<AlterConfigOp>();
    configOp.add(new AlterConfigOp(compaction, AlterConfigOp.OpType.SET)); ❸
    Map<ConfigResource, Collection<AlterConfigOp>> alterConf = new HashMap<>();
    alterConf.put(configResource, configOp);
    admin.incrementalAlterConfigs(alterConf).all().get();
} else {
    System.out.println("Topic " + TOPIC_NAME + " is compacted topic");
}
```


❶ Как упоминалось ранее, существует несколько типов `ConfigResource`, здесь мы проверяем конфигурацию для конкретного топика. В одном запросе можно указать несколько ресурсов разных типов.

❷ Результатом `describeConfigs` является карта сопоставления каждого `ConfigResource` с коллекцией конфигураций. Каждая запись конфигурации имеет метод `isDefault()`, который позволяет узнать, какие конфигурации были изменены. Конфигурация топика считается не заданной по умолчанию, если пользователь настроил топик на значение не по умолчанию или если была изменена конфигурация на уровне брокера и созданный топик унаследовал от него это значение не по умолчанию.

❸ Чтобы изменить конфигурацию, укажите карту `ConfigResource`, который вы хотите изменить, и набор операций. Каждая операция по изменению конфигурации состоит из элемента конфигурации (имя и значение конфигурации, в данном случае `cleanup.policy`, — это имя конфигурации, а `compact` — значение) и типа операции. В Kafka конфигурацию изменяют четыре типа операций: `SET` (установить), которая устанавливает значение конфигурации; `DELETE` (удалить), которая удаляет значение и сбрасывает его к значению по умолчанию; `APPEND` (добавить) и `SUBTRACT` (вычесть). Последние две операции применяются только к конфигурациям с типом `List` и позволяют добавлять и удалять значения из списка, не отправляя каждый раз весь список в Kafka.

Описание конфигурации может оказаться на удивление удобным в чрезвычайной ситуации. Мы помним случай, когда во время обновления конфигурационный файл для брокеров был случайно заменен неработающей копией. Это было выявлено после перезапуска первого брокера и обнаружения того, что его не удалось запустить. У команды не было возможности восстановить оригинал, и мы приготовились к многочисленным пробам и ошибкам, пытаясь восстановить правильную конфигурацию и вернуть брокер к жизни. Инженер по надежности сайта (SRE) спас положение, подключившись к одному из оставшихся брокеров и сбросив его конфигурацию с помощью `AdminClient`.

Управление группами потребителей

Мы уже упоминали, что, в отличие от большинства очередей сообщений, Kafka позволяет обрабатывать данные именно в том порядке, в котором они были получены и обработаны ранее. В главе 4, где обсуждались группы потребителей, мы объяснили, как использовать API потребителей, чтобы вернуться назад и повторно прочитать старые сообщения из топика. Но использование этих API означает, что вы заранее запрограммировали возможность повторной обработки

данных в своем приложении. Ваше приложение само должно предоставлять функциональность повторной обработки.

Существует несколько сценариев, в которых вы захотите заставить приложение повторно обрабатывать сообщения, даже если эта возможность не была встроена в приложение заранее. Одним из таких сценариев является устранение неполадок в работе приложения во время инцидента. Другой случай — подготовка приложения к запуску на новом кластере во время сценария аварийного восстановления после сбоя (более подробно мы поговорим об этом в главе 9, когда будем обсуждать методы аварийного восстановления).

В этом разделе мы рассмотрим, как можно использовать `AdminClient` для программного исследования и изменения групп потребителей и смещений, которые были зафиксированы этими группами. В главе 10 рассмотрим внешние инструменты для выполнения тех же операций.

Изучение групп потребителей

Если вы хотите изучить и изменить потребительские группы, первым шагом будет составление их списка:

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

Обратите внимание на то, что при использовании метода `valid()` коллекция, которую вернет `get()`, будет содержать только те группы потребителей, которые кластер вернул без ошибок, если таковые имеются. Любые ошибки будут полностью игнорироваться, а не выдаваться в качестве исключений. Для получения всех исключений можно использовать метод `errors()`. Если использовать метод `all()`, как мы делали в других примерах, то в качестве исключения будет выдана только первая ошибка, которую вернул кластер. Вероятными причинами таких ошибок являются авторизация, когда у вас нет разрешения на просмотр группы, или случаи, когда недоступен координатор для некоторых групп потребителей.

Если нам нужна дополнительная информация о некоторых группах, мы можем описать их:

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

Описание содержит большое количество информации о группе. В нее входят члены группы, их идентификаторы и хосты, назначенные им разделы, алгоритм, использованный для назначения, и хост координатора группы. Это описание очень полезно при устранении неполадок в работе групп потребителей. Но в нем отсутствует одна из самых важных частей информации о группе потребителей — нам обязательно нужно будет узнать, каким было последнее смещение, зафиксированное группой для каждого раздела, который она использует, и насколько оно отстает от последних сообщений в журнале.

В прошлом единственным способом получить эту информацию был анализ сообщений о фиксации, которые группы потребителей отправляли во внутренний топик Kafka. Хотя этот метод и выполнял свою задачу, Kafka не гарантирует совместимость форматов внутренних сообщений, поэтому использовать старый метод не рекомендуется. Мы рассмотрим, как AdminClient Kafka позволяет получить эту информацию:

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

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

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

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

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

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

❶ Мы получаем карту всех топиков и разделов, которые обрабатывает группа потребителей, и последнее зафиксированное смещение для каждой из них.

Обратите внимание на то, что, в отличие от `describeConsumerGroups`, `listConsumerGroupOffsets` принимает только одну группу потребителей, а не коллекцию.

❷ Для каждого топика и раздела в результатах мы хотим получить смещение последнего сообщения в разделе. `OffsetSpec` имеет три очень удобные реализации: `earliest()`, `latest()` и `forTimestamp()`, которые позволяют нам получить более ранние и поздние смещения в разделе, а также смещение записи, внесенной в указанное время или сразу после него.

❸ Наконец, мы перебираем все разделы и для каждого из них выводим последнее зафиксированное смещение, последнее смещение в разделе и задержку между ними.

Модификация групп потребителей

До сих пор мы просто изучали доступную информацию. В `AdminClient` также есть методы для изменения потребительских групп: удаление групп, удаление участников, удаление зафиксированных смещений и изменение смещений. Инженеры по надежности сайта обычно используют эти методы, чтобы создать специальные инструменты для восстановления после аварийных ситуаций.

Модификация смещений является наиболее полезной из всех этих функций. Удаление смещений может показаться простым способом заставить потребителя начать с нуля, но на самом деле это зависит от конфигурации потребителя: если он запускается, а смещений не найдено, начнет ли он с самого начала? Или перейдет к последнему сообщению? Пока у нас нет значения `auto.offset.reset`, мы этого не узнаем. Явное изменение зафиксированных смещений на самые ранние доступные смещения заставит потребителя начать обработку с начала топика и, по сути, приведет к «перезапуску» потребителя.

Следует помнить, что группы потребителей не получают обновлений при изменении смещений в топике смещений. Они читают смещения только тогда, когда потребителю назначается новый раздел, или при запуске. Чтобы предотвратить внесение изменений в смещения, о которых потребители не будут знать (и, следовательно, переопределять), Kafka не позволит вам изменять смещения, пока группа потребителей активна.

Также следует помнить, что, если приложение потребителя сохраняет состояние (а большинство приложений потоковой обработки делают это), сброс смещений и принуждение группы потребителей начать обработку с начала топика могут странно повлиять на сохраненное состояние. Например, у вас есть потоковое приложение, которое непрерывно подсчитывает обувь, проданную в вашем магазине, и предположим, что в 8:00 вы обнаруживаете, что во входных данных

была ошибка, и хотите полностью перепроверить счет с 3:00. Если вы сбросите смещения до 3:00 без соответствующего изменения сохраненного агрегата данных, то будете дважды считать каждую пару обуви, которая была продана сегодня (а также обработаете все данные между 3:00 и 8:00, но предположим, что это необходимо для исправления ошибки). Вам нужно позаботиться о том, чтобы соответствующим образом обновить сохраненное состояние. В среде разработки мы обычно полностью удаляем хранилище состояний перед сбросом смещений на начало топика ввода.

Учитывая эти предупреждения, рассмотрим пример:

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

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

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

❶ Чтобы сбросить группу потребителей и начать обработку с самого раннего смещения, нам нужно сначала получить самые ранние смещения. Этот процесс аналогичен получению самых последних, как показано в предыдущем примере.

❷ В этом цикле мы преобразуем карту со значениями `ListOffsetsResultInfo`, которые были возвращены `listOffsets`, в карту со значениями `OffsetAndMetadata`, которые требуются для `alterConsumerGroupOffsets`.

❸ После вызова `alterConsumerGroupOffsets` мы ждем завершения работы `Future`, чтобы увидеть, успешно ли она завершилась.

❹ Одна из наиболее распространенных причин сбоя `alterConsumerGroupOffsets` заключается в том, что мы сначала не остановили группу потребителей (это должно быть сделано непосредственным завершением работы приложения-потребителя — для завершения работы группы потребителей не существует команды администратора). Если группа все еще активна, наша попытка изменить смещения будет отображаться для координатора потребителей так, как будто клиент, не являющийся членом группы, фиксирует для нее смещение. В этом случае мы получим исключение `UnknownMemberIdException`.

Метаданные кластера

Приложению редко приходится явно узнавать что-либо о кластере, к которому оно подключилось. Вы можете создавать и потреблять сообщения, даже не узнав, сколько брокеров существует и какой из них является контроллером. Клиенты Kafka абстрагируются от этой информации — их должны интересовать только топики и разделы.

Но если вам интересно, этот небольшой фрагмент удовлетворит ваше любопытство:

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

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

❶ Идентификатор кластера представляет собой глобально уникальный идентификатор GUID и поэтому не читается человеком. Тем не менее полезно проверить, подключился ли ваш клиент к правильному кластеру.

Расширенные операции администратора

В этом разделе мы обсудим несколько методов, которые редко используются и могут быть рискованными, но невероятно полезными в случае необходимости. Они важны в основном для инженеров по надежности сайта (SRE) во время инцидентов, но не ждите, пока что-то случится, чтобы научиться их использовать. Читайте и практикуйтесь, пока не стало слишком поздно. Обратите внимание на то, что приведенные здесь методы имеют мало общего друг с другом, за исключением того, что все они попадают в эту категорию.

Добавление разделов в топик

Обычно количество разделов в топике задается при его создании. И поскольку каждый раздел может иметь очень высокую пропускную способность, столкновение с ограничениями пропускной способности топика происходит редко. Кроме того, если сообщения в топике имеют ключи, то потребители могут предположить, что все сообщения с одним и тем же ключом всегда будут направляться в один и тот же раздел и обрабатываться в одном и том же порядке одним и тем же потребителем.

По этим причинам добавление разделов в топик требуется редко и может быть рискованным. Вам нужно убедиться, что операция не нарушит работу какого-либо приложения, которое использует данные из топика. Однако иногда вы действительно достигаете предела пропускной способности, которую можете обработать с помощью существующих разделов, и у вас не останется другого выбора, кроме как добавить несколько разделов.

Можно добавить разделы в коллекцию топиков с помощью метода `createPartitions`. Обратите внимание: если вы попытаетесь расширить несколько топиков одновременно, возможно, некоторые из них будут успешно расширены, в то время как другие завершатся неудачей:

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

❶ При расширении топиков необходимо указать общее количество разделов, которое будет иметь топик после их добавления, а не количество новых разделов.



Поскольку метод `createPartition` принимает в качестве параметра общее количество разделов в топике после добавления новых разделов, вам может понадобиться описать топик и выяснить, сколько разделов в нем существует, прежде чем его расширять.

Удаление записей из топика

Действующие законы о конфиденциальности требуют соблюдения определенных правил хранения данных. К сожалению, хотя в Kafka есть правила хранения для топиков, они не были реализованы таким образом, чтобы гарантировать соблюдение закона. Топик с правилами хранения в течение 30 дней может хранить более старые данные, если все данные помещаются в один сегмент в каждом разделе.

Метод `deleteRecords` помечает как удаленные все записи со смещениями старше, чем указано при вызове метода, и делает их недоступными для потребителей Kafka. Метод возвращает наибольшие смещения удаленных записей, так что мы можем проверить, действительно ли удаление произошло так, как ожидалось. Полная очистка с диска будет происходить асинхронно. Помните, что метод `listOffsets` можно использовать для получения смещений для записей, которые были записаны в определенное время или сразу после него. Вместе

эти методы можно применять для удаления записей старше определенного момента времени:

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

Выборы лидера

Этот метод позволяет вам инициировать два различных типа выборов лидера.

- *Выборы предпочтительного лидера.* В каждом разделе есть реплика, которая назначается предпочтительным лидером. Он является таковым, потому что, если все разделы используют свою предпочтительную реплику лидера в качестве лидера, количество лидеров на каждом брокере должно быть сбалансировано. По умолчанию Kafka каждые 5 минут будет проверять, действительно ли реплика предпочтительного лидера является лидером, и если это не так, но она имеет право стать лидером, то будет выбирать реплику предпочтительного лидера в качестве лидера. Если `auto.leader.rebalance.enable` имеет значение `false` или если вы хотите, чтобы все происходило быстрее, метод `electLeader()` может запустить этот процесс.
- *Выборы нечистого лидера.* Если ведущая реплика раздела становится недоступной, а другие реплики не имеют права становиться ведущими (обычно из-за отсутствия данных), раздел остается без ведущей реплики и, следовательно, оказывается недоступным. Один из способов решения этой проблемы — инициировать запуск выбора нечистого лидера, что означает избрание лидером реплики, которая в ином случае не имела бы права стать им. Это приведет к потере данных — все события, которые были записаны на старого лидера и не были реплицированы на нового, будут потеряны. Метод `electLeader()` также может быть использован для запуска выборов нечистого лидера.

Метод является асинхронным, что означает: даже после его успешного завершения потребуется некоторое время, пока все брокеры узнают о новом состоянии, из-за чего вызовы функции `descriptionTopics()` могут возвращать противоречивые результаты. Если вы запускаете выборы лидера для нескольких разделов, возможно, операция будет успешной для одних разделов и завершится неудачей для других:


```

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

```

❶ Выбираем предпочтительного лидера в одном разделе конкретного топика. Мы можем указать любое количество разделов и топиков. Если вы вызовете команду со значением `null` вместо набора разделов, она запустит указанный вами тип выборов для всех разделов.

❷ Если кластер находится в исправном состоянии, команда ничего не сделает. Выборы как предпочтительного, так и нечистого лидера вступают в силу только в том случае, если текущим лидером является реплика, отличная от предпочтительного лидера.

Перераспределение реплик

Иногда вам может не нравиться текущее расположение некоторых реплик. Возможно, брокер перегружен и вы хотите переместить несколько реплик. Возможно, вы хотите добавить больше реплик. Может быть, вы хотите переместить все реплики из брокера, чтобы можно было удалить машину. Или, может быть, несколько топиков настолько активные, что вам нужно изолировать их от остальной рабочей нагрузки. Во всех этих сценариях `alterPartitionReassignments` дает вам тонкий контроль над размещением каждой отдельной реплики для раздела. Имейте в виду, что перераспределение реплик с одного брокера на другой может потребовать копирования большого количества данных из одного брокера в другой. Помните о доступной пропускной способности сети и при необходимости регулируйте репликацию с помощью квот. Квоты — это конфигурация брокера, поэтому вы можете описать и обновить их с помощью `AdminClient`.

Для примера предположим, что у нас есть один брокер с идентификатором 0. Наш топик имеет несколько разделов, все с одной репликой на этом брокере. После добавления нового брокера мы хотим использовать его для хранения некоторых реплик топика. Мы собираемся назначить каждый раздел топика немного иным способом:

```

Map<TopicPartition, Optional<NewPartitionReassignment>> reassignment = new
HashMap<>();
reassignment.put(new TopicPartition(TOPIC_NAME, 0),
    Optional.of(new NewPartitionReassignment(Arrays.asList(0,1)))); ❶

```

```

reassignment.put(new TopicPartition(TOPIC_NAME, 1),
    Optional.of(new NewPartitionReassignment(Arrays.asList(1)))); ❷
reassignment.put(new TopicPartition(TOPIC_NAME, 2),
    Optional.of(new NewPartitionReassignment(Arrays.asList(1,0)))); ❸
reassignment.put(new TopicPartition(TOPIC_NAME, 3), Optional.empty()); ❹

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

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

```

❶ Мы добавили еще одну реплику в раздел 0, разместили новую реплику в новый брокер, который имеет идентификатор 1, но оставили лидера без изменений.

❷ Мы не добавляли никаких реплик в раздел 1 — просто переместили одну существующую реплику на новый брокер. Поскольку у нас есть только одна реплика, она также является лидером.

❸ Мы добавили еще одну реплику в раздел 2 и сделали ее предпочтительным лидером. При следующих выборах предпочтительного лидера лидерство перейдет к новой реплике на новом брокере. Существующая реплика станет последователем.

❹ Для раздела 3 нет текущего переназначения, но, если бы оно было, это отменило бы его и вернуло состояние к тому, каким оно было до начала операции переназначения.

❺ Можем перечислить текущие переназначения.

❻ Мы также можем распечатать новое состояние, но помните, что это может занять некоторое время, пока не появятся согласованные результаты.

Тестирование

Apache Kafka предоставляет тестовый класс `MockAdminClient`, который вы можете инициализировать с любым количеством брокеров и использовать для проверки корректности работы ваших приложений, не запуская реальный кластер Kafka, и действительно выполнять на нем административные операции. Хотя `MockAdminClient` не является частью Kafka API и поэтому может быть изменен без предупреждения, он имитирует общедоступные методы, поэтому сигнатуры методов остаются совместимыми. Есть некоторый компромисс в том, оправды-

вает ли удобство этого класса риск того, что он изменится и сломает ваши тесты, поэтому учитывайте это.

Особенно привлекательным этот тестовый класс делает то, что некоторые из распространенных методов имеют полномасштабное макетирование: вы можете создать топики с помощью `MockAdminClient`, и последующий вызов `listTopics()` выведет список топики, которые вы «создали».

Однако не все методы моделируются. Если вы используете `AdminClient` версии 2.5 или более ранней и вызовете `incrementalAlterConfigs()` из `MockAdminClient`, то получите исключение `UnsupportedOperationException`, но можете справиться с этой ситуацией, внедрив собственную реализацию.

Чтобы продемонстрировать, как проводить тестирование с помощью `MockAdminClient`, начнем с реализации класса, который обрабатывается с помощью клиента администратора и использует его для создания топики:

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

// Example of a method that will create a topic if its name starts with "test"
public void maybeCreateTopic(String topicName)
    throws ExecutionException, InterruptedException {
    Collection<NewTopic> topics = new ArrayList<>();
    topics.add(new NewTopic(topicName, 1, (short) 1));
    if (topicName.toLowerCase().startsWith("test")) {
        admin.createTopics(topics);

        // alter configs just to demonstrate a point
        ConfigResource configResource =
            new ConfigResource(ConfigResource.Type.TOPIC, topicName);
        ConfigEntry compaction =
            new ConfigEntry(TopicConfig.CLEANUP_POLICY_CONFIG,
                TopicConfig.CLEANUP_POLICY_COMPACT);
        Collection<AlterConfigOp> configOp = new ArrayList<AlterConfigOp>();
        configOp.add(new AlterConfigOp(compaction, AlterConfigOp.OpType.SET));
        Map<ConfigResource, Collection<AlterConfigOp>> alterConf =
            new HashMap<>();
        alterConf.put(configResource, configOp);
        admin.incrementalAlterConfigs(alterConf).all().get();
    }
}
```

Логика здесь несложная: `maybeCreateTopic` создаст топик, если его название начинается с `test`. Мы также изменяем конфигурацию топика, чтобы показать, как действовать в случае, когда используемый нами метод не реализован в макетном клиенте.



Мы используем фреймворк тестирования Mockito (<https://site.mockito.org>) для проверки того, что методы MockAdminClient вызываются так, как ожидается, и для замещения нереализованных методов. Mockito — это довольно простой фреймворк для макетирования с хорошими API, что делает его подходящим для небольшого примера модульного теста.

Начнем тестирование с создания экземпляра имитированного клиента:

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

    // without this, the tests will throw
    // `java.lang.UnsupportedOperationException: Not implemented yet`
    AlterConfigsResult emptyResult = mock(AlterConfigsResult.class);
    doReturn(KafkaFuture.completedFuture(null)).when(emptyResult).all();
    doReturn(emptyResult).when(admin).incrementalAlterConfigs(any()); ❷
}
```

❶ MockAdminClient создается со списком брокеров (здесь мы используем только один) и одним брокером, который будет контроллером. Брокеры — это просто идентификатор брокера, имя хоста и порт, естественно, все они ненастоящие. Никакие брокеры не будут запускаться во время выполнения этих тестов. Мы будем использовать шпионскую инъекцию агента Mockito, чтобы позже проверить правильность выполнения метода TopicCreator.

❷ Здесь используем методы doReturn от Mockito, чтобы убедиться, что макет клиента администратора не выдает исключения. Тестируемый метод ожидает объект AlterConfigResult с методом all(), который возвращает KafkaFuture. Мы убедились, что фиктивный incrementalAlterConfigs возвращает именно это.

Теперь, когда у нас есть правильно смакетированный AdminClient, можем использовать его для проверки правильности работы метода maybeCreateTopic():

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

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

❶ Название топика начинается с `test`, поэтому мы ожидаем, что функция `maybeCreateTopic()` создаст топик. Проверяем, была ли функция `createTopics()` вызвана один раз.

❷ Если название топика не начинается с `test`, мы убеждаемся, что функция `createTopics()` не вызывалась вообще.

И последнее замечание: Apache Kafka опубликовал `MockAdminClient` в тестовом `jar`-файле, поэтому убедитесь, что ваш `pom.xml` включает тестовую зависимость:

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

Резюме

`AdminClient` — это полезный инструмент, который необходимо иметь в своем наборе средств разработки Kafka. Он полезен для разработчиков приложений, которые хотят создавать топик «на лету» и проверять правильность настройки используемых ими топиков для своих приложений. Он также полезен для операторов и инженеров-технологов, которые хотят создать инструменты автоматизации на базе Kafka или нуждаются в восстановлении после инцидента. `AdminClient` имеет так много полезных методов, что инженеры-технологии считают его аналогом швейцарского армейского ножа для операций с Kafka.

В этой главе мы рассмотрели основы использования `AdminClient` Kafka: управление топиками, конфигурацией и группами потребителей, а также несколько других полезных методов, которые стоит иметь под рукой, — никогда не знаешь, когда они понадобятся.

ГЛАВА 6

Внутреннее устройство Kafka

Для промышленной эксплуатации Kafka или написания использующих ее приложений знать внутреннее устройство платформы не обязательно. Однако понимание особенностей ее работы помогает при отладке и выяснении того, почему в конкретной ситуации она ведет себя так, а не иначе. Рассмотрение всех подробностей реализации и проектных решений Kafka выходит за рамки данной книги, так что в этой главе мы остановимся на нескольких вопросах, особенно актуальных для тех, кому приходится иметь дело с этой платформой.

- Контроллер Kafka.
- Функционирование репликации Kafka.
- Обработка Kafka запросов от производителей и потребителей.
- Хранение данных в Kafka: форматы файлов и индексы.

Глубокое понимание этих вопросов особенно полезно при тонкой настройке Kafka: разбираться в механизмах, контролируемых параметрами настройки, важно для того, чтобы применять их осознанно, а не изменять хаотично.

Членство в кластере

Для поддержания списка состоящих в настоящий момент в кластере брокеров Kafka использует Apache ZooKeeper. У каждого брокера есть уникальный идентификатор (ID), задаваемый в его файле конфигурации или генерируемый автоматически. При каждом запуске процесса брокер регистрируется с этим ID в ZooKeeper посредством создания *временного узла* (ephemeral node) (<http://bit.ly/2s3MYNh>). Брокеры Kafka, контроллер и некоторые инструменты экосистемы подписываются на путь регистрации брокеров `/brokers/ids` в ZooKeeper, чтобы получать уведомления при добавлении или удалении брокеров.

Если попробовать запустить второй брокер с тем же ID, будет возвращена ошибка — новый брокер попытается зарегистрироваться, но ему это не удаст-

ся, поскольку для данного идентификатора брокера уже существует узел ZooKeeper.

При потере связи брокера с ZooKeeper (обычно в результате останова брокера, а иногда из-за нарушения связности сети или длительной паузы на сборку мусора) созданный при запуске брокера временный узел будет автоматически удален из ZooKeeper. Подписанные на список брокеров компоненты Kafka будут уведомлены об удалении брокера.

И хотя соответствующий брокеру узел удаляется при его останове, ID брокера по-прежнему присутствует в других структурах данных. Например, список реплик всех топиков (см. далее раздел «Репликация») содержит идентификаторы брокеров для реплик. Таким образом, в случае невосстановимого сбоя брокера можно запустить новый брокер с тем же ID, и он немедленно присоединится к кластеру вместо старого и получит те же разделы и топики.

Контроллер

Контроллер — это брокер Kafka, который, помимо выполнения обычных своих функций, отвечает за выбор ведущих реплик для разделов. Первый запущенный в кластере брокер становится контроллером, создавая в ZooKeeper временный узел под названием `/controller`. Остальные брокеры, запускаясь, также попытаются создать этот узел, но получают исключение «узел уже существует», в результате чего поймут, что узел-контроллер уже имеется и у данного кластера есть контроллер. Брокеры создают *таймеры ZooKeeper* (<http://www.bit.ly/2sKoTTN>) на узле-контроллере для оповещения о производимых на нем изменениях. Таким образом мы гарантируем, что у кластера в каждый заданный момент времени будет только один контроллер.

В случае останова брокера-контроллера или разрыва его соединения с ZooKeeper временный узел исчезнет. Это происходит при любом сценарии, когда клиент ZooKeeper, используемый контроллером, перестает отправлять контрольные сигналы в ZooKeeper дольше, чем значение `zookeeper.session.timeout.ms`. Когда временный узел исчезает, другие брокеры из кластера будут оповещены об этом посредством наблюдателя ZooKeeper и попытаются сами создать узел-контроллер в ZooKeeper. Первый узел, создавший контроллер, становится следующим узлом-контроллером, а остальные получают исключение «узел уже существует» и пересоздадут свои таймеры на новом узле-контроллере. Каждый вновь выбранный контроллер получает новое большее значение *номера эпохи контроллера* (`controller epoch`) с помощью операции условного инкремента ZooKeeper. Текущий *номер эпохи* контроллера известен брокерам, так что они будут игнорировать полученные от контроллера сообщения с более старым значением. Это важно, поскольку брокер контроллера может отключиться от

ZooKeeper из-за длительной паузы сборки мусора — во время нее будет избран новый контроллер. Когда предыдущий лидер возобновляет работу после паузы, он может продолжать отправлять сообщения брокерам, не зная о существовании нового контроллера, — в этом случае старый контроллер считается зомби. Эпоха контроллера в сообщении, которая позволяет брокерам игнорировать сообщения от старых контроллеров, является формой ограждения зомби.

Когда контроллер впервые появляется, он должен прочитать последнюю карту состояния реплики из ZooKeeper, прежде чем сможет начать управлять метаданными кластера и выполнять выборы лидера. В процессе загрузки используются асинхронные API и конвейерные запросы на чтение в ZooKeeper, чтобы скрыть задержки. Но даже в этом случае в кластерах с большим количеством разделов процесс загрузки может занять несколько секунд — несколько тестов и сравнений описаны в блоге Apache Kafka 1.1.0 (<https://oreil.ly/mQpl4>).

Если контроллер посредством отслеживания соответствующего пути ZooKeeper либо получения запроса `ControlledShutdownRequest` от брокера обнаруживает, что брокер покинул кластер, то понимает, что всем разделам, ведущая реплика которых находилась на этом брокере, понадобится новая ведущая реплика. Он проходит по всем требующим новой ведущей реплики разделам, выбирает ее (просто берет следующую из списка реплик этого раздела) и отправляет запрос всем брокерам, содержащим или новые ведущие реплики, или существующие ведомые реплики для данных разделов. Затем он сохраняет новое состояние в ZooKeeper (опять же используя конвейерные асинхронные запросы для уменьшения задержки), а затем отправляет запрос `LeaderAndISR` всем брокерам, которые содержат реплики для этих разделов. Запрос содержит информацию о новой ведущей и ведомым репликам для этих разделов. Эти запросы группируются для повышения эффективности, поэтому каждый запрос включает новую информацию о лидерстве для нескольких разделов, имеющих реплику на одном и том же брокере. Новая ведущая реплика знает, что должна начать обслуживать запросы на генерацию и потребление от клиентов, а ведомые — что должны приступить к репликации сообщений от новой ведущей. Поскольку каждый брокер в кластере имеет кэш метаданных `MetadataCache`, который включает в себя карту всех брокеров и всех реплик в кластере, контроллер отправляет всем брокерам информацию о смене лидера в запросе обновления метаданных `UpdateMetadata`, чтобы они могли обновить свои кэши. Практически аналогичный процесс повторяется, когда брокер запускает резервное копирование, — основное отличие заключается в том, что все реплики в брокере начинаются как последователи и должны догнать лидера, прежде чем получают право быть избранными в качестве лидеров.

Контроллер, получив информацию о присоединении брокера к кластеру, действует идентификатор брокера для выяснения того, есть ли на этом брокере

реплики. Если да, контроллер уведомляет как новый, так и уже существующие брокеры об изменении, а реплики на новом брокере приступают к репликации сообщений от имеющихся ведущих реплик.

Подытожим. Kafka использует временные узлы ZooKeeper для выбора контроллера, его уведомления о присоединении узлов к кластеру и их выходе из его состава. Контроллер отвечает за выбор ведущих разделов и реплик при обнаружении присоединения узлов к кластеру и выходе из его состава. Для предотвращения разделения полномочий, когда два узла считают себя текущим контроллером, применяется значение начала отсчета контроллера.

KRaft: новый контроллер Kafka на основе Raft

Начиная с 2019 года сообщество Apache Kafka приступило к реализации амбициозного проекта — переходу от контроллера на базе ZooKeeper к кворуму контроллеров на базе Raft. Предварительная версия нового контроллера, названного KRaft, входит в состав выпуска Apache Kafka 2.8. Выпуск Apache Kafka 3.0, запланированный на середину 2021 года, будет включать первую производственную версию KRaft, а кластеры Kafka смогут работать либо с традиционным контроллером на базе ZooKeeper, либо с KRaft.

Почему сообщество Kafka решило заменить контроллер? Ее существующий контроллер уже претерпел несколько изменений, но, несмотря на улучшения в способе, которым он использует ZooKeeper для хранения информации о топиках, разделах и репликах, стало ясно, что существующая модель не будет масштабироваться до того количества разделов, которые мы хотим, чтобы поддерживала Kafka. Изменения были вызваны несколькими известными проблемами.

- Обновления метаданных записываются в ZooKeeper синхронно, но отправляются брокерам асинхронно. Кроме того, получение обновлений от ZooKeeper происходит асинхронно. Все это приводит к крайним случаям, когда метаданные не согласуются между брокерами, контроллером и ZooKeeper. Такие случаи сложно обнаружить.
- При каждом перезапуске контроллер должен прочитать все метаданные для всех брокеров и разделов из ZooKeeper, а затем отправить эти метаданные всем брокерам. Несмотря на многолетние усилия, это остается основным узким местом — по мере увеличения количества разделов и брокеров перезапуск контроллера становится все медленнее.
- Внутренняя архитектура владения метаданными не очень хороша: некоторые операции выполнялись через контроллер, другие — через любой брокер, а третьи — непосредственно на ZooKeeper.

- ZooKeeper — это собственная распределенная система, и, как и Kafka, она требует определенных знаний и опыта для работы. Поэтому разработчикам, которые хотят использовать Kafka, необходимо изучить две распределенные системы, а не одну.

Учитывая все эти проблемы, сообщество Apache Kafka решило заменить существующий контроллер на базе ZooKeeper.

В существующей архитектуре ZooKeeper выполняет две важные функции: он используется для выбора контроллера и хранения метаданных кластера — зарегистрированных брокеров, конфигурации, топиков, разделов и реплик. Кроме того, контроллер сам управляет метаданными — он используется для выбора лидеров, создания и удаления топиков и переназначения реплик. Вся эта функциональность должна быть заменена в новом контроллере.

Основная идея нового дизайна контроллера заключается в том, что сама Kafka имеет архитектуру, основанную на журналах, где пользователи представляют состояние в виде потока событий. Преимущества такого представления хорошо понятны сообществу — множество потребителей могут быстро узнать о последнем состоянии, воспроизводя события. Журнал устанавливает четкий порядок между событиями и гарантирует, что потребители всегда перемещаются по единой временной шкале. Новая архитектура контроллера обеспечивает те же преимущества для управления метаданными Kafka.

В новой архитектуре узлы контроллера представляют собой кворум Raft, который управляет журналом событий метаданных. Этот журнал содержит информацию о каждом изменении метаданных кластера. Все, что в настоящее время хранится в ZooKeeper, например топика, разделы, обработчики прерываний ISR, конфигурации и т. д., будет храниться в этом журнале.

Используя алгоритм Raft, узлы контроллера будут выбирать лидера из своего числа, не полагаясь на какую-либо внешнюю систему. Лидер журнала метаданных называется активным контроллером. Он обрабатывает все механизмы вызова удаленных процедур RPC, поступающие от брокеров. Контроллеры-последователи реплицируют данные, которые записываются в активный контроллер, и служат в качестве горячих резервных копий на тот случай, если активный контроллер выйдет из строя. Поскольку все контроллеры теперь будут отслеживать последнее состояние, при отказе контроллера не потребуются длительный период перезагрузки, в течение которого мы переносим все состояние на новый контроллер.

Вместо того чтобы контроллер рассылал обновления другим брокерам, эти брокеры будут получать обновления от активного контроллера с помощью нового API `MetadataFetch`. Подобно запросу на выборку, брокеры будут отслеживать смещение последнего изменения метаданных, которое они получили, и запрашивать у контроллера только более новые обновления. Брокеры станут

сохранять метаданные на диск, что позволит им быстро запускаться даже при наличии миллионов разделов (<https://oreil.ly/TsU0w>).

Брокеры регистрируются в кворуме контроллера и остаются зарегистрированными до тех пор, пока их не снимет с регистрации администратор, поэтому, как только брокер завершит работу, он будет отключен, но все еще останется зарегистрированным. Брокеры, которые находятся в режиме онлайн, но не имеют актуальных метаданных, будут изолированы и не смогут обслуживать запросы клиентов. Новое изолированное состояние предотвратит случаи, когда клиент отправляет события брокеру, который больше не является лидером, но слишком устарел, чтобы знать, что он не является лидером.

В рамках перехода на кворум контроллера все операции, которые ранее выполнялись клиентами или брокерами, взаимодействующими непосредственно с ZooKeeper, будут маршрутизироваться через контроллер. Это позволит обеспечить бесшовную миграцию путем замены контроллера без необходимости менять что-либо на каком-либо брокере.

Общий дизайн новой архитектуры описан в KIP-500 (<https://oreil.ly/KAsp9>). Подробности адаптации протокола Raft для Kafka изложены в KIP-595 (<https://oreil.ly/XbI8L>). Подробное описание нового кворума контроллеров, включая конфигурацию контроллеров и новый интерфейс командной строки CLI для взаимодействия с метаданными кластера, содержится в KIP-631 (<https://oreil.ly/rpOjK>).

Репликация

Репликация — основа основ архитектуры Kafka. Действительно, Kafka часто описывается как «распределенный, секционированный сервис реплицируемых журналов фиксации». Репликация критически важна, поскольку с ее помощью Kafka обеспечивает доступность и сохраняемость данных при неизбежных сбоях отдельных узлов.

Как мы уже говорили, данные в Kafka сгруппированы по топикам. Последние разбиваются на разделы, у каждого из которых может быть несколько реплик. Эти реплики хранятся на брокерах, причем каждый из них обычно хранит сотни или даже тысячи реплик, относящихся к разным топикам и разделам.

Существует два типа реплик.

- *Ведущие.* Одна реплика из каждого раздела назначается ведущей (leader). Через нее выполняются все запросы на генерацию, чтобы обеспечить согласованность. Клиенты могут потреблять как от ведущей реплики, так и от ее последователей.

- *Ведомые.* Все реплики раздела, не являющиеся ведущими, называются ведомыми (followers). Если не настроено иначе, ведомые реплики не обслуживают клиентские запросы, их основная задача — реплицировать сообщения от ведущей реплики и поддерживать актуальное по сравнению с ней состояние. В случае аварийного сбоя ведущей реплики раздела одна из ведомых будет повышена в ранге и станет новой ведущей.

ЧТЕНИЕ ИЗ ВЕДОМОЙ РЕПЛИКИ

Возможность чтения из ведомых реплик была добавлена в KIP-392 (<https://oreil.ly/2xfxa>). Основная цель этой функции — снизить затраты на сетевой трафик, позволяя клиентам получать данные не из ведущей реплики, а из ближайшей синхронизированной. Для использования этой функции конфигурация потребителя должна включать `client.rack`, идентифицирующий местоположение клиента. Конфигурация брокера должна включать `replica.selector.class`. Эта конфигурация по умолчанию имеет значение `LeaderSelector` (всегда выполняет запрос от лидера), но может быть установлена на `RackAwareReplicaSelector`, которая будет выбирать реплику, расположенную на брокере с конфигурацией `rack.id`, соответствующей `client.rack` клиента. Мы также можем реализовать нашу собственную логику выбора реплики, реализовав интерфейс `ReplicaSelector` и используя вместо него нашу собственную реализацию.

Протокол репликации был расширен, чтобы гарантировать, что только зафиксированные сообщения будут доступны при получении из ведомой реплики. Это означает: мы имеем те же гарантии надежности, что и всегда, даже при получении сообщений из ведомой реплики. Чтобы обеспечить эту гарантию, все реплики должны знать, какие сообщения были зафиксированы лидером. Для этого лидер включает текущую максимальную отметку последнего зафиксированного смещения (high-water mark) в данные, которые он отправляет ведомой реплике. Распространение отметки high-water mark вносит небольшую задержку, это означает, что данные будут доступны для потребления от лидера раньше, чем они доступны для последователя. Важно помнить об этой дополнительной задержке, поскольку очень заманчиво попытаться уменьшить задержку потребителя, потребляя данные из реплики лидера.

Еще одна обязанность ведущей реплики — знать, какие ведомые реплики актуальны по сравнению с ней, а какие — нет. Ведомые реплики поддерживают актуальность посредством репликации всех сообщений от ведущей по мере их поступления. Но они могут отставать вследствие множества причин, например замедления репликации в результате перегруженности сети или аварийного останова брокера, из-за чего все его реплики начинают отставать и отстают до тех пор, пока он не будет запущен снова и репликация не возобновится.

Чтобы не отстать от ведущей, реплики посылают ей запросы `Fetch`, такие же, какие потребители отправляют для получения сообщений. В ответ на них ведущая реплика отправляет ведомым сообщения. В каждом из запросов `Fetch` содержится смещение сообщения, которое реплика желает получить

следующим, что обеспечивает поддержание нужного порядка. Это значит, что ведущая реплика знает, что ведомая реплика получила все сообщения вплоть до последних, которые получала реплика, и не получила ни одно из последующих сообщений. Ведущая реплика на основе последних запрошенных репликами смещений может определить, насколько отстает каждая из них. Если реплика не запрашивала сообщений более 10 с или запрашивала, но не отстает более чем на 10 с, то она считается *рассогласованной* (out of sync). Если реплика отстает от ведущей, то не может более надеяться стать новой ведущей в случае отказа нынешней — в конце концов, в ней же нет всех сообщений.

Напротив, стабильно запрашивающие новые сообщения реплики называются *согласованными* (in-sync). Только согласованная реплика может быть избрана ведущей репликой раздела в случае сбоя действующей ведущей реплики.

Параметр настройки `replica.lag.time.max.ms` задает промежуток времени, по истечении которого бездействующая или отстающая ведомая реплика будет сочтена рассогласованной. Это допустимое отставание влияет на поведение клиентов и сохранение данных при выборе ведущей реплики. Мы обсудим это подробнее в главе 7, когда будем говорить о гарантиях надежности.

Помимо действующей ведущей реплики, в каждом разделе есть *предпочтительная ведущая реплика* (preferred leader) — та, которая была ведущей в момент создания топика. Предпочтительная она потому, что при первоначальном создании разделов ведущие реплики распределяются между брокерами. В результате можно ожидать, что, когда ведущие реплики всех разделов кластера будут одновременно и предпочтительными, распределяться нагрузка по брокерам станет равномерно. По умолчанию в конфигурации Kafka задан параметр `auto.leader.rebalance.enable=true`, при котором она будет проверять, является ли предпочтительная реплика ведущей и согласована ли она, инициируя в этом случае выбор ведущей реплики, чтобы сделать предпочтительную ведущую реплику действующей.



Нахождение предпочтительных ведущих реплик

Проще всего найти предпочтительную ведущую реплику с помощью списка реплик разделов. (Подробные данные о разделах и репликах можно найти в выводимой утилитой `kafka-topics.sh` информации. Мы обсудим ее и другие инструменты администратора в главе 13.) Предпочтительная ведущая реплика всегда стоит первой в списке. На самом деле не имеет значения, какая реплика является ведущей в данный момент или что реплики распределены по разным брокерам с помощью утилиты переназначения реплик. При переназначении реплик вручную важно помнить, что указываемая первой реплика будет предпочтительной, так что их нужно распределять по разным брокерам, чтобы не перегружать одни брокеры ведущими, оставляя другие без законной доли нагрузки.

Обработка запросов

Основная доля работы брокера Kafka заключается в обработке запросов, поступающих ведущим репликам разделов от клиентов, реплик разделов и контроллера. У Kafka есть двоичный протокол (работает по TCP), определяющий формат запросов и ответ на них брокеров как при успешной обработке запроса, так и при возникновении ошибок во время обработки.

Проект Apache Kafka включает в себя Java-клиентов, которые были реализованы и поддерживаются участниками проекта Apache Kafka; есть также клиенты на других языках программирования, таких как C, Python, Go, и многих других. Полный список можно посмотреть на сайте Apache Kafka (<http://bit.ly/2sKvTjx>). Все они взаимодействуют с брокерами Kafka с помощью этого протокола.

Клиенты всегда выступают в роли стороны, иницилирующей подключения и отправляющей запросы, а брокер обрабатывает запросы и отвечает на них. Все полученные брокером от конкретного клиента запросы обрабатываются в порядке поступления. Благодаря этому Kafka может служить очередью сообщений и гарантировать упорядоченность хранимых сообщений.

Каждый запрос имеет стандартный заголовок, включающий:

- тип запроса (называется также ключом API);
- версию запроса (так что брокеры могут работать с клиентами разных версий и отвечать на их запросы соответствующим образом);
- идентификатор корреляции — число, уникально идентифицирующее запрос и включаемое также в ответ и журналы ошибок (этот идентификатор применяется для диагностики и устранения неполадок);
- идентификатор клиента — используется для идентификации отправившего запрос приложения.

Мы не станем описывать этот протокол, поскольку он подробно изложен в документации Kafka (<http://kafka.apache.org/protocol.html>). Однако не помешает разобраться с тем, как брокеры обрабатывают запросы, — далее, когда мы будем обсуждать мониторинг Kafka и различные параметры конфигурации, вам станет понятнее, к каким очередям и потокам выполнения относятся показатели и параметры конфигурации.

Для каждого порта, на котором брокер выполняет прослушивание, запускается *принимающий поток* (acceptor thread), создающий соединение и передающий контроль над ним *обрабатывающему потоку* (processor thread). Число потоков-обработчиков, также называемых *сетевыми потоками* (network threads), можно задать в конфигурации. Сетевые потоки отвечают за получение запросов из клиентских соединений, помещение их в *очередь запросов* (request queue), сбор

ответов из *очереди ответов* (response queue) и отправку их клиентам. Иногда ответы клиентам приходится получать с задержкой — потребители получают ответы только тогда, когда данные доступны, а клиенты администратора получают ответ на запрос `DeleteTopic` после удаления топика. Отложенные ответы хранятся в чистилище (purgatory) (<https://oreil.ly/2jWos>) до тех пор, пока они не будут завершены. Наглядно этот процесс показан на рис. 6.1.

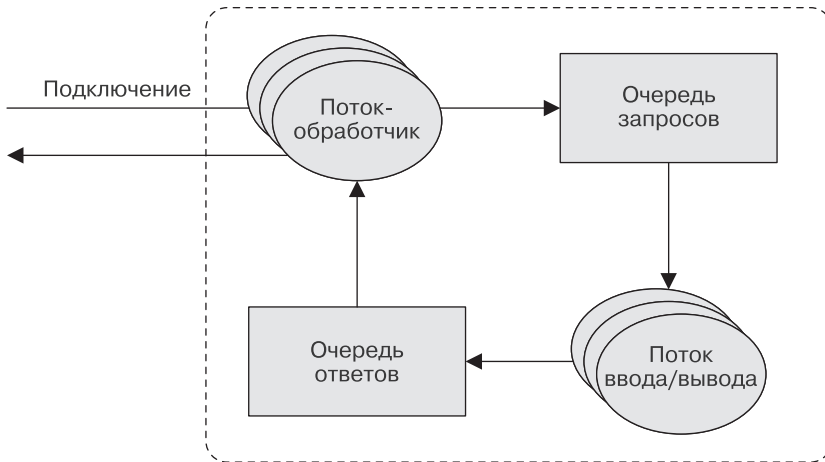


Рис. 6.1. Обработка запросов внутри Apache Kafka

После помещения запросов в очередь ответственность за их обработку передается *потокам ввода/вывода* (IO threads) (также называются потоками обработчика запросов — handler threads). Наиболее распространенные типы клиентских запросов:

- *запросы от производителей* — отправляются производителями и содержат сообщения, записываемые клиентами в брокеры Kafka;
- *запросы на извлечение* — отправляются потребителями и ведомыми репликами при чтении ими сообщений от брокеров Kafka;
- *запросы администратора* — отправляются клиентами-администраторами при выполнении операций с метаданными, таких как создание и удаление топиков.

Как запросы от производителей, так и запросы на извлечение должны отправляться ведущей реплике раздела. Если брокер получает запрос от производителя, относящийся к конкретному разделу, ведущая реплика которого находится на другом брокере, то отправивший запрос клиент получит сообщение об ошибке «Не является ведущей репликой для раздела» (Not a leader for partition). Та же ошибка возникнет при запросе на извлечение из конкретного раздела, полученном на брокере, на котором нет для нее ведущей реплики. Клиенты Kafka отвечают за то, чтобы запросы производителей и запросы на извлечение

отправлялись на брокер, содержащий ведущую реплику для соответствующего запросу раздела.

Откуда клиенты знают, куда им отправлять запросы? Клиенты Kafka применяют для этой цели еще один вид запроса, называемый *запросом метаданных* (metadata request) и включающий список топиков, интересующих клиента. Ответ сервера содержит информацию о существующих в этих топиках разделах, репликах для каждого из разделов, а также ведущей реплике. Запросы метаданных можно отправлять любому брокеру, поскольку у каждого из них есть содержащий эту информацию кэш метаданных.

Клиенты обычно кэшируют эту информацию и используют ее для направления запросов производителей и запросов на извлечение нужному брокеру для каждого из разделов. Им также приходится иногда обновлять эту информацию (интервал обновления задается параметром конфигурации `metadata.max.age.ms`) посредством отправки дополнительных запросов метаданных для выяснения, не поменялись ли метаданные топика, например, не был ли добавлен еще один брокер и не была ли перенесена на него часть реплик (рис. 6.2). Кроме того, при получении на один из запросов ответа «Не является ведущей репликой» клиент обновит метаданные перед попыткой отправить запрос повторно, поскольку эта ошибка указывает на использование им устаревшей информации и отправку запроса не тому брокеру.

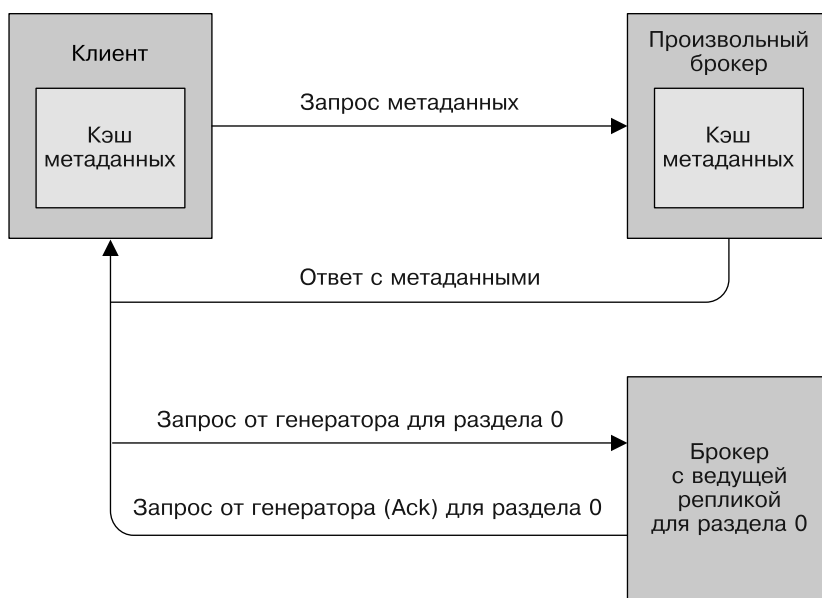


Рис. 6.2. Маршрутизация запросов клиентов

Запросы от производителей

Как мы уже видели в главе 3, параметр конфигурации `acks` определяет число брокеров, которые должны подтвердить получение сообщения, чтобы операция записи считалась успешной. Можно настроить производители так, чтобы они считали сообщение записанным успешно, если его прием был подтвержден только ведущей репликой (`acks=1`) или всеми согласованными репликами (`acks=all`) либо как только оно отправлено, не дожидаясь его приема брокером (`acks=0`).

Брокер, на котором находится ведущая реплика раздела, при получении запроса к ней от производителя начинает с нескольких проверок.

- Есть ли у отправляющего данные пользователя права на запись в этот топик?
- Допустимо ли указанное в запросе значение параметра `acks` (допустимые значения `0`, `1` и `all`)?
- Если параметр `acks` установлен в значение `all`, достаточно ли согласованных реплик для безопасной записи сообщения? (Можно настроить брокеры так, чтобы они отказывались принимать новые сообщения, если число согласованных реплик меньше заданного в конфигурации значения. Мы поговорим об этом подробнее в главе 7, когда будем обсуждать гарантии сохраняемости и надежности Kafka.)

Затем брокер записывает новые сообщения на локальный диск. На операционной системе Linux сообщения записываются в кэш файловой системы, и нет никаких гарантий, что они будут записаны на диск. Kafka не ждет сохранения данных на диск — сохраняемость сообщений обеспечивается посредством репликации.

После записи сообщения на ведущую реплику раздела брокер проверяет значение параметра `acks`. Если оно равно `0` или `1`, брокер отвечает сразу же, если же `all`, запрос хранится в буфере-*чистилище* (purgatory) до тех пор, пока ведущая реплика не удостоверится, что ведомые реплики выполнили репликацию сообщения. Затем клиенту будет отправлен ответ.

Запросы на извлечение

Брокеры обрабатывают запросы на извлечение примерно так же, как и запросы от производителей. Клиент посылает запрос, в котором просит брокер отправить сообщения в соответствии со списком топиков, разделов и смещений, — что-то вроде «Пожалуйста, отправьте мне сообщения, начинающиеся

со смещения 53 раздела 0 топика Test, и сообщения, начинающиеся со смещения 64 раздела 3 топика Test». Клиенты также указывают ограничения на объем возвращаемых из каждого раздела данных. Это ограничение важно, потому что клиентам требуется выделять память под ответ брокера. Без него отправляемые брокерами ответы могли бы оказаться настолько велики, что клиентам не хватило бы памяти.

Как мы уже обсуждали, запрос должен быть отправлен ведущим репликам указанных в запросе разделов, для чего клиенты предварительно запрашивают метаданные, чтобы гарантировать правильную маршрутизацию запросов на извлечение. Ведущая реплика, получив запрос, первым делом проверяет, корректен ли он — существует ли по крайней мере данное смещение в этом разделе. Если клиент запрашивает настолько старое смещение, что оно уже удалено из раздела, или еще не существующее, брокер вернет сообщение об ошибке.

Если смещение существует, брокер читает сообщения из раздела вплоть до указанного клиентом в запросе ограничения и отправляет сообщения клиенту. Kafka знаменита своим использованием метода *zero-copy* для отправки сообщений клиентам — это значит, что она отправляет сообщения напрямую из файлов (или, скорее, кэша файловой системы Linux) без каких-либо промежуточных буферов. В большинстве же баз данных, в отличие от Kafka, перед отправкой клиентам данные сохраняются в локальном буфере. Эта методика позволяет избавиться от накладных расходов на копирование байтов и управление буферами памяти и существенно повышает производительность.

Помимо ограничения сверху объема возвращаемых брокером данных, клиенты могут задать и ограничение снизу. Например, ограничение снизу в 10 Кбайт эквивалентно указанию брокеру возвращать результаты только при накоплении хотя бы 10 Кбайт для отправки. Это отличный способ снижения загруженности процессора и сети в случаях, когда клиенты читают данные из топиков с не слишком большими объемами трафика. Вместо отправки брокерам запросов данных каждые несколько секунд с получением в ответ лишь одного-двух (а то и ни одного) сообщений клиент отправляет запрос, а брокер ждет, пока не накопится порядочный объем данных, возвращает их, и лишь тогда клиент запрашивает новые данные (рис. 6.3). При этом читается в целом тот же самый объем данных при намного меньшем объеме взаимодействий, а следовательно, меньших накладных расходах.

Конечно, не следует заставлять клиентов ждать бесконечно долго, пока брокер не накопит достаточно данных. По прошествии определенного времени имеет смысл обрабатывать имеющиеся данные, а не ждать дальше. Следовательно, клиенты тоже могут задать промежуток времени и сообщить брокеру: «Если за x миллисекунд у тебя не появится минимально достаточный для отправки объем данных, просто отправляй все, что есть».

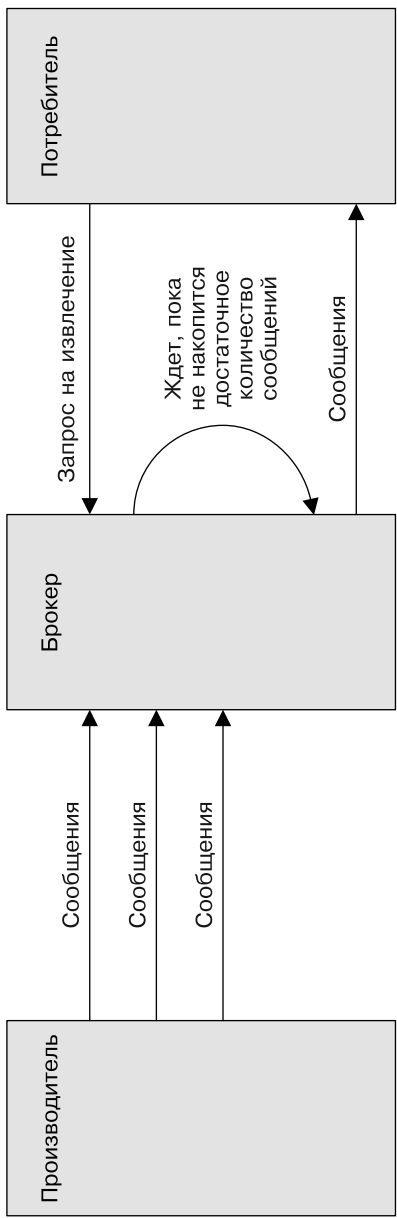


Рис. 6.3. Брокер откладывает ответ до тех пор, пока не накопит достаточно данных

Интересно отметить, что не все данные из ведущей реплики в этом разделе доступны клиентам для считывания. Большинство клиентов могут читать только те сообщения, которые записаны во все согласованные реплики (ведомых реплик это не касается, хотя они и потребляют данные, иначе не смогла бы функционировать репликация). Мы уже обсуждали, что ведущая реплика раздела знает, какие сообщения были реплицированы на какие реплики, и сообщение не будет отправлено потребителю, пока оно не записано во все согласованные реплики. Попытки прочесть подобные сообщения приведут к возврату пустого ответа, а не сообщения об ошибке.

Причина в том, что не реплицированные на достаточное количество реплик сообщения считаются небезопасными — в случае аварийного сбоя ведущей реплики и ее замены другой они пропадут из Kafka. Если разрешить клиентам чтение сообщений, имеющихся только на ведущей реплике, возникнет рассогласованность. Например, если во время чтения потребителем такого сообщения ведущая реплика аварийно прекратит работу, а этого сообщения нет больше ни на одном брокере, то оно будет утрачено. Больше ни один потребитель его прочесть не сможет, что приведет к рассогласованию с уже прочитавшим его потребителем. Вместо этого необходимо дожидаться получения сообщения всеми согласованными репликами и лишь затем разрешать потребителям его читать (рис. 6.4). Такое поведение означает также, что в случае замедления по какой-либо причине репликации между брокерами доставка новых сообщений потребителям будет занимать больше времени, поскольку мы сначала ждем репликации сообщений. Эта задержка ограничивается параметром `replica.lag.time.max.ms` — промежуток времени, по истечении которого реплика, отстающая при репликации новых сообщений, будет сочтена рассогласованной.

В некоторых случаях потребитель получает события из большого количества разделов. Отправка списка всех интересующих его разделов брокеру с каждым запросом и передача брокером всех метаданных обратно может быть очень неэффективной — набор разделов редко меняется, их метаданные редко меняются, и во многих случаях нужно возвращать не так уж много данных. Чтобы минимизировать эти накладные расходы, в Kafka есть кэш сессии выборки. Потребители могут попытаться создать кэшированную сессию, хранящую список разделов, из которых они получают данные, и их метаданные. После создания сессии потребителям больше не нужно указывать все разделы в каждом запросе, вместо этого они могут использовать инкрементные запросы на выборку. Брокеры будут включать метаданные в ответ только в том случае, если произошли какие-либо изменения. Кэш сессий имеет ограниченное пространство, и Kafka отдает приоритет ведомым репликам и потребителям с большим набором разделов, поэтому в некоторых случаях сессия не будет создана или будет вытеснена. В обоих этих случаях брокер вернет клиенту соответствующую ошибку, а потребитель прозрачно прибегнет к запросам на полную выборку, включающим все метаданные раздела.

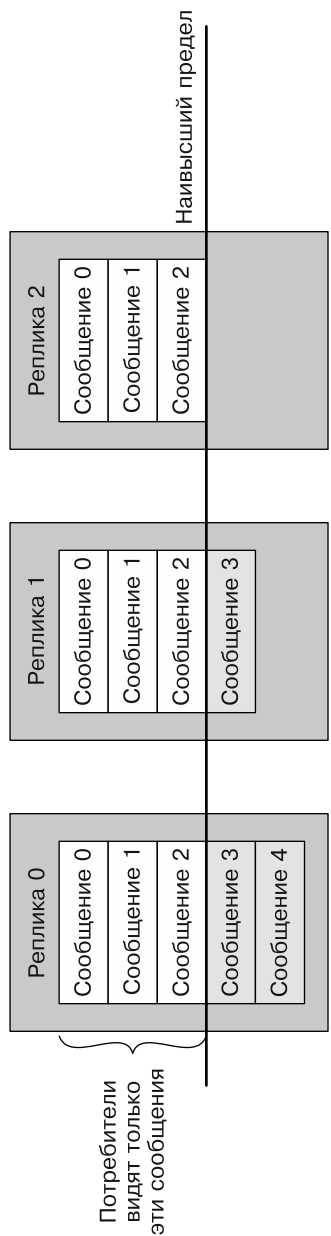


Рис. 6.4. Потребители видят только те сообщения, которые реплицированы на согласованные реплики

Другие запросы

Мы только что обсудили самые распространенные типы запросов, применяемые клиентами Kafka: `Metadata`, `Produce` и `Fetch`. В настоящее время протокол Kafka обрабатывает 61 тип запросов (<https://oreil.ly/hBmNc>), и в дальнейшем будут добавлены новые. Только потребители используют 15 типов запросов для формирования групп, координации потребления и предоставления разработчикам возможности управлять группами потребителей. Существует также большое количество запросов, связанных с управлением метаданными и безопасностью.

Кроме того, этот же протокол применяется для взаимодействия между самими брокерами Kafka. Это внутренние запросы, их не должны использовать клиенты. Например, контроллер, извещая о новой ведущей реплике раздела, отправляет запрос `LeaderAndIsr` новой ведущей реплике, чтобы она начала принимать запросы клиентов, и ведомым репликам, чтобы они ориентировались на новую ведущую реплику.

Этот протокол постоянно совершенствуется — он развивается по мере того, как сообщество Kafka добавляет все больше функциональных возможностей для клиентов. Например, в прошлом потребители Kafka задействовали Apache ZooKeeper для отслеживания получаемых от платформы смещений. Так что потребитель после запуска может выяснить в ZooKeeper, каково последнее прочитанное из соответствующих разделов смещение, и будет знать, с какого места начинать обработку. По различным причинам сообщество решило больше не использовать для этого ZooKeeper, а хранить смещения в отдельном топике Kafka. Для этого участникам пришлось добавить в протокол несколько типов запросов: `OffsetCommitRequest`, `OffsetFetchRequest` и `ListOffsetsRequest`. Теперь, когда приложение обращается к API клиента для проведения потребительских взаимозачетов, клиент ничего не записывает в ZooKeeper, а отправляет запрос `OffsetCommitRequest` в Kafka.

Раньше создание топиков обрабатывалось с помощью утилит командной строки, непосредственно обновляющих список топиков в ZooKeeper. С тех пор сообщество Kafka добавило запрос `CreateTopicRequest` и аналогичные запросы для управления метаданными Kafka. Java-приложения выполняют эти операции с метаданными через `AdminClient` Kafka, подробно описанный в главе 5. Поскольку данные операции теперь являются частью протокола Kafka, это позволяет клиентам в языках программирования, где нет библиотеки ZooKeeper, создавать топик, обращаясь непосредственно к брокерам Kafka.

Помимо усовершенствования протокола путем добавления новых типов запросов, разработчики Kafka иногда меняют существующие запросы, добавляя не-

которые новые возможности. Например, при переходе от Kafka 0.9.0 к 0.10.0 они решили, что клиентам не мешает информация о текущем контроллере, и добавили ее в ответ `Metadata`. В результате была добавлена новая версия запроса и ответа `Metadata`. Теперь клиенты 0.9.0 отправляют запросы `Metadata` версии 0, поскольку версия 1 в них еще не существовала, а брокеры вне зависимости от их версии возвращают ответ версии 0, в котором нет информации о контроллере. Это нормально, ведь клиенты 0.9.0 не ждут информации о контроллере и все равно не сумеют выполнить ее синтаксический разбор. Клиент же 0.10.0 отправит запрос `Metadata` версии 1, в результате чего брокеры версии 0.10.0 вернут ответ версии 1 с информацией о контроллере, которой клиенты 0.10.0 смогут воспользоваться. Брокер же версии 0.9.0 при получении от клиента 0.10.0 запроса `Metadata` версии 1 не будет знать, что с ним делать, и вернет сообщение об ошибке. Именно поэтому мы рекомендуем сначала обновить все брокеры и только потом обновлять клиенты — новые брокеры смогут обработать старые запросы, но не наоборот.

В версии Kafka 0.10.0 сообщество Kafka добавило запрос `ApiVersionRequest`, позволяющий клиентам запрашивать у брокера поддерживаемые версии запросов и применять соответствующую версию. Клиенты, правильно реализующие эту новую возможность, смогут взаимодействовать со старыми брокерами благодаря использованию поддерживаемых ими версий протокола. В настоящее время ведется работа по добавлению API, которые позволят клиентам узнавать, какие функции поддерживаются брокерами, и дадут возможность брокерам блокировать функции, существующие в определенной версии. Это улучшение было предложено в KIP-584 (<https://oreil.ly/dxg8N>), и на данный момент вполне вероятно, что оно станет частью версии 3.0.0.

Физическое хранилище

Основная единица хранения Kafka — реплика раздела. Разделы нельзя разносить по нескольким брокерам или даже по различным дискам одного брокера, так что размер раздела ограничивается доступным на отдельной точке монтирования местом. (Точка монтирования может быть отдельным диском при использовании дискового массива JBOD или состоять из нескольких дисков, когда действуется RAID, — см. главу 2.)

При настройке Kafka администратор задает список каталогов для хранения разделов с помощью параметра `log.dirs` (не путайте его с местом хранения журнала ошибок Kafka, настраиваемым в файле `log4j.properties`). Обычная конфигурация предусматривает по одному каталогу для каждой точки монтирования Kafka.

Разберемся, как Kafka использует доступные каталоги для хранения данных. Во-первых, посмотрим, как данные распределяются по брокерам в кластере и каталогам на брокере. Затем поговорим о том, как брокеры обращаются с файлами, уделив особое внимание гарантиям сохранения информации. Затем заглянем внутрь файлов и изучим форматы файлов и индексов. Наконец, обсудим сжатие журналов — продвинутую возможность, благодаря которой вы можете превратить Kafka в долговременное хранилище данных, и опишем, как она функционирует.

Многоуровневое хранилище

С конца 2018 года сообщество Apache Kafka участвует в амбициозном проекте по добавлению в Kafka возможностей многоуровневого хранения данных. Работа над проектом продолжается, его выпуск запланирован в версии 3.0.

Мотивация довольно проста: в настоящее время Kafka используется для хранения больших объемов данных либо из-за высокой пропускной способности, либо из-за длительных периодов хранения. В связи с этим возникают следующие проблемы.

- Объем данных, которые можно хранить в одном разделе, ограничен. В результате максимальный срок хранения и количество разделов определяются не только требованиями продукта, но и ограничениями размеров физического диска.
- Выбор размера диска и кластера определяется требованиями к объему памяти. Кластеры часто оказываются больше, чем могли бы быть, если бы основными соображениями были задержка и пропускная способность, что приводит к увеличению затрат.
- Время, необходимое для перемещения разделов от одного брокера к другому, например, при расширении или сжатии кластера, зависит от размера разделов. Большие разделы делают кластер менее эластичным. В наши дни архитектуры разрабатываются с учетом максимальной эластичности с использованием преимуществ гибких вариантов развертывания в облаке.

При многоуровневом подходе к хранению кластер Kafka конфигурируется с двумя уровнями хранения — локальным и удаленным. Локальный уровень такой же, как и текущий уровень хранения Kafka, — он использует локальные диски на брокерах Kafka для хранения сегментов журнала. Новый удаленный уровень использует специальные системы хранения, такие как HDFS или S3, для хранения завершенных сегментов журнала.

Пользователи Kafka могут установить отдельную политику хранения для каждого уровня. Поскольку локальное хранилище обычно намного дороже, чем

удаленный уровень, период хранения для локального уровня обычно составляет всего несколько часов или даже меньше, а период хранения для удаленного уровня может быть намного больше — дни или даже месяцы.

Локальное хранилище имеет значительно меньшее время задержки, чем удаленное хранилище. Это хорошо работает, поскольку приложения, чувствительные к задержкам, выполняют чтение последних элементов хвоста (последних сообщений) и обслуживаются с локального уровня, поэтому они выигрывают от существующего механизма Kafka по эффективному использованию кэша страниц для обслуживания данных. Backfill (обратное заполнение) и другие приложения, восстанавливающиеся после сбоя, которым требуются более старые данные, чем те, что находятся на локальном уровне, обслуживаются с удаленного уровня.

Двухуровневая архитектура, применяемая в многоуровневом хранилище, позволяет масштабировать хранилище независимо от памяти и процессоров в кластере Kafka. Это позволяет использовать Kafka в качестве долгосрочного хранилища данных. Также это уменьшает объем данных, хранящихся локально на брокерах Kafka, и, следовательно, объем данных, которые необходимо копировать во время восстановления и перебалансировки. Не нужно восстанавливать сегменты журнала, доступные на удаленном уровне, на брокере, или откладывать восстановление, они обслуживаются с удаленного уровня. Поскольку не все данные хранятся на брокерах, увеличение периода хранения больше не требует масштабирования хранилища кластера Kafka и добавления новых узлов. В то же время общий срок хранения данных может быть гораздо больше, что устраняет необходимость в отдельных конвейерах данных для копирования данных из Kafka во внешние хранилища, как делается сейчас во многих развертываниях.

Дизайн многоуровневого хранилища, включая новый компонент — **RemoteLogManager** и взаимодействие с существующими функциональными возможностями, такими как догоняющие лидера реплики и выборы лидера, подробно документирован в KIP-405 (<https://oreil.ly/yZP6w>).

Одним из интересных результатов, задокументированных в KIP-405, является влияние многоуровневого хранения на производительность. Команда, внедряющая многоуровневое хранилище, измерила производительность в нескольких сценариях использования. В первом сценарии использовалась обычная для Kafka высокопроизводительная рабочая нагрузка. В этом случае задержка немного увеличилась (с 21 мс в r99 до 25 мс), поскольку брокеры также должны были отправлять сегменты в удаленное хранилище. Второй сценарий использования — когда некоторые потребители читают старые данные. Без многоуровневого хранилища потребители, читающие старые данные, сильно влияют на задержку (21 мс против 60 мс в r99), но при включенном многоуровневом хранилище это влияние значительно меньше (25 мс против 42 мс в r99).

Это происходит потому, что чтение из многоуровневого хранилища происходит из HDFS или S3 по сетевому пути. Сетевые операции чтения не конкурируют с локальными операциями чтения за дисковый ввод-вывод или кэш страниц и оставляют кэш страниц нетронутым со свежими данными. Это означает, что в дополнение к бесконечной емкости, более низкой стоимости и эластичности многоуровневое хранилище обеспечивает также изоляцию между архивными чтениями и чтениями в реальном времени.

Распределение разделов

При создании топика Kafka сначала принимает решение о распределении разделов по брокерам. Допустим, у нас есть шесть брокеров и мы хотим создать топик на десять разделов с коэффициентом репликации 3. Kafka нужно распределить 30 реплик разделов по шести брокерам. Основные задачи распределения следующие.

- Равномерно распределить реплики по брокерам — в нашем случае выделить на каждый брокер пять разделов.
- Гарантировать, что все реплики для каждого из разделов находятся на разных брокерах. Если ведущая реплика раздела 0 располагается на брокере 2, ее ведомые реплики можно поместить на брокеры 3 и 4, но не на 2 (и не обе на 3).
- Если у брокеров имеется информация о размещении в стойках (доступны в Kafka, начиная с версии 0.10.0), то нужно по возможности разместить реплики для каждого из разделов на различных стойках. Это гарантирует, что отсутствие связи или неработоспособность целой стойки не приведет к полной недоступности разделов.

Чтобы добиться этого, мы начнем с произвольного брокера (допустим, 4) и станем циклически назначать разделы каждому из брокеров для определения местоположения ведущих реплик. Так, ведущая реплика раздела 0 окажется на брокере 4, ведущая реплика раздела 1 — на брокере 5, раздела 2 — на брокере 0 (поскольку у нас всего шесть брокеров) и т. д. Далее для каждого из разделов будем размещать реплики в соответствии со все увеличивающимися смещениями по отношению к ведущей реплике. Если ведущая реплика раздела 0 находится на брокере 4, то первая ведомая реплика попадет на брокер 5, а вторая — на брокер 0. Ведущая реплика раздела 1 находится на брокере 5, так что первая ведомая реплика попадет на брокер 0, а вторая — на брокер 1.

Если учитывать информацию о стойках, то вместо выбора брокеров в числовом порядке подготовим список брокеров с чередованием стоек. Допустим, нам известно, что брокеры 0 и 1 находятся в одной стойке, а брокеры 2 и 3 — в другой. Вместо того чтобы подбирать брокеры по порядку от 0 до 3, мы сортируем их в следующем порядке: 0, 2, 1, 3 — за каждым брокером следует брокер из другой

стойки (рис. 6.5). В таком случае, если ведущая реплика раздела 0 находится на брокере 2, первая реплика окажется на брокере 1, находящемся в другой стойке. Это замечательно, потому что, если первая стойка выйдет из строя, у нас все равно имеется работающая реплика и раздел по-прежнему будет доступен. Это справедливо для всех реплик, так что мы гарантировали доступность в случае отказа одной из стоек.

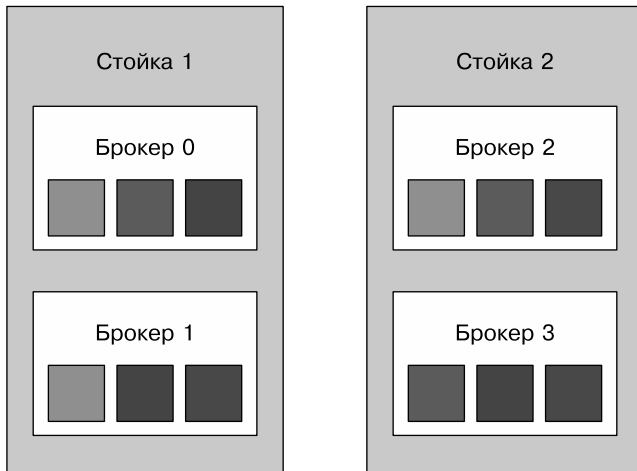


Рис. 6.5. Разделы и реплики распределяются по брокерам, находящимся в разных стойках

Выбрав нужные брокеры для всех разделов и реплик, мы должны определиться с каталогом для новых разделов. Сделаем это отдельно для каждого раздела. Принцип прост: подсчитывается число разделов в каждом каталоге и новые разделы добавляются в каталог с минимальным числом разделов. Это значит, что при добавлении нового диска все новые разделы будут создаваться на нем, поскольку до того, как все выровняется, на новом диске всегда будет меньше всего разделов.



Не забывайте про место на диске

Обратите внимание на то, что распределение разделов по брокерам не учитывает наличие места или имеющуюся нагрузку, а распределение разделов по дискам учитывает только число разделов, но не их размер. Так что если на некоторых брокерах больше дискового пространства, чем на других (допустим, из-за того, что в кластере есть и более старые, и более новые серверы), а среди разделов попадаются очень большие или на брокере есть диски разного размера, необходимо соблюдать осторожность при распределении разделов.

Управление файлами

Сохранение информации имеет в Kafka большое значение — платформа не хранит данные вечно и не ждет, когда все потребители прочтут сообщение, перед тем как его удалить. Администратор Kafka задает для каждого топика срок хранения — или промежуток времени, в течение которого сообщения хранятся перед удалением, или объем хранимых данных, по исчерпанию которого старые сообщения удаляются.

Поскольку поиск сообщений, которые нужно удалить, в большом файле и последующее удаление его части — процесс затратный и грозящий возникновением ошибок, то вместо этого разделы разбиваются на *сегменты*. По умолчанию каждый сегмент содержит 1 Гбайт данных или данные за неделю в зависимости от того, что оказывается меньше. По достижении этого лимита при записи брокером Kafka данных в раздел файл закрывается и начинается новый.

Сегмент, в который в настоящий момент производится запись, называется *активным* (active segment). Он никогда не удаляется, так что если в конфигурации журналов задано хранить данные лишь за день, но каждый сегмент содержит данные за пять дней, то в действительности будут храниться данные за пять дней, поскольку удалить их до закрытия сегмента невозможно. Если вы решите хранить данные неделю и создавать новый сегмент каждый день, то увидите, что каждый день будет создаваться новый сегмент и удаляться наиболее старый, так что почти все время раздел будет насчитывать семь сегментов.

Как вы знаете из главы 2, брокеры Kafka держат открытыми дескрипторы файлов для всех сегментов раздела, даже неактивных. Из-за этого число открытых дескрипторов файлов стабильно высоко, так что операционная система должна быть настроена соответствующим образом.

Формат файлов

Каждый сегмент хранится в отдельном файле данных, в котором находятся сообщения Kafka и их смещения. Формат файла на диске идентичен формату сообщений, отправляемых от производителя брокеру, а затем от брокера — потребителям. Одинаковый формат данных на диске и передаваемых дает Kafka возможность использовать оптимизацию zero-copy при передаче сообщений потребителям, а также избежать распаковки и повторного сжатия данных, уже сжатых производителем. В итоге при изменении формата сообщений придется поменять как формат данных на диске, так и протокол передачи данных, а бро-

керы Kafka должны будут знать, что делать в случаях, когда из-за появления новых версий файлы содержат сообщения в двух форматах.

В Kafka сообщения состоят из полезной нагрузки пользователя и системных заголовков. Полезная нагрузка пользователя включает в себя необязательный ключ, значение и необязательный набор заголовков, каждый из которых представляет собой собственную пару «ключ/значение».

Начиная с версии 0.11 (и формата сообщений v2), производители Kafka всегда отправляют сообщения пакетами. Если вы отправляете одно сообщение, пакетирование немного увеличивает накладные расходы. Но при отправке двух и более сообщений на пакет пакетная отправка экономит место, что сокращает использование сети и диска. Это одна из причин, почему Kafka работает лучше при `linger.ms=10` — небольшая задержка увеличивает вероятность того, что больше сообщений будет отправлено вместе. Поскольку Kafka создает отдельный пакет для каждого раздела, производители, которые записывают в меньшее количество разделов, также будут более эффективны. Обратите внимание на то, что производители Kafka могут включать несколько пакетов в один запрос на отправку. Это означает, что, если вы используете сжатие на производителе (мы рекомендуем это делать!), отправка больших пакетов гарантирует лучшее сжатие как по сети, так и на дисках брокера.

Заголовки пакетов сообщений включают:

- магическое число, указывающее на текущую версию формата сообщения (здесь мы документируем формат сообщений v2);
- смещение первого сообщения в пакете и разницу со смещением последнего сообщения — они сохраняются, даже если пакет позже будет уплотнен и некоторые сообщения будут удалены. Смещение первого сообщения устанавливается в значение 0, когда производитель создает и отправляет пакет. Брокер, который первым сохраняет этот пакет (лидер раздела), заменяет это значение реальным смещением;
- временные метки первого сообщения и самую высокую временную метку в пакете. Временные метки могут быть установлены брокером, если тип временной метки установлен на время добавления, а не на время создания;
- размер пакета в байтах;
- эпоху лидера, получившего пакет (применяется при усечении сообщений после избрания лидера; KIP-101 (<https://oreil.ly/Ffa4D>) и KIP-279 (<https://oreil.ly/LO7nx>) подробно объясняют использование);
- контрольную сумму для проверки того, что пакет не поврежден;

- 16 бит, указывающих на различные атрибуты: тип сжатия, тип временной метки (временная метка может быть установлена на клиенте или на брокере), а также то, является ли пакет частью транзакции или контрольным пакетом;
- идентификатор производителя, эпоху производителя и первую последовательность в пакете — все это используется, чтобы гарантировать точное совпадение (гарантия «ровно один раз»);
- и конечно, набор сообщений, входящих в пакет.

Как вы могли заметить, заголовок пакета содержит много информации. Сами записи также имеют системные заголовки (не путайте с заголовками, которые могут быть установлены пользователями). Каждая запись включает в себя:

- размер записи в байтах;
- атрибуты — в настоящее время нет атрибутов на уровне записи, поэтому они не используются;
- разницу между смещением текущей записи и первым смещением в пакете;
- разницу между временной меткой этой записи и первой временной меткой в пакете в миллисекундах;
- полезную нагрузку пользователя — ключ, значение и заголовки.

Обратите внимание на то, что накладные расходы на каждую запись очень малы, а большая часть системной информации находится на уровне пакета. Хранение первого смещения и временной метки пакета в заголовке и хранение только разницы в каждой записи значительно снижает накладные расходы на каждую запись, что делает большие пакеты более эффективными.

Помимо пакетов сообщений, содержащих пользовательские данные, Kafka имеет также управляющие пакеты, указывающие, например, на транзакционные фиксации. Они обрабатываются потребителем и не передаются пользовательскому приложению, и в настоящее время включают в себя индикатор версии и типа: **0** — прерванная транзакция, **1** — фиксация.

Если вы хотите увидеть все это своими глазами, брокеры Kafka поставляются в комплекте с утилитой `DumpLogSegment`, позволяющей просматривать сегменты разделов в файловой системе и исследовать их содержимое. Запустить ее можно с помощью следующей команды:

```
bin/kafka-run-class.sh kafka.tools.DumpLogSegments
```

Если задать параметр `--deep-iteration`, она отобразит информацию о сжатых сообщениях, содержащихся внутри сообщений-оберток.



Преобразование формата сообщения в меньшую сторону

Формат сообщений, описанный ранее, был введен в версии 0.11. Поскольку Kafka поддерживает обновление брокеров до обновления всех клиентов, она должна была поддерживать любую комбинацию версий между брокером, производителем и потребителем. Большинство комбинаций работает без проблем — новые брокеры будут понимать старый формат сообщений от производителей, а новые производители будут знать, что нужно отправлять сообщения старого формата старым брокерам. Но сложности возникают в том случае, когда новый производитель отправляет сообщения в формате v2 новым брокерам: сообщение хранится в формате v2, но старый потребитель, не поддерживающий формат v2, пытается его прочитать. В этом случае брокеру необходимо преобразовать сообщение из формата v2 в v1, чтобы потребитель смог его разобрать. Это преобразование использует гораздо больший объем центрального процессора и памяти, чем обычное потребление, поэтому его лучше избегать. KIP-188 (<https://oreil.ly/9RwQC>) ввел несколько важных показателей работоспособности, среди которых `FetchMessageConversionsPerSec` и `Message Conversions TimeMs`. Если ваша организация все еще работает со старыми клиентами, мы рекомендуем проверить показатели и обновить клиенты как можно скорее.

Индексы

Kafka дает потребителям возможность извлекать сообщения, начиная с любого смещения. Это значит, что, если потребитель запрашивает 1 Мбайт сообщений, начиная со смещения 100, брокер сможет быстро найти сообщение со смещением 100 (которое может оказаться в любом из сегментов раздела) и начать с этого места чтение сообщений. Чтобы ускорить поиск брокерами сообщений с заданным смещением, Kafka поддерживает индексы для всех разделов. Индекс задает соответствие смещения файлу сегмента и месту в этом файле.

Аналогично в Kafka есть второй индекс, который сопоставляет временные метки со смещениями сообщений. Данный индекс применяется при поиске сообщений по метке времени. Kafka Streams широко использует этот поиск, и он также полезен в некоторых сценариях восстановления после отказа.

Индексы тоже разбиты на сегменты, так что при очистке старых сообщений можно удалять и старые записи индексов. Kafka не поддерживает для индексов контрольные суммы. В случае повреждения индекс восстанавливается из соответствующего сегмента журнала с помощью обычного повторного чтения сообщений и записи смещений и местоположений. При необходимости администраторы могут без опасений удалять сегменты индексов (хотя это может привести к длительному восстановлению) — они будут сгенерированы заново автоматически.

Сжатие

При обычных обстоятельствах Kafka хранит сообщения в течение заданного интервала времени и удаляет сообщения, чей возраст превышает срок хранения. Однако представьте себе, что вы применяете Kafka для хранения адресов доставки покупателей. В этом случае имеет смысл хранить последний адрес каждого из покупателей, а не адреса за последнюю неделю или последний год. Так вам не нужно будет волноваться об устаревших адресах и адресах тех покупателей, которые давно никуда не переезжали, — они станут храниться столько, сколько нужно. Другой сценарий использования — приложение, задействующее Kafka для хранения своего текущего состояния. При каждом изменении состояния приложение записывает свое новое состояние в Kafka. При восстановлении после сбоя оно читает эти сообщения из Kafka для восстановления последнего состояния. В этом случае приложение интересуется только последнее состояние перед сбоем, а не все происходившие во время его работы изменения.

Kafka поддерживает подобные сценарии за счет двух возможных стратегий сохранения для топика: *delete* («удалять»), при которой события, чей возраст превышает интервал хранения, удаляются, и *compact* («сжимать»), при которой сохраняется только последнее значение для каждого из ключей топика. Конечно, вторая стратегия имеет смысл только для тех топиков, для которых приложения генерируют события, содержащие как ключ, так и значение. В случае использования неопределенных ключей (*null*) попытка сжатия приведет к сбою.

Топики также могут иметь политику *delete.and.compact* (удаление и сжатие), которая сочетает в себе сжатие с периодом хранения. Сообщения старше периода хранения будут удалены, даже если они являются самым последним значением для ключа. Эта политика предотвращает чрезмерное увеличение размера сжатых топиков, а также используется, когда бизнес требует удаления записей по истечении определенного времени.

Как происходит сжатие

Каждый из журналов условно делится на две части (рис. 6.6):

- «чистую» — сжатые ранее сообщения. Она содержит только по одному значению для каждого ключа — последнему на момент предыдущего сжатия;
- «грязную» — сообщения, записанные после последнего сжатия.

Если при запуске Kafka сжатие было активировано (с помощью довольно неудачно названного параметра *log.cleaner.enabled*), то каждый из брокеров будет запущен с потоком диспетчера сжатия и несколькими потоками сжатия, которые отвечают за выполнение задач сжатия. Каждый из потоков выбирает

раздел с максимальным отношением числа «грязных» сообщений к полному размеру раздела и очищает его.

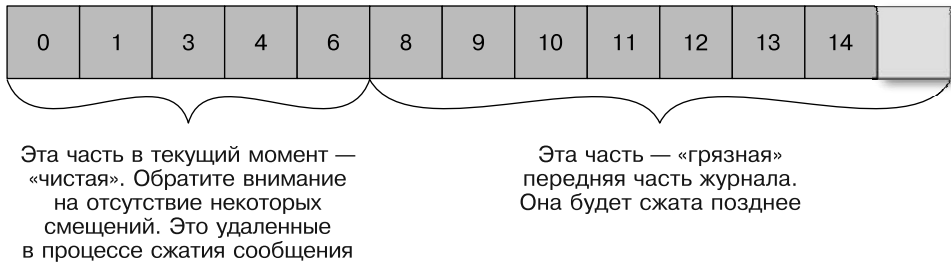


Рис. 6.6. Журнал с «чистым» и «грязным» разделами

Для сжатия раздела поток очистки читает «грязную» часть раздела и создает ассоциативный массив (карту) в оперативной памяти. Каждая запись этого массива состоит из 16-байтного хеша ключа сообщения и 8-байтного смещения предыдущего сообщения с тем же ключом. Это значит, что каждая запись массива использует только 24 байта. Если мы предположим, что в сегменте размером 1 Гбайт каждое сообщение занимает 1 Кбайт, то сегмент может содержать 1 млн сообщений, а для его сжатия понадобится ассоциативный массив всего 24 Мбайт (возможно, даже намного меньше — при повторении ключей одни и те же хеш-записи будут часто использоваться повторно и займут меньше памяти). Весьма эффективно!

При настройке Kafka администратор задает объем памяти, который потоки сжатия могут использовать для этой карты смещений. И хотя у каждого потока будет своя карта, соответствующий параметр задает для всех них общий объем памяти. Если задать его равным 1 Гбайт при пяти потоках очистки, каждый из них получит 200 Мбайт памяти для своего ассоциативного массива. Для Kafka не обязательно, чтобы вся «грязная» часть раздела помещалась в выделенное для этого ассоциативного массива пространство, но по крайней мере один полный сегмент туда помещаться должен. Если это не так, Kafka зафиксирует в журнале ошибку и администратору придется или выделить больше памяти под карты смещений, или использовать меньше потоков очистки. Если помещается лишь несколько сегментов, Kafka начнет со сжатия самых старых сегментов ассоциативного массива. Остальные останутся «грязными», им придется подождать следующего сжатия.

После того как поток очистки сформирует карту смещений, он станет считывать «чистые» сегменты, начиная с самого старого, и сверять их содержимое с картой смещений. Для каждого сообщения поток очистки проверяет, существует ли

ключ сообщения в карте смещений. Если его нет, значит, значение только что прочитанного сообщения актуальное, поэтому сообщение копируется в сменный сегмент. Если же ключ в карте присутствует, сообщение пропускается, поскольку далее в этом разделе есть сообщение с таким же ключом, но с более свежим значением. После копирования всех сообщений, содержащих актуальные ключи, мы меняем сменный сегмент местами с исходным и поток переходит к следующему сегменту. В конце этого процесса для каждого ключа остается одно значение — самое новое (рис. 6.7).

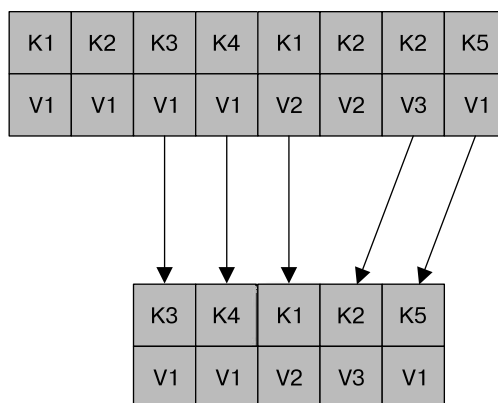


Рис. 6.7 Сегмент раздела до и после сжатия

Удаленные события

Допустим, мы всегда сохраняем последнее сообщение для каждого ключа. Но что делать, если нужно удалить все сообщения для конкретного ключа, например, если пользователь перестал у нас обслуживаться и мы по закону обязаны убрать все его следы из системы?

Чтобы удалить ключ из системы полностью, без сохранения даже последнего сообщения, приложение должно сгенерировать сообщение, содержащее этот ключ, и пустое значение. Наткнувшись на подобное сообщение, поток очистки сначала выполнит обычное сжатие и сохранит только сообщение с пустым значением. Это особое сообщение, известное как *отметка об удалении* (tombstone), будет храниться в течение настраиваемого промежутка времени. На всем его протяжении потребители смогут видеть это сообщение и будут знать, что значение удалено. Так что потребитель, копирующий данные из Kafka в реляционную базу данных, увидит отметку об удалении и будет знать, что нужно убрать пользователя из базы данных. По истечении этого промежутка времени поток

очистки удалит сообщение — отметку об удалении, и ключ пропадет из раздела Kafka. Важно выделить достаточно времени, чтобы потребители успели увидеть его, ведь если потребитель не функционировал несколько часов и пропустил это сообщение, он просто не увидит ключа и не будет знать, что тот был удален из Kafka и нужно удалить его из базы данных.

Стоит помнить, что клиент администратора Kafka включает в себя также метод `deleteRecords`. Последний удаляет все записи до указанного смещения, используя совершенно иной механизм. При вызове этого метода Kafka переместит отметку нижнего предела (*lowwater mark*) — свою запись о первом смещении раздела — на указанное смещение. Это не позволит потребителям использовать записи ниже новой отметки нижнего предела и фактически сделает эти записи недоступными до тех пор, пока они не будут удалены более чистым потоком. Этот метод можно применять для топиков с политикой хранения и сжатых топиков.

Когда выполняется сжатие топиков

Подобно тому как при стратегии `delete` никогда не удаляются активные в настоящий момент сегменты, при стратегии `compact` никогда не сжимается текущий сегмент. Сжатие сообщений возможно только в неактивных сегментах.

По умолчанию Kafka начинает сжатие, когда 50 % топика содержит «грязные» сообщения. Задача заключается в том, чтобы не сжимать слишком часто — это может негативно повлиять на производительность чтения/записи для данного топика, но и не хранить слишком много «грязных» сообщений, поскольку они занимают место на диске. Разумный компромисс состоит в том, чтобы дожидаться, когда «грязные» записи займут 50 % используемого топиком дискового пространства, после чего сжать их за один раз. Этот параметр настраивает администратор.

Кроме того, администраторы могут контролировать время сжатия с помощью двух параметров конфигурации.

- `min.compaction.lag.ms` может использоваться для гарантии минимальной задержки после записи сообщения, прежде чем его можно будет сжать.
- `max.compaction.lag.ms` может использоваться для гарантии максимальной задержки между моментом записи сообщения и моментом, когда оно становится пригодным для сжатия. Эта конфигурация часто применяется в ситуациях, когда в бизнесе есть причина гарантировать сжатие в течение определенного периода времени. Например, общий регламент по защите персональных данных (GDPR) требует, чтобы определенная информация была удалена в течение 30 дней после того, как был сделан запрос на удаление.

Резюме

Тема внутреннего устройства Kafka намного обширнее, чем можно было охватить в этой главе. Но мы надеемся, что вы смогли прочувствовать реализованные сообществом Kafka во время работы над ней проектные решения и оптимизации и, вероятно, разобрались в некоторых малопонятных видах ее поведения и настроек.

Если внутреннее устройство Kafka вас действительно интересует, другого выхода, кроме чтения кода, нет. Среди разработчиков Kafka (почтовая рассылка dev@kafka.apache.org) — очень дружелюбного сообщества — всегда найдется кто-нибудь, готовый ответить на вопросы о функционировании платформы. А во время чтения кода, возможно, вы сумеете исправить одну-две ошибки — проекты с открытым исходным кодом всегда приветствуют любую помощь.

Надежная доставка данных

Надежность — это свойство всей системы, а не отдельного компонента, поэтому, когда мы будем говорить о гарантиях надежности Apache Kafka, нам нужно учитывать всю систему в целом и сценарии ее использования. В том, что касается надежности, интегрируемые с Kafka системы так же важны, как и сама Kafka. А поскольку надежность относится к системе в целом, ответственность за нее не может лежать на одном человеке. Все — администраторы Kafka, администраторы Linux, администраторы сети и систем хранения, а также разработчики приложения — должны сотрудничать для создания надежной системы.

Apache Kafka очень гибка в том, что касается надежной доставки данных. У нее есть множество сценариев использования, начиная от отслеживания нажатий на веб-сайте и заканчивая оплатой по кредитным картам. Некоторые из этих сценариев требуют максимальной надежности, а для других важнее быстрое действие и простота. Kafka спроектирована в расчете на довольно широкие возможности настройки, а ее клиентский API достаточно гибок для любых компромиссов.

Но его гибкость может оказаться ахиллесовой пятой Kafka — легко можно счесть надежной на самом деле ненадежную систему. Эту главу мы начнем с обсуждения различных видов надежности и их значения в контексте Apache Kafka. Затем поговорим о механизме репликации Kafka и его вкладе в надежность системы. Далее обсудим брокеры и топики Kafka и их настройку для различных сценариев. Затем рассмотрим клиенты, производители и потребители, а также их правильное использование в различных ситуациях, связанных с надежностью. И наконец, обсудим тему проверки надежности системы, поскольку недостаточно верить, что система надежна, — необходимо знать это наверняка.

Гарантии надежности

В разговоре о надежности речь обычно идет в терминах *гарантий*, означающих, что поведение системы гарантированно не меняется при различных обстоятельствах.

Вероятно, лучшая из известных гарантий надежности — ACID, стандартная гарантия надежности, поддерживаемая практически всеми реляционными базами данных. Этот акроним расшифровывается как Atomicity, Consistency, Isolation, Durability — *атомарность, согласованность, изоляция и сохраняемость*. Если производитель СУБД говорит, что их база данных удовлетворяет ACID, значит, она гарантирует определенное поведение относительно транзакций. Благодаря этим гарантиям люди доверяют реляционным базам данных, имеющимся в наиболее критических приложениях, — они точно знают, что обещает система и как она будет вести себя в различных условиях. Эти гарантии понятны и позволяют писать на их основе безопасные приложения.

Понимание того, в чем состоят предоставляемые Kafka гарантии, чрезвычайно важно для желающих создавать надежные приложения. Это позволяет разработчикам системы предугадать ее поведение в случае различных сбоев. Итак, что же гарантирует Apache Kafka?

- Упорядоченность сообщений в разделе. Если сообщение B было записано после сообщения A с помощью одного производителя в одном разделе, то Kafka гарантирует, что смещение сообщения B будет превышать смещение сообщения A и потребители прочитают сообщение B после сообщения A.
- Сообщения от производителей считаются зафиксированными, когда они записаны во все согласованные реплики раздела, но не обязательно уже сброшены на диск. Производители могут выбирать разные варианты оповещения о получении сообщений: при полной фиксации сообщения, записи на ведущую реплику или отправке по сети.
- Зафиксированные сообщения не будут потеряны, если функционирует хотя бы одна реплика.
- Потребители могут читать только зафиксированные сообщения.

Эти основные гарантии можно использовать при создании надежной системы, но сами по себе они не делают ее абсолютно надежной. Создание надежной системы допускает различные компромиссы, и Kafka дает возможность администраторам и разработчикам самим определять, насколько надежная система им требуется, с помощью задания параметров конфигурации, контролирующих эти компромиссы. Обычно речь идет о компромиссе между степенью важности надежного и согласованного хранения сообщений и другими важными сооб-

ражениями, такими как доступность, высокая пропускная способность, малое значение задержки и стоимость аппаратного обеспечения. Далее мы рассмотрим механизм репликации Kafka, познакомим вас с терминологией и обсудим фундамент надежности платформы. После этого пройдемся по упомянутым параметрам конфигурации.

Репликация

В основе гарантий надежности Kafka лежит механизм репликации, предусматривающий создание нескольких реплик для каждого раздела. Kafka обеспечивает сохраняемость сообщений в случае аварийного сбоя благодаря записи сообщений в несколько реплик.

Мы детально рассмотрели механизм репликации Kafka в главе 5, а здесь вкратце резюмируем основные положения.

Топики Kafka разбиваются на *разделы*, представляющие собой основные стандартные блоки данных. Разделы сохраняются на отдельные диски. Kafka гарантирует упорядоченность событий в пределах раздела, который может быть подключен (доступен) или отключен (недоступен). У каждого раздела может быть несколько реплик, одна из которых назначается ведущей. Все события направляются в ведущую реплику и обычно также потребляются из ведущей реплики. Другие реплики просто должны быть согласованы с ведущей и своевременно реплицировать все недавние события. В случае недоступности ведущей реплики одна из синхронизированных реплик становится новой ведущей (из этого правила есть исключение, о котором мы говорили в главе 6).

Реплика считается согласованной, если она является ведущей репликой раздела или ведомой, которая:

- отправляла в ZooKeeper контрольный сигнал в последние 6 с (настраивается), что означает наличие текущего сеанса связи с ZooKeeper;
- извлекала сообщения из ведущей реплики в последние 10 с (настраивается);
- извлекала наиболее свежие сообщения из ведущей реплики в последние 10 с (настраивается). То есть недостаточно, чтобы ведомая реплика продолжала получать сообщения от ведущей, требуется еще и отсутствие задержки хотя бы один раз за последние 10 с (настраивается).

Если реплика теряет соединение с ZooKeeper, прекращает извлекать новые сообщения или отстает более чем на 10 с, она считается рассогласованной. И снова становится согласованной после повторного подключения к ZooKeeper и догоняет ведущую вплоть до самого свежего сообщения. Обычно

это происходит довольно быстро после восстановления сети от временных неполадок, но может занять и много времени, если брокер, на котором находится реплика, долго не работал.



Рассогласованные реплики

В более старых версиях Kafka нередко можно было наблюдать, как одна или несколько реплик быстро переходили из состояния синхронизации в несинхронизированное. Это было верным признаком того, что в кластере что-то не так. Довольно распространенной причиной был большой максимальный размер запроса и большая куча JVM, что требовало настройки для предотвращения длительных пауз сборки мусора, которые могли привести к временному отключению брокера от ZooKeeper. В настоящее время эта проблема встречается очень редко, особенно при использовании Apache Kafka версии 2.5.0 и выше с настройками по умолчанию для тайм-аута соединения ZooKeeper и максимального времени задержки реплики. Применение JVM версии 8 и выше (теперь это минимальная версия, поддерживаемая Kafka) со сборщиком мусора G1 (<https://oreil.ly/oDL86>) помогло устранить эту проблему, хотя для больших сообщений все еще может потребоваться настройка. В целом протокол репликации Kafka стал значительно более надежным за годы, прошедшие с момента публикации первого издания этой книги. Подробную информацию об эволюции протокола репликации Kafka можно найти в отличном докладе Джейсона Густафсона (Jason Gustafson) «Усиление репликации Apache Kafka» (<https://oreil.ly/Z1R1w>) и обзоре улучшений Kafka от Гвен Шапира «Пожалуйста, обновите Apache Kafka сейчас» (<https://oreil.ly/vKnVl>).

Чуть-чуть отстающая согласованная реплика может замедлять работу производителей и потребителей, поскольку они считают сообщение *зафиксированным* только после его получения всеми согласованными репликами. А когда реплика становится рассогласованной, то ждать получения ею сообщений не надо. Она по-прежнему отстает, но на производительность это больше не влияет. Нюанс в том, что чем меньше согласованных реплик, тем ниже фактический коэффициент репликации, а следовательно, выше риск простоя или потери данных.

В следующем разделе мы увидим, что это значит на практике.

Настройка брокера

На поведение Kafka в смысле надежного хранения сообщений влияют три параметра конфигурации. Как и многие другие переменные конфигурации брокера, их можно использовать на уровне брокера для управления настройками всех топиков системы, а также на уровне отдельных топиков.

Возможность управлять связанными с надежностью компромиссами на уровне отдельных топиков означает, что один и тот же кластер Kafka можно использовать как для надежных, так и для ненадежных топиков. Например, в банке администратор, вероятно, захочет установить очень высокий уровень надежности по умолчанию для всего кластера, за исключением топиков, в которых хранятся жалобы пользователей, где определенные потери данных допустимы.

Рассмотрим эти параметры по очереди и выясним, как они влияют на надежность хранения данных в Kafka и на какие компромиссы приходится идти.

Коэффициент репликации

Соответствующий параметр уровня топика называется `replication.factor`. А на уровне брокера для автоматически создаваемых топиков используется параметр `default.replication.factor`.

До сих пор мы предполагали, что коэффициент репликации топиков равен 3, то есть каждый раздел реплицируется три раза на трех различных брокерах. Это было вполне разумное допущение, соответствующее умолчаниям Kafka, но пользователи могут менять это поведение. Даже после создания топика можно добавлять или удалять реплики, меняя таким образом коэффициент репликации, с помощью инструмента назначения реплик Kafka.

Коэффициент репликации N означает возможность потери $N - 1$ брокеров при сохранении чтения из топика и записи в него. Так что повышение коэффициента репликации означает повышение доступности и надежности и снижение количества аварийных ситуаций. В то же время для обеспечения равного N коэффициента репликации нам понадобится как минимум N брокеров и придется хранить N копий данных, то есть нужно будет в N раз больше дискового пространства. Фактически мы повышаем доступность за счет дополнительного аппаратного обеспечения.

Как же нам определить правильное число реплик для топика? Есть несколько ключевых соображений.

- *Доступность.* Раздел, содержащий только одну реплику, станет недоступным даже при обычном перезапуске одного брокера. Чем больше у нас реплик, тем более высокой доступности мы можем ожидать.
- *Долговечность.* Каждая реплика — это копия всех данных в разделе. Если в разделе есть одна-единственная реплика и диск по какой-либо причине становится непригодным для использования, мы теряем все данные в разделе. При большем количестве копий, особенно на разных устройствах хранения, вероятность потери всех копий уменьшается.

- *Пропускная способность.* С каждой дополнительной репликой мы умножаем межброкерский трафик. Если мы передаем данные в раздел со скоростью 10 Мбайт/с, то одна реплика не будет генерировать никакого репликационного трафика. Если у нас две реплики, то трафик репликации будет 10 Мбайт/с, при трех репликах — 20 Мбайт/с, а при пяти — 40 Мбайт/с. Это необходимо учитывать при планировании размера и пропускной способности кластера.
- *Сквозная задержка.* Каждая созданная запись должна быть реплицирована во все синхронизированные реплики, прежде чем станет доступной для потребителей. Теоретически при большем количестве реплик существует более высокая вероятность того, что одна из этих реплик будет немного медленней и, следовательно, станет замедлять работу потребителей. На практике, если один брокер становится медленным по какой-либо причине, он будет замедлять работу каждого клиента, который пытается его использовать, независимо от коэффициента репликации.
- *Стоимость.* Это наиболее распространенная причина применения коэффициента репликации ниже 3 для некритических данных. Чем больше у нас копий данных, тем выше затраты на хранение и сеть. Поскольку многие системы хранения уже реплицируют каждый блок три раза, иногда имеет смысл снизить затраты, настроив Kafka с коэффициентом репликации 2. Обратите внимание на то, что это все равно снизит доступность по сравнению с коэффициентом репликации 3, но долговечность будет гарантирована устройством хранения.

Размещение реплик также играет важную роль. Kafka всегда размещает каждую реплику раздела на отдельном брокере. В некоторых случаях этот вариант недостаточно безопасен. Если все реплики раздела размещены на брокерах в одной стойке, в случае сбоя коммутатора верхней части стойки раздел станет недоступен вне зависимости от коэффициента репликации. Для защиты от подобных проблем на уровне стойки мы рекомендуем распределять брокеры по нескольким стойкам и использовать параметр конфигурации брокеров `broker.rack`, чтобы задавать имя стойки для каждого брокера. При заданных именах стоек Kafka обеспечит распределение реплик раздела по нескольким стойкам, что гарантирует еще более высокую доступность. При работе Kafka в облачной среде принято рассматривать зоны доступности как отдельные стойки. В главе 6 мы подробно описали, как платформа распределяет реплики по брокерам и стойкам.

«Нечистый» выбор ведущей реплики

Параметр, доступный только на уровне брокера (и на практике для всего кластера), называется `unclean.leader.election.enable`. По умолчанию его значение — `false`.

Как объяснялось ранее, если ведущая реплика раздела становится недоступной, одна из согласованных реплик выбирается новой ведущей. Такой выбор

ведущей реплики является «чистым» в смысле гарантий отсутствия потерь данных — по определению зафиксированные данные имеются во всех согласованных репликах.

Но что делать, если нет никаких согласованных реплик, кроме только что ставшей недоступной ведущей?

Такая ситуация может возникнуть при одном из двух сценариев.

- У раздела три реплики, две ведомые реплики стали недоступными (например, два брокера отказали). В этом случае, поскольку производители продолжают записывать данные на ведущую реплику, получение всех сообщений подтверждается и они фиксируются, так как ведущая реплика является единственной согласованной. Теперь предположим, что ведущая реплика становится недоступной (упс, еще один брокер отказал). Если при таком сценарии развития событий сначала запустится одна из несогласованных ведомых реплик, единственной доступной репликой для раздела окажется несогласованная.
- У раздела три реплики, и из-за проблем с сетью две ведомые отстали так, что хотя они работают и выполняют репликацию, но уже не согласованы. Ведущая реплика продолжает получать сообщения как единственная согласованная. Если теперь ведущая реплика станет недоступной, лидерами могут стать только несинхронизированные реплики.

При развитии обоих сценариев необходимо принять непростое решение.

- Если мы запрещаем рассогласованным репликам становиться ведущими, то раздел будет отключен до тех пор, пока не восстановим работу старой ведущей реплики. В некоторых случаях (например, при необходимости замены модуля памяти) это может занять многие часы.
- Если разрешить рассогласованной реплике стать ведущей, то мы потеряем все сообщения, записанные на старую ведущую реплику за то время, пока она была рассогласованной, а заодно получим проблемы с рассогласованием на потребителях. Почему? Представьте себе, что за время недоступности реплик 0 и 1 мы записали сообщения со смещениями 100–200 на реплику 2, ставшую затем ведущей. Теперь реплика 2 недоступна, а реплика 0 вернулась в работу. На реплике 0 содержатся только сообщения 0–100, но нет сообщений 100–200. Так что у новой ведущей реплики будут совершенно новые сообщения 100–200. Отметим, что часть потребителей могли уже прочитать старые сообщения 100–200, часть — новые, а часть — некую их смесь. Это приведет к довольно неприятным последствиям с точки зрения дальнейших отчетов. Кроме того, реплика 2 вернется в работу и станет ведомой у новой ведущей реплики. При этом она удалит все сообщения, которых не существует на текущем лидере. В дальнейшем никто из потребителей не сможет их увидеть.

Резюмируя: разрешение рассогласованным репликам становиться ведущими увеличивает риск потери данных и того, что они станут противоречивыми. Если же запретить это, уменьшится доступность из-за необходимости ждать, пока первоначальная ведущая реплика станет доступной и можно будет восстановить работу раздела.

По умолчанию параметру `unclean.leader.election.enable` установлено значение `false`, что не позволит рассогласованным репликам стать лидерами. Это самый безопасный вариант, поскольку он обеспечивает наилучшие гарантии от потери данных. Однако это означает, что в экстремальных сценариях недоступности, которые мы описывали ранее, некоторые разделы останутся недоступными до тех пор, пока не будут восстановлены вручную. Администратор всегда может оценить ситуацию, принять решение о потере данных, чтобы сделать разделы доступными, и переключить эту конфигурацию на значение `true` перед запуском кластера. Только не забудьте вернуть значение `false` после восстановления кластера.

Минимальное число согласованных реплик

Соответствующий параметр как уровня топика, так и уровня брокера называется `min.insync.replicas`.

Как мы уже видели, в некоторых случаях даже при трех репликах в топике согласованной может остаться только одна. В случае ее недоступности придется выбирать между доступностью и согласованностью. Это всегда непростой выбор. Проблема усугубляется следующим: так как Kafka гарантирует надежность, данные считаются зафиксированными после их записи во все согласованные реплики, даже если такая реплика только одна, и в случае ее недоступности они будут утеряны.

Если нам нужно гарантировать, что зафиксированные данные будут записаны более чем в одну реплику, можно задать более высокое минимальное число согласованных реплик. Если в топике три реплики и для параметра `min.insync.replicas` установлено значение 2, то производители смогут записывать в раздел топика только тогда, когда по крайней мере две из трех реплик согласованы.

Если все три реплики согласованы, то все работает нормально. То же самое будет и в случае недоступности одной из реплик. Однако если недоступны две из трех реплик, брокер перестанет принимать запросы производителей. Вместо этого производителям, пытающимся отправить данные, будет возвращено исключение `NotEnoughReplicasException`. При этом потребители могут продолжать читать существующие данные. Фактически в подобной ситуации единственная согласованная реплика превращается в реплику только для чтения. Это предотвращает

нежелательную ситуацию, при которой данные производятся и потребляются лишь для того, чтобы пропасть в никуда при «нечистом» выборе. Для выхода из состояния «только для чтения» мы должны вновь обеспечить доступность одной из двух недоступных реплик (возможно, перезапустить брокер) и подождать, пока она нагонит упущенное и станет согласованной.

Поддержание синхронизации реплик

Как упоминалось ранее, рассинхронизация реплик снижает общую надежность, поэтому важно по возможности избегать ее. Мы также объяснили, что реплика может рассинхронизироваться одним из двух способов: либо она теряет связь с ZooKeeper, либо не успевает за лидером и создает задержку репликации. В Kafka есть две конфигурации брокера, которые контролируют чувствительность кластера к этим двум условиям.

Параметр `zookeeper.session.timeout.ms` — это интервал времени, в течение которого брокер Kafka может прекратить посылать контрольные сигналы в ZooKeeper без того, чтобы ZooKeeper посчитал брокер мертвым и удалил его из кластера. В версии 2.5.0 это значение было увеличено с 6 до 18 с, чтобы повысить стабильность кластеров Kafka в облачных средах, где сетевые задержки имеют более высокую дисперсию. В целом мы хотим, чтобы это время было достаточно большим, чтобы избежать случайных зависаний, вызванных сборкой мусора или состоянием сети, но при этом достаточно низким, чтобы гарантировать, что брокеры, которые действительно заморожены, будут своевременно обнаружены.

Если реплика не получала сообщения от лидера или не отслеживала последние сообщения от лидера дольше, чем указано в значении параметра `replica.lag.time.max.ms`, она становится рассинхронизированной. Это значение было увеличено с 10 до 30 с в версии 2.5.0, чтобы повысить устойчивость кластера и избежать ненужного зависания. Обратите внимание на то, что это более высокое значение влияет также на максимальную задержку для потребителей — при более высоком значении может потребоваться до 30 с, пока сообщение дойдет до всех реплик и потребители смогут его использовать.

Долговременное хранение на диске

Мы уже несколько раз упоминали, что Kafka подтверждает сообщения, которые не были сохранены на диск, в зависимости от количества реплик, получивших сообщение. Kafka будет сбрасывать сообщения на диск при ротации сегментов (по умолчанию размером 1 Гбайт) и перед перезагрузками, но в остальном будет полагаться на кэш страниц Linux для сброса сообщений, когда он заполняется.

Идея заключается в том, что наличие трех машин в отдельных стойках или зонах доступности, каждая из которых имеет копию данных, более безопасно, чем запись сообщений на диск в лидере, поскольку одновременные сбои на двух разных стойках или в зонах маловероятны. Однако можно настроить брокеры на более частое сохранение сообщений на диск. Конфигурационный параметр `flush.messages` позволяет нам контролировать максимальное количество сообщений, не синхронизированных на диск, а `flush.ms` — частоту синхронизации на диск. Прежде чем использовать эту функцию, стоит прочесть, как `fsync` влияет на производительность Kafka и как уменьшить его недостатки (<https://oreil.ly/Ai1hl>).

Использование производителей в надежной системе

Даже если конфигурация брокеров самая надежная из всех возможных, система в целом потенциально может терять данные, если не настроить производители достаточно надежно.

Вот два возможных сценария для иллюстрации сказанного.

- Брокеры настроены на использование трех реплик, а возможность «нечистого» выбора ведущей реплики отключена. Так что мы вроде бы не должны потерять ни одного сообщения, зафиксированного в кластере Kafka. Однако производитель настроили так, чтобы отправлять сообщения с `acks=1`. Мы отправили сообщение с производителя, и оно уже было записано на ведущую реплику, но еще не было занесено на ведомые согласованные реплики. Ведущая реплика вернула производителю ответ, гласящий: «Сообщение было записано успешно», и сразу же после этого, еще до репликации данных на другие реплики, потерпела аварийный сбой. Другие реплики по-прежнему считаются согласованными (как вы помните, до объявления реплики рассогласованной проходит некоторое время), и одна из них становится ведущей. Так как сообщение на них не записано, оно было утрачено. Но приложение-производитель считает, что оно записано успешно. Система согласована, поскольку ни один потребитель сообщения не видит (оно так и не было зафиксировано, поэтому реплики его не получили), но с точки зрения производителя сообщение было потеряно.
- Брокеры настроены на использование трех реплик, а возможность «нечистого» выбора ведущей реплики отключена. Мы извлекли урок из своей ошибки и стали генерировать сообщения с `acks=all`. Допустим, мы пытаемся записать сообщение в Kafka, но ведущая реплика раздела, в который записываются данные, только что потерпела аварийный сбой, а новая еще не выбрана. Kafka вернет ошибку «Ведущая реплика недоступна». Если при этом про-

изводитель не поступит должным образом и не будет пытаться отправить сообщение вплоть до успешного выполнения операции записи, оно может быть потеряно. Опять же это не проблема надежности брокера, поскольку он вообще не получал сообщения, и не проблема согласованности, потому что потребители тоже его не получали. Но если производители не обрабатывают ошибки должным образом, то могут столкнуться с потерей данных.

Как показывают эти примеры, все, кто пишет приложения, которые служат производителями для Kafka, должны обращать внимание на две вещи:

- использование соответствующего требованиям надежности значения параметра `acks`;
- правильную обработку ошибок как в настройках, так и в исходном коде.

Конфигурации производителей мы подробно обсуждали в главе 3, но остановимся на важнейших нюансах еще раз.

Отправка подтверждений

Производители могут выбрать один из трех режимов подтверждения.

- `acks=0` означает, что сообщение считается успешно записанным в Kafka, если производитель сумел отправить его по сети. Возможно возникновение ошибок, если отправляемый объект не удастся сериализовать или произошел сбой сетевой карты. Но если раздел находится в автономном режиме, выполняется выбор лидера или даже если весь кластер Kafka недоступен, никакие ошибки возвращены не будут. Работа при `acks=0` имеет низкую задержку производства (именно поэтому мы видим много сравнительных тестов с такой конфигурацией), но это не улучшает сквозную задержку (помните, что потребители не увидят сообщения, пока они не будут реплицированы во все доступные реплики).
- `acks=1` означает, что ведущая реплика в момент получения сообщения и записи его в файл данных раздела (но не обязательно на диск) отправила подтверждение или сообщение об ошибке. Возможна потеря данных в случае выключения или аварийного сбоя ведущей реплики, если часть успешно записанных на нее и подтвержденных сообщений не были реплицированы на ведомые реплики до сбоя. При такой конфигурации можно также выполнять запись на лидер быстрее, чем он может реплицировать сообщения, и в итоге получить недостаточно реплицированные разделы, поскольку лидер будет подтверждать сообщения от производителя, прежде чем реплицировать их.
- `acks=all` означает, что ведущая реплика, прежде чем отправлять подтверждение или сообщение об ошибке, дожидается получения сообщения всеми

согласованными репликами. В сочетании с параметром `min.insync.replicas` на брокере это позволяет нам контролировать число реплик, которые должны получить сообщение для его подтверждения. Это самый безопасный вариант — производитель будет пытаться отправить сообщение вплоть до его полной фиксации. Он характеризуется и наибольшей задержкой производителя — тот ждет получения сообщения всеми синхронизированными репликами, прежде чем отметить пакет сообщений как обработанный и продолжить работу.

Настройка повторов отправки производителями

Обработка ошибок на стороне производителя состоит из двух частей: автоматической обработки производителем и обработки с помощью библиотеки производителя, которую должны выполнять вы как разработчик.

Сам производитель может справиться с *возвращаемыми* брокером ошибками. При отправке производителем сообщения брокеру последний может вернуть или код, соответствующий успешному выполнению, или код ошибки. Коды ошибок делятся на две категории: коды ошибок, которые можно разрешить путем повтора отправки, и коды ошибок, которые разрешить нельзя. Например, если брокер вернул код ошибки `LEADER_NOT_AVAILABLE`, то производитель может попробовать повторить отправку сообщения в надежде, что выбор уже сделан и вторая попытка завершится успешно. Это значит, что `LEADER_NOT_AVAILABLE` — *ошибка, которую можно разрешить путем повтора отправки* (retriable error). Но если брокер вернул исключение `INVALID_CONFIG`, то повтор отправки того же сообщения никак не поменяет настройки. Это пример *ошибки, которую нельзя разрешить путем повтора* (nonretriable error).

В общем, если наша цель — не терять ни одного сообщения, то лучше всего настроить производитель на повторные отправки сообщений в тех случаях, когда он сталкивается с ошибкой, которую можно разрешить путем повтора отправки. И лучший подход к повторным попыткам, как рекомендовано в главе 3, заключается в том, чтобы оставить количество повторных попыток равным текущему значению по умолчанию (`MAX_INT`, или фактически бесконечному) и использовать параметр `delivery.timeout.ms` для настройки максимального количества времени, которое мы готовы ждать до отказа от отправки сообщения, — производитель будет повторять попытку отправки сообщения столько раз, сколько возможно в течение этого промежутка времени.

Попытка повтора отправки сообщения в случае неудачи означает риск того, что оба сообщения будут успешно записаны на брокер, а это приведет к дублированию. Повторы отправки и тщательная обработка ошибок позволяют гарантировать сохранение сообщения *по крайней мере один раз*, но не *ровно один раз*. Использование параметра `enable.idempotence=true` приведет к тому, что произ-

водитель включит дополнительную информацию в свои записи, которую брокеры будут использовать для пропуска повторяющихся сообщений, вызванных повторными попытками. В главе 8 мы подробно обсудим, как и когда это работает.

Дополнительная обработка ошибок

С помощью встроенного в производители механизма повторов можно легко должным образом обработать множество разнообразных ошибок без потери сообщений, но, как разработчикам, нам нужно иметь возможность обрабатывать и другие типы ошибок, включающие:

- ошибки брокеров, которые нельзя разрешить путем повтора отправки, например ошибки, связанные с размером сообщений, ошибки авторизации и т. п.;
- ошибки, произошедшие до отправки сообщения брокеру, например ошибки сериализации;
- ошибки, связанные с тем, что производитель достиг предельного количества попыток повтора отправки или исчерпал во время этого доступную ему память на хранение сообщений;
- тайм-ауты.

В главе 3 мы обсуждали написание обработчиков ошибок как для синхронного, так и для асинхронного метода отправки сообщений. Логика этих обработчиков ошибок меняется в зависимости от вашего приложения и его задач: выбрасываем ли мы «плохие» сообщения? Заносим ли ошибки в журнал? Прекращаем ли чтение сообщений из исходной системы? Применяем ли к исходной системе обратное давление, чтобы прекратить отставку сообщений на некоторое время? Хранить ли эти сообщения в каталоге на локальном диске? Эти решения зависят от вашей архитектуры и требований к продуктам. Просто отметим, что если все, что делает обработчик ошибок, — это повторная попытка отправить сообщение, то в этом лучше положиться на функциональность повторной попытки производителей.

Использование потребителей в надежной системе

Теперь, разобравшись, как учесть гарантии надежности Kafka при работе с производителями данных, можно обсудить потребление данных.

Как мы видели в первой части главы, данные становятся доступными потребителям лишь после их фиксации в Kafka, то есть записи во все согласованные реплики. Это значит, что потребителям поступают заведомо согласованные данные. Им остается лишь обеспечить учет того, какие сообщения они уже прочитали, а какие — нет. Это ключ к тому, чтобы не терять сообщения при потреблении.

Во время чтения данных из раздела потребитель извлекает пакет сообщений, находит в нем последнее смещение и запрашивает следующий пакет сообщений, начиная с последнего полученного смещения. Благодаря этому потребители Kafka всегда получают новые данные в правильной последовательности и не пропускают событий.

В случае остановки потребителя другому потребителю понадобится информация о том, с какого места продолжить работу, — каково последнее из обработанных предыдущим потребителем перед остановом смещений. Этот другой потребитель может даже оказаться тем же самым потребителем, только перезапущенным. Это неважно: какой-то потребитель продолжит получать данные из этого раздела, и ему необходимо знать, с какого смещения начинать работу. Именно поэтому потребители должны фиксировать обработанные смещения. Потребитель для каждого раздела, из которого берет данные, сохраняет текущее местоположение, так что после перезагрузки он или другой потребитель будет знать, с какого места продолжить работу. Потребители в основном теряют сообщения, когда фиксируют смещения для прочитанных, но еще не полностью обработанных событий. В этом случае другой потребитель, продолжающий работу, пропустит эти сообщения и они так никогда и не будут обработаны. Именно поэтому чрезвычайно важно тщательно отслеживать, когда и как фиксируются смещения.



Фиксация сообщений и фиксация событий

Зафиксированное смещение отличается от зафиксированного сообщения (committed message), которое, как обсуждалось ранее, представляет собой сообщение, записанное во все согласованные реплики и доступное потребителям. Зафиксированные смещения (committed offsets) — это смещения, отправленные потребителем в Kafka в подтверждение получения и обработки ею всех сообщений в разделе вплоть до этого конкретного смещения.

В главе 4 мы подробно обсуждали API потребителей и видели множество методов фиксации смещений. Здесь рассмотрим некоторые важные соображения и доступные альтернативы, но за подробностями использования API отправляем вас к главе 4.

Свойства конфигурации потребителей, важные для надежной обработки

Существует четыре параметра конфигурации потребителей, без понимания которых не получится настроить их достаточно надежное поведение.

Первый из них, `group.id`, очень подробно описан в главе 4. Основная его идея: если у двух потребителей одинаковый идентификатор группы и они подписаны

на один топик, каждый получает в обработку подмножество разделов топика и будет читать только часть сообщений, но группа в целом прочитает все сообщения. Если нам необходимо, чтобы отдельный потребитель увидел каждое из сообщений топика, то у него должен быть уникальный `group.id`.

Второй параметр — `auto.offset.reset`. Он определяет, что потребитель будет делать, если никаких смещений не было зафиксировано (например, при первоначальном запуске потребителя) или он запросил смещения, которых нет в брокере (почему так случается, вы можете узнать из главы 4). У этого параметра есть только два значения. Если выбрать `earliest`, то при отсутствии корректного смещения потребитель начнет с начала раздела. Это приведет к повторной обработке множества сообщений, но гарантирует минимальные потери данных. Если же выбрать `latest`, то потребитель начнет с конца раздела. Это минимизирует повторную обработку, но почти наверняка приведет к пропуску потребителем некоторых сообщений.

Третий из этих параметров — `enable.auto.commit`. Нужно принять непростое решение: разрешить ли потребителю фиксировать смещения вместо нас по расписанию или самостоятельно фиксировать смещения в своем коде. Основное преимущество автоматической фиксации смещений в том, что при использовании потребителей в нашем приложении одной заботой окажется меньше. Если выполнять всю обработку прочитанных записей внутри цикла опроса потребителя, то автоматическая фиксация смещений гарантирует невозможность случайной фиксации необработанного смещения. Основной недостаток автоматической фиксации смещений — отсутствие контроля числа дубликатов, которые, возможно, придется обработать приложению, поскольку оно остановилось после обработки части записей, но до запуска автоматической фиксации. Если приложение имеет более сложную обработку наподобие передачи записей другому потоку для обработки в фоновом режиме, нет другого выбора, кроме как использовать ручную фиксацию смещения, поскольку автоматическая может зафиксировать смещения для уже прочитанных, но, возможно, еще не обработанных потребителем записей.

Четвертый параметр называется `auto.commit.interval.ms`, и он связан с третьим. Если мы выберем автоматическую фиксацию смещений, то этот параметр даст вам возможность настроить ее частоту. Значение по умолчанию — каждые 5 с. В целом более частая фиксация увеличивает вычислительные расходы, но снижает число дубликатов, которые могут возникать при останове потребителя.

Хотя это не имеет прямого отношения к надежной обработке данных, трудно считать потребителя надежным, если он часто прекращает потребление для восстановления баланса. Глава 4 содержит рекомендации о том, как настроить потребителей таким образом, чтобы свести к минимуму ненужную перебалансировку и паузы при перебалансировке.

Фиксация смещений в потребителях явным образом

Если мы решим, что нам нужен больший контроль, и выберем фиксацию смещений вручную, то нужно будет позаботиться о корректности и последствиях для производительности.

Мы не станем рассматривать здесь механизм и API фиксации смещений, поскольку уже сделали это в главе 4. Вместо этого обсудим важные соображения относительно разработки потребителей для надежной обработки данных. Начнем с простых и, наверное, очевидных вещей и постепенно перейдем к более сложным схемам.

Всегда фиксируйте смещения после обработки сообщений

Это не составит проблемы, если вся обработка происходит внутри цикла опроса, а состояние между итерациями цикла опроса не сохраняется (например, производится агрегирование). Можно воспользоваться параметром автоматической фиксации, или фиксировать смещения в конце цикла опроса, или фиксировать смещения внутри цикла с частотой, которая уравнивает требования как к накладным расходам, так и к отсутствию дублирующей обработки. Если в процесс вовлечены дополнительные потоки или обработка с сохранением состояний, это становится сложнее, особенно если объект-потребитель не является потокобезопасным. В главе 4 мы обсудили, как это можно сделать, и предоставили ссылки с дополнительными примерами.

Частота фиксации — компромисс между производительностью и числом дубликатов, возникающих при аварийном сбое

Даже в простейшем случае, когда вся обработка происходит внутри цикла опроса, а состояние между итерациями цикла опроса не сохраняется, можно или выполнять фиксацию несколько раз внутри цикла, или фиксировать точно один раз в несколько итераций. Фиксация, подобно отправке сообщений при `acks=all`, обуславливает значительные накладные расходы на производительность, но все фиксации смещений одной группы потребителей передаются одному и тому же брокеру, который может оказаться перегруженным. Частота фиксации должна сбалансировать требования к производительности и отсутствию дубликатов. Фиксация после каждого сообщения должна выполняться только в топиках с очень низкой пропускной способностью.

Фиксируйте правильные смещения в нужное время

Распространенная ошибка при фиксации во время цикла опроса состоит в случайной фиксации последнего прочитанного при опросе смещения, а не смещения после обработки последнего смещения. Помните, что чрезвычайно важно всегда

фиксировать смещения сообщений после их обработки — фиксация смещений для прочитанных, но еще не обработанных сообщений может привести к потере сообщений потребителями. В главе 4 приведены примеры, показывающие, как это сделать.

Переназначение

При проектировании приложения необходимо помнить, что время от времени будут происходить переназначение потребителей и вам придется обрабатывать их должным образом. В главе 4 приведены несколько примеров. Обычно это включает в себя фиксацию смещений перед отменой разделов и очистку любого состояния, которое приложение сохраняет при назначении ему новых разделов.

Потребителям может понадобиться повторить попытку

В некоторых случаях после выполнения опроса и обработки записей оказывается, что часть записей обработана не полностью и их придется обработать позже. Допустим, мы пытались перенести записи из Kafka в базу данных, но оказалось, что она в данный момент недоступна, так что нужно будет повторить попытку. Отметим, что, в отличие от обычных систем обмена сообщениями по типу «публикация/подписка», потребители Kafka фиксируют смещения, но не подтверждают получение отдельных сообщений. Это значит, что если мы не смогли обработать запись № 30, но успешно обработали запись № 31, то фиксировать 31-е смещение не следует — это приведет к маркировке как обработанных всех записей до 31-й, включая 30-ю, что нежелательно. Вместо этого попробуйте один из следующих вариантов.

1. Столкнувшись с ошибкой, которую можно разрешить путем повтора, зафиксируйте последнюю успешно обработанную запись. Затем сохраните ожидающие обработки записи в буфере (чтобы следующая итерация опроса их не затерла), воспользуйтесь методом `pause()` потребителя для упрощения повторов, чтобы гарантировать, что дополнительные опросы не вернут данные, и продолжайте попытки обработки записей.
2. Столкнувшись с ошибкой, которую можно разрешить путем повтора, запишите ее в отдельный топик и продолжайте выполнение. Для обработки записей из этого топика для повторов можно воспользоваться отдельной группой потребителей. Или один и тот же потребитель может подписаться как на основной топик, так и на топик для повторов с приостановкой между повторами потребления данных из топика для повторов. Эта схема работы напоминает очереди зависших сообщений (`dead-letter queue`), используемые во многих системах обмена сообщениями.

Потребителям может потребоваться сохранение состояния

В некоторых приложениях необходимо сохранять состояние между вызовами опроса. Например, если нужно вычислить скользящее среднее, приходится обновлять значение среднего при каждом опросе Kafka на предмет новых сообщений. В случае перезапуска процесса необходимо не только начать получение с последнего смещения, но и восстановить соответствующее скользящее среднее. Сделать это можно, в частности записав последнее накопленное значение в топик для результатов одновременно с фиксацией смещения приложением. Это значит, что поток может начать работу с того места, где остановился, подхватив последнее накопленное значение. В главе 8 мы обсудим, как приложение может записывать результаты фиксации смещений в одной транзакции. В целом это довольно сложная проблема, и мы рекомендуем обратиться к таким библиотекам, как Kafka Streams или Flink, предоставляющим высокоуровневые DSL-подобные API для агрегирования, реализации соединений, оконных функций и другой сложной аналитики.

Проверка надежности системы

Пройдя процесс выяснения требований надежности, настроив брокеры и клиенты, воспользовавшись API оптимальным для конкретного сценария использования образом, можно расслабиться и запускать систему в промышленную эксплуатацию в полной уверенности, что ни одно событие не будет пропущено, правда?

Мы рекомендуем сначала выполнить хотя бы небольшую проверку и советуем выполнять три уровня проверки: проверку конфигурации, проверку приложения и мониторинг приложения при промышленной эксплуатации. Рассмотрим каждый из этих этапов и разберемся, что нужно проверять и как.

Проверка конфигурации

Можно легко протестировать настройки брокера и клиента независимо от логики приложения, так рекомендуется поступить по двум причинам.

- Благодаря этому можно проверить, соответствует ли выбранная конфигурация нашим требованиям.
- Это хорошее упражнение на прослеживание ожидаемого поведения системы.

Kafka включает две утилиты, предназначенные для такой проверки. Пакет `org.apache.kafka.tools` включает классы `VerifiableProducer` и `VerifiableCon-`

sumer. Их можно запускать в виде утилит командной строки или встраивать во фреймворк автоматизированного тестирования.

Смысл процедуры состоит в генерации контрольным производителем последовательности сообщений с номерами от 1 до выбранного вами значения. Этот контрольный производитель мы можем настраивать точно так же, как и свой собственный, задавая нужное значение параметра `acks`, количество попыток повторов, параметр `delivery.timeout.ms` и частоту, с которой генерируются сообщения. Он выведет для каждого отправленного брокеру сообщения уведомление об ошибке или успехе отправки в зависимости от полученных подтверждений. Контрольный потребитель позволяет выполнить дополнительную проверку. Он потребляет события (обычно исходящие от контрольного производителя) и выводит их в соответствующем порядке. А также выводит информацию о фиксациях и переназначении.

Важно также задуматься о том, какие тесты имеет смысл выполнить. Например, такие.

- Выбор ведущей реплики: что произойдет, если мы остановим ведущую реплику? Сколько времени займет возобновление нормальной работы производителя и потребителя?
- Выбор контроллера: через какое время система возобновит работу после перезапуска контроллера?
- Плавающий перезапуск: можно ли перезапускать брокеры по одному без потери сообщений?
- «Нечистый» выбор ведущей реплики: что произойдет, если отключать все реплики раздела по одной, чтобы они точно переставали быть согласованными, после чего запустить несогласованный брокер? Что должно произойти для возобновления работы? Приемлемо ли это?

Выбрав сценарий тестирования, мы запускаем контрольный производитель и контрольный потребитель и выполняем выбранный сценарий — например, останавливаем ведущую реплику раздела, для которой генерирует данные производитель. Если мы ожидаем лишь небольшой паузы, после которой функционирование возобновится без потери каких-либо данных, то нужно проверить, совпадает ли число сообщений, сгенерированных производителем, и число сообщений, потребленных потребителем.

Репозиторий исходного кода Apache Kafka включает обширный набор тестов (<https://oreil.ly/IjX8>). Многие из них основаны на одном и том же принципе и используют контрольный производитель и контрольный потребитель для проверки функционирования плавающих обновлений.

Проверка приложений

Убедившись, что настройки брокера и клиента соответствуют требованиям, мы можем приступить к проверке того, обеспечивает ли приложение необходимые гарантии. Это включает проверку таких вещей, как пользовательский код обработки ошибок, фиксация смещений, переназначение прослушивателей и других мест, в которых логика приложения взаимодействует с клиентскими библиотеками Kafka.

Конечно, поскольку логика приложения может значительно отличаться от нашей, вам виднее, как его тестировать, мы можем лишь подсказать некоторые вещи. Рекомендуем использовать интеграционные тесты для приложения как часть любого процесса разработки и запускать их при различных сбойных состояниях:

- при потере клиентами соединения с одним из брокеров;
- большой задержке между клиентом и брокером;
- заполнении диска;
- зависании диска, также называемом затмением (brown out);
- выборе ведущей реплики;
- плавающем перезапуске брокеров;
- плавающем перезапуске потребителей;
- плавающем перезапуске производителей.

Существует множество инструментов, которые можно использовать для выявления сбоев в сети и на диске, многие из них превосходны, поэтому мы не будем пытаться давать конкретные рекомендации. Сама Apache Kafka включает в себя тестовый фреймворк Trogdor (<https://oreil.ly/P3ai1>) для внедрения неисправностей. В каждом из этих сценариев существует *ожидаемое поведение* — то, что мы хотели получить, когда создавали приложение. Затем мы выполняем тест, чтобы увидеть, что произойдет на самом деле. Например, когда вы планировали плавающий перезапуск потребителей, то ожидали небольшой паузы вследствие переназначения потребителей, после которой потребление продолжилось бы при не более чем 1000 дублирующихся значений. Наш тест покажет, действительно ли приложение фиксирует смещения и выполняет переназначение подобным образом.

Мониторинг надежности при промышленной эксплуатации

Тестирование приложения играет важную роль, но не заменяет необходимости непрерывного мониторинга системы для контроля потоков данных при промышленной эксплуатации. В главе 12 приведены подробные советы по мониторингу состояния кластера Kafka, но, помимо этого, важно контролировать также клиенты и потоки данных в системе.

Java-клиенты Kafka включают показатели JMX, позволяющие выполнять мониторинг событий и состояния клиентов. Два наиболее важных для производителей показателя — число ошибок и число повторов в секунду (агрегированные). Следите за ними, ведь рост числа ошибок или повторов означает проблему с системой в целом. По журналам производителей также отслеживайте ошибки отправки событий, помеченные как **WARN**, которые выглядят примерно так: «Получен ответ об ошибке при генерации сообщения с идентификатором корреляции 5689 в разделе топика [топик-1, 3], повторяю попытку (осталось две попытки). Ошибка...» (Got error produce response with correlation id 5689 on topic-partition [topic-1,3], retrying (two attempts left). Error...). Когда мы видим события, у которых осталось 0 попыток, это означает, что у производителя закончились повторные попытки. В главе 3 мы обсудили, как настроить параметр `delivery.timeout.ms` и повторные попытки, чтобы улучшить обработку ошибок в производителе и избежать преждевременного исчерпания повторных попыток. Конечно, всегда лучше в первую очередь решить проблему, которая вызвала ошибки. Сообщения журнала уровня **ERROR** на производителе, скорее всего, указывают на то, что отправка сообщения не удалась из-за неустранимой ошибки, устранимой ошибки, которая исчерпала количество повторных попыток, или тайм-аута. Когда это применимо, точная ошибка от брокера также будет записана в журнал.

На стороне потребителя важнейшим показателем является задержка потребителя, показывающая, насколько он отстает от последнего зафиксированного в разделе на брокере сообщения. В идеале задержка всегда должна быть равна 0 и потребитель всегда читает последнее сообщение. На практике же она будет колебаться в определенных пределах вследствие того, что вызов метода `poll()` возвращает несколько сообщений, после чего потребителю приходится тратить время на их обработку, прежде чем извлечь новые. Главное, чтобы потребители в конце концов наверстали упущенное, а не отставали все больше и больше. Из-за ожидаемых колебаний задержки потребителя задание уведомлений на основе этого показателя представляет собой непростую задачу. Упростить ее может утилита проверки задержки Burrow от LinkedIn (<https://oreil.ly/supY1>).

Мониторинг потока данных означает также проверку того, чтобы все сформированные производителями данные были потреблены своевременно, а смысл слова «своевременно» определяется требованиями бизнеса. Чтобы обеспечить это, нам нужно знать, когда данные поступили от производителей. Для облегчения этого Kafka, начиная с версии 0.10.0, включает во все сообщения метку даты/времени (хотя обратите внимание на то, что это может быть переопределено либо приложением, отправляющим события, либо самими брокерами, если они настроены на это).

Чтобы убедиться в том, что все сформированные производителями сообщения были потреблены за приемлемое время, необходимо, чтобы приложение, производящее сообщения, регистрировало число сгенерированных событий

(обычно в виде количества событий в секунду). Потребители должны будут регистрировать как количество событий, потребляемых в единицу времени, так и информацию о задержках между генерацией сообщений и их потреблением на основе меток даты/времени событий. Далее нам понадобится система для сверки количества событий в секунду от производителей и потребителей, чтобы гарантировать, что никакие сообщения не потерялись по дороге, а также чтобы убедиться, что промежутки времени между генерацией и потреблением разумны. Реализация подобных систем сквозного мониторинга не проста и требует значительных затрат времени. Насколько нам известно, подобных систем с открытым исходным кодом не существует, но Confluent предлагает коммерческую реализацию как часть продукта Confluent Control Center (<https://oreil.ly/KnvVW>).

В дополнение к мониторингу клиентов и сквозного потока данных брокеры Kafka включают показатели, которые показывают частоту ответов об ошибках, отправляемых брокерами клиентам. Мы рекомендуем собирать `kafka.server:type=BrokerTopicMetrics, name=FailedProduceRequestsPerSec` и `kafka.server:type=BrokerTopicMetrics, name=FailedFetchRequestsPerSec`. Иногда ожидается определенный уровень ответов с ошибками — например, если мы закрываем брокер на техническое обслуживание, а на другом брокере избираются новые лидеры, ожидается, что производители получат ошибку `NOT_LEADER_FOR_PARTITION`, которая заставит их запросить обновленные метаданные, прежде чем продолжать производить события в обычном режиме. Необъяснимое увеличение числа неудачных запросов всегда должно расследоваться. Чтобы помочь в этом, показатели неудачных запросов помечаются конкретным ответом на ошибку, который отправил брокер.

Резюме

Как мы говорили в начале главы, надежность обеспечивается не только конкретными возможностями Kafka. Необходимо сделать надежной систему в целом, включая архитектуру приложения, методы применения приложениями API производителей и потребителей, настройки производителей и потребителей, настройки топиков и брокеров. Обеспечение надежности приложения всегда означает определенный компромисс между сложностью приложения, его производительностью, доступностью и использованием дискового пространства. Разбираясь во всех доступных вариантах, распространенных схемах действий и требованиях для конкретных сценариев, можно принимать разумные решения по поводу нужной степени надежности приложения и развертывания Kafka, а также того, на какие компромиссы имеет смысл пойти в конкретном случае.

Семантика «точно один раз»

В главе 7 мы обсудили параметры конфигурации и лучшие практики, которые позволяют пользователям Kafka контролировать гарантии надежности Kafka. Мы сосредоточились на доставке «по крайней мере один раз» — гарантии того, что Kafka не потеряет сообщения, которые были подтверждены системой в качестве принятых. Это все еще оставляет открытой вероятность дублирования сообщений.

В простых системах, где сообщения создаются, а затем потребляются различными приложениями, дубликаты являются досадной неприятностью, с которой довольно легко справиться. Большинство практических приложений содержат уникальные идентификаторы, которые приложения-потребители могут использовать для дедупликации сообщений.

Ситуация усложняется, когда мы рассматриваем приложения для обработки потоков, которые агрегируют события. При проверке приложения, которое обрабатывает события, вычисляет среднее значение и выдает результаты, часто тем, кто проверяет результаты, становится невозможно обнаружить, что среднее значение неверно, потому что событие было обработано дважды во время вычисления среднего значения. В таких случаях важно обеспечить более надежную гарантию — семантику обработки «точно один раз».

В этой главе мы обсудим, как использовать Kafka с семантикой «точно один раз», рассмотрим рекомендуемые сценарии использования и ограничения. Как и в случае с гарантиями «по крайней мере один раз», мы погрузимся немного глубже и дадим некоторое представление о том, как реализуется эта гарантия. Эти детали можно пропустить при первом чтении главы, но их будет полезно понять перед использованием функции — они помогут прояснить значение различных конфигураций и интерфейсов API и то, как их лучше использовать.

Семантика «точно один раз» в Kafka представляет собой сочетание двух ключевых особенностей: идемпотентных производителей, которые помогают избежать дублирования, вызванного повторными попытками производителей,

и транзакционной семантики, которая гарантирует обработку точно один раз в приложениях для обработки потоков. Мы обсудим и то и другое, начав с более простого и в целом более полезного идемпотентного производителя.

Идемпотентный производитель

Сервис называется идемпотентным, если выполнение одной и той же операции несколько раз приводит к тому же результату, что и выполнение ее один раз. В базах данных это обычно демонстрируется как разница между `UPDATE t SET x=x+1 WHERE y = 5`, и `UPDATE t SET x=18 WHERE y = 5`. Первый пример не идемпотентный: если мы вызовем его три раза, то в итоге получим совсем иной результат, чем если бы вызвали его один раз. Второй пример идемпотентный — независимо от того, сколько бы раз мы ни запускали этот оператор, `x` будет равен 18.

Как это связано с производителем Kafka? Если мы настроим производитель на семантику «по крайней мере один раз», а не на идемпотентную семантику, это будет означать, что в случае неопределенности производитель повторит попытку отправки сообщения, чтобы оно пришло хотя бы один раз. Повторные попытки могут привести к появлению дубликатов.

Классический случай — когда лидер раздела получил запись от производителя, успешно реплицировал ее последователям, а затем брокер, на котором находится лидер, вышел из строя, прежде чем смог отправить ответ производителю. Производитель, не получив ответа в течение определенного времени, повторно отправляет сообщение. Оно придет новому лидеру, у которого уже есть копия сообщения, полученного при предыдущей попытке, что приводит к дублированию.

В некоторых приложениях дубликаты не имеют большого значения, но в других они могут привести к неправильному подсчету запасов, созданию неверной финансовой отчетности или отправке кому-то двух зонтиков вместо одного заказанного.

Идемпотентный производитель Kafka решает эту проблему путем автоматического обнаружения и устранения таких дубликатов.

Как работает идемпотентный производитель

Когда мы включаем идемпотентный производитель, каждое сообщение будет содержать уникальный идентификатор производителя (PID) и порядковый номер. Они вместе с целевым топиком и разделом однозначно идентифицируют каждое сообщение. Брокеры используют эти уникальные идентификаторы для отслеживания последних пяти сообщений, созданных для каждого раздела на

брокере. Чтобы ограничить количество предыдущих порядковых номеров, которые необходимо отслеживать для каждого раздела, мы также требуем, чтобы производители применили значение параметра `max.inflight.requests=5` или меньше (по умолчанию 5).

Когда брокер получает сообщение, которое уже принимал ранее, он отклоняет дубликат с соответствующей ошибкой. Эта ошибка регистрируется производителем и отражается в его показателях, но не вызывает никаких исключений и не должна никого беспокоить. На клиенте производителя она будет добавлена в показатель частоты ошибок записи. На брокере она станет частью показателя `ErrorsPerSec` типа `RequestMetrics`, который включает в себя отдельный подсчет для каждого типа ошибок.

Что делать, если брокер получает неожиданно высокий порядковый номер? Брокер ожидает, что за сообщением 2 последует сообщение 3, а что произойдет, если вместо него он получит сообщение 27? В таких случаях он ответит ошибкой «Нарушена последовательность», но если мы используем идемпотентный производитель без применения транзакций, эту ошибку можно проигнорировать.



Производитель продолжит работу в обычном режиме после того, как столкнется с исключением «Нарушена последовательность». Эта ошибка обычно указывает на то, что сообщения были потеряны между производителем и брокером: если брокер получил сообщение 2, за которым следует сообщение 27, что-то должно было произойти с сообщениями с 3-го до 26-го. При обнаружении такой ошибки в журналах следует пересмотреть конфигурацию производителя и топика и убедиться, что производитель настроен с рекомендуемыми значениями для обеспечения высокой надежности, а также проверить, не произошло ли «нечистое» избрание лидера.

Как всегда бывает в распределенных системах, интересно рассмотреть поведение идемпотентного производителя в условиях сбоя. Рассмотрим два случая: перезапуск производителя и отказ брокера.

Перезапуск производителя

Когда производитель выходит из строя, обычно вместо него создается новый производитель — либо вручную человеком, перезагружающим машину, либо с помощью более сложной платформы, такой как Kubernetes, которая обеспечивает автоматическое восстановление после сбоя. Ключевым моментом является то, что при запуске производителя, если включена функция идемпотентного производителя, он инициализируется и обращается к брокеру Kafka для создания идентификатора производителя. Каждая инициализация производителя будет

приводить к созданию совершенно нового идентификатора (при условии, что мы не включили транзакции). Это означает, что, если производитель выйдет из строя, а тот, который его заменит, отправит сообщение, ранее уже отправленное старым производителем, брокер не обнаружит дубликатов — два сообщения получат разные идентификаторы производителя и разные порядковые номера и будут рассматриваться как два разных сообщения. Обратите внимание: то же самое верно, если старый производитель завис, а затем вернулся в строй после начала работы его замены, — оригинальный производитель не будет распознаваться как зомби, потому что у нас есть два совершенно разных производителя с разными идентификаторами.

Отказ брокера

Когда брокер выходит из строя, контроллер выбирает новых лидеров для разделов, у которых были лидеры на вышедшем из строя брокере. Предположим, у нас есть производитель, который создавал сообщения для топика А раздела 0, который имел ведущую реплику на брокере 5 и реплику последователя на брокере 3. После сбоя брокера 5 новым лидером становится брокер 3. Производитель обнаружит, что новым лидером является брокер 3, через протокол метаданных и начнет производить для него. Но как брокер 3 узнает, какие последовательности уже были произведены, чтобы отбросить дубликаты?

Лидер продолжает обновлять свое состояние производителя в памяти при помощи пяти последних идентификаторов последовательностей каждый раз, когда создается новое сообщение. Реплики-последователи обновляют собственные буферы в памяти каждый раз, когда реплицируют новые сообщения от лидера. Это означает, что, когда последователь становится лидером, у него уже есть последние номера последовательностей в памяти и проверка новых сообщений может продолжаться без каких-либо проблем или задержек.

Но что произойдет, когда вернется старый лидер? После перезагрузки старое состояние производителя больше не будет храниться в памяти. Чтобы помочь в восстановлении, брокеры делают моментальный снимок состояния производителя в файл при завершении работы или при каждом создании сегмента. Когда брокер запускается, он считывает последнее состояние из файла. Затем вновь перезапущенный брокер продолжает обновлять состояние производителя по мере того, как он догоняет его, реплицируясь с текущего лидера, и имеет в памяти самые последние идентификаторы последовательностей к моменту, когда он готов снова стать лидером.

Что делать, если брокер вышел из строя и последний снимок состояния не был обновлен? Идентификатор производителя и идентификатор последовательности также являются частью формата сообщений, которые записываются в жур-

налы Kafka. Во время восстановления после сбоя состояние производителя будет восстановлено путем чтения более старого моментального снимка состояния, а также сообщений из последнего сегмента каждого раздела. Новый снимок будет сохранен, как только завершится процесс восстановления.

Интересный вопрос: что произойдет, если сообщений не будет? Представьте себе, что определенный топик хранится в течение двух часов, но за последние два часа не поступило ни одного нового сообщения, то есть не будет никаких сообщений, которые можно было бы использовать для восстановления состояния в случае сбоя брокера. К счастью, отсутствие сообщений означает также отсутствие дубликатов. Мы начнем принимать сообщения немедленно (при этом в журнал будет выведено предупреждение об отсутствии состояния) и создадим состояние производителя из вновь поступающих сообщений.

Ограничения идемпотентного производителя

Идемпотентный производитель Kafka предотвращает дублирование только в случае повторных попыток, вызванных внутренней логикой производителя. Вызов функции `producer.send()` дважды с одним и тем же сообщением создаст дубликат, и идемпотентный производитель не предотвратит этого. Это происходит потому, что у производителя нет возможности узнать, что две отправленные записи на самом деле одна и та же запись. Всегда лучше использовать встроенный механизм повторных попыток производителя, чем перехватывать исключения производителя и повторять попытку из самого приложения. Идемпотентный производитель делает этот шаблон еще более привлекательным — это самый простой способ избежать дубликатов при повторных попытках.

Кроме того, довольно часто встречаются приложения, имеющие несколько экземпляров или даже один экземпляр с несколькими производителями. Если два из этих производителей попытаются отправить идентичные сообщения, идемпотентный производитель не обнаружит дублирования. Такой сценарий довольно часто встречается в приложениях, которые получают данные из источника, например из каталога с файлами, и отправляют их в Kafka. Если в приложении оказалось два экземпляра, считывающих один и тот же файл и создающих записи в Kafka, мы получим несколько копий записей в этом файле.



Идемпотентный производитель предотвратит появление только дубликатов, порожденных механизмом повторных попыток самого производителя, независимо от того, вызваны ли эти повторные попытки ошибками производителя, сети или брокера. Но не более того.

Как использовать идемпотентный производитель Kafka

Это самая простая часть. Добавьте параметр `enable.idempotence=true` в конфигурацию производителя. Если последний уже настроен на значение параметра `acks=all`, разницы в производительности не будет. При включении идемпотентного производителя изменится следующее.

- Чтобы получить идентификатор производителя, производитель сделает один дополнительный вызов API при запуске.
- Каждый отправляемый пакет записей будет включать в себя идентификатор производителя и идентификатор последовательности для первого сообщения в пакете (идентификаторы последовательности для каждого сообщения в пакете представляют собой сумму идентификатора последовательности первого сообщения и дельты). Эти новые поля добавляют 96 бит к каждому пакету записей (идентификатор производителя длинный, а последовательность — целое число), что практически не требует дополнительных расходов для большинства рабочих нагрузок.
- Брокеры будут проверять порядковые номера от любого отдельного экземпляра производителя и гарантировать отсутствие дубликатов сообщений.
- Порядок сообщений, отправляемых в каждый раздел, будет гарантирован при всех сценариях сбоя, даже если для параметра `max.in.flight.requests.per.connection` установлено значение больше 1 (5 — это значение по умолчанию, а также максимальное значение, поддерживаемое идемпотентным производителем).



Логика идемпотентного производителя и обработка ошибок значительно улучшились в версии 2.5 (как на стороне производителя, так и на стороне брокера) в результате KIP-360. До версии 2.5 состояние производителя не всегда сохранялось достаточно долго, что приводило к фатальным ошибкам `UNKNOWN_PRODUCER_ID` в различных сценариях (переназначение разделов имело известный пограничный случай, когда новая реплика становилась лидером до того, как происходила запись от определенного производителя, что означало, что новый лидер не имел состояния для этого раздела). Кроме того, предыдущие версии пытались переписать идентификаторы последовательностей в некоторых сценариях ошибок, что могло привести к появлению дубликатов. В новых версиях, если мы сталкиваемся с фатальной ошибкой для пакета записей, этот пакет и все пакеты, находящиеся в процессе выполнения, будут отклонены. Пользователь, который пишет приложение, может обработать исключение и решить, пропустить ли эти записи или повторить попытку, рискуя получить дубликаты и изменение порядка.

Транзакции

Как мы уже упоминали во введении к этой главе, транзакции были добавлены в Kafka, чтобы гарантировать корректность приложений, разработанных с использованием потоков Kafka. Для того чтобы приложение обработки потоков генерировало корректные результаты, каждая входная запись должна обрабатываться ровно один раз, и результат ее обработки будет отражен ровно один раз даже в случае сбоя. Транзакции в Apache Kafka позволяют приложениям обработки потоков генерировать точные результаты. Это, в свою очередь, позволяет разработчикам использовать приложения потоковой обработки в тех случаях, когда точность является ключевым требованием.

Важно помнить, что транзакции в Kafka были разработаны специально для приложений потоковой обработки. И поэтому они были созданы для работы с шаблоном «потребление — обработка — производство», который лежит в основе приложений потоковой обработки. Использование транзакций может гарантировать семантику «точно один раз» в этом контексте — обработка каждой входной записи будет считаться завершенной после обновления внутреннего состояния приложения и успешной выдачи результатов на выходные данные в топики. В подразделе «Какие проблемы не решаются транзакциями» далее в этой главе мы рассмотрим несколько сценариев, в которых гарантии Kafka «точно один раз» неприменимы.



Транзакции — это название базового механизма. Семантика «точно один раз» или гарантии «точно один раз» — это поведение приложения для обработки потоков. Потоки Kafka применяют транзакции для реализации своих гарантий «точно один раз». Другие механизмы обработки потоков, такие как Spark Streaming или Flink, задействуют различные механизмы для предоставления своим пользователям семантики «точно один раз».

Сценарии использования транзакций

Транзакции полезны для любого приложения обработки потоков, где важна точность, и особенно там, где обработка потоков включает агрегацию и/или объединение. Если приложение потоковой обработки выполняет только преобразование и фильтрацию одиночных записей, нет необходимости обновлять внутреннее состояние, и даже если в процессе обработки появились дубликаты, их достаточно просто отфильтровать из выходного потока. Когда приложение обработки потока объединяет несколько записей в одну, гораздо сложнее проверить, не является ли запись с результатами ошибочной из-за того, что некоторые входные записи были подсчитаны более одного раза: исправить результат без повторной обработки входных данных невозможно.

Финансовые приложения являются типичными примерами сложных приложений потоковой обработки, в которых для обеспечения точного суммирования используются возможности «точно один раз». Однако, поскольку настроить любое приложение Kafka Streams для обеспечения гарантий «точно один раз» довольно просто, мы видели, как оно используется в более простых случаях, включая, например, чат-боты.

Какие проблемы решают транзакции

Рассмотрим простое приложение для обработки потоков: оно считывает события из исходного топика, возможно, обрабатывает их и записывает результаты в другой топик. Мы хотим быть уверены, что для каждого сообщения, которое мы обрабатываем, результаты записываются ровно один раз. Что может пойти не так?

Оказывается, довольно многое может пойти не так. Давайте рассмотрим два сценария.

Повторная обработка, вызванная сбоем приложения

После получения сообщения из исходного кластера и его обработки приложение должно выполнить два действия: выдать результат в выходной топик и зафиксировать смещение сообщения, которое мы получили. Предположим, что эти два отдельных действия происходят в таком порядке. Что произойдет, если приложение завершит работу после выдачи результата, но до того, как будет зафиксировано смещение входного сообщения?

В главе 4 мы обсудили, что происходит, когда потребитель выходит из строя. Через несколько секунд отсутствие контрольных сигналов вызовет перебалансировку и разделы, из которых потребитель потреблял данные, будут переназначены другому потребителю. Этот потребитель начнет потреблять записи из данных разделов, начиная с последнего зафиксированного смещения. Это означает, что все записи, которые были обработаны приложением между последним зафиксированным смещением и сбоем, будут обработаны снова и результаты снова записаны в выходной топик, что приведет к появлению дубликатов.

Повторная обработка, вызванная приложениями-зомби

Что произойдет, если наше приложение только что потребило пакет записей из Kafka, а затем зависло или потеряло связь с ней, прежде чем успело сделать что-либо еще с этим пакетом записей?

Точно так же, как и в предыдущем сценарии, после нескольких пропущенных контрольных сигналов приложение будет считаться мертвым, а его разделы

будут переназначены другому потребителю в группе потребителей. Этот потребитель перечитает пакет записей, обработает его, выдаст результаты в выходной топик и продолжит работу.

Тем временем первый экземпляр приложения — тот, который завис, — может возобновить свою деятельность: обработать пакет записей, который он недавно потреблял, и выдать результаты в выходной топик. Он может сделать все это до того, как подаст запрос в Kafka на наличие записей или отправит контрольный сигнал и обнаружит, что он должен был быть мертвым, а эти разделы теперь принадлежат другому экземпляру.

Потребитель, который мертв, но не знает об этом, называется зомби. В этом сценарии мы видим, что без дополнительных гарантий зомби могут выдавать данные в выходной топик и вызывать дублирование результатов.

Как транзакции гарантируют «точно один раз»

Возьмем простое приложение для обработки потоков. Оно считывает данные из одного топика, обрабатывает их и записывает результат в другой топик. Обработка «точно один раз» означает, что потребление, обработка и выдача данных выполняются *атомарно*. Либо смещение исходного сообщения зафиксировано и результат успешно получен, либо не происходит ни того ни другого. Нам нужно убедиться, что частичных результатов — когда смещение зафиксировано, но результат не получен или наоборот, — быть не может.

Для поддержки такого поведения транзакции Kafka вводят идею *атомарной многораздельной записи*. Суть ее заключается в том, что фиксация смещений и получение результатов включают запись сообщений в разделы. Однако результаты записываются в выходной топик, а смещения — в топик `_consumer_offsets`. Если мы можем открыть транзакцию, записать оба сообщения и зафиксировать их, если они записаны успешно, или прервать и повторить попытку, если они не были записаны, то получим нужную нам семантику «точно один раз».

На рис. 8.1 показано простое приложение для обработки потоков, выполняющее атомарную многораздельную запись в два раздела с одновременной фиксацией смещений для потребляемого события.

Чтобы использовать транзакции и выполнять атомарную многораздельную запись, мы применяем *транзакционный производитель*. Это просто производитель Kafka, который был настроен с помощью параметра `transactional.id` и инициализирован с помощью функции `initTransactions()`. В отличие от параметра `producer.id`, который генерируется автоматически брокерами Kafka, параметр `transactional.id` является частью конфигурации производителя

и, как ожидается, будет сохраняться между перезапусками. Фактически основная роль `transactional.id` заключается в идентификации одного и того же производителя при разных перезапусках. Брокеры Kafka поддерживают сопоставление параметров `transactional.id` и `producer.id`, поэтому, если функция `initTransactions()` будет вызвана снова с существующим `transactional.id`, производителю будет присвоен тот же `producer.id`, а не новое случайное число.

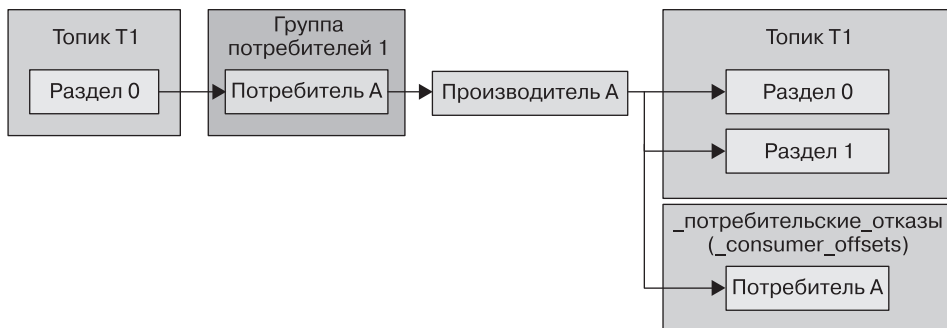


Рис. 8.1. Транзакционный производитель с атомарной многораздельной записью

Для предотвращения создания дубликатов экземплярами приложения-зомби требуется механизм защиты от зомби, то есть предотвращения записи результатов в выходной поток *зомбированными экземплярами приложения*. Здесь реализуется обычный способ защиты от зомби — использование эпохи. Kafka увеличивает номер эпохи, связанный с `transactional.id`, когда вызывается `initTransaction()` для инициализации производителя транзакций. Запросы на отправку, фиксацию и отмену от производителей с одинаковым значением параметра `transactional.id`, но с меньшей эпохой будут отклонены с ошибкой `FencedProducer`. Более старый производитель не сможет выполнять запись в выходной поток и будет принудительно закрыт с помощью `close()`, предотвращая появление дубликатов записей от зомби. В Apache Kafka 2.5 и более поздних версиях также есть возможность добавить метаданные группы потребителей в метаданные транзакции. Эти метаданные также будут использоваться для защиты от зомби, что позволит производителям с разными идентификаторами транзакций выполнять записи в одни и те же разделы, но при этом сохранять защиту от зомбированных экземпляров.

Транзакции по большей части являются функцией производителя: мы создаем транзакционный производитель, начинаем транзакцию, заносим записи в несколько разделов, создаем смещения, чтобы пометить записи как уже обработанные, и фиксируем или прерываем транзакцию. Все это делаем из производителя.

Однако этого недостаточно — записи, сделанные транзакционно, даже те, что являются частью транзакций, которые в конечном итоге были прерваны, записываются в разделы точно так же, как и любые другие. Потребители должны быть настроены с правильными гарантиями изоляции, иначе мы не получим ожидаемых гарантий «точно один раз».

Мы контролируем потребление сообщений, записанных транзакционно, путем установки параметра `isolation.level`. Если установлено значение `read_committed`, вызов функции `consumer.poll()` после подписки на набор топиков вернет сообщения, которые либо были частью успешно зафиксированной транзакции, либо записаны нетранзакционно. Но он не вернет сообщения, которые были частью прерванной или все еще открытой транзакции. Параметр `isolation.level` по умолчанию указан со значением `read_uncommitted`, он возвращает все записи, включая принадлежащие открытым или прерванным транзакциям. Настройка режима `read_committed` не гарантирует, что приложение получит все сообщения, являющиеся частью конкретной транзакции. Можно подписаться только на подмножество топиков, которые были частью транзакции, и, следовательно, получить подмножество сообщений. Кроме того, приложение не может знать, когда транзакции начинаются или заканчиваются или какие сообщения являются частью какой транзакции.

На рис. 8.2 показано, какие записи видны потребителю в режиме `read_committed` по сравнению с потребителем в режиме по умолчанию `read_uncommitted`.

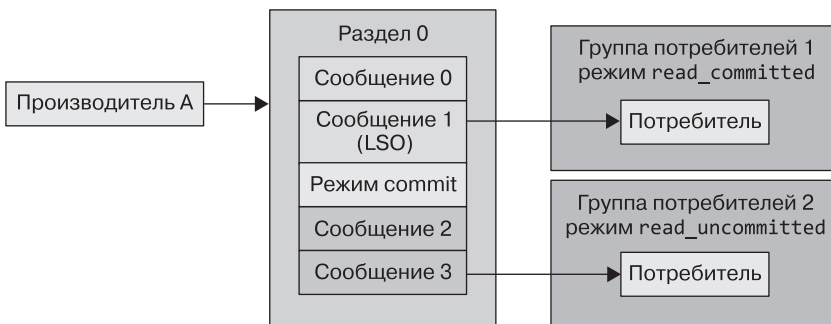


Рис. 8.2. Потребители в режиме `read_committed` будут отставать от потребителей с конфигурацией по умолчанию

Чтобы гарантировать, что сообщения будут прочитаны по порядку, режим `read_committed` не будет возвращать сообщения, созданные после момента начала первой, еще открытой транзакции, известной как последнее стабильное смещение (Last Stable Offset — LSO). Эти сообщения будут храниться до тех

пор, пока транзакция не будет зафиксирована или прервана производителем или пока они не достигнут значения `transaction.timeout.ms` (по умолчанию 15 мин) и не будут прерваны брокером. Если держать транзакцию открытой в течение длительного времени, это приведет к увеличению конечной задержки за счет задержки потребителей.

Наше простое задание по обработке потока будет иметь на выходе гарантии «точно один раз», даже если входные данные были записаны нетранзакционно. Атомарное многораздельное производство гарантирует, что если выходные записи были зафиксированы в топике вывода, то для этого потребителя будет зафиксировано также смещение входных записей и в результате они не будут обрабатываться повторно.

Какие проблемы не решаются транзакциями

Как объяснялось ранее, транзакции были добавлены в Kafka для обеспечения многораздельной атомарной записи (но не чтения) и защиты от производителей-зомби в приложениях обработки потоков. В результате они обеспечивают гарантии «точно один раз» при использовании в цепочках задач обработки потоков «потребление — обработка — производство». В других контекстах транзакции либо не будут работать, либо потребуют дополнительных усилий для получения желаемых гарантий.

Две основные ошибки заключаются в предположении, что гарантии «точно один раз» применимы к иным действиям, кроме производства в Kafka, и что потребители всегда читают транзакции целиком и обладают информацией об их границах.

Далее приведены несколько сценариев, в которых транзакции Kafka не помогут получить гарантии «точно один раз».

Побочные эффекты при обработке потока

Предположим, что этап обработки записей в нашем приложении для обработки потоков включает в себя отправку электронной почты пользователям. Включение семантики «точно один раз» в приложение не гарантирует, что электронное письмо будет отправлено только один раз. Гарантия распространяется только на записи, сделанные в Kafka. Применение порядковых номеров для дедупликации записей или маркеров — для прерывания или отмены транзакции — работает в Kafka, но оно не отменяет отправку электронного письма. То же самое верно для любого действия с внешними эффектами, которое выполняется внутри приложения для обработки потоков: вызова REST API, записи в файл и т. д.

Чтение из топика Kafka и запись в базу данных

В этом случае приложение записывает данные не в Kafka, а во внешнюю базу данных. В этом сценарии производитель не участвует — записи заносятся в базу данных с помощью драйвера базы данных (скорее всего, JDBC), а смещения фиксируются в Kafka потребителем. Не существует механизма, который позволял бы записывать результаты во внешнюю базу данных и фиксировать смещения в Kafka в рамках одной транзакции. Вместо этого мы можем управлять смещениями в базе данных (как объяснялось в главе 4) и фиксировать данные и смещения в базе данных в одной транзакции — это будет полагаться на транзакционные гарантии базы данных, а не Kafka.



Микросервисам часто требуется обновить базу данных и опубликовать сообщение в Kafka в рамках одной атомарной транзакции, поэтому либо произойдет и то и другое, либо ни то ни другое. Как мы только что объяснили в двух последних примерах, транзакции Kafka этого не сделают.

Общее решение этой распространенной проблемы известно как шаблон исходящих сообщений (outbox). Микросервис публикует сообщение только в топик Kafka («исходящие»), а отдельная служба ретрансляции сообщений считывает событие из Kafka и обновляет базу данных. Поскольку, как мы только что увидели, Kafka не гарантирует обновление базы данных «точно один раз», важно убедиться, что обновление является идемпотентным.

Использование этого шаблона гарантирует, что сообщение в конечном итоге попадет в Kafka, к потребителям топиков и в базу данных или не попадет ни к кому из них.

Применяется и обратный шаблон, когда таблица базы данных служит исходящим ящиком, а служба ретрансляции следит за тем, чтобы обновления таблицы также поступали в Kafka в виде сообщений. Этот шаблон предпочтителен, когда полезны встроенные ограничения РСУБД, такие как уникальность и внешние ключи. Проект Debezium опубликовал подробную статью в блоге о шаблоне исходящих сообщений (<https://oreil.ly/PB3Vb>) с подробными примерами.

Считывание данных из базы данных, запись в Kafka, а оттуда — в другую базу данных

Очень заманчиво полагать, что мы можем создать приложение, которое будет считывать данные из базы данных, идентифицировать транзакции базы данных, делать записи в Kafka, а оттуда вносить записи в другую базу данных, сохраняя исходные транзакции из исходной базы данных.

К сожалению, транзакции Kafka не обладают необходимой функциональностью для поддержки такого рода сквозных гарантий. В дополнение к проблеме

фиксации записей и смещений в рамках одной транзакции существует еще одна сложность — гарантии `read_committed` в потребителях Kafka слишком слабы для сохранения транзакций базы данных. Да, потребитель не увидит записи, которые не были зафиксированы. Но не гарантируется, что он увидит все записи, которые были зафиксированы в рамках транзакции, потому что он может отставать по некоторым топикам: у него нет информации для определения границ транзакции, поэтому он не может знать, когда началась и закончилась транзакция и видел ли он лишь некоторые из своих записей, видел их все или не видел вообще никаких.

Копирование данных из одного кластера Kafka в другой

Этот вариант еще менее очевидный — можно поддерживать гарантии «точно один раз» при копировании данных из одного кластера Kafka в другой. Описание того, как это делается, содержится в предложении по улучшению Kafka для добавления возможностей «точно один раз» в версии MirrorMaker 2.0 (<https://oreil.ly/EoM6w>). На момент написания этой книги предложение все еще находится в стадии разработки, но алгоритм четко описан. Это предложение включает гарантию того, что каждая запись в исходном кластере будет скопирована в целевой кластер ровно один раз.

Однако это не гарантирует, что транзакции будут атомарными. Если приложение производит несколько записей и выполняет смещения транзакционно, а затем MirrorMaker 2.0 копирует их в другой кластер Kafka, транзакционные свойства и гарантии будут потеряны в процессе копирования. Они теряются по той же причине, что и при копировании данных из Kafka в реляционную базу данных: потребитель, считывающий данные из Kafka, не может знать или гарантировать, что он получает все события в транзакции. Например, он может реплицировать часть транзакции, если он подписан только на подмножество топигов.

Шаблон публикации/подписки

Вот еще несколько более тонкий случай. Мы уже обсуждали точно такой же случай в контексте шаблона «потребление — обработка — производство», но шаблон «публикация/подписка» — очень распространенный сценарий использования. Применение транзакций в шаблоне публикации/подписки обеспечивает некоторые гарантии: потребители, настроенные на режим `read_committed`, не увидят записей, которые были опубликованы как часть прерванной транзакции. Но эти гарантии не соответствуют принципу «точно один раз». Потребители могут обрабатывать сообщение более одного раза в зависимости от их собственной логики фиксации смещения.

Гарантии, которые Kafka предоставляет в этом случае, похожи на те, которые предоставляются транзакциями JMS, но зависят от потребителей в режиме `read_committed`, чтобы гарантировать, что незафиксированные транзакции останутся невидимыми. Брокеры JMS скрывают незафиксированные транзакции от всех потребителей.



Важным шаблоном, которого следует избегать, является публикация сообщения, а затем ожидание ответа другого приложения до фиксации транзакции. Другое приложение получит сообщение только после фиксации транзакции, что приведет к взаимоблокировке.

Как использовать транзакции

Транзакции являются функцией брокера и частью протокола Kafka, поэтому существует множество клиентов, поддерживающих транзакции.

Самый распространенный и наиболее рекомендуемый способ использования транзакций — это включение в потоки Kafka Streams гарантий «точно один раз». Таким образом, мы не будем применять транзакции напрямую, а скорее потоки Kafka Streams будут использовать их для нас за кулисами, чтобы обеспечить необходимые нам гарантии. Транзакции были разработаны с учетом этого сценария, поэтому их использование через потоки Kafka является самым простым и с наибольшей вероятностью будет работать так, как ожидается.

Чтобы включить гарантии «точно один раз» для приложения Kafka Streams, мы просто устанавливаем в конфигурации `processing.guarantee` значение `exactly_once` или `exactly_once_beta`. Вот и все.



`exactly_once_beta` — это немного другой метод обработки экземпляров приложений, которые выходят из строя или зависают при выполнении транзакций. Данный метод введен в версии 2.5 для брокеров Kafka и в версии 2.6 для потоков Kafka. Основное преимущество этого метода — возможность обрабатывать множество разделов с помощью одного транзакционного производителя и, следовательно, создавать более масштабируемые приложения Kafka Streams. Более подробную информацию об изменениях можно найти в предложении по улучшению Kafka, где они обсуждались впервые (<https://oreil.ly/O3dSA>).

Но что, если мы хотим получить гарантии «точно один раз» без использования Kafka Streams? В этом случае будем использовать транзакционные API напрямую. Вот фрагмент кода, показывающий, как это будет работать. Полный пример есть в Apache Kafka GitHub, который включает в себя демонстрационный

драйвер (<https://oreil.ly/45dE4>) и простой процессор «точно один раз» (<https://oreil.ly/CrXHU>), который выполняется в отдельных потоках:

```
Properties producerProps = new Properties();
producerProps.put(ProducerConfig.BootstrapServersConfig, "localhost:9092");
producerProps.put(ProducerConfig.ClientIdConfig, "DemoProducer");
producerProps.put(ProducerConfig.TransactionIdConfig, transactionalId); ❶

producer = new KafkaProducer<>(producerProps);

Properties consumerProps = new Properties();
consumerProps.put(ConsumerConfig.BootstrapServersConfig, "localhost:9092");
consumerProps.put(ConsumerConfig.GroupIdConfig, groupId);
props.put(ConsumerConfig.EnableAutoCommitConfig, "false"); ❷
consumerProps.put(ConsumerConfig.IsolationLevelConfig, "read_committed"); ❸

consumer = new KafkaConsumer<>(consumerProps);

producer.initTransactions(); ❹

consumer.subscribe(Collections.singleton(inputTopic)); ❺

while (true) {
    try {
        ConsumerRecords<Integer, String> records =
            consumer.poll(Duration.ofMillis(200));
        if (records.count() > 0) {
            producer.beginTransaction(); ❻
            for (ConsumerRecord<Integer, String> record : records) {
                ProducerRecord<Integer, String> customizedRecord =
                    transform(record); ❼
                producer.send(customizedRecord);
            }
            Map<TopicPartition, OffsetAndMetadata> offsets = consumer.offsets();
            producer.sendOffsetsToTransaction(offsets, consumer.groupMetadata()); ❽
            producer.commitTransaction(); ❾
        }
    } catch (ProducerFencedException|InvalidProducerEpochException e) { ❿
        throw new KafkaException(String.format(
            "The transactional.id %s is used by another process", transactionalId));
    } catch (KafkaException e) {
        producer.abortTransaction(); ⓫
        resetToLastCommittedPositions(consumer);
    }
}
```

❶ Настройка производителя с помощью параметра `transactional.id` делает его транзакционным производителем, способным делать атомарную много-раздельную запись. Идентификатор транзакции должен быть уникальным и с длительным жизненным циклом. По сути, он определяет экземпляр приложения.

❷ Потребители, которые являются частью транзакций, не фиксируют собственные смещения — производитель записывает смещения как часть транзакции. Поэтому фиксация смещений должна быть отключена.

❸ В этом примере потребитель читает из входного топика. Мы будем предполагать, что записи во входном топике также были записаны транзакционным производителем (просто для развлечения — для входных данных это не требуется). Для чистого чтения транзакций (то есть игнорирования запущенных и прерванных транзакций) установим уровень изоляции потребителя на `read_committed`. Обратите внимание на то, что потребитель по-прежнему будет читать нетранзакционные записи в дополнение к чтению зафиксированных транзакций.

❹ Первое, что должен сделать производитель транзакций, — это инициализация. При этом регистрируется идентификатор транзакции, увеличивается эпоха, чтобы гарантировать, что другие производители с тем же идентификатором будут считаться зомби, и прерываются более старые транзакции, находящиеся в процессе выполнения, с тем же идентификатором транзакции.

❺ Здесь мы используем API потребителя подписки, что означает: разделы, назначенные данному экземпляру приложения, могут измениться в любой момент в результате перебалансировки. До выхода версии 2.5, когда были внесены изменения в API на основе KIP-447, это было гораздо сложнее. Производители транзакций должны были статически назначать набор разделов, поскольку механизм ограждения транзакций полагался на то, что один и тот же идентификатор транзакции будет использоваться для одних и тех же разделов (при изменении идентификатора транзакции не было защиты от зомбирования). KIP-447 добавил новые API, используемые в данном примере, которые прикрепляют информацию о группе потребителей к транзакции, и эта информация используется для ограждения. При использовании этого метода имеет смысл также фиксировать транзакции каждый раз, когда отменяются соответствующие разделы.

❻ Мы получили записи, а теперь хотим обработать их и выдать результаты. Этот метод гарантирует, что все, что производится с момента его вызова до тех пор, пока транзакция не будет зафиксирована или прервана, является частью одной атомарной транзакции.

❼ Именно здесь мы обрабатываем записи — вся бизнес-логика находится здесь.

❽ Как объяснялось ранее в этой главе, важно зафиксировать смещения как часть транзакции. Это гарантирует, что если мы не получим результатов, то не зафиксируем смещения для записей, которые на самом деле не были обработаны. Этот метод фиксирует смещения как часть транзакции. Обратите внимание: важно не фиксировать смещения каким-либо другим способом — отключите автоматическую фиксацию смещений и не вызывайте никаких API потребительской

фиксации. Фиксация смещений любым другим методом не обеспечивает транзакционных гарантий.

❸ Мы произвели все необходимое, зафиксировали смещения как часть транзакции, пришло время зафиксировать транзакцию и завершить процесс. Как только этот метод вернется успешно, вся транзакция будет завершена и мы сможем продолжить чтение и обработку следующего пакета событий.

❹ Получение этого исключения означает, что мы — зомби. Каким-то образом приложение зависло или отключилось, и в данный момент запущен новый экземпляр приложения с нашим идентификатором транзакции. Скорее всего, транзакция, которую мы начали, уже прервана и кто-то другой обрабатывает эти записи. Ничего не остается, кроме как изящно умереть.

❺ Если мы получили ошибку во время записи транзакции, то можем прервать ее, вернуть позицию потребителя и повторить попытку.

Идентификаторы транзакций и ограждения

Выбор идентификатора транзакции для производителей очень важен и немного сложнее, чем кажется. Неправильное назначение идентификатора транзакции может привести либо к ошибкам в работе приложения, либо к потере гарантий «точно один раз». Основные требования заключаются в том, чтобы идентификатор транзакции был последовательным для одного и того же экземпляра приложения между перезапусками и различался для разных экземпляров приложения, иначе брокеры не смогут отсеять зомбированные экземпляры.

До версии 2.5 единственным способом обеспечить ограждение было статическое сопоставление идентификатора транзакции с разделами. Это гарантировало, что каждый раздел всегда будет потребляться с одним и тем же идентификатором транзакции. Если производитель с транзакционным идентификатором А обрабатывал сообщения из топика Т и потерял подключение, а новый производитель, заменивший его, имеет транзакционный идентификатор В и позже производитель А возвращается как зомби, то зомби А не будет огражден, потому что его идентификатор не совпадает с идентификатором нового производителя В. Мы хотим, чтобы производитель А всегда заменялся производителем А, а новый производитель А имел более высокий номер эпохи, тогда зомби А будет огражден должным образом. В этих версиях предыдущий пример будет некорректным — идентификаторы транзакций назначаются потокам случайным образом и не гарантируют того, чтобы один и тот же идентификатор транзакции всегда использовался для записи в один и тот же раздел.

В Apache Kafka 2.5 KIP-447 ввел второй метод ограждения, основанный на метаданных группы потребителей для ограждения в дополнение к идентифи-

каторам транзакций. Мы используем метод фиксации смещения производителя и передаем в качестве аргумента метаданные группы потребителей, а не только ее идентификатор.

Допустим, у нас есть топик T1 с двумя разделами, t-0 и t-1. Каждый из них потребляется отдельным потребителем в той же группе. Каждый потребитель передает записи соответствующему производителю транзакций — одному с транзакционным идентификатором A, а другому с транзакционным идентификатором B, и они записывают выходные данные в разделы топика T2 0 и 1 соответственно. Рисунок 8.3 иллюстрирует этот сценарий.

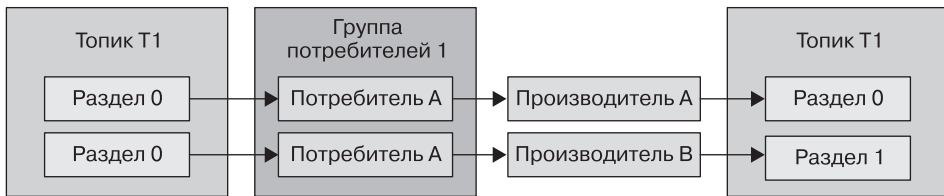


Рис. 8.3. Обработчик транзакционных записей

Как показано на рис. 8.4, если экземпляр приложения с потребителем А и производителем А станет зомби, потребитель В начнет обрабатывать записи из обоих разделов. Если мы хотим гарантировать, что никакие зомби не будут записывать в раздел 0, потребитель В не может просто начать читать из раздела 0 и записывать в раздел 0 с транзакционным идентификатором В. Вместо этого приложению потребуется создать экземпляр нового производителя с транзакционным идентификатором А, чтобы безопасно записывать в раздел 0, и отменить старый транзакционный идентификатор А. Это расточительно. Вместо этого мы включаем информацию о группе потребителей в транзакции. Транзакции от производителя В покажут, что они относятся к более новому поколению группы потребителей, и поэтому пройдут, в то время как транзакции от теперь уже зомбированного производителя А покажут старое поколение группы потребителей и будут оградены.

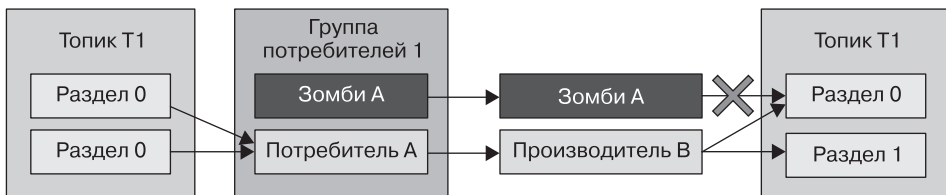


Рис. 8.4. Обработчик транзакционных записей после переконфигурации

Как работают транзакции

Мы можем использовать транзакции, вызывая API и не понимая, как они работают. Но наличие некоторой мысленной модели того, что происходит внутри, поможет нам устранить неполадки в приложениях, которые ведут себя не так, как ожидалось.

Основной алгоритм транзакций в Kafka был вдохновлен моментальными снимками Чанди — Лампорта (Chandy — Lamport), в которых управляющие сообщения-маркеры отправляются в каналы связи, а согласованное состояние определяется на основе прибытия маркера. Транзакции Kafka используют сообщения с маркерами для указания фиксации или отмены транзакций в нескольких разделах — когда производитель решает зафиксировать транзакцию, он отправляет сообщение «зафиксировать» координатору транзакций, который затем записывает маркеры фиксации во все разделы, участвующие в транзакции. Но что произойдет, если производитель выйдет из строя после записи сообщений о фиксации только в подмножество разделов? Транзакции Kafka решают эту проблему с помощью двухфазной фиксации и журнала транзакций. На высоком уровне алгоритм будет следующим.

1. Запишите в журнал существование текущей транзакции, включая вовлеченные разделы.
2. Зарегистрируйте намерение зафиксировать или прервать транзакцию — как только это будет сделано, мы будем обречены совершить фиксацию или прерывание в конечном итоге.
3. Запишите все маркеры транзакции во все разделы.
4. Запротоколируйте в журнале завершение транзакции.

Чтобы реализовать этот базовый алгоритм, Kafka необходим журнал транзакций. Мы используем внутренний топик под названием `transaction_state`.

Давайте посмотрим, как этот алгоритм работает на практике, изучив внутреннюю работу транзакционных вызовов API, которые мы применяли в предыдущем фрагменте кода.

Прежде чем начать первую транзакцию, производители должны зарегистрироваться как транзакционные, вызвав функцию `initTransaction()`. Этот запрос отправляется брокеру, который будет координатором транзакций для данного производителя транзакций. Каждый брокер является координатором транзакций для подмножества производителей, так же как каждый брокер является координатором группы потребителей для подмножества потребительских групп. Координатор транзакций для каждого идентификатора транзакции является

лидером раздела журнала транзакций, к которому привязан этот идентификатор транзакции.

API функции `initTransaction()` регистрирует новый идентификатор *транзакции с координатором* или увеличивает эпоху существующего идентификатора транзакции, чтобы отсеять предыдущих производителей, которые могли стать зомби. Когда эпоха увеличивается, ожидающие транзакции будут прерваны.

Следующим шагом для производителя будет вызов функции `beginTransaction()`. Этот вызов API не является частью протокола — он просто сообщает производителю, что теперь транзакция находится в процессе выполнения. Координатор транзакций на стороне брокера все еще не знает, что транзакция началась. Однако, как только производитель начинает отправлять записи, каждый раз, когда производитель обнаруживает, что он отправляет записи в новый раздел, он также отправляет брокеру запрос `Add PartitionsToTxnRequest`, информируя его о том, что для этого производителя выполняется транзакция и что дополнительные разделы являются ее частью. Эта информация будет записана в журнал транзакций.

Закончив выдачу результатов и подготовившись к фиксации, мы начнем с фиксации смещений для записей, которые обработали в этой транзакции. Фиксация смещений может быть выполнена в любое время, но это необходимо сделать до фиксации транзакции. Вызов `sendOffsetsToTransaction()` отправит координатору транзакции запрос, который включает смещения, а также идентификатор группы потребителей. Координатор транзакции будет использовать идентификатор группы потребителей, чтобы найти координатора группы и зафиксировать смещения, как это обычно делает группа потребителей.

Теперь пришло время зафиксировать или прервать транзакцию. Вызов функции `commitTransaction()` или `abortTransaction()` отправит запрос `EndTransactionRequest` координатору транзакции. Координатор транзакций запишет намерение зафиксировать или прервать транзакцию в журнал транзакций. После успешного завершения этого шага координатор транзакции обязан завершить процесс фиксации (или прерывания). Он записывает маркер фиксации во все разделы, участвующие в транзакции, а затем указывает в журнале транзакций, что фиксация завершилась успешно. Обратите внимание на то, что если координатор транзакции выключится или выйдет из строя после регистрации намерения зафиксировать транзакцию и до завершения процесса, то будет избран новый координатор транзакции, который получит намерение зафиксировать транзакцию из журнала транзакций и завершит процесс.

Если транзакция не будет зафиксирована или прервана в течение `transaction.timeout.ms`, координатор транзакций прервет ее автоматически.



Каждый брокер, получающий записи от транзакционных или идемпотентных производителей, будет хранить в памяти идентификаторы производителей/транзакций, а также соответствующее им состояние для каждого из последних пяти пакетов, отправленных производителем: порядковые номера, смещения и т. п. Это состояние хранится в течение `transactional.id.expiration.ms` миллисекунд после того, как производитель перестал быть активным (по умолчанию семь дней). Это позволяет производителю возобновить активность, не сталкиваясь с ошибками `UNKNOWN_PRODUCER_ID`. Можно вызвать нечто похожее на утечку памяти в брокере, создавая новые идемпотентные производители или новые идентификаторы транзакций с очень высокой скоростью, но никогда не используя их повторно. Три новых идемпотентных производителя в секунду, накопленные в течение недели, дадут 1,8 млн записей состояния производителя с сохранением в общей сложности 9 млн пакетных метаданных, задействуя около 5 Гбайт оперативной памяти. Это может привести к нехватке памяти или серьезным проблемам со сборкой мусора на брокере. Мы рекомендуем при разработке архитектуры приложения инициализировать несколько производителей с длительным сроком службы при запуске приложения, а затем использовать их повторно в течение всего времени работы приложения. Если это невозможно (применение сценария «функция как сервис» усложняет эту задачу), рекомендуем снизить значение параметра `transactional.id.expiration.ms` таким образом, чтобы срок действия идентификаторов истекал быстрее и, следовательно, старое состояние, которое никогда не будет повторно использоваться, не занимало значительную часть памяти брокера.

Производительность транзакций

Транзакции умеренно увеличивают накладные расходы для производителя. Запрос на регистрацию идентификатора транзакции выполняется один раз в жизненном цикле производителя. Дополнительные вызовы для регистрации разделов как части транзакции выполняются не более одного раза на раздел для каждой транзакции, затем каждая транзакция посылает запрос на фиксацию, что вызывает запись дополнительного маркера фиксации в каждом разделе. Запросы на инициализацию транзакций и на фиксацию транзакций выполняются синхронно, поэтому никакие данные не будут отправляться до тех пор, пока они не завершатся успешно, не произойдет сбой или не истечет тайм-аут, что еще больше увеличивает накладные расходы.

Обратите внимание на то, что накладные расходы на транзакции для производителя не зависят от количества сообщений в них. Таким образом, большее количество сообщений в транзакции уменьшит относительные накладные расходы и сократит количество синхронных остановок, что увеличит общую пропускную способность.

На стороне потребителя существуют некоторые накладные расходы, связанные с чтением маркеров фиксации. Основное влияние, которое транзакции оказывают на производительность потребителей, связано с тем, что потребители в режиме `read_committed` не будут возвращать записи, являющиеся частью открытой транзакции. Длительные интервалы между фиксациями транзакций означают, что потребителю придется ждать дольше, прежде чем вернуть сообщения, и в результате увеличивается сквозная задержка.

Однако обратите внимание на то, что потребителю не нужно буферизовать сообщения, принадлежащие открытым транзакциям. Брокер не будет возвращать их в ответ на запросы на выборку от потребителя. Поскольку потребителю не приходится выполнять дополнительную работу при чтении транзакций, пропускная способность также не снижается.

Резюме

Семантика «точно один раз» в Kafka — это противоположность шахматам: она сложна для понимания, но проста в использовании.

В этой главе мы рассмотрели два ключевых механизма, обеспечивающих гарантии «точно один раз» в Kafka: идемпотентный производитель, который позволяет избежать дублирования, вызванного механизмом повторных попыток, и транзакции, составляющие основу семантики «точно один раз» в Kafka Streams.

Оба варианта могут быть включены в одну конфигурацию и позволяют применять Kafka для приложений, требующих меньшего количества дубликатов и более надежных гарантий корректности. Мы подробно обсудили конкретные сценарии и примеры их использования, чтобы показать ожидаемое поведение, и даже рассмотрели некоторые детали реализации. Эти сведения важны при устранении неполадок в приложениях или при непосредственном применении транзакционных API.

Понимая, что гарантирует семантика Kafka «точно один раз» в том или ином случае, мы можем разрабатывать приложения, которые будут использовать ее, когда это необходимо. Поведение приложений не должно удивлять, и информация, содержащаяся в этой главе, поможет избежать неожиданностей.

ГЛАВА 9

Создание конвейеров данных

При обсуждении создания конвейеров данных с помощью Apache Kafka обычно подразумевают несколько сценариев использования. Первый — создание конвейера данных, в котором Apache Kafka представляет собой одну из двух конечных точек. Например, перемещение данных из Kafka в S3 или из MongoDB в Kafka. Второй сценарий включает создание конвейера данных между двумя различными системами с Kafka в качестве промежуточной. Примером может служить перемещение данных из Twitter в Elasticsearch путем отправки их сначала в Kafka, а затем из Kafka в Elasticsearch.

Увидев, что Kafka используется в обоих этих сценариях в LinkedIn и других крупных компаниях, мы добавили в Apache Kafka фреймворк Kafka Connect. Мы обратили внимание на специфические задачи по интеграции Kafka в конвейеры данных, которые приходилось решать каждой из этих компаний, и решили добавить в Kafka API, которые решали бы некоторые из этих задач, вместо того чтобы заставлять всех делать это с нуля.

Главная ценность Kafka для конвейеров данных состоит в том, что она может служить очень большим надежным буфером между различными этапами конвейера. Тем самым разделяет производителей данных и их потребителей внутри конвейера и позволяет использовать одни и те же данные из источника в нескольких целевых приложениях и системах, которые имеют различные требования к своевременности и доступности. Благодаря этому, а также своим надежности и эффективности Kafka очень хорошо подходит для большинства конвейеров данных.



Учет интеграции данных

Некоторые компании рассматривают Kafka как конечную точку конвейера. Они формулируют свои вопросы примерно так: «Как мне передать данные из Kafka в Elastic?» Это вполне резонный вопрос, особенно если данные, находящиеся сейчас в Kafka, нужны вам в Elastic, и мы рассмотрим способы добиться

этого. Но начнем с обсуждения использования Kafka в более широком контексте, включающем по крайней мере две (а может, и намного больше) конечные точки, не считая Kafka. Рекомендуем каждому, кто столкнулся с задачей интеграции данных, исходить из более широкой картины, а не заикливаться на непосредственных конечных точках. Сосредоточить усилия только на текущих задачах интеграции — верный способ в итоге получить сложную и дорогостоящую в поддержке мешанину вместо системы.

В этой главе мы обсудим некоторые распространенные нюансы, которые необходимо учитывать при создании конвейеров данных. Они представляют собой не что-то специфичное для Kafka, а общие проблемы интеграции данных. Однако мы покажем, что Kafka отлично подходит для связанных с интеграцией данных сценариев использования и продемонстрируем решение многих из этих проблем с ее помощью. Мы обсудим отличия API Kafka Connect от обычных клиентов-производителей и клиентов-потребителей, а также обстоятельства, при которых должен применяться каждый из типов клиентов. Хотя полномасштабное изучение Kafka Connect выходит за рамки данной главы, мы продемонстрируем простейшие примеры использования этого фреймворка, чтобы познакомить вас с ним, и укажем, где искать более детальную информацию. Наконец, обсудим другие системы интеграции данных и их объединение с Kafka.

Соображения по поводу создания конвейеров данных

Здесь мы не будем углубляться во все нюансы создания конвейеров данных, однако хотелось бы подчеркнуть некоторые детали, которые важно учесть при проектировании архитектур программного обеспечения, нацеленных на интеграцию нескольких систем.

Своевременность

В части систем ожидается, что данные будут поступать большими порциями раз в день, в других данные должны доставляться через несколько миллисекунд после генерации. Большинство конвейеров данных представляют собой что-то среднее между этими двумя крайностями. Хорошие системы интеграции данных способны соответствовать различным требованиям к своевременности для разных конвейеров, а также переходить от одного графика к другому при изменении бизнес-требований. Kafka как платформу потоковой обработки с масштабируемым и надежным хранилищем можно использовать для поддержки чего угодно, начиная от конвейеров, работающих практически в режиме реального

времени, до пакетов, поступающих раз в час. Производители могут записывать данные в Kafka с такой частотой, с какой требуется, а потребители могут читать и доставлять самые свежие события по мере их поступления. Возможна также реализация пакетного режима работы потребителей с запуском раз в час, подключением к Kafka и чтением накопившихся за этот час событий.

В этом контексте удобно рассматривать Kafka как гигантский буфер, который разделяет требования к интервалам времени, относящимся к производителям и потребителям. Производители могут записывать события в режиме реального времени, а потребители — обрабатывать пакеты событий, или наоборот. Появляется также возможность приостановки процесса — Kafka сама контролирует обратный поток в производителях за счет отсрочки подтверждений при необходимости, поскольку скорость получения данных целиком зависит от потребителей.

Надежность

Нам хотелось бы избежать отдельных критических точек и обеспечить быстрое автоматическое восстановление после разнообразных сбоев. Данные часто поступают по конвейерам в критичные для бизнеса системы, и сбой длительностью более нескольких секунд может иметь разрушительные последствия, особенно если в требованиях к своевременности упоминаются величины порядка нескольких миллисекунд. Еще один важный фактор надежности — гарантии доставки данных. Хотя в некоторых системах потери данных допустимы, чаще всего требуется *как минимум однократная* их доставка. Это означает, что все события, отправленные из системы-источника, должны достичь пункта назначения, хотя иногда возможно появление дубликатов из-за повторной отправки. Часто выдвигается даже требование *строго однократной* доставки — все события, отправленные из системы-источника, должны достичь пункта назначения без каких-либо потерь или дублирования.

Доступность и гарантии надежности Kafka подробно обсуждались в главе 7. Как мы говорили, самостоятельно Kafka способна обеспечить как минимум однократную доставку, а в сочетании с внешним хранилищем с поддержкой транзакционной модели или уникальных ключей — и строго однократную. Поскольку многие из конечных точек представляют собой хранилища данных, обеспечивающие возможность строго однократной доставки, конвейер на основе Kafka можно сделать строго однократным. Стоит упомянуть, что API Kafka Connect при обработке смещений предоставляет API для интеграции с внешними системами, упрощающие построение сквозных конвейеров строго однократной доставки. Разумеется, многие из существующих коннекторов с открытым исходным кодом поддерживают строго однократную доставку.

Высокая/переменная нагрузка

Создаваемые конвейеры данных должны масштабироваться до очень высокой производительности, часто необходимой в современных информационных системах. И что еще важнее, они должны уметь приспосабливаться к внезапному повышению нагрузки.

Благодаря Kafka, служащей буфером между производителями и потребителями, теперь не требуется связывать производительность потребителей с производительностью производителей. Больше не нужен сложный механизм контроля обратного потока данных, поскольку, если производительность производителя превышает производительность потребителя, данные будут просто накапливаться в Kafka до тех пор, пока потребитель не догонит производителя. Умение Kafka масштабироваться за счет независимого добавления производителей и потребителей дает возможность динамически и независимо масштабировать любую из сторон конвейера, чтобы приспособиться к меняющимся требованиям.

Kafka — распределенная система с высокой пропускной способностью, которая может обрабатывать сотни мегабайт данных в секунду даже на не очень мощных кластерах, так что можно не бояться, что конвейер не сможет масштабироваться в соответствии с растущими требованиями. Кроме того, API Kafka Connect фокусируется на распараллеливании работы и может выполнять ее как на одном узле, так и путем масштабирования в зависимости от системных требований. В следующем разделе мы опишем, как платформа Kafka дает возможность источникам и приемникам данных распределять работы по нескольким потокам выполнения и использовать доступные ресурсы процессора даже при работе на отдельной машине.

Kafka также поддерживает несколько типов сжатия, благодаря чему пользователи и администраторы могут контролировать использование ресурсов сети и устройств хранения при росте нагрузки.

Форматы данных

Одна из важнейших задач конвейеров данных — согласование их форматов и типов. Различные базы данных и другие системы хранения поддерживают разные форматы данных. Вам может потребоваться загрузить в Kafka данные в формате XML и реляционные данные, использовать внутри Kafka формат Avro, а затем преобразовать данные в формат JSON для записи в Elasticsearch, или в формат Parquet для записи в HDFS, или в CSV для записи в S3.

Самой Kafka и API Kafka Connect форматы данных совершенно неважны. Как мы видели в предыдущих главах, производители и потребители могут

применить любой сериализатор для представления данных в любом формате. Хранящиеся в оперативной памяти собственные объекты Kafka Connect включают типы и схемы данных, но, как мы скоро узнаем, Kafka Connect позволяет использовать подключаемые преобразователи формата для хранения этих записей в произвольном формате. Это значит, что вне зависимости от задействованного в ней формата данных Kafka не ограничивает выбор преобразователей.

У многих источников и приемников данных есть схемы: можно прочитать схему из источника вместе с данными, сохранить ее и воспользоваться ею в дальнейшем для проверки совместимости или даже обновить ее в базе данных приемника. Классический пример — конвейер данных из MySQL в Snowflake. Хороший конвейер данных при добавлении столбца в MySQL обеспечивает добавление его и в Snowflake, чтобы можно было загрузить туда новые данные.

Кроме того, при записи данных из Kafka во внешние системы коннекторы приемников данных отвечают за формат записываемых данных. Некоторые из них делают этот формат подключаемым. Например, коннектор S3 позволяет выбирать между форматами Avro и Parquet.

Просто поддерживать различные типы данных недостаточно. Универсальный фреймворк интеграции данных должен также решать проблемы различия поведения разных источников и приемников данных. Например, Syslog представляет собой источник, «проталкивающий» данные, а реляционные базы данных требуют, чтобы фреймворк извлекал данные из них. HDFS — это файловая система, предназначенная только для добавления данных, так что их в нее можно только записывать, в то время как большинство систем дают возможность как дописывать данные, так и обновлять существующие записи.

Преобразования

Преобразования — самые неоднозначные из всех требований. Существует две парадигмы создания конвейеров данных: ETL и ELT. ETL (расшифровывается как Extract — Transform — Load — *«извлечь — преобразовать — загрузить»*) означает, что конвейер данных отвечает за изменение проходящих через него данных. Это дает ощутимую экономию времени и места, поскольку не требуется сохранять данные, менять их и сохранять снова. В зависимости от преобразований иногда это преимущество реально, а иногда просто перекладывает бремя вычислений и хранения на сам конвейер данных, что может быть нежелательным. Основной недостаток такого подхода заключается в том, что производимые в конвейере данных преобразования могут лишить нас возможности обрабатывать данные в дальнейшем. Если создатель конвейера между MongoDB и MySQL решил отфильтровать часть событий или убрать из записей некоторые поля, то у всех обращающихся к данным в MySQL пользователей и приложений окажется доступ лишь к части данных. Если им потребуется

доступ к отсутствующим полям, придется перестраивать конвейер и повторно обрабатывать уже обработанные данные (если они еще доступны).

ELT расшифровывается как *«извлечь — загрузить — преобразовать»* (Extract — Load — Transform) и означает, что конвейер лишь минимально преобразует данные (в основном это касается преобразования типов данных) с тем, чтобы попадающие по месту назначения данные как можно меньше отличались от исходных. В них целевая система собирает сырые данные и обрабатывает их должным образом. Их преимущество заключается в максимальной гибкости: у пользователей целевой системы есть доступ ко всем данным. В этих системах также проще искать причины проблем, поскольку вся обработка данных выполняется в одной системе, а не распределяется между конвейером и дополнительными приложениями. Недостаток — в расходе ресурсов CPU и хранилища в целевой системе. В некоторых случаях эти ресурсы обходятся недешево, и желательно по возможности вынести обработку из этих систем.

Kafka Connect включает функцию преобразования одного сообщения (Single Message Transformation), которая преобразует записи во время их копирования из источника в Kafka или из Kafka в цель. Она включает в себя маршрутизацию сообщений в различные топики, фильтрацию сообщений, изменение типов данных, редактирование определенных полей и многое другое. Более сложные преобразования, включающие объединение и агрегирование, обычно выполняются с помощью Kafka Streams, и мы подробно рассмотрим их в отдельной главе.



При создании ETL-системы с помощью Kafka следует помнить, что Kafka позволяет создавать конвейеры «один ко многим», где исходные данные записываются в нее один раз, а затем потребляются несколькими приложениями и записываются в несколько целевых систем. Ожидаются некоторые предварительные обработка и очистка, например стандартизация временных меток и типов данных, добавление истории и, возможно, удаление личной информации — преобразования, которые принесут пользу всем потребителям данных. Но не стоит преждевременно очищать и оптимизировать данные при поступлении, потому что в другом месте они могут понадобиться менее обработанными.

Безопасность

Безопасность должна быть важна всегда. В терминологии конвейеров данных основные проблемы безопасности обычно состоят в следующем.

- Кто имеет доступ к данным, поступающим в Kafka?
- Можем ли мы гарантировать шифрование проходящих через конвейер данных? В основном это важно для конвейеров, проходящих через границы ЦОД.
- Кому разрешено вносить в конвейер изменения?

- Может ли конвейер при чтении им данных из мест с контролируемым доступом обеспечить должную аутентификацию?
- Соответствует ли наша работа с персонально идентифицируемой информацией (ПИ) законам и нормативным актам, касающимся ее хранения, использования и доступа к ней?

Kafka предоставляет возможность шифрования данных при передаче, когда она встроена в конвейер между источниками и приемниками данных. Она также поддерживает аутентификацию (через SASL) и авторизацию, так что вы можете быть спокойны: если топик содержит конфиденциальную информацию, никто не уполномоченный на это не передаст ее в менее защищенные системы. В Kafka также имеется журнал аудита для отслеживания доступа — санкционированного и несанкционированного. Написав немного дополнительного кода, можно отследить, откуда поступили события, находящиеся в каждом топике, кто их менял, и таким образом получить полную историю каждой записи.

Безопасность Kafka подробно рассматривается в главе 11. Однако Kafka Connect и его коннекторы должны иметь возможность подключения к внешним системам данных и аутентификации в них, а конфигурация коннекторов будет включать учетные данные для аутентификации во внешних системах данных.

В настоящее время не рекомендуется хранить учетные данные в конфигурационных файлах, поскольку это означает, что нужно очень осторожно обращаться с такими файлами и ограничить доступ к ним. Распространенным решением является использование внешней системы управления учетными секретами, такой как HashiCorp Vault (<https://www.vaultproject.io>). Kafka Connect включает поддержку внешней конфигурации секретов (<https://oreil.ly/5eVRU>). Apache Kafka включает только фреймворк, который позволяет внедрять подключаемые внешние провайдеры конфигурации, пример провайдера, который считывает конфигурацию из файла, а также разработанные сообществом внешние провайдеры конфигурации (<https://oreil.ly/ovntG>), которые интегрируются с Vault, AWS и Azure.

Обработка сбоев

Считать, что все данные всегда будут в полном порядке, очень опасно. Важно заранее предусмотреть обработку сбоев. Можно ли сделать так, чтобы дефектные записи никогда не попадали в конвейер? Можно ли восстановить работу системы после обработки не поддающихся разбору записей? Можно ли исправить «плохие» записи (возможно, при вмешательстве оператора) и обработать их заново? Что, если «плохая» запись выглядит точно так же, как нормальная, и проблема вскроется лишь через несколько дней?

Благодаря тому, что Kafka может быть настроена на долгое хранение всех событий, можно при необходимости вернуться назад во времени и исправить ошибки. Это также позволяет воспроизводить события, хранящиеся в Kafka, в целевой системе, если они были утеряны.

Связывание и гибкость

Одной из важнейших задач реализации конвейеров данных является расцепление источников и приемников данных. Случайное связывание может возникнуть множеством способов.

- *Узкоспециализированные конвейеры.* Некоторые компании создают по отдельному конвейеру для каждой пары приложений, которые нужно связать. Например, они используют Logstash, чтобы выгрузить журналы в Elasticsearch, Flume, чтобы выгрузить журналы в HDFS, Oracle GoldenGate для передачи данных из Oracle в HDFS, Informatica для переброски данных из MySQL и XML-файлов в Oracle и т. д. Такая практика приводит к сильному связыванию конвейера данных с конкретными конечными точками и образует мешанину из точек интеграции, требующую немалых затрат труда для развертывания, сопровождения и мониторинга. Из-за этого возрастают затраты на внедрение новых технологий и усложняются инновации, ведь для каждой новой системы, появляющейся в компании, приходится создавать дополнительные конвейеры.
- *Потери метаданных.* Если конвейер данных не сохраняет метаданные схемы и не позволяет ей эволюционировать, производящее данные программное обеспечение окажется в конечном итоге сильно связанным с программным обеспечением, их использующим. Без информации о схеме каждый из этих программных продуктов должен будет содержать информацию о способе разбора данных и их интерпретации. Если данные движутся из Oracle в HDFS и администратор базы данных добавил в Oracle новое поле, не сохранив информацию о схеме и не разрешив ей эволюционировать, то всем разработчикам придется одновременно модифицировать свои приложения. В противном случае все приложения, читающие из HDFS данные, перестанут работать. Оба эти варианта отнюдь не означают, что адаптация будет быстрой. Если конвейер поддерживает эволюцию схемы, то все команды разработчиков могут менять свои приложения независимо друг от друга, не волнуясь, что далее по конвейеру что-то перестанет работать.
- *Чрезмерная обработка.* Как мы уже упоминали при обсуждении преобразований данных, определенная обработка данных — неотъемлемое свойство конвейеров. В конце концов, данные перемещаются между разными системами, в которых используются разные форматы данных и поддерживаются

различные сценарии. Однако чрезмерная обработка ограничивает располагающиеся далее по конвейеру системы решениями, принятыми при создании конвейера: о том, какие поля сохранять, как агрегировать данные и т. д. Часто из-за этого конвейер постоянно изменяется по мере смены требований от приложений, располагающихся далее по конвейеру, что неэффективно, небезопасно и плохо соответствует концепции быстрой адаптации. Чтобы адаптация была быстрой, стоит сохранить как можно больше необработанных данных и разрешить располагающимся далее по конвейеру приложениям, включая приложения Kafka Streams, самим решать, как их обрабатывать и агрегировать.

Когда использовать Kafka Connect, а когда — клиенты-производители и клиенты-потребители

При записи данных в Kafka или чтении из нее можно использовать традиционные клиент-производитель и клиент-потребитель, как описано в главах 3 и 4, или воспользоваться API Kafka Connect и коннекторами, как мы покажем в следующих разделах. Прежде чем углубиться в нюансы Kafka Connect, возможно, вы уже задаетесь вопросом, когда каждую из этих возможностей применить.

Как мы уже видели, клиенты Kafka представляют собой клиенты, встраиваемые в ваше же приложение. Благодаря этому приложение может читать данные из Kafka и записывать данные в нее. Используйте клиенты Kafka тогда, когда у вас есть возможность модифицировать код приложения, к которому вы хотите подключиться, и когда вы хотели бы поместить данные в Kafka или извлечь их из нее.

Kafka Connect же вы будете использовать для подключения Kafka к хранилищам данных, созданным не вами, код или API которых вы не можете или не должны менять. Kafka Connect применяется для извлечения данных из внешнего хранилища данных в Kafka или помещения данных из нее во внешнее хранилище. Чтобы использовать Kafka Connect, вам нужен коннектор для хранилища данных, к которому вы хотите подключиться, и в настоящее время таких коннекторов очень много. Это означает, что на практике пользователям Kafka Connect нужно только написать конфигурационные файлы.

Если же нужно подключить Kafka к хранилищу данных, для которого еще не существует коннектора, можно написать приложение, задействующее или клиенты Kafka, или Kafka Connect. Рекомендуется работать с Connect, поскольку он предоставляет такие готовые возможности, как управление настройками, хранение смещений, распараллеливание, обработка ошибок, поддержка различных типов данных и стандартные REST API для управления коннекторами. Кажется, что написать маленькое приложение для подключения Kafka к хранилищу данных очень просто, но вам придется учесть много мелких нюансов,

относящихся к типам данных и настройкам, так что задача окажется не такой уж простой. Более того, вам нужно будет поддерживать это конвейерное приложение и документировать его, а ваши коллеги должны будут научиться его использовать. Kafka Connect является стандартной частью экосистемы Kafka и выполняет большую часть этой работы за вас, позволяя вам сосредоточиться на транспортировке данных во внешние хранилища и обратно.

Kafka Connect

Фреймворк Kafka Connect — часть Apache Kafka, обеспечивающая масштабируемый и гибкий способ копирования данных между Kafka и другими хранилищами данных. Он предоставляет API и среду выполнения для разработки и запуска *плагинов-коннекторов* (connector plugins) — исполняемых Kafka Connect библиотек, отвечающих за перемещение данных. Kafka Connect выполняется в виде кластера процессов-исполнителей (worker processes). Необходимо установить плагины-коннекторы на исполнителях, после чего использовать API REST для настройки коннекторов, выполняемых с определенными конфигурациями, и управления ими. Коннекторы запускают дополнительные *задачи* (tasks) для параллельного перемещения больших объемов данных и эффективного использования доступных ресурсов рабочих узлов. Задачам коннектора источника необходимо лишь прочитать данные из системы-источника и передать объекты данных коннектора процессам-исполнителям. Задачи коннектора приемника получают объекты данных коннектора от исполнителей и отвечают за их запись в целевую информационную систему. Для обеспечения хранения этих объектов данных в Kafka в различных форматах Kafka Connect применяет *преобразователи формата* (convertors) — поддержка формата JSON встроена в Apache Kafka, а реестр схем Confluent предоставляет преобразователи форматов Avro, Protobuf и JSON Schema. Благодаря этому пользователи могут выбирать формат хранения данных в Kafka независимо от задействованных коннекторов, а также того, как обрабатывается схема данных (если вообще обрабатывается).

В этой главе мы, разумеется, не можем обсудить все нюансы Kafka Connect и множества его коннекторов. Это потребовало бы отдельной книги. Однако сделаем обзор Kafka Connect и того, как он применяется, а также укажем, где искать дополнительную справочную информацию.

Запуск Kafka Connect

Kafka Connect поставляется вместе с Apache Kafka, так что устанавливать его отдельно не требуется. Для промышленной эксплуатации, особенно если вы собираетесь использовать Connect для перемещения больших объемов данных или запускать большое число коннекторов, желательно установить Kafka Connect на

отдельном сервере из ваших брокеров Kafka. В этом случае установите Apache Kafka на все машины, запустив на части серверов брокеры, а на других серверах — Connect.

Запуск исполнителя Kafka Connect напоминает запуск брокера. Нужно просто вызвать сценарий запуска, передав ему файл с параметрами:

```
bin/connect-distributed.sh config/connect-distributed.properties
```

Вот несколько основных настроек исполнителей Connect.

- **bootstrap.servers** — список брокеров Kafka, с которыми будет работать Connect. Коннекторы будут передавать данные в эти брокеры или из них. Указывать в этом списке все брокеры кластера не нужно, но рекомендуется хотя бы три.
- **group.id** — все исполнители с одним идентификатором группы образуют один кластер Connect. Запущенный на нем коннектор может оказаться запущенным на любом из исполнителей кластера, как и его задачи.
- **plugin.path**. Kafka Connect использует подключаемую архитектуру, в которой коннекторы, конвертеры, преобразования и провайдеры учетных цифровых идентификационных данных могут быть загружены и добавлены на платформу. Для этого Kafka Connect должен иметь возможность находить и загружать эти плагины.

Мы можем настроить один или несколько каталогов в качестве мест, где могут быть найдены коннекторы и их зависимости. Например, можем настроить `plugin.path=/opt/connectors,/home/gwenshap/connectors`. Внутри одного из этих каталогов мы обычно создаем подкаталог для каждого коннектора, поэтому в предыдущем примере создадим `/opt/connectors/jdbc` и `/opt/connectors/elastic`. Внутри каждого подкаталога мы разместим сам jar-файл коннектора и все его зависимости. Если коннектор поставляется как `uberJar` и не имеет зависимостей, его можно поместить непосредственно в `plugin.path` и он не требует подкаталога. Но обратите внимание на то, что размещение зависимостей в пути верхнего уровня не будет работать.

Альтернативный вариант — добавление коннекторов и всех их зависимостей в путь к классу Kafka Connect, но делать это не рекомендуется, так как могут возникнуть ошибки, если вы используете коннектор, который вносит зависимость, конфликтующую с одной из зависимостей Kafka. Рекомендуемый подход заключается в применении конфигурации `plugin.path`.

- **key.converter** и **value.converter** — Connect может работать с несколькими форматами данных, хранимых в Kafka. Эти две настройки задают преобразователь формата для ключа и значения сообщения, сохраняемого в Kafka. По умолчанию используется формат JSON с включенным в Apache Kafka преобразователем `JSONConverter`. Можно также установить их равными

AvroConverter, ProtobufConverter или JscSchemaConverter — это составные части реестра схем Confluent.

У некоторых преобразователей формата есть особые параметры конфигурации. Перед ними нужно поставить префикс `key.converter.` или `value.converter.` в зависимости от того, к какому преобразователю — ключу или значению — вы хотите их применить. Например, сообщения в формате JSON могут включать или не включать схему. Чтобы конкретизировать эту возможность, нужно установить параметр `key.converter.schemas.enable=true` или `false` соответственно. Аналогичную настройку можно выполнить для преобразователя значений, установив параметр `value.converter.schemas.enable` в `true` или `false`. Сообщения Avro также содержат схему, но необходимо задать местоположение реестра схем с помощью свойств `key.converter.schema.registry.url` и `value.converter.schema.registry.url`.

- `rest.host.name` и `rest.port`. Для настройки и контроля коннекторов обычно используется API REST или Kafka Connect. При необходимости вы можете задать конкретный порт для API REST.

Настроив исполнителей, убедитесь, что ваш кластер работает, с помощью API REST:

```
$ curl http://localhost:8083/
{"version":"3.0.0-SNAPSHOT", "commit":"fae0784ce32a448a", "kafka_cluster_id":"pfkYIGZQSXm8RylvACQHdg"}%
```

В результате обращения к корневому URI REST должна быть возвращена текущая версия. Мы работаем с предварительным выпуском Kafka 3.0.0. Можно также просмотреть доступные плагины коннекторов:

```
$ curl http://localhost:8083/connector-plugins

[
  {
    "class": "org.apache.kafka.connect.file.FileStreamSinkConnector",
    "type": "sink",
    "version": "3.0.0-SNAPSHOT"
  },
  {
    "class": "org.apache.kafka.connect.file.FileStreamSourceConnector",
    "type": "source",
    "version": "3.0.0-SNAPSHOT"
  },
  {
    "class": "org.apache.kafka.connect.mirror.MirrorCheckpointConnector",
    "type": "source",
    "version": "1"
  },
  {
```

```
    "class": "org.apache.kafka.connect.mirror.MirrorHeartbeatConnector",  
    "type": "source",  
    "version": "1"  
  },  
  {  
    "class": "org.apache.kafka.connect.mirror.MirrorSourceConnector",  
    "type": "source",  
    "version": "1"  
  }  
]
```

У нас запущена простая Apache Kafka, так что доступны только плагины коннекторов для файлового источника, файлового приемника и коннекторы, которые входят в состав MirrorMaker 2.0.

Взглянем теперь на настройку и использование этих образцов коннекторов, после чего перейдем к более сложным примерам, требующим настройки внешних информационных систем, к которым будет осуществляться подключение.



Автономный режим

Отметим, что у Kafka Connect имеется автономный режим. Он схож с распределенным режимом — нужно просто запустить сценарий `bin/connect-standalone.sh` вместо `bin/connect-distributed.sh`. Файл настроек коннекторов можно передать в командной строке, а не через API REST. В этом режиме все коннекторы и задачи выполняются на одном автономном исполнителе. Он используется в случаях, когда коннекторы и задачи должны выполняться на конкретной машине (например, коннектор `syslog` прослушивает порт, так что вам нужно знать, на каких машинах он работает).

Пример коннектора: файловый источник и файловый приемник

Здесь мы воспользуемся файловыми коннекторами и преобразователем формата для JSON, включенными в состав Apache Kafka. Чтобы следить за ходом рассуждений, убедитесь, что у вас установлены и запущены ZooKeeper и Kafka.

Для начала запустите распределенный исполнитель Connect. При промышленной эксплуатации должны работать хотя бы два или три таких исполнителя, чтобы обеспечить высокую доступность. В данном примере он будет один:

```
bin/connect-distributed.sh config/connect-distributed.properties &
```

Теперь можно запустить файловый источник. В качестве примера настроим его на чтение файла конфигурации Kafka — фактически передадим конфигурацию Kafka в топик Kafka:

```
echo '{"name":"load-kafka-config", "config":{"connector.class":
"FileStreamSource", "file":"config/server.properties", "topic":
"kafka-config-topic"}}' | curl -X POST -d @- http://localhost:8083/connectors
-H "Content-Type: application/json"
```

```
{
  "name": "load-kafka-config",
  "config": {
    "connector.class": "FileStreamSource",
    "file": "config/server.properties",
    "topic": "kafka-config-topic",
    "name": "load-kafka-config"
  },
  "tasks": [
    {
      "connector": "load-kafka-config",
      "task": 0
    }
  ],
  "type": "source"
}
```

Для создания коннектора мы написали JSON-текст, включающий название коннектора — `load-kafka-config` — и ассоциативный массив его настроек, включающий класс коннектора, загружаемый файл и топик, в который мы хотим его загрузить.

Воспользуемся консольным потребителем Kafka для проверки загрузки настроек в топик:

```
gwen$ bin/kafka-console-consumer.sh --new-consumer --bootstrap-
server=localhost:9092
--topic kafka-config-topic --from-beginning
```

Если все прошло успешно, вы увидите что-то вроде:

```
{"schema":{"type":"string","optional":false},"payload":"# Licensed to the
Apache Software Foundation (ASF) under one or more"}
<more stuff here>

{"schema":{"type":"string","optional":false},"payload":"# Server Basics
#####"}
{"schema":{"type":"string","optional":false},"payload":""}
{"schema":{"type":"string","optional":false},"payload":"# The id of the broker.
This must be set to a unique integer for each broker."}
{"schema":{"type":"string","optional":false},"payload":"broker.id=0"}
{"schema":{"type":"string","optional":false},"payload":""}

<more stuff here>
```

Фактически это содержимое файла `config/server.properties`, преобразованного нашим коннектором построчно в формат JSON и помещенного в `kafka-config-topic`. Обратите внимание на то, что по умолчанию преобразователь формата JSON вставляет схему в каждую запись. В данном случае схема очень проста — всего один столбец `payload` типа `string`, содержащий по одной строке из файла для каждой записи.

А сейчас воспользуемся преобразователем формата файлового приемника для сброса содержимого этого топика в файл. Итоговый файл должен оказаться точно таким же, как и исходный `config/server.properties`, поскольку преобразователь JSON преобразует записи в формате JSON в обычные текстовые строки:

```
echo '{"name":"dump-kafka-config", "config":
{"connector.class":"FileStreamSink", "file":"copy-of-server-
properties", "topics":"kafka-config-topic"}}' | curl -X POST -d @- http://local-
host:8083/connectors --header "content-Type:application/json"
```

```
{"name":"dump-kafka-config", "config":
{"connector.class":"FileStreamSink", "file":"copy-of-server-
properties", "topics":"kafka-config-topic", "name":"dump-kafka-config"}, "tasks":
[]}
```

Обратите внимание на отличие от исходных настроек: сейчас мы используем класс `FileStreamSink`, а не `FileStreamSource`. У нас по-прежнему есть свойство `file`, но теперь оно указывает на целевой файл, а не на источник записей, и вместо `topic` мы указываем `topics`. Внимание, множественное число! С помощью приемника можно записывать несколько топиков в один файл, в то время как источник позволяет записывать только в один топик.

В случае успешного выполнения вы получите файл `copy-of-server-properties`, совершенно идентичный файлу `config/server.properties`, на основе которого мы заполняли `kafka-config-topic`.

Удалить коннектор можно с помощью команды:

```
curl -X DELETE http://localhost:8083/connectors/dump-kafka-config
```



В этом примере используются коннекторы `FileStream`, поскольку они просты и встроены в Kafka, что позволит вам создать свой первый конвейер, не устанавливая ничего, кроме Kafka. Их не следует применять для реальных производственных конвейеров, поскольку они имеют множество ограничений и не гарантируют надежности. Существует несколько альтернатив, которые можно использовать, если вы хотите получать данные из файлов: `FilePulse Connector` (<https://oreil.ly/VLCf2>), `FileSystem Connector` (<https://oreil.ly/Fcryw>) или `SpoolDir` (<https://oreil.ly/qgsI4>).

Пример коннектора: из MySQL в Elasticsearch

Теперь, когда заработал простой пример, пора заняться чем-то более полезным. Возьмем таблицу MySQL, отправим ее в топик Kafka, загрузим оттуда в Elasticsearch и проиндексируем ее содержимое.

Мы выполняем эксперименты на MacBook. Для установки MySQL и Elasticsearch, достаточно выполнить команды:

```
brew install mysql
brew install elasticsearch
```

Следующий шаг — проверить наличие коннекторов. Существует несколько вариантов.

1. Загрузите и установите с помощью клиента Confluent Hub (<https://oreil.ly/c7S5z>).
2. Загрузите с сайта Confluent Hub (<https://www.confluent.io/hub>) или с любого другого сайта, на котором размещен интересующий вас коннектор.
3. Соберите из исходного кода. Для этого вам потребуется:

- клонировать исходный код коннектора:


```
git clone https://github.com/confluentinc/kafka-connect-elasticsearch
```
- выполнить команду `install -DskipTests` для сборки проекта;
- повторить эти действия для коннектора JDBC (<https://oreil.ly/yXg0S>).

Теперь нам нужно загрузить эти коннекторы. Создайте каталог, например, `/opt/connectors`, и обновите `config/connect-distributed.properties`, включив в него `plugin.path=/opt/connectors`.

Затем возьмите JAR-файлы, появившиеся в результате сборки в подкаталогах `target` каталогов, в которых выполнялась сборка коннекторов, и скопируйте каждый из них, а также их зависимости в соответствующие подкаталоги `plugin.path`:

```
gwen$ mkdir /opt/connectors/jdbc
gwen$ mkdir /opt/connectors/elastic
gwen$ cp ../kafka-connect-jdbc/target/kafka-connect-jdbc-10.3.x-SNAPSHOT.jar /opt/connectors/jdbc
gwen$ cp ../kafka-connect-elasticsearch/target/kafka-connect-elasticsearch-11.1.0-SNAPSHOT.jar /opt/connectors/elastic
gwen$ cp ../kafka-connect-elasticsearch/target/kafka-connect-elasticsearch-11.1.0-SNAPSHOT-package/share/java/kafka-connect-elasticsearch/* /opt/connectors/elastic
```

Кроме того, поскольку нам нужно подключиться не просто к любой базе данных, а конкретно к MySQL, потребуется загрузить и установить драйвер MySQL

JDBC. Драйвер не поставляется вместе с коннектором по лицензионным причинам. Вы можете скачать драйвер с сайта MySQL (<https://oreil.ly/KZCPw>), а затем поместить скачанный jar-файл в `/opt/connectors/jdbc`.

Перезапустите исполнители Kafka Connect и проверьте, перечислены ли новые плагины коннекторов в списке:

```
gwen$ bin/connect-distributed.sh config/connect-distributed.properties &

gwen$ curl http://localhost:8083/connector-plugins
[
  {
    "class": "io.confluent.connect.elasticsearch.ElasticsearchSinkConnector",
    "type": "sink",
    "version": "11.1.0-SNAPSHOT"
  },
  {
    "class": "io.confluent.connect.jdbc.JdbcSinkConnector",
    "type": "sink",
    "version": "10.3.x-SNAPSHOT"
  },
  {
    "class": "io.confluent.connect.jdbc.JdbcSourceConnector",
    "type": "source",
    "version": "10.3.x-SNAPSHOT"
  }
]
```

Как видите, в кластере теперь доступны новые плагины коннекторов.

Следующий шаг — создание таблицы в базе данных MySQL, которую потом можно будет передать в Kafka с помощью JDBC-коннектора:

```
gwen$ mysql.server restart
gwen$ mysql --user=root

mysql> create database test;
Query OK, 1 row affected (0.00 sec)

mysql> use test;
Database changed
mysql> create table login (username varchar(30), login_time datetime);
Query OK, 0 rows affected (0.02 sec)

mysql> insert into login values ('gwenshap', now());
Query OK, 1 row affected (0.01 sec)

mysql> insert into login values ('tpalino', now());
Query OK, 1 row affected (0.00 sec)
```

Как видите, мы создали базу данных и таблицу и вставили в нее несколько строк для примера.

Следующий шаг — настройка коннектора JDBC-источника. Можно прочитать о возможностях настройки в документации или воспользоваться API REST для их выяснения:

```
gwen$ curl -X PUT -d '{"connector.class":"JdbcSource"}' localhost:8083/
connector-plugins/JdbcSourceConnector/config/validate --header "content-
Type:application/json"
```

```
{
  "configs": [
    {
      "definition": {
        "default_value": "",
        "dependents": [],
        "display_name": "Timestamp Column Name",
        "documentation": "The name of the timestamp column to use
        to detect new or modified rows. This column may not be
        nullable.",
        "group": "Mode",
        "importance": "MEDIUM",
        "name": "timestamp.column.name",
        "order": 3,
        "required": false,
        "type": "STRING",
        "width": "MEDIUM"
      },
      <more stuff>
    }
  ]
}
```

Мы запросили у API REST проверку настроек коннектора, передав ему конфигурацию, содержащую только имя класса (это минимально необходимая настройка). В качестве ответа получили JSON-описание всех доступных настроек.

Теперь приступим к созданию и настройке JDBC-коннектора:

```
echo '{"name":"mysql-login-connector", "config":{"connector.class":"JdbcSource-
Connector", "connection.url":"jdbc:mysql://127.0.0.1:3306/test?
user=root", "mode":"timestamp", "table.whitelist":"login", "vali-
date.non.null":false, "timestamp.column.name":"login_time", "topic.pre-
fix":"mysql."}}' | curl -X POST -d @- http://localhost:8083/connectors --header
"content-Type:application/json"
```

```
{
  "name":"mysql-login-connector",
  "config":{"
    "connector.class":"JdbcSourceConnector",
    "connection.url":"jdbc:mysql://127.0.0.1:3306/test?user=root",
    "mode":"timestamp",
    "table.whitelist":"login",
    "validate.non.null":"false",
    "timestamp.column.name":"login_time",
    "topic.prefix":"mysql.",
  }
```

```

    "name": "mysql-login-connector"
  },
  "tasks": []
}

```

Прочитаем данные из топика `mysql.login`, чтобы убедиться, что он работает:

```

gwen$ bin/kafka-console-consumer.sh --new --bootstrap-server=localhost:9092 --topic
mysql.login --from-beginning

```

Если вы не увидите никаких данных или получите сообщение, гласящее, что топика не существует, поищите в журналах Connect следующие ошибки:

```

[2016-10-16 19:39:40,482] ERROR Error while starting connector mysql-login-
connector (org.apache.kafka.connect.runtime.WorkerConnector:108)
org.apache.kafka.connect.errors.ConnectException: java.sql.SQLException: Access
denied for user 'root;'@'localhost' (using password: NO)
    at io.confluent.connect.jdbc.JdbcSourceConnector.
start(JdbcSourceConnector.java:78)

```

Среди возможных проблем также отсутствие драйвера по пути к классам или прав на чтение таблицы.

Если после запуска коннектора вы вставите в таблицу `login` дополнительные строки, они должны сразу же отразиться в топике `mysql.login`.



Захват измененных данных и проект Debezium

Используемый нами коннектор JDBC использует JDBC и SQL для сканирования таблиц базы данных на наличие новых записей. Он обнаруживает новые записи с помощью полей временных меток или увеличивающегося первичного ключа. Это довольно неэффективный и порой неточный процесс. Все реляционные базы данных имеют журнал транзакций, также называемый журналом повтора (*redo log*), *binlog* или журналом упреждающей записи (*write-ahead log*), как часть своей реализации, и многие из них позволяют внешним системам считывать данные непосредственно из своего журнала транзакций — это гораздо более точный и эффективный процесс, известный как захват измененных данных. Большинство современных систем ETL зависят от захвата измененных данных в качестве источника данных. Проект Debezium (<https://debezium.io>) предоставляет коллекцию высококачественных коннекторов захвата изменений с открытым исходным кодом для различных баз данных. Если вы планируете передавать данные из реляционной базы данных в Kafka, мы настоятельно рекомендуем использовать коннектор захвата изменений Debezium, если он существует для вашей базы данных. Кроме того, документация Debezium — одна из лучших, которую мы видели: в дополнение к самим коннекторам она охватывает полезные шаблоны проектирования и примеры применения, связанные с захватом измененных данных, особенно в контексте микросервисов.

Передача данных из MySQL в Kafka полезна и сама по себе, но пойдем еще дальше и запишем эти данные в Elasticsearch.

Во-первых, запустим Elasticsearch и проверим его работу, обратившись к соответствующему локальному порту:

```
gwen$ elasticsearch &
gwen$ curl http://localhost:9200/
{
  "name" : "Chens-MBP",
  "cluster_name" : "elasticsearch_gwenshap",
  "cluster_uuid" : "X69zu3_sQNGb7zbMh7NDVw",
  "version" : {
    "number" : "7.5.2",
    "build_flavor" : "default",
    "build_type" : "tar",
    "build_hash" : "8bec50e1e0ad29dad5653712cf3bb580cd1afcdf",
    "build_date" : "2020-01-15T12:11:52.313576Z",
    "build_snapshot" : false,
    "lucene_version" : "8.3.0",
    "minimum_wire_compatibility_version" : "6.8.0",
    "minimum_index_compatibility_version" : "6.0.0-beta1"
  },
  "tagline" : "You Know, for Search"
}
```

Теперь создадим и запустим коннектор:

```
echo '{"name":"elastic-login-connector", "config":{"connector.class":"ElasticsearchSinkConnector", "connection.url":"http://localhost:9200", "type.name":"mysql-data", "topics":"mysql.login", "key.ignore":true}}' |
curl -X POST -d @- http://localhost:8083/connectors --header "Content-Type:application/json"

{
  "name":"elastic-login-connector",
  "config":{
    "connector.class":"ElasticsearchSinkConnector",
    "connection.url":"http://localhost:9200",
    "topics":"mysql.login",
    "key.ignore":"true",
    "name":"elastic-login-connector"
  },
  "tasks":[
    {
      "connector":"elastic-login-connector",
      "task":0
    }
  ]
}
```

Здесь есть несколько настроек, которые требуют пояснений. `connection.url` — просто URL локального сервера Elasticsearch, который мы настроили ранее. Каждый топик в Kafka по умолчанию становится отдельным индексом Elasticsearch с тем же именем, что и у топика. Записываем в Elasticsearch только один топик — `mysql.login`. Коннектор JDBC не заполняет ключ сообщения. В результате ключи событий в Kafka имеют значение `null`. А поскольку у событий в Kafka нет ключей, необходимо сообщить коннектору Elasticsearch, чтобы в качестве ключей для каждого события он использовал название топика, идентификаторы разделов и смещения. Это делается установкой параметра `key.ignore` в значение `true`.

Проверим, что индекс с данными таблицы `mysql.login` создан:

```
gwen$ curl 'localhost:9200/_cat/indices?v'
health status index      uuid                                pri rep docs.count
docs.deleted store.size pri.store.size
yellow open    mysql.login wkeyk9-bQea6NJmAFjv4hw  1   1           2
0      3.9kb      3.9kb
```

Если индекс не найден, поищите ошибки в журнале исполнителя Connect. Зачастую причиной ошибок становится отсутствие параметров или библиотек. Если все в порядке, можем поискать в индексе наши записи:

```
gwen$ curl -s -X "GET" "http://localhost:9200/mysql.login/_search?pretty=true"
{
  "took" : 40,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 3,
      "relation" : "eq"
    },
    "max_score" : 1.0,
    "hits" : [
      {
        "_index" : "mysql.login",
        "_type" : "_doc",
        "_id" : "mysql.login+0+0",
        "_score" : 1.0,
        "_source" : {
          "username" : "gwenshap",
          "login_time" : 1621699811000
        }
      }
    ]
  }
}
```

```

    },
    {
      "_index" : "mysql.login",
      "_type" : "_doc",
      "_id" : "mysql.login+0+1",
      "_score" : 1.0,
      "_source" : {
        "username" : "tpalino",
        "login_time" : 1621699816000
      }
    }
  ]
}
}

```

Если добавить новые записи в таблицу в MySQL, они автоматически появятся в топике `mysql.login` в Kafka и соответствующем индексе Elasticsearch.

Теперь, посмотрев на сборку и установку JDBC-источников и приемников Elasticsearch, мы сможем собрать и использовать любую пару подходящих для нашего сценария коннекторов. Confluent поддерживает набор собственных готовых коннекторов, а также коннекторов от сообщества и других поставщиков на сайте Confluent Hub (<https://www.confluent.io/hub>). Вы можете выбрать из списка любой коннектор, какой вам только хочется попробовать, загрузить его, настроить, прочитав документацию или получив настройки из API REST, и запустить на своем кластере исполнителей Connect.



Создание собственных коннекторов

API коннекторов общедоступен, так что каждый может создать новый коннектор. Мы призываем вас написать собственный коннектор, если хранилище данных, к которому вам нужно подключиться, не имеет готового коннектора. Вы можете затем представить его в Confluent Hub, чтобы другие люди его увидели и смогли использовать. Обсуждение всех нюансов создания коннектора выходит за рамки данной главы, но в блоге есть несколько статей, которые объясняют, как это сделать (<https://oreil.ly/WUqlZ>), а также хорошие доклады с саммита Kafka, состоявшегося в Нью-Йорке в 2019 году (<https://oreil.ly/rV9RH>), с саммита Kafka Sum, прошедшего в Лондоне в 2018-м (<https://oreil.ly/Jz7XV>), и ApacheCon (<https://oreil.ly/8QsOL>). Мы также рекомендуем рассмотреть уже существующие коннекторы в качестве образцов и, возможно, начать с применения шаблона maven (<http://www.bit.ly/2sc9E9q>). Мы всегда будем рады ответить на ваши вопросы и откликнуться на просьбы о помощи, а также увидеть демонстрацию новых коннекторов в почтовой рассылке сообщества разработчиков Apache Kafka (users@kafka.apache.org). Или отправьте их в Confluent Hub, чтобы их можно было легко найти.

Преобразования одиночных сообщений

Копирование записей из MySQL в Kafka и оттуда в Elastic довольно полезно само по себе, но конвейеры ETL обычно включают этап преобразования. В экосистеме Kafka мы разделяем преобразования на преобразования одиночных сообщений (SMTs), которые не имеют состояния, и потоковую обработку, которая может происходить с сохранением состояния. SMT можно выполнять в Kafka Connect, преобразуя сообщения во время их копирования, часто без написания какого-либо кода. Для более сложных преобразований, которые обычно включают в себя соединения или агрегирование, потребуется фреймворк Kafka Streams с сохранением состояния. Мы обсудим Kafka Streams в одной из последующих глав.

Apache Kafka включает следующие SMT.

- *Cast*. Изменение типа данных поля.
- *MaskField*. Замена содержимого поля на `null`. Это полезно для удаления конфиденциальных или идентифицирующих личность данных.
- *Filter*. Удаление или включение всех сообщений, которые соответствуют определенному условию. Встроенные условия включают совпадение по названию топика, определенному заголовку или по тому, является ли сообщение удаленным объектом (то есть имеет нулевое значение).
- *Flatten*. Преобразование вложенной структуры данных в плоскую. Это делается путем объединения всех имен всех полей в пути к определенному значению.
- *HeaderFrom*. Перемещение или копирование полей из сообщения в заголовок.
- *InsertHeader*. Добавление статической строки в заголовок каждого сообщения.
- *InsertField*. Добавление нового поля в сообщение с использованием либо значения из его метаданных, например смещения, либо статического значения.
- *RegexRouter*. Изменение топика назначения с помощью регулярного выражения и строки замены.
- *ReplaceField*. Удаление или переименование поля в сообщении.
- *TimestampConverter*. Изменение формата времени поля, например, из Unix Epoch в строку String.
- *TimestampRouter*. Изменение топика на основе временной метки сообщения. В основном это полезно в коннекторах приемника, когда мы хотим скопировать сообщения в определенные разделы таблицы на основе их временной метки, а поле топика используется для поиска эквивалентного набора данных в системе назначения.

Кроме того, преобразования доступны от участников, не входящих в основную кодовую базу Apache Kafka. Их можно найти на GitHub (полезные коллек-

ции есть у Lenses.io (<https://oreil.ly/fWAYh>), Aiven (<https://oreil.ly/oQRG5>) и Jeremy Custenborder (<https://oreil.ly/OdPHW>) или на Confluent Hub (<https://oreil.ly/Up8dM>).

Чтобы узнать больше о Kafka Connect SMT, вы можете прочитать подробные примеры многих преобразований в серии блогов «Двенадцать дней SMT» (<https://oreil.ly/QnpQV>). Кроме того, вы можете узнать, как написать собственные преобразования, следуя руководству и подробному описанию (<https://oreil.ly/rw4CU>).

В качестве примера предположим, что мы хотим добавить заголовок записи (<https://oreil.ly/ISiWs>) к каждой записи, выданной коннектором MySQL, который мы создали ранее. Заголовок будет указывать на то, что запись была создана этим коннектором MySQL, что будет полезно в том случае, если аудиторы захотят проверить происхождение записей.

Для этого мы заменим предыдущую конфигурацию коннектора MySQL на следующую:

```
echo '{
  "name": "mysql-login-connector",
  "config": {
    "connector.class": "JdbcSourceConnector",
    "connection.url": "jdbc:mysql://127.0.0.1:3306/test?user=root",
    "mode": "timestamp",
    "table.whitelist": "login",
    "validate.non.null": "false",
    "timestamp.column.name": "login_time",
    "topic.prefix": "mysql.",
    "name": "mysql-login-connector",
    "transforms": "InsertHeader",
    "transforms.InsertHeader.type":
      "org.apache.kafka.connect.transforms.InsertHeader",
    "transforms.InsertHeader.header": "MessageSource",
    "transforms.InsertHeader.value.literal": "mysql-login-connector"
  }}' | curl -X POST -d @- http://localhost:8083/connectors --header "content-
Type:application/json"
```

Теперь, если вы вставите еще несколько записей в таблицу MySQL, которую мы создали в предыдущем примере, вы сможете увидеть, что новые сообщения в топике `mysql.login` имеют заголовки (обратите внимание на то, что вам требуется Apache Kafka версии 2.7 или выше для печати заголовков в консольном потребителе):

```
bin/kafka-console-consumer.sh --bootstrap-server=localhost:9092 --topic
mysql.login --from-beginning --property print.headers=true
```

```
NO_HEADERS      {"schema":{"type":"struct","fields":
[{"type":"string","optional":true,"field":"username"},
{"type":"int64","optional":true,"name":"org.apache.kafka.connect.data.Time-
stamp","version":1,"field":"login_time"}],"optional":false,"name":"login"},
```

```

"pay-load":{"username":"tpalino","login_time":1621699816000}}
MessageSource:mysql-login-connector      {"schema":{"type":"struct","fields":
[{"type":"string","optional":true,"field":"username"},
{"type":"int64","optional":true,"name":"org.apache.kafka.connect.data.Time-
stamp","version":1,"field":"login_time"}], "optional":false,"name":"login"},"pay-
load":{"username":"rajini","login_time":1621803287000}}

```

Как можно видеть, старые записи показывают NO_HEADERS, но новые записи показывают MessageSource:mysql-login-connector.



Обработка ошибок и очереди мертвых писем

Преобразования — это пример конфигурации коннектора, которая не является специфической для одного коннектора, но может быть использована в конфигурации любого коннектора. Еще одна очень полезная конфигурация коннектора, которую можно использовать в любом коннекторе приемника, — это `error.tolerance`: вы можете настроить любой коннектор так, что он будет автоматически отбрасывать поврежденные сообщения или перенаправлять в специальный топик, называемый «очередью мертвых писем» (dead letter queue). Более подробную информацию вы можете найти в статье блога «Глубокое погружение в Kafka Connect — обработка ошибок и очереди мертвых писем» (<https://oreil.ly/935hH>).

Взглянем на Kafka Connect поближе

Чтобы понять, как работает Kafka Connect, необходимо разобраться с тремя основными его понятиями и их взаимодействием друг с другом. Как мы уже объясняли и демонстрировали на примерах, для использования Kafka Connect вам нужен работающий кластер исполнителей и потребуется запускать/оставлять коннекторы. Еще один нюанс, в который мы ранее особо не углублялись, — обработка данных преобразователями форматов — компонентами, преобразующими строки MySQL в записи JSON, заносимые в Kafka коннектором.

Заглянем в каждую из систем чуть глубже и разберемся, как они взаимодействуют друг с другом.

Коннекторы и задачи

Плагины коннекторов реализуют API коннекторов, состоящий из двух частей.

- *Коннекторы.* Отвечают за выполнение трех важных вещей:
 - определение числа задач для коннектора;
 - разбиение работы по копированию данных между задачами;
 - получение от исполнителей настроек для задач и передачу их далее.

Например, коннектор JDBC-источника подключается к базе данных, находит таблицы для копирования и на основе этой информации определяет, сколько требуется задач, выбирая меньшее из значений параметра `tasks.max` и числа таблиц. После этого он генерирует конфигурацию для каждой из задач на основе своих настроек (например, параметра `connection.url`) и списка таблиц, которые должны будут копировать все задачи. Метод `taskConfigs()` возвращает список ассоциативных массивов, то есть настроек для каждой из запускаемых задач. Исполнители отвечают за дальнейший запуск задач и передачу каждой из них ее индивидуальных настроек, на основе которых она должна будет скопировать уникальный набор таблиц из базы данных. Отметим, что коннектор при запуске посредством API REST может быть запущен на любом узле, а значит, и запускаемые им задачи тоже могут выполняться на любом из узлов.

- *Задачи.* Отвечают за получение данных из Kafka и вставку туда данных. Исполнители инициализируют все задачи путем передачи контекста. Контекст источника включает объект, предназначенный для хранения задач смещений записей источника (например, в файловом коннекторе смещения представляют собой позиции в файле, в коннекторе JDBC-источника они могут быть столбцом временной метки в таблице). Контекст для коннектора приемника включает методы, с помощью которых он может контролировать получаемые из Kafka записи. Они используются, в частности, для приостановки обратного потока данных, а также повторения отправки и сохранения смещений во внешнем хранилище для обеспечения строго однократной доставки. После инициализации задания запускаются с объектом `Properties`, содержащим настройки, созданные для данной задачи коннектором. После запуска задачи источника опрашивают внешнюю систему и возвращают списки записей, отправляемые исполнителем брокерам Kafka. Задачи приемника получают записи из Kafka через исполнитель и отвечают за отправку этих записей во внешнюю систему.

Исполнители

Процессы-исполнители Kafka Connect представляют собой процессы-контейнеры, выполняющие коннекторы и задачи. Они отвечают за обработку HTTP-запросов с описанием коннекторов и их настроек, а также за хранение настроек коннекторов во внутреннем топики Kafka, запуск коннекторов и их задач, включая передачу соответствующих настроек. В случае останова или аварийного сбоя процесса-исполнителя кластер узнает об этом из контрольных сигналов протокола потребителей Kafka и переназначает работающие на данном исполнителе коннекторы и задачи оставшимся исполнителям. Другие исполнители тоже заметят, если к кластеру Connect присоединится новый исполнитель, и назначат ему коннекторы или задачи, чтобы равномерно распределить нагрузку

между всеми исполнителями. Исполнители отвечают также за автоматическую фиксацию смещений для коннекторов как источника, так и приемника во внутреннем топике Kafka и за выполнение повторов в случае генерации задачами исключений.

Чтобы разобраться в том, что такое исполнители, лучше всего представить себе, что коннекторы и задачи отвечают за ту часть интеграции данных, которая относится к перемещению данных, а исполнители отвечают за API REST, управление настройками, надежность, высокую доступность, масштабирование и распределение нагрузки.

Если сравнивать с классическими API потребителей/производителей, то главное преимущество API Connect состоит именно в этом разделении обязанностей. Опытные разработчики знают, что написание кода для чтения данных из Kafka и вставки их в базу данных занимает, наверное, день или два. Но если вдобавок нужно отвечать за настройки, ошибки, API REST, мониторинг, развертывание, повышающее и понижающее вертикальное масштабирование, а также обработку сбоев, то корректная реализация всего этого может занять несколько месяцев. И большинство конвейеров интеграции данных включают в себя нечто большее, чем просто один источник или цель. Итак, примем во внимание, что усилия, затраченные на создание уникального кода только для интеграции с базой данных, многократно повторяются для других технологий. При воплощении в жизнь копирования данных с помощью коннекторов последние подключаются к исполнителям, которые берут на себя заботы о множестве сложных эксплуатационных вопросов, так что вам не придется об этом беспокоиться.

Преобразователи форматов и модель данных Connect

Последний фрагмент пазла API Connect — модель данных коннектора и преобразователи форматов. API Kafka Connect включают API данных, который, в свою очередь, включает как объекты данных, так и описывающие эти данные схемы. Например, JDBC-источник читает столбец из базы данных и формирует объект Connect Schema на основе типов данных столбцов, которые вернула база данных. Для каждого столбца сохраняются имя столбца и значение. Все коннекторы источников выполняют схожие функции — читают события из системы источника и генерируют пары Schema/Value. У коннекторов приемников функции противоположные — они получают пары Schema/Value и используют объекты Schema для разбора значений и вставки их в целевую систему.

Хотя коннекторы источников знают, как генерировать объекты на основе API данных, остается актуальным вопрос о сохранении этих объектов в Kafka исполнителями Connect. Именно тут вносят свой вклад преобразователи форматов. Пользователи, настраивая исполнитель (или коннектор), выбирают преобразо-

ватель формата, который будет применяться для сохранения данных в Kafka. В настоящий момент в этом качестве можно использовать примитивные типы, массивы байтов, строки, Avro, JSON, схемы JSON или буферы протокола. Преобразователь JSON можно настроить так, чтобы он включал или не включал схему в итоговую запись, таким образом можно обеспечить поддержку как структурированных, так и полуструктурированных данных. Получив от коннектора запись API данных, исполнитель с помощью уже настроенного преобразователя формата преобразует запись в объект Avro, JSON или строковое значение, после чего сохраняет результат в Kafka.

Противоположное происходит с коннекторами приемников. Исполнитель Connect, прочитав запись из Kafka, с помощью уже настроенного преобразователя преобразует запись из формата Kafka (то есть примитивные типы, массивы байтов, строки, Avro, JSON, схемы JSON или Protobuf) в запись API данных Connect, после чего передает ее коннектору приемника, вставляющему ее в целевую систему. Благодаря этому API Connect может поддерживать различные типы хранимых в Kafka данных вне зависимости от реализации коннекторов (то есть можно использовать любой коннектор для любого типа записей, был бы только доступен преобразователь формата).

Управление смещениями

Управление смещениями — один из удобных сервисов, предоставляемых исполнителями коннекторам, помимо развертывания и управления настройками через API REST. Суть его в том, что коннекторам необходимо знать, какие данные они уже обработали, и они могут воспользоваться предоставляемыми Kafka API для хранения информации о том, какие события уже обработаны.

Для коннекторов источников это значит, что записи, возвращаемые коннектором исполнителям Connect, включают информацию о логическом разделе и логическом смещении. Это не разделы и смещения Kafka, а разделы и смещения в том виде, в каком они нужны в системе источника. Например, в файловом источнике раздел может быть файлом, а смещение — номером строки или символа в нем. В JDBC-источнике раздел может быть таблицей базы данных, а смещение — идентификатором записи или временной меткой в таблице. При написании коннектора источника нужно принять одно из важнейших проектных решений: как секционировать данные в системе источника и как отслеживать смещения. Оно влияет на возможный уровень параллелизма коннектора, а также вероятность обеспечения по крайней мере однократной или строго однократной доставки.

После возвращения коннектором источника списка записей, включающего разделы и смещения в источнике для всех записей, исполнитель отправляет записи

брокерам Kafka. Если брокеры сообщили, что получили записи, исполнитель сохраняет смещения отправленных в Kafka записей. Благодаря этому коннекторы могут начинать обработку событий с последнего сохраненного после перезапуска или аварийного сбоя смещения. Механизм хранения является подключаемым и обычно представляет собой топик Kafka, вы можете управлять названием топика с помощью конфигурации `offset.storage.topic`. Кроме того, Connect использует топика Kafka для хранения конфигурации всех созданных нами коннекторов и статуса каждого коннектора — для них используются имена, сконфигурированные с помощью `config.storage.topic` и `status.storage.topic` соответственно.

Последовательность действий, выполняемых коннекторами приемников, аналогична с точностью до наоборот: они читают записи Kafka, в которых уже есть идентификаторы топика, раздела и смещения. Затем вызывают метод `put()` коннектора для сохранения этих записей в целевой системе. В случае успешного выполнения этих действий они фиксируют переданные коннектору смещения в Kafka с помощью обычных методов фиксации потребителей.

Отслеживание смещений самим фреймворком должно облегчить разработчикам задачу написания коннекторов и до определенной степени гарантировать согласованное поведение при использовании различных коннекторов.

Альтернативы Kafka Connect

Мы подробно рассмотрели API Kafka Connect. Хотя нам очень понравились их удобство и надежность, эти API — не единственный метод передачи данных в Kafka и из нее. Посмотрим, какие еще варианты существуют и как их обычно применяют.

Фреймворки ввода и обработки данных для других хранилищ

Хотя мы привыкли считать Kafka центром мироздания, кое-кто с нами не согласен. Некоторые разработчики ставят в основу своих архитектур данных такие системы, как Hadoop или Elasticsearch. В некоторых системах есть собственные утилиты ввода и обработки данных — Flume для Hadoop и Logstash или Fluentd для Elasticsearch. Мы рекомендуем использовать API Kafka Connect в случаях, когда Kafka является неотъемлемой частью архитектуры, а цель состоит в соединении большого числа источников и приемников. Если же вы создаете систему, ориентированную на Hadoop или Elasticsearch, а Kafka — лишь одно из многих средств ввода данных в нее, то имеет смысл воспользоваться Flume или Logstash.

ETL-утилиты на основе GUI

Множество классических систем наподобие Informatica, а также их альтернатив с открытым исходным кодом, например Talend и Pentaho, и даже более новых вариантов, таких как Apache NiFi и StreamSets, поддерживают использование Apache Kafka в качестве как источника данных, так и целевой системы. Если вы уже работаете с этими системами, например, везде применяете Pentaho, то зачем добавлять еще одну систему интеграции данных специально для Kafka? Это имеет смысл также, если вы создаете конвейеры ETL на основе GUI. Основным недостатком таких систем в том, что они обычно предназначены для запутанных технологических процессов и окажутся несколько тяжеловесным и сложным программным решением для случая, когда нужно всего лишь передать данные в Kafka и из нее. Мы убеждены, что основной целью интеграции данных должна быть добросовестная передача сообщений при любых условиях, в то время как большинство ETL-утилит лишь привносят ненужную сложность.

Мы рекомендуем рассматривать Kafka в качестве платформы, способной как на интеграцию данных (с помощью Connect) и приложений (с использованием производителей и потребителей), так и на потоковую обработку. Kafka может послужить прекрасной заменой для ETL-утилиты, которая занимается только интеграцией хранилищ данных.

Фреймворки потоковой обработки

Практически все фреймворки потоковой обработки могут читать данные из Kafka и записывать их в некоторые другие системы. Если ваша целевая система поддерживается и вы все равно собираетесь использовать конкретный фреймворк потоковой обработки для обработки поступающих из Kafka событий, имеет смысл использовать его же для интеграции данных. Это часто позволяет исключить из технологического процесса потоковой обработки необходимость хранить обработанные события в Kafka — можно просто читать их и записывать в другую систему. Правда, у этого решения есть и недостаток в виде усложнения отладки в случае, например, потери и повреждения сообщений.

Резюме

В этой главе разговор шел о применении Kafka для интеграции данных. Начав с оснований для использования Kafka при реализации данной задачи, мы рассмотрели несколько общих вопросов, относящихся к решениям для интеграции данных. Мы продемонстрировали, почему, с нашей точки зрения, Kafka и ее API Connect хорошо для этого подходят. Привели несколько примеров работы Kafka

Connect в различных сценариях, посмотрели, как работает Connect, после чего обсудили несколько его альтернатив.

На каком бы программном решении для интеграции данных вы ни остановились в итоге, важнейшим его свойством будет способность доставить все сообщения при любых сбойных ситуациях. Мы убеждены, что Kafka Connect исключительно надежен, поскольку основан на проверенных временем характеристиках надежности самой Kafka, но важно, чтобы вы всегда тестировали выбранную систему, как делаем мы. Убедитесь, что ваша система интеграции данных способна без потерь сообщений выдерживать остановку процессов, аварийные сбои машин, сетевые задержки и высокие нагрузки. В конце концов, по сути, единственная задача систем интеграции данных — доставить сообщения.

Важнейшим требованием при интеграции информационных систем обычно является надежность, однако это лишь одно из требований. При выборе информационной системы важно сначала просмотреть свои требования (см. примеры в разделе «Соображения по поводу создания конвейеров данных» ранее в этой главе), после чего убедиться, что система им удовлетворяет. Но этого недостаточно — вы должны хорошо разобраться в выбранном программном решении для интеграции данных, чтобы быть уверенными в том, что при его использовании ваши требования будут удовлетворяться. Того, что Kafka поддерживает строго однократную доставку, недостаточно — вы должны убедиться, что случайно не настроили ее так, что она окажется надежной лишь частично.

Зеркальное копирование между кластерами

В большей части данной книги мы обсуждаем настройку, поддержку и использование одного кластера Kafka. Однако существует несколько сценариев, при которых архитектура должна состоять из большого количества кластеров.

В некоторых случаях кластеры совершенно независимы — действуют в разных подразделениях организации или в различных сценариях. Иногда вследствие различий в соглашениях об уровне предоставления услуг (SLA) или нагрузках бывает сложно приспособить один кластер для обслуживания нескольких сценариев использования. В других случаях разнятся требования к информационной безопасности. Эти ситуации не представляют проблем — управлять несколькими отдельными кластерами, по сути, все равно что управлять несколькими экземплярами одного и того же кластера.

В других сценариях различные кластеры взаимосвязаны и администраторам приходится непрерывно копировать данные между ними. В большинстве баз данных непрерывное копирование данных между серверами баз данных называется *репликацией*. Но поскольку мы используем термин «репликация» для описания перемещения данных между узлами Kafka в пределах одного кластера, то для копирования данных между разными кластерами Kafka будем употреблять термин «*зеркальное копирование*» (mirroring). Встроенный в Apache Kafka репликатор данных между кластерами называется *MirrorMaker*.

В этой главе мы обсудим зеркальное копирование части или всех данных между кластерами. Начнем с обсуждения некоторых распределенных сценариев зеркального копирования данных между кластерами. Затем продемонстрируем несколько применяемых для реализации этих сценариев архитектур и обсудим плюсы и минусы каждой из них. После этого перейдем к разговору о самом MirrorMaker и его использовании. Мы дадим вам несколько советов по его эксплуатации, включая развертывание и настройку производительности. В завершение обсудим несколько альтернатив MirrorMaker.

Сценарии зеркального копирования данных между кластерами

Вот несколько ситуаций, в которых может оказаться полезно зеркальное копирование данных между кластерами.

- *Региональные и центральные кластеры.* В некоторых случаях один или несколько ЦОД компании находятся в различных географических регионах, городах или даже на разных континентах. В каждом ЦОД есть свой кластер Kafka. Части приложений достаточно для взаимодействия лишь с локальным кластером, а части требуются данные из нескольких ЦОД (в противном случае нам не нужны были бы программные решения для репликации данных между ЦОД). Такое требование выдвигается при множестве сценариев использования, но классический пример — компания, меняющая цены товаров в зависимости от их запасов и спроса. У нее могут быть ЦОД во всех городах, где она работает, в которые стекается информация о запасах товаров на местных складах и спросе на них с соответствующей корректировкой цен. Всю эту информацию необходимо зеркально копировать в центральный кластер, чтобы бизнес-аналитики могли сформировать отчеты о прибыли в масштабе всей компании.
- *Высокая доступность (HA) и аварийное восстановление (DR).* Приложения выполняются в одном кластере Kafka, для них не требуются данные из других мест, но вас беспокоит вероятность недоступности кластера по каким-либо причинам. Для обеспечения избыточности вам потребуется еще один кластер Kafka со всеми данными из первого кластера, чтобы в случае аварии можно было перенаправить приложения во второй кластер и продолжать работу как ни в чем не бывало.
- *Соответствие нормативным требованиям.* Компаниям, работающим в разных странах, может потребоваться использовать различные конфигурации и политики для соответствия законодательным и нормативным требованиям, существующим в каждой стране. Например, некоторые наборы данных могут храниться в отдельных кластерах со строгим контролем доступа, при этом подмножества данных реплицируются в другие кластеры с более широким доступом. Чтобы соответствовать нормативным требованиям, регулирующим период хранения данных в каждом регионе, наборы данных могут храниться в кластерах в разных регионах с различными конфигурациями.
- *Миграция в облако.* Сейчас многие компании пользуются как локальными ЦОД, так и услугами облачных провайдеров. Часто ради избыточности приложения выполняются в различных регионах провайдеров облачных услуг, а иногда задействуются и различные провайдеры. В подобных случаях часто

используется как минимум по одному кластеру Kafka в каждом локальном ЦОД и каждом регионе облака. Эти кластеры Kafka применяются приложениями из каждого ЦОД и региона для эффективной передачи данных между ЦОД. Например, если развернутое в облаке новое приложение требует каких-либо данных, хранимых в локальной базе данных и обновляемых работающими в локальном ЦОД приложениями, то можно воспользоваться Kafka Connect для сбора изменений в базе данных в локальном кластере Kafka с последующим их зеркальным копированием в облачный кластер Kafka, где их сможет использовать новое приложение. Это помогает контролировать затраты на трафик между ЦОД, а также повысить его безопасность.

- *Агрегирование данных из граничных кластеров.* Некоторые отрасли, включая розничную торговлю, телекоммуникации, транспорт и здравоохранение, генерируют данные с небольших устройств с ограниченными возможностями подключения. Агрегированный кластер с высокой доступностью можно использовать для поддержки аналитики и других сценариев использования данных из большого количества граничных кластеров. Это снижает требования к подключению, доступности и долговечности граничных кластеров с низкими ресурсами, например, при использовании технологии «Интернет вещей» (IoT). Агрегатный кластер с высокой доступностью обеспечивает непрерывность бизнеса даже при отключении граничных кластеров и упрощает разработку приложений, которым не нужно напрямую работать с большим количеством граничных кластеров с нестабильными сетями.

Мультикластерные архитектуры

Теперь, когда мы взглянули на несколько сценариев использования, для которых требуется больше одного кластера Kafka, пришло время описать несколько распространенных архитектурных паттернов, которые мы успешно применяли при реализации этих сценариев. Прежде чем заняться этими архитектурами, рассмотрим вкратце реалии взаимодействия между различными ЦОД. Если не понимать, что предлагаемые программные решения — компромиссы, учитывающие конкретные условия работы сети, может показаться, что они переусложнены.

Реалии взаимодействия между различными ЦОД

При обсуждении взаимодействия между различными ЦОД имеет смысл учитывать следующее:

- *высокую длительность задержек.* Задержки при взаимодействии между двумя кластерами Kafka возрастают пропорционально расстоянию и числу транзитных участков сети между ними;

- *ограниченную пропускную способность сети.* Пропускная способность глобальных сетей (Wide Area networks, WAN) обычно значительно ниже пропускной способности сети в пределах одного ЦОД, причем еще и меняется ежеминутно. Кроме того, более высокая длительность задержек усложняет использование всей доступной полосы пропускания;
- *повышенные по сравнению с работой в пределах одного ЦОД затраты.* Вне зависимости от того, где работает Kafka, локально или в облаке, взаимодействие между кластерами означает повышенные затраты. Это происходит отчасти из-за ограниченности пропускной способности сети и слишком высокой стоимости ее повышения, а также из-за запрашиваемой провайдерами стоимости передачи данных между различными ЦОД, регионами и облаками.

Брокеры и клиенты Apache Kafka проектируются, разрабатываются, тестируются и настраиваются в условиях одного ЦОД, причем в расчете на низкую задержку и широкую полосу пропускания между брокерами и клиентами. Это очевидно из значений времени ожидания по умолчанию и размеров различных буферов. Поэтому не рекомендуется (за исключением особых случаев, которые мы обсудим позже) устанавливать часть брокеров Kafka в одном ЦОД, а часть — в другом.

В большинстве случаев стоит избегать генерации данных для удаленного ЦОД, но если вам все же приходится это делать, рассчитывайте на более длительные задержки и, возможно, большее число сетевых ошибок. С ошибками можно справиться, увеличив число попыток повтора производителей, а с длительными задержками — увеличив размеры буферов для хранения записей между попытками отправки.

Если нам требуется репликация между кластерами и мы исключили возможность взаимодействия между брокерами, а также взаимодействия производителей с брокерами, то остается только взаимодействие брокеров с потребителями. Безусловно, это наиболее безопасный вид межкластерного взаимодействия, поскольку в случае невозможности чтения данных потребителем из-за нарушения связности сети записи будут в безопасности в брокерах Kafka до тех пор, пока связь не восстановится и потребители не смогут их прочитать. Риск случайной потери данных вследствие нарушения связности сети отсутствует. Тем не менее если в одном ЦОД есть несколько приложений, которым требуется читать данные из брокеров Kafka, находящихся в другом ЦОД, то из-за ограниченности полосы пропускания лучше установить кластер Kafka в каждом из ЦОД и выполнить зеркальное копирование данных между ними, вместо того чтобы допустить, чтобы несколько приложений потребляли одни и те же данные через глобальную сеть.

Мы еще поговорим подробнее о тонкой настройке Kafka для взаимодействия между ЦОД, но следующие принципы будут руководящими для всех дальнейших архитектур.

- Не менее одного кластера на ЦОД.
- Каждое событие реплицируется ровно один раз (повторы в случае ошибок запрещены) между каждой парой ЦОД.
- По возможности предпочитаем потребление данных из удаленного ЦОД отправке данных в него.

Архитектура с топологией типа «звезда»

Эта архитектура предназначена для ситуации с несколькими локальными и одним центральным кластером Kafka (рис. 10.1).

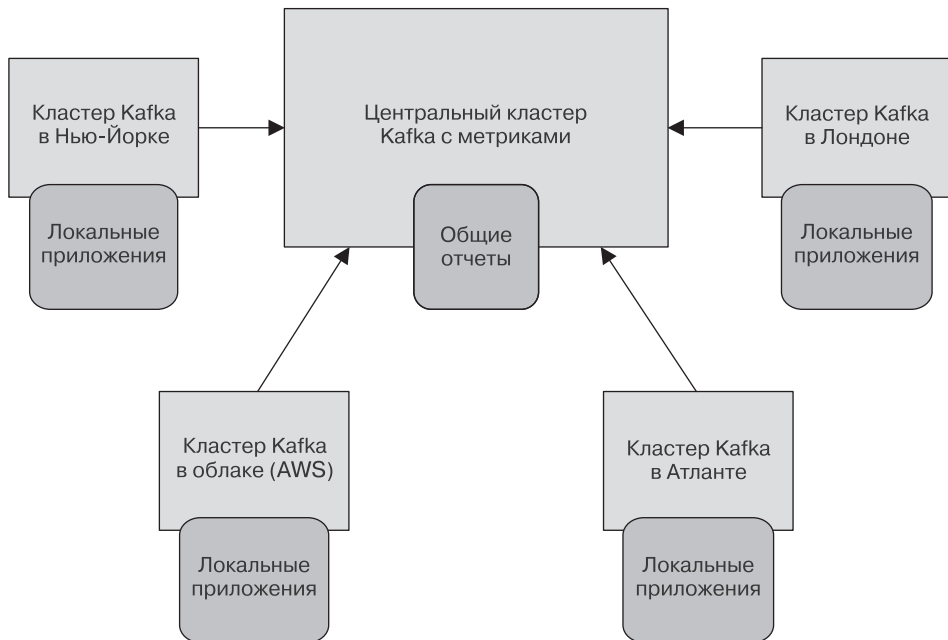


Рис. 10.1. Архитектура с топологией типа «звезда»

Существует также упрощенный вариант этой архитектуры с двумя кластерами — ведущим и ведомым (рис. 10.2).

Эта архитектура применяется, когда данные генерируются в нескольких ЦОД, причем части потребителей необходим доступ ко всему набору данных. Она также дает возможность приложениям в каждом ЦОД обрабатывать только локальные по отношению к нему данные. Но при этом не обеспечивается доступ ко всему набору данных из всех ЦОД.

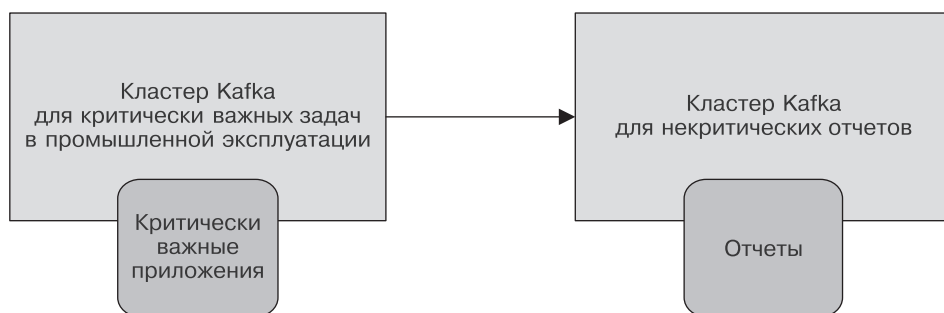


Рис. 10.2. Упрощенная версия архитектуры с топологией типа «звезда»

Основное достоинство этой архитектуры заключается в том, что данные всегда генерируются для локального ЦОД, а события из каждого ЦОД реплицируются однократно — в центральный ЦОД. Приложения, обрабатывающие данные из одного ЦОД, можно разместить в нем же. Приложения, которым необходимо обрабатывать данные из нескольких ЦОД, размещаются в центральном ЦОД, куда зеркально копируются все события. А поскольку репликация всегда происходит в одну сторону и каждый потребитель всегда читает данные из одного кластера, то такую архитектуру легко развертывать, настраивать и контролировать.

Основной недостаток этой архитектуры вытекает из ее достоинств и простоты. Процессоры одного регионального ЦОД не могут обращаться к данным другого. Чтобы лучше разобраться, почему это обстоятельство ограничивает наши возможности, рассмотрим пример данной архитектуры.

Допустим, у большого банка есть филиалы в нескольких городах и было решено хранить профили пользователей и историю их счетов в кластерах Kafka в каждом из городов. Вся информация реплицируется в центральный кластер, применяемый для целей бизнес-аналитики. При входе пользователей на сайт банка или посещении ими местного филиала события отправляются в локальный кластер и читаются тоже оттуда. Однако представьте себе, что пользователь посетил филиал банка в другом городе. Информации о нем в этом городе нет, так что филиалу придется связаться с удаленным кластером (не рекомендуется), иначе никакой возможности получить информацию о пользователе у него не будет (весьма неприятная ситуация). Поэтому применяться данный паттерн может только для тех частей набора данных, которые можно полностью разделить между региональными ЦОД.

При реализации такой архитектуры необходим как минимум один процесс зеркального копирования в центральном ЦОД для каждого регионального. Этот процесс будет потреблять данные из всех удаленных региональных кластеров

и отправлять ее в центральный кластер. Если в нескольких ЦОД существует один и тот же топик, можно записать все события из него в один топик с тем же названием на центральном кластере или заносить события из каждого ЦОД в отдельный топик.

Архитектура типа «активный — активный»

Эта архитектура реализуется, когда два или более ЦОД совместно используют часть данных или их все, причем каждый из них может как генерировать, так и потреблять события (рис. 10.3).

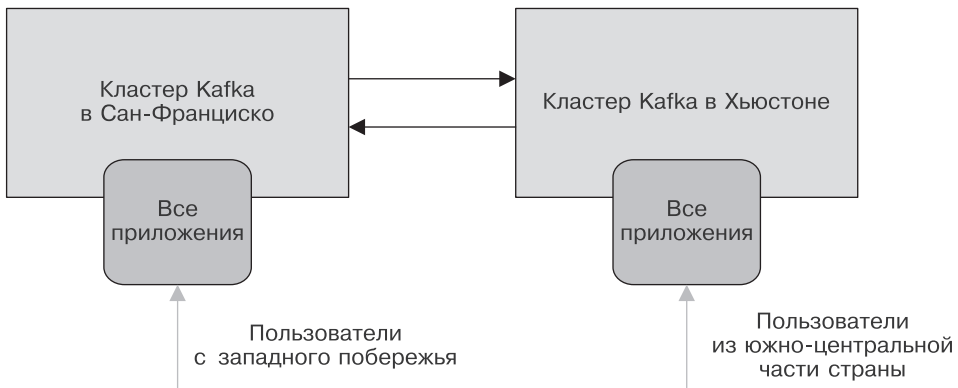


Рис. 10.3. Модель архитектуры типа «активный — активный»

Основное преимущество архитектуры — возможность обслуживания пользователей ближайшим ЦОД, что обычно повышает производительность, причем без потерь функциональности из-за ограниченной доступности данных, что мы наблюдали в архитектуре с топологией типа «звезда». Преимуществами являются также его избыточность и отказоустойчивость. Поскольку каждый из ЦОД обладает всей полнотой функциональности, в случае недоступности одного из них можно перенаправить пользователей на оставшийся. Для такого восстановления после сбоев нужно лишь сетевое перенаправление пользователей, обычно это самый простой и прозрачный тип восстановления.

Основной недостаток подобной архитектуры заключается в том, что весьма непросто избежать конфликтов в случае асинхронного чтения и обновления данных в нескольких местах. Существуют также технические сложности зеркального копирования событий. Например, как гарантировать, что одно и то же событие не будет бесконечно зеркально копироваться туда и обратно? Но еще более значимы сложности при поддержании согласованности данных между

этими двумя ЦОД. Вот несколько примеров трудностей, с которыми вам придется столкнуться.

- При отправке пользователем события в один ЦОД и чтении событий из другого существует вероятность того, что записанное им событие еще не попало во второй ЦОД. С точки зрения пользователя, это будет выглядеть так, будто он добавил книгу в свой список желаемых, заглянул в него, а книги там нет. Поэтому при использовании такой архитектуры разработчики обычно стараются привязывать пользователей к конкретному ЦОД и гарантировать, что они всегда задействуют один кластер, за исключением случаев их подключения из удаленной точки или недоступности ЦОД.
- Событие из одного ЦОД гласит, что пользователь заказал книгу А, а относящееся примерно к тому же моменту времени событие из другого ЦОД — что он же заказал книгу Б. После зеркального копирования оба события окажутся в обоих ЦОД, так что в каждом из них будут находиться два конфликтующих события. Приложениям в каждом из ЦОД необходимо знать, что делать в такой ситуации: следует ли выбрать одно из событий как правильное? Если да, то должны существовать согласованные правила выбора события, чтобы приложения из обоих ЦОД приняли одинаковое решение. Или следует счесть, что оба события правильные, просто отправить пользователю две книги, а дальше пусть другой отдел разбирается с возвратом? Amazon обычно так и решает подобные проблемы, но при торговле акциями это невозможно. При каждом сценарии применения есть свой способ минимизации конфликтов и обработки возникающих конфликтных ситуаций. Важно лишь не забывать, что при такой архитектуре конфликты *неизбежны* и их придется как-то решать.

Если вам удастся решить проблемы с асинхронными операциями чтения одних и тех же данных из нескольких мест и записи в них, то данная архитектура — очень неплохой вариант. Это наиболее масштабируемая, отказоустойчивая, гибкая и затратноэффективная архитектура из известных нам. Так что имеет смысл поискать возможности избежать циклов репликации, ограничить пользователей в основном одним ЦОД и найти способ решения конфликтов при их возникновении.

Часть проблемы зеркального копирования по типу «активный — активный», особенно в случае более чем двух ЦОД, состоит в том, что вам понадобятся задачи зеркального копирования для каждой пары ЦОД и каждого направления. Многие инструменты зеркального копирования в наши дни могут совместно использовать процессы, например использовать один и тот же процесс для всех зеркальных копирований на целевой кластер.

Кроме того, необходимо избегать заикливания — бесконечного зеркального копирования одного и того же события туда и обратно. Для этого можно вы-

делить для каждого логического топика отдельный топик в каждом ЦОД и не реплицировать топика, производителем которых является удаленный ЦОД. Например, логический топик `users` будет называться `SF.users` в одном ЦОД и `NYC.users` — в другом. При зеркальном копировании `SF.users` будет скопирован из Сан-Франциско (SF) в Нью-Йорк (NYC), а `NYC.users` — из Нью-Йорка в Сан-Франциско. В результате все события будут зеркально копироваться однократно, но в каждом из этих ЦОД будут в наличии оба топика, `SF.users` и `NYC.users`, то есть информация по всем пользователям. Потребителям придется читать данные из `*.users`, чтобы получить события по всем пользователям. Эту схему можно рассматривать как отдельное пространство имен для каждого ЦОД, включающее все его топика. В нашем примере речь идет о пространствах имен NYC и SF. Некоторые инструменты зеркального копирования, например MirrorMaker, предотвращают циклы репликации, используя аналогичное соглашение об именах.

Заголовки записей, введенные в Apache в версии 0.11.0, позволяют пометить события с указанием ЦОД, из которого они происходят. Информация заголовка может быть использована также для того, чтобы избежать бесконечных циклов зеркального копирования и организовывать обработку событий из разных ЦОД по отдельности. Эту функциональность можно реализовать и с помощью структурированного формата данных для значений записей (наш излюбленный пример — Avro), включая теги и заголовки в самое событие. Однако это потребует дополнительных усилий при зеркальном копировании, поскольку ни одна из существующих утилит зеркального копирования не будет поддерживать пользовательский формат заголовка.

Архитектура типа «активный — резервный»

В некоторых случаях единственное требование к мультикластерной архитектуре — наличие сценария действий в случае аварийного сбоя. Допустим, у вас есть два кластера в одном ЦОД. Один кластер используется для всех приложений, а второй, содержащий практически все события из первого, предназначен для применения при полной недоступности первого. Или, возможно, для вас важна отказоустойчивость в географическом смысле. Работа всего бизнеса осуществляется из ЦОД в Калифорнии, но на случай землетрясения вы хотели бы иметь второй ЦОД в Техасе, большую часть времени практически не работающий. В техасском ЦОД, вероятно, будут бездействующие («холодные») копии всех приложений, которые администраторы могут запустить в случае чрезвычайной ситуации и которые будут применять второй кластер (рис. 10.4). Зачастую это требование закона, а не реальный план действий, но все равно лучше быть готовыми к подобной ситуации.

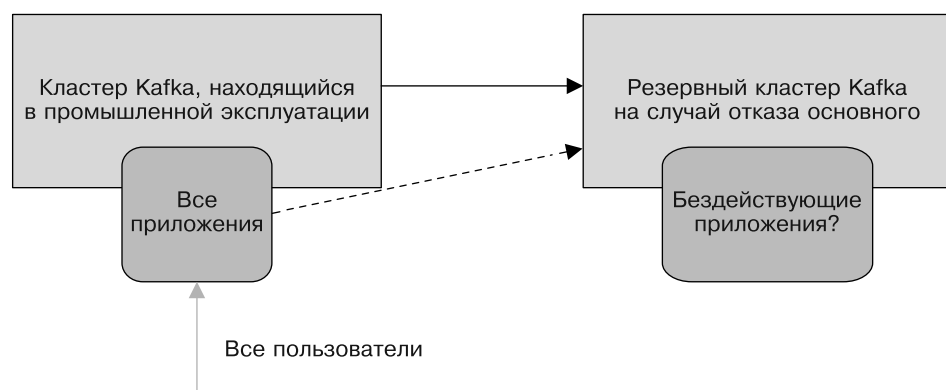


Рис. 10.4. Архитектура типа «активный — резервный»

Преимущества подобной схемы — простота настройки и возможность использования практически в любом сценарии. Нужно просто установить второй кластер и настроить процесс зеркального копирования, который перебрасывал бы данные из одного кластера в другой. И никаких проблем с доступом к данным, обработкой конфликтов и другими архитектурными трудностями.

Недостатки же таковы: простаивает отличный кластер, да и переключение с одного кластера Kafka на другой на практике не такая уж простая вещь. Определяющий фактор — то, что в настоящий момент невозможно переключиться с одного кластера Kafka на другой без потери данных или дублирования событий. А часто и того и другого. Можно минимизировать эти неприятные эффекты, но не исключить их полностью.

Очевидно, что кластер, не выполняющий никакой полезной работы, а лишь простаивающий в ожидании аварийного сбоя, — пустая трата ресурсов. Поскольку аварийные сбои — событие редкое (или по крайней мере должно быть таковым), большую часть времени этот кластер машин вообще ничего не будет делать. Некоторые компании пытаются справиться с этой проблемой за счет так называемого DR-кластера (disaster recovery, кластер для переключения в случае аварийного сбоя), намного меньшего, чем кластер для промышленной эксплуатации. Но это довольно рискованное решение, ведь нет уверенности, что такой минималистичный кластер сможет справиться с реальной нагрузкой при нештатной ситуации. Другие компании предпочитают, чтобы в обычное время, когда нет нештатных ситуаций, резервный кластер приносил пользу, и делегируют ему рабочую нагрузку «только для чтения», то есть фактически используют архитектуру с топологией типа «звезда», где у «звезды» только один луч.

Более насущный вопрос: а как, собственно, переключиться на DR-кластер в Apache Kafka?

Какой бы из существующих методов вы ни выбрали, команда специалистов по обеспечению надежности должна регулярно практиковаться в его выполнении. Прекрасно работающий сегодня план может перестать действовать после обновления, а существующий инструментарий может оказаться устаревшим при новых сценариях применения. Тренировки по восстановлению после сбоя следует проводить не реже чем раз в квартал. А хорошие команды специалистов по обеспечению надежности делают это гораздо чаще. Знаменитая утилита Chaos Monkey компании Netflix, вызывающая аварийные сбои в случайные моменты времени, доводит этот тезис до абсолюта: любой день может оказаться днем тренировки восстановления после сбоя.

Посмотрим теперь, что включает в себя восстановление после сбоя.

Планирование восстановления после аварийного сбоя

При планировании восстановления после аварийного сбоя важно учитывать два ключевых показателя. Цель времени восстановления (RTO) определяет максимальное количество времени, в течение которого все сервисы должны возобновить работу после аварии. Цель точки восстановления (RPO) определяет максимальный промежуток времени, в течение которого данные могут быть потеряны в результате аварийного сбоя. Чем ниже RTO, тем важнее избегать ручных процессов и перезапусков приложений, поскольку очень низкий RTO может быть достигнут только при автоматизированном восстановлении после сбоя. Низкий RPO требует зеркального копирования в реальном времени с низкими задержками, а $RPO = 0$ требует синхронной репликации.

Потери данных и несогласованность при внеплановом восстановлении после сбоя

Поскольку все программные решения для зеркального копирования Kafka асинхронны (мы обсудим синхронные решения в следующем разделе), то в DR-кластере не будет последних сообщений из основного кластера. Следует всегда контролировать отставание DR-кластера от основного и не позволять ему отставать слишком сильно. Но при перегруженной системе следует ожидать, что DR-кластер будет отставать от основного на несколько сотен или даже тысяч сообщений. Если ваш кластер Kafka обрабатывает 1 млн сообщений в секунду, а отставание DR-кластера от основного равно 5 мс, то DR-кластер даже при наилучшем сценарии будет отставать на 5000 сообщений. Так что будьте готовы к потере данных в случае незапланированного переключения. Если же переключение запланировано, можно остановить основной кластер и подождать, пока в ходе зеркального копирования будут скопированы оставшиеся сообщения, прежде чем переключать приложения на DR-кластер, исключив таким образом

потерю данных. На случай незапланированного переключения учтите, что решения для зеркального копирования сейчас не поддерживают транзакции, так что если часть событий из нескольких топиков связаны друг с другом (например, продажи и позиции заказа), то часть событий может поступить в DR-кластер вовремя (для переключения), а часть — нет. Приложения должны уметь обрабатывать позиции заказов без соответствующих продаж после переключения на DR-кластер.

Начальное смещение для приложений после аварийного переключения

Одна из наиболее сложных задач при переключении на другой кластер — гарантировать, что приложения начнут потреблять данные с нужного места. Существует несколько возможных решений этой проблемы. Некоторые просты, но могут вызвать дополнительную потерю данных или обработку дубликатов, другие сложнее, но позволяют минимизировать потери данных и повторную обработку. Рассмотрим некоторые из них.

- *Автоматический сброс смещения.* У потребителей Apache Kafka имеется параметр, определяющий их поведение при отсутствии ранее зафиксированных смещений, — приступить к чтению с начала или с конца раздела. Если не производится зеркальное копирование этих смещений в рамках плана DR, необходимо выбрать один из следующих параметров: либо читать с начала имеющихся данных и обрабатывать большое количество дубликатов, либо перескочить в конец раздела, пропустив при этом некое (надеемся, небольшое) число событий. Если ваше приложение обрабатывает дубликаты без проблем или отсутствие некоторых данных не имеет большого значения, это, безусловно, проще всего. Переход к концу топика о восстановлении после сбоя является популярным методом обработки восстановления после сбоя из-за своей простоты.
- *Репликация топика для смещений.* Потребители версий Kafka 0.9.0 и выше будут фиксировать смещения в специальный топик `__consumer_offsets`. Если зеркально копировать его в DR-кластер, то потребители смогут начать читать данные из DR-кластера с тех смещений, на которых они закончили чтение. Все просто, но не без некоторых нюансов.

Во-первых, нет никаких гарантий, что смещения в основном и дополнительном кластерах будут совпадать. Допустим, что данные хранятся в основном кластере только три дня и вы начинаете зеркально копировать топик через неделю после его создания. В этом случае первым смещением в основном кластере будет, например, смещение `57000000` (старые события за первые четыре дня уже удалены), а первым смещением в DR-кластере — `0`. Так что

попытка потребителя прочитать из DR-кластера смещение 57000003, потому что именно его он должен читать следующим, приведет к сбою.

Во-вторых, даже если начать зеркально копировать сразу же после создания топика, так что как топики основного кластера, так и DR-топики будут начинаться со смещения 0, повторы попыток отправки производителем могут привести к расхождению смещений. В конце этой главы мы обсудим альтернативный вариант зеркального копирования, которое сохраняет смещения при переходе от основного к DR-кластеру.

В-третьих, даже если смещения сохраняются в точности, из-за отставания DR-кластера от основного кластера и из-за того, что в настоящий момент решения по зеркальному копированию не поддерживают транзакции, информация о фиксации потребителем Kafka смещения может прибыть раньше или позже записи с данным смещением. При переключении после сбоя потребитель может обнаружить смещения без соответствующих им записей. Или может оказаться, что последнее зафиксированное смещение в DR старше, чем последнее зафиксированное смещение основного кластера (рис. 10.5).

Необходимо смириться с некоторым количеством дубликатов, если последнее зафиксированное смещение в DR-кластере старше, чем зафиксированное в основном кластере, или если смещения в записях DR-кластера опережают смещение основного из-за повторов попыток отправки. Вам придется также решить, что делать со случаями, когда для последнего зафиксированного смещения в DR-кластере не существует соответствующей записи, — начать обработку с начала топика или перескочить к его концу?

Как видите, у данного подхода есть ограничения. Тем не менее этот вариант позволяет переключиться на другой DR-кластер при меньшем числе дубликатов или пропущенных событий по сравнению с другими подходами, оставаясь при этом простым в реализации.

- *Переключение по времени.* Начиная с версии 0.10.0 и выше каждое сообщение включает метку даты/времени, соответствующую времени отправки сообщения в Kafka. Начиная с версии 0.10.1.0 и выше, брокеры включают индекс и API для поиска смещения по метке даты/времени. Таким образом, если вы переключились на DR-кластер и знаете, что проблемы начались в 4:05, то можете сказать потребителям, что нужно начинать обработку данных с 4:03. Конечно, будет несколько дубликатов, относящихся к этим двум минутам, но другие варианты еще хуже. К тому же такое поведение системы гораздо легче объяснить сотрудникам: «Мы откатились к времени 4:03» звучит намного лучше, чем «Мы откатились к, возможно, последним зафиксированным смещениям». Так что это зачастую неплохой компромисс. Единственный вопрос: как сообщить потребителям о необходимости начинать обработку данных с 4:03?

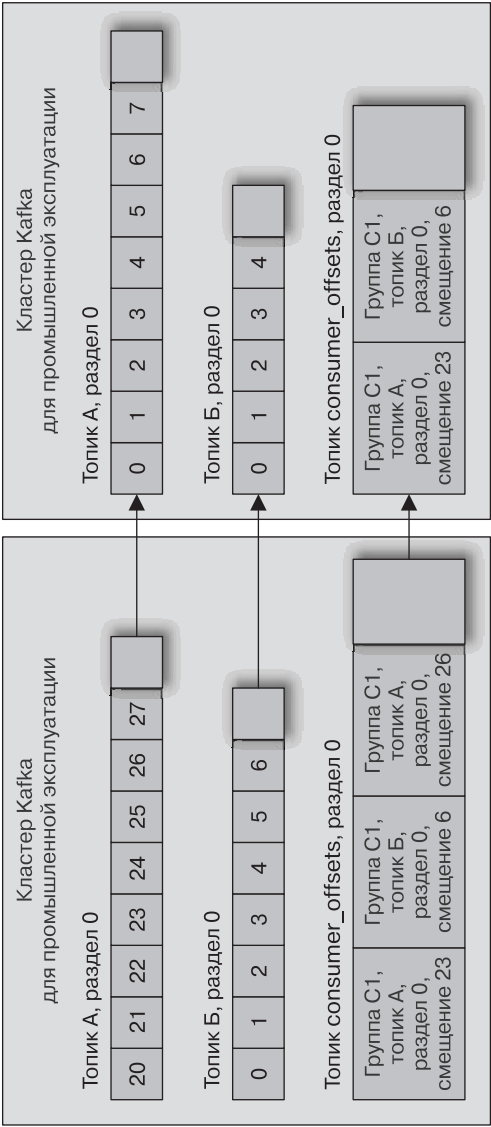


Рис. 10.5. Переключение при сбое приводит к появлению зафиксированных смещений без соответствующих записей

Один из вариантов — добавить настраиваемый пользователем параметр, который задавал бы начальное время для приложения. При задании этого параметра приложение может воспользоваться новыми API для извлечения смещений по времени, перейти к этому моменту и начать потребление данных с нужной точки, фиксируя смещения обычным образом.

Этот вариант хорош, если сразу писать все приложения таким образом. Но что, если вы уже создали приложение без него? Apache Kafka предоставляет инструмент `kafka-consumer-groups` для сброса смещений на основе ряда опций, включая сброс на основе временных меток, добавленный в версии 0.11.0. При запуске этой утилиты необходимо приостановить работу группы потребителей и возобновить ее непосредственно после завершения работы утилиты. Например, следующая команда сбрасывает смещения потребителей для всех топиков, принадлежащих определенной группе, на определенное время:

```
bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --reset-
offsets --all-topics --group my-group --to-datetime
2021-03-31T04:03:00.000 --execute
```

Этот вариант рекомендуется для развертываний, в которых необходимо гарантировать определенный уровень надежности при отказоустойчивости.

- *Соответствия смещений.* Одна из главных проблем при зеркальном копировании топика для смещений состоит в возможных расхождениях смещений в основном и DR-кластере. Из-за этого в прошлом некоторые компании предпочитали использовать внешнее хранилище данных, например Apache Cassandra, для хранения соответствий смещений одного кластера смещениям другого. При генерации события для отправки в DR-кластер оба смещения отправляются во внешнее хранилище данных инструментом зеркального копирования при расхождении смещений. В настоящее время решения для зеркального копирования, включая MirrorMaker, используют топик Kafka для хранения метаданных перевода смещений. Смещения сохраняются в случае изменения различия между ними. Например, если смещение 495 в основном кластере соответствует смещению 500 в DR-кластере, мы записываем во внешнее хранилище или в топик перевода со смещением пару (495, 500). Если в дальнейшем разница меняется и смещение 596 соответствует уже смещению 600, то записываем новое соответствие (596, 600). Нет необходимости хранить все промежуточные соответствия смещений между 495 и 596, мы просто предполагаем, что разница остается неизменной и смещение 550 в основном кластере будет соответствовать смещению 555 в DR-кластере. А в случае переключения на резервный кластер вместо задания соответствий меток даты/времени (всегда немного неточных) используются соответствия смещений основного кластера смещениям DR-кластера. Один из двух описанных ранее методов может быть реализован, чтобы потребители начинали

чение с новых смещений из карты соответствий. При этом сохраняется проблема с фиксацией смещений, прибывших раньше самих записей, а также смещений, которые не были вовремя зеркально скопированы в DR-кластер, но часть проблемных случаев все равно будет охвачена.

После аварийного переключения

Предположим, что аварийное переключение было успешным. Все прекрасно работает в DR-кластере. Теперь нам нужно что-то сделать с основным кластером. Например, создать из него новый DR-кластер.

Очень заманчиво просто изменить направление процесса зеркального копирования и начать зеркально копировать данные из нового основного кластера в старый. Однако при этом возникает два важных вопроса.

- Как узнать, с какого места начинать зеркальное копирование? Нам придется решить ту же проблему, которую мы описали ранее, применительно ко всем потребителям. И не забывайте, что для всех наших решений существуют сценарии использования, при которых возникают дубликаты или пропущенные данные — иногда и то и другое.
- Кроме того, по обсуждавшимся ранее причинам вполне возможно, что в исходном основном кластере содержатся события, отсутствующие в DR-кластере. Если начать зеркальное копирование новых данных в обратную сторону, эти дополнительные события никуда не денутся и два кластера окажутся несогласованными.

По этой причине для сценариев, в которых гарантии согласованности и упорядоченности являются критически важными, простейшим решением будет сначала почистить исходный кластер — удалить все данные и зафиксированные смещения, после чего начать зеркальное копирование из нового основного кластера в новый DR-кластер. Благодаря этому вы получите чистый DR-кластер, идентичный новому основному.

Несколько слов об обнаружении кластеров

При планировании резервного кластера очень важно учесть, что в случае аварийного переключения ваши приложения должны будут откуда-то узнать, как начать взаимодействие с резервным кластером. Если вы «зашили» имена хостов брокеров основного кластера в свойства производителей и потребителей, то это будет нелегко. Большинство компаний для простоты создают имя DNS, которое в обычных обстоятельствах указывает на брокеры основного кластера. В случае аварии можно сделать так, чтобы оно ссылалось на резервный кластер. Сервис обнаружения (DNS или какой-то другой) не должен включать все брокеры — кли-

ентам Kafka достаточно успешно обратиться к одному брокеру, чтобы получить метаданные кластера и обнаружить все остальные. Так что обычно достаточно включить всего три брокера. Независимо от метода обнаружения большинство сценариев восстановления после сбоя требуют перезапуска приложений-потребителей после переключения, чтобы они смогли обнаружить новые смещения, с которых нужно начинать чтение. Для автоматического восстановления после отказа без перезапуска приложения, чтобы достичь очень низкого RTO, логика аварийного восстановления должна быть встроена в клиентские приложения.

Эластичные кластеры

Архитектуры типа «активный — резервный» служат для защиты бизнеса при сбое кластера Kafka за счет переключения приложения на взаимодействие с другим кластером в случае отказа кластера. *Эластичные кластеры* (stretch clusters) предназначены для предотвращения отказа кластера Kafka в случае перебоев в работе ЦОД. Это достигается за счет установки одного кластера Kafka в нескольких ЦОД.

Эластичные кластеры принципиально отличаются от других мультикластерных архитектур. Начать нужно с того, что они не мультикластерные — речь идет об одном кластере. В результате оказывается не нужен процесс зеркального копирования для синхронизации двух кластеров. Для обеспечения согласованности всех брокеров кластера используется обычный механизм репликации Kafka. Эта схема может включать синхронную репликацию. Обычно производители после успешной записи сообщения в Kafka получают подтверждение от брокера Kafka. В случае же эластичного кластера можно настроить все так, чтобы подтверждение отправлялось после успешной записи сообщения в брокеры Kafka в двух ЦОД. Для этого потребуются соответствующие описания стоек, чтобы у каждого раздела были реплики в нескольких ЦОД. Придется также воспользоваться параметрами `min.insync.replicas` и `acks=all`, чтобы гарантировать подтверждение каждой записи как минимум из двух ЦОД. Начиная с версии 2.4.0, брокеры также можно настроить таким образом, чтобы потребители могли получать данные из ближайшей реплики, используя определения стоек. Брокеры сопоставляют свою стойку со стойкой потребителя, чтобы найти локальную реплику, которая является наиболее актуальной, и возвращаются к лидеру, если подходящая локальная реплика недоступна. Потребители, получающие данные от последователей в своем локальном ЦОД, достигают более высокой пропускной способности, более низкой задержки и более низкой стоимости за счет сокращения трафика между ЦОД.

Преимущества такой архитектуры заключаются в синхронной репликации — для некоторых коммерческих предприятий необходимо, чтобы их DR-сайт всегда был на 100 % согласован с основным сайтом. Зачастую это нужно согласно

законодательству, так что компания вынуждена соблюдать это требование во всех своих хранилищах данных, включая Kafka. Еще одно преимущество — используются оба ЦОД и все брокеры кластера. Ничего не простаивает, как в архитектурах типа «активный — резервный».

Однако эта архитектура защищает от ограниченного списка типов аварий — только от отказов ЦОД, но не от отказов приложений или Kafka. Ограничена также эксплуатационная сложность. Кроме того, архитектура требует физической инфраструктуры, обеспечить которую под силу не всем компаниям.

Использовать эту архитектуру имеет смысл, если у вас есть возможность установить Kafka по крайней мере в трех ЦОД с высокой пропускной способностью и низкой сетевой задержкой взаимодействия между ними. Достичь этого можно, если вашей компании принадлежат три здания на одной улице или (чаще всего) три зоны доступности в пределах одного региона облачного провайдера.

Причина, по которой нужны три ЦОД, заключается в том, что для ZooKeeper требуется, чтобы в кластере было нечетное число узлов: он остается доступным, пока доступно большинство из них. При двух ЦОД и нечетном числе узлов в одном из ЦОД будет больше узлов, так что при его недоступности окажется недоступен и ZooKeeper, а значит, и Kafka. При трех ЦОД можно легко распределить узлы так, что ни у одного из них не будет большинства. И если один из ЦОД станет недоступен, в других двух останется большинство, значит, останется доступен кластер ZooKeeper. А следовательно, и кластер Kafka.



Архитектура 2,5 DC

Популярной моделью для эластичных кластеров является архитектура 2,5 DC (datacenter — ЦОД), в которой Kafka и ZooKeeper работают в двух ЦОД, а третий 0,5 ЦОД с одним узлом ZooKeeper обеспечивает кворум в случае сбоя одного из ЦОД.

Работу ZooKeeper и Kafka в двух ЦОД можно обеспечить с помощью таких настроек групп ZooKeeper, которые позволяли бы выполнять ручное аварийное переключение между этими ЦОД. Однако такая схема реализуется редко.

Утилита MirrorMaker (Apache Kafka)

В Apache Kafka включена утилита под названием MirrorMaker для зеркального копирования данных между двумя ЦОД. Ее ранние версии использовали набор потребителей, которые были членами одной группы, для того чтобы читать данные из набора исходных топиков и общего производителя Kafka в каждом процессе MirrorMaker для отправки этих событий в целевой кластер. Хотя

в некоторых сценариях этого было достаточно для зеркального копирования данных между кластерами, все же там имелся ряд проблем, в частности скачки задержки при изменении конфигурации и добавлении новых топиков, что приводило к перебалансировке в режиме полной остановки (stop-the-world). Утилита MirrorMaker версии 2.0 — это решение нового поколения для мультикластерного зеркального копирования Apache Kafka, основанное на фреймворке Kafka Connect и устраняющее многие недостатки своего предшественника. Сложные топологии могут быть легко настроены для поддержки широкого спектра сценариев использования, таких как аварийное восстановление, резервное копирование, миграция и агрегация данных.



Еще о MirrorMaker

Утилита MirrorMaker представляется очень простой, но в силу нашего стремления к эффективности и максимальной приближенности к строго однократной доставке ее корректная реализация — задача довольно хитрая. Утилита MirrorMaker многократно переписывалась. Приведенные в следующих разделах описание и различные подробности относятся к MirrorMaker 2.0, которая была представлена в версии 2.4.0.

Утилита MirrorMaker использует коннектор источника для получения данных из другого кластера Kafka, а не из базы данных. Использование фреймворка Kafka Connect сводит к минимуму накладные расходы на администрирование для загруженных ИТ-отделов предприятий. Если вы помните архитектуру Kafka Connect из главы 9, то знаете, что все коннекторы распределяют работу между настраиваемым количеством задач. В MirrorMaker каждая задача представляет собой пару из потребителя и производителя. Фреймворк Connect назначает эти задачи различным рабочим узлам Connect по мере необходимости — таким образом, у вас может быть несколько задач на одном сервере или задачи будут распределены по нескольким серверам. Это заменяет ручную работу по определению того, сколько потоков MirrorMaker должно запускаться на каждом экземпляре и сколько экземпляров на каждой машине. Connect также имеет REST API для централизованного управления конфигурацией коннекторов и задач. Если предположить, что большинство развертываний Kafka включают Kafka Connect по другим причинам (отправка событий изменения базы данных в Kafka — очень популярный сценарий использования), то, запустив MirrorMaker внутри Connect, мы сможем сократить количество кластеров, которыми нам нужно управлять.

MirrorMaker равномерно распределяет разделы между задачами, не используя протокол управления группами потребителей Kafka, чтобы избежать скачков задержки из-за перебалансировки при добавлении новых топиков или разделов. События из каждого раздела в исходном кластере зеркально отображаются на тот же раздел в целевом кластере, сохраняя семантическое разделение и поддерживая упорядоченность событий для каждого раздела. Если в исходные топик

добавляются новые разделы, они автоматически создаются в целевом топике. Помимо репликации данных, MirrorMaker поддерживает также миграцию пользовательских смещений, конфигурацию топиков и механизмов управления доступом к топикам, что делает его полноценным решением для зеркального копирования для мультикластерных развертываний. Поток репликации определяет конфигурацию направленного потока от исходного кластера к целевому. В MirrorMaker можно определить несколько потоков репликации для создания сложных топологий, включая архитектурные паттерны, которые мы обсуждали ранее, такие как архитектуры с топологиями типа «звезда», «активный — резервный» и «активный — активный». На рис. 10.6 показано использование MirrorMaker в архитектуре с топологией типа «активный — резервный».

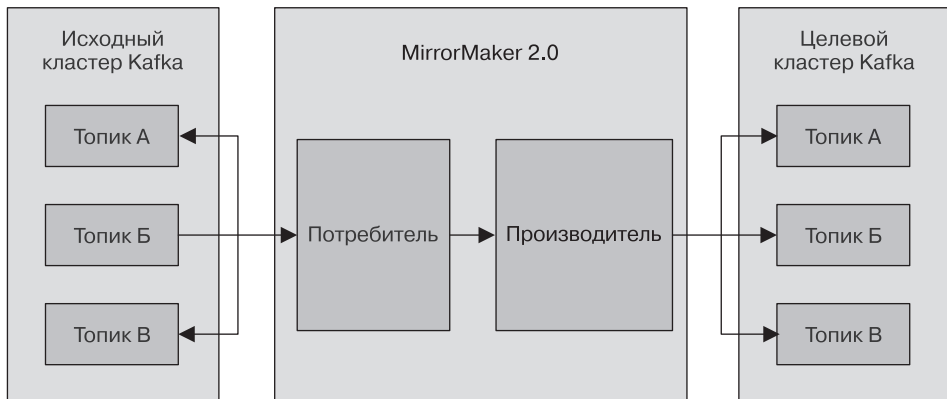


Рис. 10.6. Процесс MirrorMaker в Kafka

Настройка MirrorMaker

У MirrorMaker очень широкие возможности настройки. В дополнение к настройке кластера для определения топологии, Kafka Connect и настройке коннектора можно настроить все параметры конфигурации базового производителя, потребителей и клиента администратора, используемого MirrorMaker. Мы продемонстрируем несколько примеров и отметим некоторые наиболее важные параметры конфигурации, но всестороннее описание MirrorMaker выходит за рамки данной книги.

С учетом сказанного взглянем на пример использования MirrorMaker. Следующая команда запускает MirrorMaker с параметрами конфигурации, указанными в файле свойств:

```
bin/connect-mirror-maker.sh etc/kafka/connect-mirror-maker.properties
```

Рассмотрим некоторые параметры конфигурации MirrorMaker.

- *Поток репликации.* В следующем примере показаны параметры конфигурации для настройки потока репликации в режиме «активный — резервный» между двумя ЦОД в Нью-Йорке и Лондоне:

```
clusters = NYC, LON
NYC.bootstrap.servers = kafka.nyc.example.com:9092
LON.bootstrap.servers = kafka.lon.example.com:9092
NYC->LON.enabled = true
NYC->LON.topics = .*
```

- ❶ Определите псевдонимы для кластеров, используемых в потоках репликации.
 - ❷ Настройте начальную загрузку для каждого кластера, применяя псевдоним кластера в качестве префикса.
 - ❸ Включите поток репликации между парой кластеров, используя префикс `source->target`. Все параметры конфигурации для этого потока используют один и тот же префикс.
 - ❹ Настройте топики, которые будут зеркально копироваться для этого потока репликации.
- *Зеркальное копирование топиков.* Как показано в примере, для каждого потока репликации можно задать регулярное выражение для названий топиков, которые будут зеркально копироваться. В данном примере мы реплицируем их все, но часто имеет смысл использовать регулярное выражение вроде `prod.*`, чтобы не реплицировать тестовые топики. Отдельный список исключения топиков, содержащий их названия или шаблоны типа `test.*`, также может быть указан для исключения топиков, которые не требуют зеркального копирования. По умолчанию к названиям целевых топиков автоматически добавляется псевдоним исходного кластера. Например, в архитектуре типа «активный — активный» MirrorMaker, реплицирующий топики из нью-йоркского ЦОД в лондонский ЦОД, будет зеркально копировать заказы топиков из Нью-Йорка в топик `NYC.orders` в Лондоне. Эта стратегия именования по умолчанию предотвращает циклы репликации, приводящие к бесконечному зеркальному копированию событий между двумя кластерами в режиме «активный — активный», если топики зеркально копируются из Нью-Йорка в Лондон и из Лондона в Нью-Йорк. Различие между локальными и удаленными топиками поддерживает также случаи применения агрегации, поскольку потребители могут выбирать схемы подписки для потребления данных, полученных только из локального региона, или подписываться на топики из всех регионов для получения полного набора данных.

MirrorMaker периодически проверяет наличие новых топиков в исходном кластере и начинает их автоматическое зеркальное копирование, если они

соответствуют настроенным шаблонам. Если в исходный топик добавляется больше разделов, то такое же количество разделов автоматически добавляется в целевой топик, обеспечивая появление в целевом топике событий, имеющихся в исходном топике, в тех же разделах в том же порядке.

- *Миграция смещений потребителей.* Утилита MirrorMaker содержит класс `RemoteClusterUtils`, позволяющий потребителям обращаться к последнему контрольному смещению в DR-кластере с трансляцией смещений при аварийном сбое первичного кластера. В версии 2.7.0 была добавлена поддержка периодической миграции потребительских смещений для автоматической фиксации переведенных смещений в целевой топик `__consumer_offsets`, чтобы потребители, переходящие в DR-кластер, могли начать работу с того места, где они остановились в первичном кластере, без потери данных и с минимальной обработкой дубликатов. Можно настроить группы потребителей, для которых переносятся смещения, а для дополнительной защиты MirrorMaker не перезаписывает смещения, если потребители в целевом кластере активно используют целевую группу потребителей, что позволяет избежать любых случайных конфликтов.
- *Миграция конфигурации топика и механизма управления доступом.* В дополнение к зеркальному копированию записей данных MirrorMaker может быть настроен на зеркальное копирование конфигурации топиков и механизмов управления доступом (ACL) топиков, чтобы сохранить то же поведение для зеркально копируемого топика. Конфигурация по умолчанию обеспечивает такую миграцию с разумными периодическими интервалами обновления, чего может быть достаточно в большинстве случаев. Большинство параметров конфигурации топика из источника применяются к целевому топiku, но некоторые из них, например `min.insync.replicas`, по умолчанию не применяются. Список исключенных конфигураций можно настроить.

Переносятся только буквальные механизмы управления доступом к топикам, которые соответствуют зеркально копируемым топикам, поэтому, если вы используете механизмы управления доступом с префиксом или подстановочным знаком или альтернативные механизмы авторизации, вам нужно будет явно настроить их в целевом кластере. Механизмы управления доступом для `Topic:Write` не переносятся, чтобы гарантировать, что только MirrorMaker разрешено записывать в целевой топик. Соответствующий доступ должен быть явно предоставлен во время восстановления после отказа, чтобы обеспечить работу приложений с вторичным кластером.

- *Задачи коннектора.* Параметр конфигурации `tasks.max` ограничивает максимальное количество задач, которые может использовать коннектор, связанный с MirrorMaker. По умолчанию берется значение 1, но рекомендуется минимум 2. При репликации большого количества разделов топиков следует

по возможности использовать более высокие значения для увеличения параллельной обработки данных.

- *Префиксы конфигурации.* MirrorMaker поддерживает настройку параметров конфигурации для всех своих компонентов, включая коннекторы, производителей, потребителей и клиентов администратора. Конфигурации Kafka Connect и коннекторов могут быть заданы без префикса. Но поскольку конфигурация MirrorMaker может включать конфигурацию для нескольких кластеров, префиксы можно использовать для указания конфигураций для конкретного кластера или конкретного потока репликации. Как мы видели в предыдущем примере, кластеры идентифицируются с помощью псевдонимов, которые применяются в качестве префикса конфигурации для параметров, относящихся к этому кластеру. Префиксы можно использовать для построения иерархической конфигурации, при этом более конкретная конфигурация с префиксом имеет более высокий приоритет, чем менее конкретная конфигурация или конфигурация без префикса. В MirrorMaker применяются следующие префиксы:

- `{cluster}.{connector_config};`
- `{cluster}.admin.{admin_config};`
- `{source_cluster}.consumer.{consumer_config};`
- `{target_cluster}.producer.{producer_config};`
- `{source_cluster}->{target_cluster}.{replication_flow_config}.`

Топология мультикластерной репликации

Мы рассмотрели пример конфигурации для простого потока репликации в режиме «активный — резервный» для MirrorMaker. Теперь давайте поговорим о расширении конфигурации для поддержки других распространенных архитектурных шаблонов.

Топология типа «активный — активный» между Нью-Йорком и Лондоном может быть настроена путем включения потока репликации в обоих направлениях. В этом случае, несмотря на то что все топики из Нью-Йорка зеркально копируются в Лондон и наоборот, MirrorMaker гарантирует, что одно и то же событие не будет постоянно зеркально копироваться туда и обратно между двумя кластерами, поскольку удаленные топики используют псевдоним кластера в качестве префикса. Рекомендуется использование одного и того же файла конфигурации, содержащего полную топологию репликации, для разных процессов MirrorMaker, поскольку это позволяет избежать конфликтов при совместном задействовании конфигураций с помощью внутреннего топика конфигураций в целевом ЦОД. Процессы MirrorMaker могут быть запущены в целевом ЦОД

с использованием общего файла конфигурации путем указания целевого кластера при запуске процесса MirrorMaker с помощью параметра `--clusters`:

```
clusters = NYC, LON
NYC.bootstrap.servers = kafka.nyc.example.com:9092
LON.bootstrap.servers = kafka.lon.example.com:9092
NYC->LON.enabled = true
NYC->LON.topics = .*
LON->NYC.enabled = true
LON->NYC.topics = .*
```

❶
❷
❸
❹

- ❶ Включите репликацию из Нью-Йорка в Лондон.
- ❷ Укажите топики, которые будут реплицироваться из Нью-Йорка в Лондон.
- ❸ Включите репликацию из Лондона в Нью-Йорк.
- ❹ Укажите топики, которые реплицируются из Лондона в Нью-Йорк.

В топологию также можно добавить больше потоков репликации с дополнительными исходными или целевыми кластерами. Например, мы можем расширить конфигурацию для поддержки веерной репликации из Нью-Йорка в Сан-Франциско и Лондон, добавив новый поток репликации для Сан-Франциско:

```
clusters = NYC, LON, SF
SF.bootstrap.servers = kafka.sf.example.com:9092
NYC->SF.enabled = true
NYC->SF.topics = .*
```

Обеспечение безопасности MirrorMaker

Для производственных кластеров важно обеспечить безопасность всего трафика между ЦОД. Варианты защиты кластеров Kafka описаны в главе 11. MirrorMaker должен быть настроен на использование безопасного приемника брокера как в исходном, так и в целевом кластере, а параметры безопасности на стороне клиента в каждом кластере должны быть настроены для MirrorMaker, чтобы он мог устанавливать аутентифицированные соединения. Протокол SSL должен применяться для шифрования всего трафика между ЦОД. Например, для настройки учетных данных для MirrorMaker можно использовать следующую конфигурацию:

```
NYC.security.protocol=SASL_SSL
NYC.sasl.mechanism=PLAIN
NYC.sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule \
    required username="MirrorMaker" password="MirrorMaker-password";
```

❶

❷

- ❶ Протокол безопасности должен соответствовать протоколу безопасности приемника брокера, соответствующего серверам начальной загрузки, указанным для кластера. Рекомендуются протоколы SSL или SASL_SSL.

❷ Учетные данные для MirrorMaker задаются здесь с помощью конфигурации JAAS, поскольку используется SASL. Для SSL следует указать хранилища ключей, если включена взаимная аутентификация клиентов.

Администратору доступа, связанному с MirrorMaker, должны быть предоставлены соответствующие разрешения в исходном и целевом кластерах, если на последних включена авторизация. Процессу MirrorMaker должны быть предоставлены следующие права доступа:

- **Topic:Read** в исходном кластере для потребления из исходных топиков, **Topic:Create** и **Topic:Write** в целевом кластере для создания и производства в целевых топиках;
- **Topic:DescribeConfigs** в исходном кластере для получения конфигурации исходного топика, **Topic:AlterConfigs** в целевом кластере для обновления конфигурации целевого топика;
- **Topic:Alter** в целевом кластере для добавления разделов, если обнаружены новые исходные разделы;
- **Group:Describe** в исходном кластере для получения метаданных исходной группы потребителей, включая смещения, **Group:Read** в целевом кластере для фиксации смещений для этих групп в целевом кластере;
- **Cluster:Describe** в кластере источника для получения механизмов управления доступом исходного топика, **Cluster:Alter** в целевом кластере для обновления механизмов управления доступом целевого топика;
- **Topic:Create** и **Topic:Write** — разрешения для внутренних топиков MirrorMaker в исходном и целевом кластерах.

Развертывание MirrorMaker для промышленной эксплуатации

В предыдущем примере утилита MirrorMaker запускалась в специальном режиме из командной строки. Вы можете запустить любое количество этих процессов, чтобы сформировать выделенный кластер MirrorMaker, который является масштабируемым и отказоустойчивым. Процессы, зеркально копирующие в один и тот же кластер, найдут друг друга и автоматически распределят нагрузку между собой. Обычно, когда MirrorMaker работает в среде промышленной эксплуатации, желательно запускать ее как сервис с помощью команды `nohup` и перенаправлять выводимую в консоль информацию в файл журнала. У MirrorMaker также есть параметр командной строки `-daemon`, которая делает все это за вас. У большинства использующих MirrorMaker компаний написаны собственные сценарии запуска, включающие применяемые параметры. Для автоматизации развертывания и управления параметрами конфигурации задействуются также такие системы

развертывания для промышленной эксплуатации, как Ansible, Puppet, Chef и Salt. MirrorMaker также может быть запущен в контейнере Docker. Утилита MirrorMaker совершенно не сохраняет состояние, для нее не нужно никакого дискового хранилища (все данные и состояние хранятся в самой Kafka).

Поскольку MirrorMaker основан на Kafka Connect, все режимы развертывания Connect могут быть использованы с MirrorMaker. Автономный режим может применяться для разработки и тестирования, когда MirrorMaker запускается как автономный исполнитель Connect на одной машине. MirrorMaker также может быть запущен в качестве коннектора в существующем распределенном кластере Connect путем явной настройки коннекторов. Для производственного использования мы рекомендуем запускать MirrorMaker в распределенном режиме либо в выделенном кластере MirrorMaker, либо в общем распределенном кластере Connect.

Если это возможно, запускайте MirrorMaker в целевом ЦОД. То есть при отправке данных из Нью-Йорка в Сан-Франциско MirrorMaker должен работать в Сан-Франциско и потреблять данные по США из Нью-Йорка. Причина состоит в том, что сеть внутри ЦОД более надежна, чем магистральные сети. В случае разрыва связности сети и потери связи между ЦОД потребитель, который не может подключиться к кластеру, намного безопаснее подобного производителя. Такой потребитель просто не сможет читать события, но они все равно будут сохранены в исходном кластере Kafka и могут находиться там длительное время. Риска потери событий нет. В то же время, если события уже прочитаны, а MirrorMaker не может отправить их из-за разрыва связности сети, все равно появляется риск случайной их потери MirrorMaker. Так что удаленное потребление данных безопаснее удаленной их генерации.

В каких же случаях приходится потреблять данные локально, а генерировать удаленно? Ответ: тогда, когда необходимо шифровать данные при их передаче из одного ЦОД в другой, но не нужно шифровать внутри ЦОД. Использование SSL-шифрования при подключении к Kafka существенно влияет на производительность потребителей — намного сильнее, чем на производительность производителей. Это связано с тем, что применение SSL требует копирования данных для шифрования, что означает: потребители больше не пользуются преимуществами производительности обычной оптимизации с нулевым копированием. Воздействует оно и на сами брокеры Kafka. Если трафик между ЦОД требует шифрования, а местный трафик не требует, лучше разместить MirrorMaker в исходном ЦОД, чтобы он потреблял незашифрованные данные локально, после чего отправлять посредством генерации их в удаленный ЦОД через зашифрованное SSL-соединение. Таким образом, через SSL подключается к Kafka производитель, а не потребитель, и производительность страдает не так сильно. Если вы решите использовать подход локального потребления и удаленной генерации, позаботьтесь, чтобы производитель MirrorMaker Connect никогда

не терял событий, задав параметр `acks=all` и достаточное число повторов попыток. Помимо этого, настройте MirrorMaker на быстрое завершение работы с помощью `errors.tolerance=none`, если отправить события невозможно, — обычно это безопаснее, чем продолжать работу и рисковать потерей данных. Обратите внимание на то, что новые версии Java значительно увеличили производительность SSL, поэтому локальное производство и удаленное потребление может быть жизнеспособным вариантом даже с шифрованием.

Другой случай, когда нам может понадобиться производить удаленно, а потреблять локально, — это гибридный сценарий зеркального копирования из локального в облачный кластер. Безопасные локальные кластеры, скорее всего, находятся за брандмауэром, который не разрешает входящие соединения из облака. Запуск MirrorMaker локально позволяет осуществлять все локальные подключения в облако.

При развертывании MirrorMaker для промышленной эксплуатации важно не забывать контролировать ее работу, как показано далее.

- *Мониторинг Kafka Connect.* Kafka Connect предоставляет широкий спектр показателей для мониторинга различных аспектов, таких как показатели коннектора для отслеживания состояния коннектора, показатели исходного коннектора для мониторинга всего коннектора, а также показатели исполнителя для мониторинга задержек перебалансировки. Connect также предоставляет REST API для просмотра коннекторов и управления ими.
- *Мониторинг метрик MirrorMaker.* В дополнение к показателям из Connect MirrorMaker добавляет показатели для мониторинга пропускной способности зеркального копирования и задержки репликации. Показатель задержки репликации `replication-latency-ms` показывает интервал времени между отметкой времени записи и временем, в течение которого запись была успешно создана в целевом кластере. Это полезно для определения того, что целевой кластер не успевает за исходным. Увеличение задержки в часы пик может быть приемлемым, если имеется достаточная пропускная способность, чтобы наверстать упущенное позже, но постоянное увеличение задержки может указывать на недостаточную пропускную способность. Другие показатели, такие как `record-age-ms`, который показывает время существования записей на момент репликации, `byte-rate`, который показывает репликацию на протяжении всего времени, и `checkpoint-latency-ms`, который показывает задержку миграции со смещением, тоже могут быть очень полезны. По умолчанию MirrorMaker также создает периодические контрольные сигналы, которые можно использовать для мониторинга его работоспособности.
- *Мониторинг отставания.* Безусловно, нужно знать, отстает ли целевой кластер от исходного. Отставание равно разнице смещений между последним сообщением в исходном кластере Kafka и последним сообщением в целевом кластере (рис. 10.7).

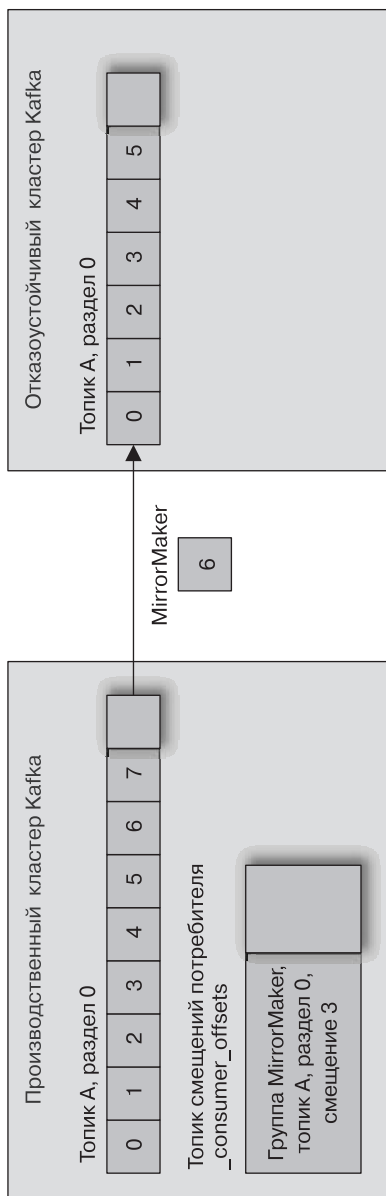


Рис. 10.7. Мониторинг величины отставания в смещениях

На рис. 10.7 последнее смещение исходного кластера равно 7, а последнее смещение целевого — 5, то есть величина отставания составляет два сообщения.

Существует два способа отслеживания этого отставания, ни один из них не идеален.

- Проверка последнего зафиксированного MirrorMaker смещения в исходном кластере Kafka. Можно воспользоваться утилитой `kafka-consumer-groups`, чтобы для каждого читаемого MirrorMaker раздела выяснить смещение последнего сообщения раздела, последнее зафиксированное смещение и отставание одного от другого. Этот показатель не совсем точен, ведь MirrorMaker не фиксирует смещения непрерывно. По умолчанию он делает это раз в минуту, так что вы обнаружите, что отставание растёт в течение минуты, после чего неожиданно резко уменьшается. На схеме фактическое отставание равно 2, но утилита `kafka-consumer-groups` покажет, что оно равно 5, поскольку MirrorMaker пока еще не зафиксировал смещения для более новых сообщений. Утилита `Bitrow` от LinkedIn служит для мониторинга той же информации, но использует более сложный метод для определения того, действительно ли отставание представляет собой проблему, так что ложной тревоги она не поднимет.
- Проверка последнего прочитанного MirrorMaker смещения, даже если оно не зафиксировано. Встраиваемые в MirrorMaker потребители публикуют важнейшие показатели в JMX. Один из них — максимальное отставание потребителя по всем читаемым разделам. Этот показатель тоже не вполне точен, поскольку обновляется в зависимости от прочитанных потребителем сообщений, но не учитывает, удалось ли производителю отправить эти сообщения в целевой кластер Kafka и было ли их получение подтверждено. В данном примере потребитель MirrorMaker проинформирует об отставании в одно сообщение, а не в два, поскольку уже прочитал сообщение 6, хотя оно еще не было сгенерировано для отправки в целевой кластер.

Обратите внимание на то, что ни один из описанных методов не обнаружит проблемы в случае, когда MirrorMaker пропускает или отбрасывает сообщения, поскольку они лишь отслеживают последнее смещение. Продукт Confluent Control Center (<https://oreil.ly/KnvVV>) — это коммерческий инструмент, который контролирует количество сообщений и контрольные суммы.

- *Мониторинг показателей производителей и потребителей.* Фреймворк Kafka Connect, используемый MirrorMaker, включает производитель и потребитель. У каждого из них есть множество показателей, которые рекомендуется собирать и отслеживать. Список всех доступных показателей приведен в документации Kafka (<http://bit.ly/2sMfZWf>). Здесь же мы перечислим лишь некоторые из них, удобные при тонкой настройке производительности MirrorMaker:
 - *показатели потребителя* — `fetch-size-avg`, `fetch-size-max`, `fetch-rate`, `fetch-throttle-time-avg` и `fetch-throttle-time-max`;

- *показатели производителя* — `batch-size-avg`, `batch-size-max`, `requests-in-flight` и `record-retry-rate`;
 - *показатели обоих* — `io-ratio` и `io-wait-ratio`.
- «Канарейка». Если вы контролируете все остальное, «канарейка» (canary) не нужна, но иметь ее в качестве дополнительного слоя мониторинга не мешает. Она представляет собой процесс, который ежеминутно отправляет событие в специальный топик в исходном кластере, после чего пытается прочитать это событие из целевого кластера. И уведомляет вас в случае, если передача события занимает слишком много времени. Это может означать, что MirrorMaker отстаёт или вообще недоступен.

Тонкая настройка MirrorMaker

MirrorMaker масштабируется по горизонтали. Выбор кластера MirrorMaker зависит от требуемой пропускной способности и допустимого отставания. Если даже минимальное отставание недопустимо, мощности MirrorMaker должно быть достаточно, чтобы выдержать максимально возможную нагрузку. Если же некоторое отставание допустимо, можно выбрать такие параметры, при которых MirrorMaker 95–99 % времени использовался бы на 75–80 %. Вы должны понимать, что при пиковой нагрузке возможно возникновение небольшого отставания, но поскольку у MirrorMaker большую часть времени есть резервы производительности, то по окончании пиковой нагрузки он его наверстает.

Далее неплохо бы оценить пропускную способность MirrorMaker при различном количестве задач коннектора — оно настраивается с помощью параметра `tasks.max`. Поскольку очень многое зависит от вашего аппаратного обеспечения, ЦОД и провайдера облачных сервисов, будет правильнее выполнить собственные тесты. В комплект поставки Kafka включена утилита `kafka-performance-producer`. Ею можно воспользоваться для генерации данных (нагрузки) в исходном кластере, а затем подключить MirrorMaker и начать эти данные зеркально копировать. Протестируйте MirrorMaker при 1, 2, 4, 8, 16 и 32 задачах коннектора. Найдите число потоков, при котором производительность начинает падать, и установите значение параметра `tasks.max` чуть меньше его. В случае потребления или генерации сжатых событий (что рекомендуется, поскольку ширина полосы пропускания — наиболее узкое место зеркального копирования между ЦОД) MirrorMaker придется распаковывать и снова упаковывать события. При этом активно используются ресурсы CPU, так что следите за применением CPU при увеличении числа задач. Таким образом вы сможете определить максимальную пропускную способность, достижимую с помощью одного исполнителя MirrorMaker. Если ее недостаточно, попробуйте добавить дополнительных исполнителей. Если вы запускаете MirrorMaker в существующем кластере Connect

с другими коннекторами, обязательно учитывайте нагрузку от этих коннекторов при определении размера кластера.

Кроме того, полезно будет выделить важные топики — те, для которых обязательно низкое значение задержки, так что кластер-зеркало должен находиться как можно ближе к исходному кластеру — в отдельном кластере MirrorMaker. Это позволит предотвратить замедление работы самого важного из ваших конвейеров из-за слишком раздутого топика или вышедшего из-под контроля производителя.

По большому счету, это все, что можно настроить в самом MirrorMaker. Однако у вас есть еще возможности повысить производительность задач и исполнителей MirrorMaker.

Если вы используете MirrorMaker в разных ЦОД, настройка стека TCP может помочь увеличить эффективную пропускную способность. В главах 3 и 4 мы видели, что размеры буферов TCP можно настроить для производителей и потребителей с помощью параметров `send.buffer.bytes` и `receive.buffer.bytes`. Аналогично размеры буферов на стороне брокера могут быть настроены с помощью параметров `socket.send.buffer.bytes` и `socket.receive.buffer.bytes` на брокерах. Эти параметры конфигурации следует сочетать с оптимизацией сетевой конфигурации в Linux следующим образом:

- увеличить размер буфера TCP (`net.core.rmem_default`, `net.core.rmem_max`, `net.core.wmem_default`, `net.core.wmem_max` и `net.core.optmem_max`);
- включить автоматическое масштабирование TCP окна (выполнить команду `sysctl -w net.ipv4.tcp_window_scaling=1` или добавить `net.ipv4.tcp_window_scaling=1` в файл `/etc/sysctl.conf`);
- уменьшить интервал времени в алгоритме медленного старта TCP (установить значение `/proc/sys/net/ipv4/tcp_slow_start_after_idle` в 0).

Обратите внимание на то, что тонкая настройка сети Linux — объемная и сложная тема. Чтобы лучше разобраться в перечисленных и иных параметрах, почитайте руководство по тонкой настройке сети, например *Performance Tuning for Linux Servers* Сандры К. Джонсон (Sandra K. Johnson) и др. (издательство IBM Press).

Кроме того, вам может понадобиться настроить основные производители и потребители в MirrorMaker. Во-первых, нужно разобраться, производитель или потребитель представляет собой узкое место — производитель ожидает, пока потребитель прочитает данные, или наоборот? Один из способов выяснить это — анализ параметров производителя и потребителя. Если один процесс простаивает, а другой используется на все 100 %, то сразу становится понятно, какой из них нуждается в настройке. Еще один способ — выполнить несколько дампов потоков выполнения с помощью утилиты `jstack` и посмотреть, на что потоки MirrorMaker тратят большую часть времени — на опросы или отправку.

То, что бóльшая часть времени затрачивается на опросы, обычно означает, что узкое место — в потребителе, а если на отправку, то в производителе.

При тонкой настройке производителя могут оказаться полезными следующие параметры конфигурации:

- **linger.ms** и **batch.size**. Если мониторинг показывает, что производитель все время отправляет полупустые пакеты (то есть значения **batch-size-avg** и **batch-size-max** меньше, чем заданное в настройках значение **batch.size**), можно увеличить пропускную способность путем создания небольшой искусственной задержки. Увеличьте значение параметра **linger.ms**, и производитель будет несколько миллисекунд ожидать наполнения пакета, прежде чем отправить его. Если же вы отправляете полные пакеты и у вас есть свободная память, то можете увеличить значение параметра **batch.size**, чтобы отправлять пакеты большего размера;
- **max.in.flight.requests.per.connection**. Ограничение числа выполняемых запросов до одного в настоящее время является единственным способом для MirrorMaker гарантировать сохранение порядка сообщений, если некоторые из них требуют нескольких повторных попыток, прежде чем будут успешно подтверждены. Но это означает, что каждый запрос, отправленный производителем, должен быть подтвержден целевым кластером перед отправкой следующего сообщения. Это может ограничить пропускную способность, особенно если брокеры подтверждают получение со значительной задержкой. Если для вашего сценария использования неважен порядок сообщений, то, используя значение по умолчанию 5 для параметра **max.in.flight.requests.per.connection**, можно значительно увеличить пропускную способность.

Следующие параметры конфигурации потребителя могут помочь увеличить его пропускную способность:

- **fetch.max.bytes** — если собираемые показатели демонстрируют, что значения **fetch-size-avg** и **fetch-size-max** близки к значению параметра **fetch.max.bytes**, то потребитель читает из брокера столько данных, сколько ему разрешается. Если у вас есть резервы памяти, можете попробовать увеличить значение параметра **fetch.max.bytes**, позволив тем самым потребителю читать больший объем данных при каждом запросе;
- **fetch.min.bytes** и **fetch.max.wait.ms** — если анализ показателей потребителя свидетельствует, что значение **fetch-rate** слишком высокое, то потребитель отправляет брокерам слишком много запросов, не получая в ответ в каждом запросе достаточного количества данных. Попробуйте увеличить значения обоих параметров, **fetch.min.bytes** и **fetch.max.wait.ms**, чтобы потребитель получал в каждом запросе больше данных, а брокер перед отправкой ему данных ждал, пока не появится достаточное их количество.

Другие программные решения для зеркального копирования между кластерами

Мы так подробно изучили MirrorMaker потому, что это программное обеспечение для зеркального копирования входит в состав Apache Kafka. Однако у MirrorMaker есть определенные ограничения. Имеет смысл обратить внимание на некоторые альтернативы MirrorMaker и то, как они обходят его ограничения и проблемы. Мы описываем несколько решений с открытым исходным кодом от Uber и LinkedIn и коммерческие решения от Confluent.

uReplicator компании Uber

Компания Uber использовала старые версии MirrorMaker очень широко и по мере роста числа топиков и разделов и повышения производительности кластера столкнулась с несколькими проблемами. Как мы видели ранее, устаревший MirrorMaker использовали потребители, которые были членами одной группы потребителей, для потребления из исходных топиков. Добавление потоков и экземпляров MirrorMaker, перезапуск экземпляров MirrorMaker или даже просто добавление новых топиков, соответствующих регулярному выражению, используемому в фильтре включения, — все это приводит к переназначению потребителей. Как мы видели в главе 4, это вызывает останов всех потребителей до тех пор, пока им всем не будут назначены новые разделы. При очень большом числе топиков и разделов этот процесс может занять немало времени, особенно при использовании потребителей старой версии, как было в Uber. В некоторых случаях это приводило к простоям в течение 5–10 минут, отставанию зеркального копирования и накоплению большого объема событий, требующих зеркального копирования. Восстановление после подобной ситуации могло быть длительным. В результате возникала очень большая задержка чтения событий потребителями из целевого кластера. Чтобы избежать переконфигурирования, когда кто-то добавлял топик, соответствующий фильтру включения топиков, Uber решил ввести список точных названий топиков для зеркального копирования вместо использования фильтра регулярных выражений. Но это было трудно поддерживать, поскольку все экземпляры MirrorMaker приходилось перенастраивать и перенаправлять, чтобы добавить новый топик. Если это сделать неправильно, могут начаться бесконечные переконфигурирования, поскольку потребители не смогут договориться о топиках, на которые они подписаны.

Из-за этих проблем Uber пришлось написать собственный клон утилиты MirrorMaker, получивший название uReplicator. В компании Uber решили воспользоваться Apache Helix в качестве центрального высокодоступного

контроллера, который отвечал бы за список топиков и разделов, назначаемых различным экземплярам `uReplicator`. В нем для добавления новых топиков в список в `Helix` администраторы применяют API REST, а `uReplicator` отвечает за назначение разделов различным потребителям. Чтобы добиться этого, в Uber заменили используемых в `MirrorMaker` потребителей Kafka на потребитель Kafka, который написали инженеры Uber и назвали потребителем `Helix`. Разделы назначаются этому потребителю контроллером `Helix`, а не в результате соглашения между потребителями (подробности того, как это происходит в Kafka, вы найдете в главе 4). В результате потребитель `Helix` избегает переназначений, прослушивая на предмет поступающих от `Helix` изменений в назначениях разделов.

В блоге инженерного обеспечения Uber написали сообщение (<https://oreil.ly/SGItx>), где описали архитектуру более подробно и показали, каких положительных результатов достигли. Зависимость `uReplicator` от Apache `Helix` означает необходимость изучения нового компонента и работы с ним, что усложняет любое развертывание. Как мы видели ранее, `MirrorMaker 2.0` решает многие из этих проблем масштабируемости и отказоустойчивости устаревшего `MirrorMaker` без каких-либо внешних зависимостей.

LinkedIn Brooklin

Как и Uber, компания LinkedIn использовала старые версии `MirrorMaker` для передачи данных между кластерами Kafka. По мере роста объема данных компания столкнулась с аналогичными проблемами масштабируемости и эксплуатационными трудностями. Поэтому LinkedIn создала решение для зеркального копирования поверх своей системы потоковой передачи данных под названием `Brooklin`. `Brooklin` — это распределенный сервис, который может передавать данные между различными гетерогенными источниками данных и целевыми системами, включая Kafka. Являясь универсальным фреймворком ввода данных, который можно использовать для построения конвейеров данных, `Brooklin` поддерживает множество сценариев использования:

- мост данных для подачи данных в системы потоковой обработки из различных источников данных;
- потоковую передачу событий сбора данных об изменениях (CDC) из различных хранилищ данных;
- решение для кросс-кластерного зеркального копирования для Kafka.

`Brooklin` — это масштабируемая распределенная система, разработанная для обеспечения высокой надежности и прошедшая масштабные тестирования с Kafka.

Она используется для зеркального копирования триллионов сообщений в день и была оптимизирована для обеспечения стабильности, производительности и работоспособности. Brooklin поставляется с REST API для управления операциями. Это общий сервис, который может обрабатывать большое количество конвейеров данных, позволяя одному и тому же сервису зеркально копировать данные на нескольких кластерах Kafka.

Решения Confluent для зеркального копирования между ЦОД

Параллельно с разработкой uReplicator компанией Uber в компании Confluent независимо создали свой Confluent Replicator. Несмотря на схожесть названий, проекты не имеют практически ничего общего — это два различных решения двух различных наборов проблем MirrorMaker. Подобно MirrorMaker 2.0, который появился позже, Replicator компании Confluent основан на фреймворке Kafka Connect и был разработан для решения проблем, с которыми сталкиваются корпоративные заказчики, когда используют устаревший MirrorMaker при разрывании своих мультикластерных архитектур.

Для клиентов, которые используют эластичные кластеры благодаря простоте эксплуатации и низким показателям RTO и RPO, компания Confluent добавила мультирегиональный кластер (Multi-Region Cluster, MRC) в качестве встроенной функции сервера Confluent, который является коммерческим компонентом платформы Confluent. Мультирегиональный кластер расширяет поддержку Kafka для эластичных кластеров, используя асинхронные реплики для ограничения влияния на задержку и пропускную способность. Как и эластичные кластеры, он подходит для репликации между зонами доступности или регионами с задержками менее 50 мс, а также для прозрачного обхода аварийного сбоя клиента. Для удаленных кластеров с менее надежными сетями в сервер Confluent недавно была добавлена встроенная функция, названная *кластерным связыванием* (Cluster Linking). Кластерное связывание расширяет протокол внутрикластерной репликации Kafka с сохранением смещения для зеркального отображения данных между кластерами.

Рассмотрим функции, поддерживаемые каждым из этих решений.

- *Confluent Replicator*. Confluent Replicator — это инструмент зеркального копирования, аналогичный MirrorMaker, который использует фреймворк Kafka Connect для управления кластерами, он может работать на существующих кластерах Connect. Оба инструмента поддерживают репликацию данных для различных топологий, а также миграцию смещений

потребителей и конфигурации топиков. Между ними есть некоторые различия в функциональности. Например, MirrorMaker поддерживает миграцию механизма управления доступом ACL и перевод смещений для любого клиента, а Replicator не переносит механизм ACL и поддерживает перевод смещений (с помощью перехватчика временных меток) только для клиентов Java. В Replicator нет концепции локальных и удаленных топиков, как в MirrorMaker, но он поддерживает агрегированные топики. Как и MirrorMaker, Replicator предотвращает циклы репликации, но делает это с помощью заголовков происхождения. Replicator предоставляет ряд показателей, таких как задержка репликации, и может мониториться с помощью своего REST API или пользовательского интерфейса центра управления. Он также поддерживает миграцию схем между кластерами и может выполнять перевод схем.

- *Мультирегиональные кластеры (MRC).* Ранее мы видели, что эластичные кластеры обеспечивают простое прозрачное восстановление после аварийного сбоя и отказоустойчивость для клиентов без необходимости перевода смещения или перезапуска клиента. Однако эластичные кластеры требуют, чтобы ЦОД располагались близко друг к другу и обеспечивали стабильную сеть с низкой задержкой для синхронной репликации между ЦОД. Мультирегиональные кластеры также подходят только для ЦОД в пределах значения задержки 50 мс, но в них используется комбинация синхронной и асинхронной репликации для ограничения влияния на производительность производителя и обеспечения большей устойчивости сети.

Как мы видели ранее, Apache Kafka поддерживает выборку из последователей, что позволяет клиентам получать данные от ближайших брокеров на основе идентификатора стойки, тем самым уменьшая трафик между ЦОД. Confluent Server также добавляет концепцию наблюдателей, которые представляют собой асинхронные реплики, не присоединяющиеся к ISR и, следовательно, не влияющие на производителей, использующих `acks=all`, но способные доставлять записи потребителям. Операторы могут настроить синхронную репликацию внутри региона и асинхронную репликацию между регионами, чтобы одновременно получить преимущества как низкой задержки, так и высокой надежности. Ограничения на размещение реплик в Confluent Server позволяют указать минимальное количество реплик в регионе с помощью идентификаторов стоек, чтобы обеспечить распределение реплик по регионам для гарантии долговечности. Confluent Platform 6.1 также добавляет автоматическое продвижение наблюдателя с настраиваемыми критериями, обеспечивая быстрое восстановление после сбоя без потери данных в автоматическом режиме.

Когда значение параметра `min.insync.replicas` падает ниже настроенного минимального количества синхронных реплик, наблюдатели, которые догнали их, автоматически продвигаются, что позволяет им присоединиться к ISR, в результате чего количество ISR возвращается к необходимому минимуму. Повышенные наблюдатели используют синхронную репликацию и могут повлиять на пропускную способность, но кластер продолжает работать без потери данных даже в случае сбоя одного из регионов. Когда потерявший работоспособность регион восстанавливается, наблюдатели автоматически понижаются, возвращая кластер к нормальному уровню производительности.

- *Связывание кластеров.* Представленное в качестве функции предварительного просмотра в платформе Confluent Platform 6.0 связывание кластеров обеспечивает межкластерную репликацию непосредственно на сервере Confluent. Используя тот же протокол, что и репликация между брокерами внутри кластера, связывание кластеров выполняет репликацию между кластерами с сохранением смещения, обеспечивая бесшовную миграцию клиентов без необходимости перевода смещения. Конфигурация топика, разделы, смещения потребителей и механизм управления доступом синхронизируются между двумя кластерами, что позволяет осуществлять восстановление после сбоя с низким RTO в случае аварии. Кластерная ссылка определяет конфигурацию направленного потока от исходного кластера к целевому. Брокеры-лидеры зеркально копируемых разделов в целевом кластере получают данные раздела от соответствующих лидеров источника, а последователи в целевом кластере реплицируются от своего локального лидера, используя стандартный механизм репликации в Kafka. Зеркально скопированные топика помечаются как доступные только для чтения в месте назначения, чтобы предотвратить любое локальное создание этих топиков, гарантируя, что зеркальные топика логически идентичны исходным топикам.

Связывание кластеров обеспечивает простоту эксплуатации, не требуя создания отдельных кластеров, как кластеры Connect, и является более производительным, чем внешние инструменты, поскольку позволяет избежать декомпрессии и повторного сжатия во время зеркального копирования. В отличие от MRC, здесь нет возможности синхронной репликации, а восстановление после сбоя клиента выполняется вручную и требует перезапуска клиента. Однако связывание кластеров может применяться в удаленных ЦОД с ненадежными сетями с высокой задержкой и снижает трафик между ЦОД за счет однократной репликации между ними. Оно подходит для миграции кластеров и совместного использования топиков.

Резюме

Мы начали эту главу с описания причин, по которым вам может понадобиться более одного кластера Kafka, после чего рассмотрели несколько распространенных мультикластерных архитектур, начиная с простейшей и заканчивая чрезвычайно сложными. Мы углубились в подробности реализации архитектуры восстановления после сбоя и сравнение различных ее вариантов. Далее перешли к утилитам, начав с MirrorMaker Apache Kafka и обсудив немало нюансов ее промышленной эксплуатации. Завершили главу обзором ее альтернатив, позволяющих решить некоторые из возникающих при работе MirrorMaker проблем.

Какие бы архитектуру и утилиты вы ни выбрали, помните о необходимости мониторинга и тестирования мультикластерной конфигурации и конвейеров зеркального копирования, как и всего остального, что попадает в промышленную эксплуатацию. А поскольку управление мультикластерной системой в Kafka проще, чем при работе с реляционными базами данных, некоторые компании вспоминают об этом слишком поздно и недостаточно внимательно относятся к ее проектированию, планированию, тестированию, автоматизации развертывания, мониторингу и обслуживанию. Вы намного повысите вероятность того, что управление несколькими кластерами Kafka окажется успешным, если отнесетесь к управлению мультикластерной архитектурой всерьез, по возможности сделав его частью единого для всей организации плана восстановления после аварийного сбоя или плана географического разнесения данных.

Обеспечение безопасности Kafka

Kafka имеет различные сценарии использования, начиная от отслеживания активности на веб-сайте и конвейеров показателей до ведения историй болезней пациентов и выполнения онлайн-платежей. Каждый сценарий использования имеет свои требования к безопасности, производительности, надежности и доступности. Хотя всегда предпочтительнее задействовать самые мощные и новейшие из доступных средств безопасности, часто приходится идти на компромисс, поскольку повышение безопасности влияет на производительность, стоимость и удобство работы пользователей. Kafka поддерживает несколько стандартных технологий безопасности с рядом параметров конфигурации, позволяющих адаптировать безопасность к разным сценариям использования.

Так же как и производительность и надежность, безопасность является аспектом системы, который должен рассматриваться для системы в целом, а не для каждого компонента в отдельности. Безопасность системы сильна лишь настолько, насколько сильно ее самое слабое звено, поэтому процессы и политики безопасности должны применяться во всей системе, включая базовую платформу. Настраиваемые функции безопасности в Kafka позволяют интегрировать их с существующей инфраструктурой безопасности для создания согласованной модели безопасности, применимой ко всей системе.

В этой главе мы обсудим функции безопасности в Kafka и посмотрим, как они решают различные аспекты безопасности и вносят свой вклад в общую безопасность установки Kafka. На протяжении всей главы будем делиться передовым опытом, рассказывать о потенциальных угрозах и методах их снижения. Мы также рассмотрим дополнительные меры, которые можно принять для обеспечения безопасности ZooKeeper и остальной части платформы.

Блокировка Kafka

Kafka использует ряд процедур безопасности для установления и поддержания конфиденциальности, целостности и доступности данных.

- Аутентификация устанавливает вашу личность и определяет, кем вы являетесь.
- Авторизация определяет, что вам разрешено делать.
- Шифрование защищает ваши данные от подслушивания и фальсификации.
- Аудит отслеживает, что вы сделали или пытались сделать.
- Квоты контролируют, сколько ресурсов вы можете использовать.

Чтобы понять, как заблокировать развертывание Kafka, сначала рассмотрим, как данные проходят через кластер Kafka. На рис. 11.1 показаны основные этапы на примере потока данных. В текущей главе мы будем использовать этот пример, чтобы изучить различные способы настройки Kafka для защиты данных на каждом этапе, стремясь гарантировать безопасность всего развертывания.

1. Алиса создает запись о заказе клиента в раздел топика с названием `customerOrders`. Запись отправляется лидеру раздела.
2. Ведущий брокер вносит запись в свой локальный файл журнала.
3. Брокер-последователь получает сообщение от лидера и записывает его в файл журнала локальной реплики.
4. Ведущий брокер обновляет состояние раздела в ZooKeeper для обновления синхронизированных реплик, если это необходимо.
5. Боб использует записи заказов клиентов из топика `customerOrders`. Он получает запись, созданную Алисой.
6. Внутреннее приложение обрабатывает все сообщения, поступающие в `customerOrders`, чтобы в режиме реального времени получить показатели по популярным товарам.

Безопасное развертывание должно гарантировать:

- *подлинность клиента*. Когда Алиса устанавливает клиентское соединение с брокером, тот должен аутентифицировать (проверить подлинность) клиента, чтобы убедиться, что сообщение действительно исходит от Алисы;
- *подлинность сервера*. Перед отправкой сообщения ведущему брокеру клиент Алисы должен проверить, что соединение установлено с настоящим брокером;

- *конфиденциальность данных.* Все соединения, по которым передаются сообщения, а также все диски, на которых хранятся сообщения, должны быть зашифрованы или физически защищены, чтобы предотвратить чтение данных злоумышленниками и гарантировать, что они не могут быть украдены;

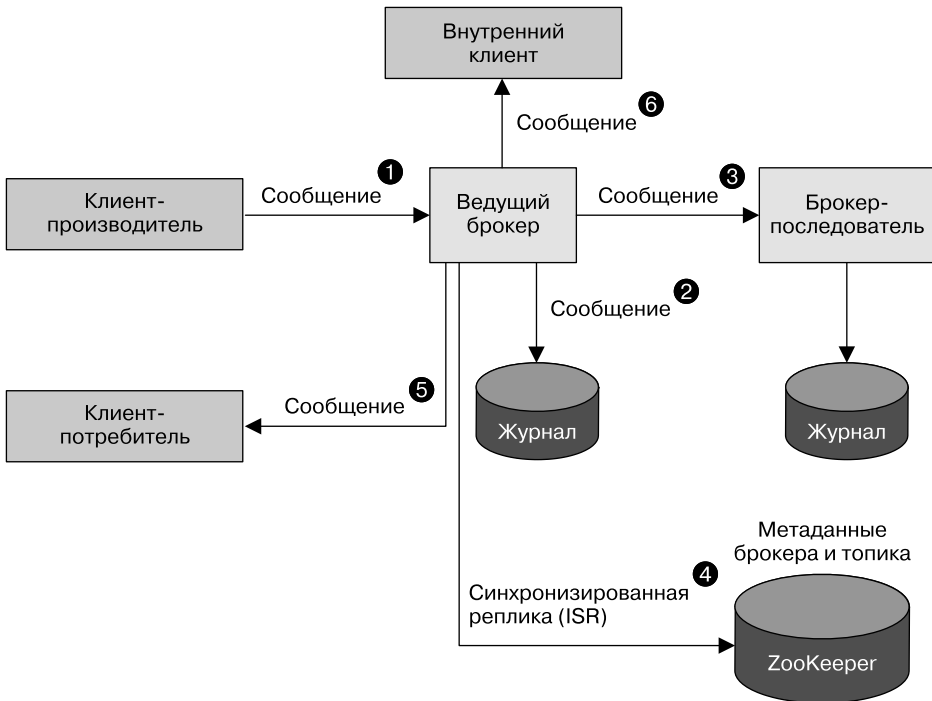


Рис. 11.1. Поток данных в кластере Kafka

- *целостность данных.* Для обнаружения фальсификации данных, передаваемых по незащищенным сетям, следует включать дайджесты сообщений;
- *контроль доступа.* Прежде чем записывать сообщение в журнал, ведущий брокер должен проверить, что Алиса имеет право писать в `customerOrders`. Перед возвратом сообщений потребителю Боба брокер должен убедиться, что Боб имеет право читать из топика. Если потребитель Боба использует управление группами, брокер также должен проверить, что Боб имеет доступ к группе потребителей;
- *возможность аудита.* Журнал аудита, показывающий все операции, которые были выполнены брокерами, Алисой, Бобом и другими клиентами, должен быть зарегистрирован;

- *доступность*. Брокеры должны применять квоты и ограничения, чтобы избежать того, что некоторые пользователи задействуют всю доступную пропускную способность или перегружат брокер атаками типа «отказ в обслуживании». ZooKeeper должен быть заблокирован, чтобы обеспечить доступность кластера Kafka, поскольку доступность брокера зависит от доступности ZooKeeper и целостности метаданных, хранящихся в ZooKeeper.

В следующих разделах мы рассмотрим функции безопасности Kafka, с помощью которых могут быть обеспечены эти гарантии. Сначала представим модель подключения Kafka и протоколы безопасности, связанные с подключениями клиентов к брокерам Kafka. Затем подробно рассмотрим все протоколы безопасности и изучим возможности аутентификации каждого протокола для установления подлинности клиента и сервера. Мы рассмотрим возможности шифрования на различных этапах, включая встроенное шифрование передаваемых данных в некоторых протоколах безопасности для обеспечения конфиденциальности и целостности данных. Затем проанализируем настраиваемую авторизацию в Kafka для управления контролем доступа и основными журналами, которые способствуют аудиту. Наконец, рассмотрим безопасность остальной части системы, включая ZooKeeper и платформу, которая необходима для поддержания доступности. Для получения подробной информации о квотах, которые помогают сделать сервис доступным за счет справедливого распределения ресурсов между пользователями, вернитесь к главе 3.

Протоколы безопасности

Брокеры Kafka настраиваются с приемниками на одной или нескольких конечных точках и принимают клиентские соединения на этих приемниках. Для каждого приемника могут быть настроены собственные параметры безопасности. Требования безопасности к частному внутреннему приемнику, который физически защищен и доступен только авторизованному персоналу, могут отличаться от требований безопасности к внешнему приемнику, получить доступ к которому можно через общедоступный Интернет. Выбор протокола безопасности определяет уровень аутентификации и шифрования данных при передаче.

Kafka поддерживает четыре протокола безопасности, используя две стандартные технологии — TLS и SASL. Протокол безопасности транспортного уровня (TLS), обычно называемый по имени своего предшественника — протокола шифрования низкого уровня для транзакций (SSL), поддерживает шифрование, а также аутентификацию клиента и сервера. Протокол простой аутентификации и безопасного соединения (SASL) — это фреймворк для обеспечения аутентификации с использованием различных механизмов в протоколах, ориентированных на соединение. Каждый протокол безопасности Kafka сочетает в себе транспортный

уровень (PLAINTEXT или SSL) с дополнительным уровнем аутентификации (SSL или SASL).

- **PLAINTEXT** (*открытый текст*). Транспортный уровень PLAINTEXT без аутентификации. Подходит только для использования в частных сетях для обработки данных, которые не являются конфиденциальными, поскольку не используется аутентификация или шифрование.
- **SSL**. Транспортный уровень SSL с дополнительной аутентификацией клиента SSL. Подходит для применения в незащищенных сетях, поскольку поддерживается аутентификация клиента и сервера, а также шифрование.
- **SASL_PLAINTEXT**. Транспортный уровень PLAINTEXT с аутентификацией клиента SASL. Некоторые механизмы SASL также поддерживают аутентификацию сервера. Не поддерживает шифрование, следовательно, подходит только для использования в частных сетях.
- **SASL_SSL**. Транспортный уровень SSL с аутентификацией SASL. Подходит для незащищенных сетей, поскольку поддерживается аутентификация клиента и сервера, а также шифрование.



TLS/SSL

Протокол TLS — это один из наиболее широко используемых криптографических протоколов в общедоступном Интернете. Такие прикладные протоколы, как HTTP, SMTP и FTP, полагаются на протокол TLS для обеспечения конфиденциальности и целостности передаваемых данных. Протокол TLS применяет инфраструктуру открытых ключей (PKI) для создания и распространения цифровых сертификатов, которые могут использоваться для асимметричного шифрования, и управления ими, что позволяет избежать необходимости распределения общих секретов между серверами и клиентами. Сеансовые ключи, создаваемые во время TLS-рукопожатия, обеспечивают симметричное шифрование с более высокой производительностью для последующей передачи данных.

Приемник, используемый для межброкерского взаимодействия, может быть выбран путем настройки параметров `inter.broker.listener.name` или `security.inter.broker.protocol`. Для протокола безопасности, используемого для межброкерского взаимодействия, в конфигурации брокера должны быть предусмотрены параметры конфигурации как на стороне сервера, так и на стороне клиента. Это связано с тем, что брокерам необходимо устанавливать клиентские соединения для этого приемника. В следующем примере SSL настраивается для межброкерского и внутреннего приемников и SASL_SSL для внешнего приемника:

```
listeners=EXTERNAL://:9092,INTERNAL://10.0.0.2:9093,BROKER://10.0.0.2:9094
advertised.listeners=EXTERNAL://broker1.example.com:9092,INTERNAL://
```

```
broker1.local:9093,BROKER://broker1.local:9094  
listener.security.protocol.map=EXTERNAL:SASL_SSL,INTERNAL:SSL,BROKER:SSL  
inter.broker.listener.name=BROKER
```

Клиенты настраиваются с помощью протокола безопасности и серверов начальной загрузки, которые определяют приемник брокера. Метаданные, возвращаемые клиентам, содержат только конечные точки, соответствующие тому же приемнику, что и серверы начальной загрузки:

```
security.protocol=SASL_SSL  
bootstrap.servers=broker1.example.com:9092,broker2.example.com:9092
```

В следующем разделе, посвященном аутентификации, мы рассмотрим специфические для протокола параметры конфигурации брокеров и клиентов для каждого протокола безопасности.

Аутентификация

Аутентификация — это процесс установления личности клиента и сервера для проверки их подлинности. Когда клиент Алисы подключается к ведущему брокеру для создания записи заказа клиента, проверка подлинности сервера позволяет клиенту установить, что сервер, с которым он разговаривает, является настоящим брокером. Аутентификация клиента устанавливает личность Алисы путем проверки ее учетных данных, таких как пароль или цифровой сертификат, чтобы определить, что соединение осуществляется от Алисы, а не от лица, выдающего себя за нее. После аутентификации идентификатор Алисы ассоциируется с подключением на протяжении всего времени действия соединения. Kafka использует экземпляр объекта `KafkaPrincipal` для представления принципа клиента и использует его для предоставления доступа к ресурсам и выделения квот для соединений с этим идентификатором клиента. `KafkaPrincipal` для каждого соединения устанавливается во время аутентификации на основе протокола аутентификации. Например, принципал `User:Alice` может быть применен для Алисы на основе имени пользователя, предоставленного для аутентификации на основе пароля. `KafkaPrincipal` может быть настроен путем конфигурирования параметра `principal.builder.class` для брокеров.



Анонимные подключения

Принципал доступа `User:ANONYMOUS` используется для неаутентифицированных подключений. Сюда входят клиенты на приемниках PLAINTEXT, а также неаутентифицированные клиенты на приемниках SSL.

SSL

Когда Kafka настроен с применением SSL или SASL_SSL в качестве протокола безопасности для приемника, протокол TLS используется в качестве безопасного транспортного уровня для соединений на этом приемнике. Когда соединение устанавливается через протокол TLS, процесс TLS-рукопожатия выполняет аутентификацию, согласовывает криптографические параметры и генерирует общие ключи для шифрования. Цифровой сертификат сервера проверяется клиентом для установления подлинности сервера. Если включена аутентификация клиента с помощью SSL, сервер также проверяет цифровой сертификат клиента, чтобы установить его личность. Весь трафик по протоколу SSL шифруется, что делает его пригодным для использования в незащищенных сетях.



Производительность SSL

Каналы SSL шифруются и, следовательно, создают заметные накладные расходы с точки зрения использования центрального процессора. Передача нулевых копий в настоящее время для SSL не поддерживается. В зависимости от структуры трафика накладные расходы могут достигать 20–30 %.

Настройка TLS

Когда TLS включен для приемника брокера с использованием протоколов SSL или SASL_SSL, брокеры должны быть настроены с хранилищем ключей, содержащим закрытый ключ и сертификат брокера, а клиенты — с хранилищем доверия, содержащим сертификат брокера или сертификат центра сертификации (CA), который подписал сертификат брокера. Сертификаты брокера должны содержать имя хоста брокера в качестве расширения альтернативного имени субъекта (SAN) или в качестве общего имени (CN), чтобы клиенты могли проверить имя хоста сервера. Сертификаты с поддержкой поддоменов можно использовать для упрощения администрирования путем использования одного хранилища ключей для всех брокеров в домене.



Проверка имени хоста сервера

По умолчанию клиенты Kafka проверяют, соответствует ли имя хоста сервера, хранящееся в сертификате сервера, хосту, к которому подключается клиент. Имя хоста соединения может быть сервером начальной загрузки, на который настроен клиент, или объявленным именем хоста приемника, которое было возвращено брокером в ответе метаданных. Проверка имени хоста — важная часть аутентификации сервера, которая защищает от атак типа «человек посередине», и поэтому в производственных системах ее отключать не следует.

Брокеры могут быть настроены на аутентификацию клиентов, подключающихся через приемники с использованием SSL в качестве протокола безопасности, путем установки параметра конфигурации брокера `ssl.client.auth=required`. Клиенты должны быть настроены с хранилищем ключей, а брокеры — с хранилищем доверия, содержащим сертификаты клиентов или сертификаты центров сертификации, подписавших сертификаты клиентов. Если для межброкерского взаимодействия используется SSL, хранилища доверия брокеров должны включать сертификат центра сертификации сертификатов брокеров, а также сертификат центра сертификации сертификатов клиентов. По умолчанию отличительное имя (DN) клиентского сертификата применяется в качестве `KafkaPrincipal` для авторизации и квот. Параметр конфигурации `ssl.principal.mapping.rules` может быть использован для предоставления списка правил для настройки принципала. Приемники, использующие протокол SASL_SSL, отключают аутентификацию клиента TLS и полагаются на аутентификацию SASL и `KafkaPrincipal`, установленный SASL.



Аутентификация клиента SSL

Аутентификацию SSL-клиента можно сделать необязательной, установив значение параметра `ssl.client.auth=requested`. Клиенты, не настроенные на хранилища ключей, в этом случае завершат подтверждение установления связи TLS, но им будет присвоено имя `User:ANONYMOUS`.

В следующих примерах показано, как создать хранилища ключей и хранилища доверия для аутентификации сервера и клиента с помощью самоподписанного сертификата центра сертификации.

Создайте самоподписанную пару ключей центра сертификации для брокеров:

```
$ keytool -genkeypair -keyalg RSA -keysize 2048 -keystore server.ca.p12 \
  -storetype PKCS12 -storepass server-ca-password -keypass server-ca-password \
  -alias ca -dname "CN=BrokerCA" -ext bc=ca:true -validity 365 ❶
$ keytool -export -file server.ca.crt -keystore server.ca.p12 \
  -storetype PKCS12 -storepass server-ca-password -alias ca -rfc ❷
```

❶ Создайте пару ключей для центра сертификации и сохраните ее в PKCS12-файле `server.ca.p12`. Мы используем его для подписания сертификатов.

❷ Экспортируйте открытый сертификат центра сертификации в `server.ca.crt`. Он будет включен в хранилища доверия и цепочки сертификатов.

Создайте хранилища ключей для брокеров с сертификатом, подписанным самоподписанным центром сертификации. При использовании подстановочных имен хостов для всех брокеров можно задействовать одно и то же хранилище ключей. В противном случае создайте хранилище ключей для каждого брокера с его полным доменным именем (FQDN):

```

$ keytool -genkey -keyalg RSA -keysize 2048 -keystore server.ks.p12 \
  -storepass server-ks-password -keystore server-ks-password -alias server \
  -storetype PKCS12 -dname "CN=Kafka,O=Confluent,C=GB" -validity 365 ❶
$ keytool -certreq -file server.csr -keystore server.ks.p12 -storetype PKCS12 \
  -storepass server-ks-password -keystore server-ks-password -alias server ❷
$ keytool -gencert -infile server.csr -outfile server.crt \
  -keystore server.ca.p12 -storetype PKCS12 -storepass server-ca-password \
  -alias ca -ext SAN=DNS:broker1.example.com -validity 365 ❸
$ cat server.crt server.ca.crt > serverchain.crt
$ keytool -importcert -file serverchain.crt -keystore server.ks.p12 \
  -storepass server-ks-password -keystore server-ks-password -alias server \
  -storetype PKCS12 -noprompt ❹

```

❶ Сгенерируйте закрытый ключ для брокера и сохраните его в PKCS12-файле `server.ks.p12`.

❷ Сгенерируйте запрос на подписание сертификата.

❸ Используйте хранилище ключей центра сертификации для подписания сертификата брокера. Подписанный сертификат хранится в файле `server.crt`.

❹ Импортируйте цепочку сертификатов брокера в хранилище ключей брокера.

Если для межброкерского взаимодействия используется протокол TLS, создайте хранилище доверия для брокеров с сертификатом центра сертификации брокера, чтобы брокеры могли аутентифицировать друг друга:

```

$ keytool -import -file server.ca.crt -keystore server.ts.p12 \
  -storetype PKCS12 -storepass server-ts-password -alias server -noprompt

```

Создайте хранилище доверия для клиентов с сертификатом центра сертификации брокера:

```

$ keytool -import -file server.ca.crt -keystore client.ts.p12 \
  -storetype PKCS12 -storepass client-ts-password -alias ca -noprompt

```

Если аутентификация клиентов TLS включена, клиенты должны быть настроены с хранилищем ключей. Следующий сценарий генерирует самоподписанный центр сертификации для клиентов и создает хранилище ключей для клиентов с сертификатом, подписанным клиентским центром сертификации. Клиентский центр сертификации добавляется в хранилище доверия брокера, чтобы брокеры могли проверять подлинность клиента:

```

# Создание самоподписывающейся пары ключей центра сертификации для клиентов
keytool -genkeypair -keyalg RSA -keysize 2048 -keystore client.ca.p12 \
  -storetype PKCS12 -storepass client-ca-password -keystore client-ca-password \
  -alias ca -dname CN=ClientCA -ext bc=ca:true -validity 365 ❶
keytool -export -file client.ca.crt -keystore client.ca.p12 -storetype PKCS12 \
  -storepass client-ca-password -alias ca -rfc

```

```
# Создание хранилища ключей для клиентов
keytool -genkey -keyalg RSA -keysize 2048 -keystore client.ks.p12 \
  -storepass client-ks-password -keypass client-ks-password -alias client \
  -storetype PKCS12 -dname "CN=Metrics App,O=Confluent,C=GB" -validity 365 ❷
keytool -certreq -file client.csr -keystore client.ks.p12 -storetype PKCS12 \
  -storepass client-ks-password -keypass client-ks-password -alias client \
keytool -gencert -infile client.csr -outfile client.crt \
  -keystore client.ca.p12 -storetype PKCS12 -storepass client-ca-password \
  -alias ca -validity 365
cat client.crt client.ca.crt > clientchain.crt
keytool -importcert -file clientchain.crt -keystore client.ks.p12 \
  -storepass client-ks-password -keypass client-ks-password -alias client \
  -storetype PKCS12 -noprompt ❸

# Добавление сертификата центра сертификации клиента в хранилище доверия брокера
keytool -import -file client.ca.crt -keystore server.ts.p12 -alias client \
  -storetype PKCS12 -storepass server-ts-password -noprompt ❹
```

- ❶ В этом примере мы создаем новый центр сертификации для клиентов.
- ❷ Клиенты, аутентифицирующиеся с помощью этого сертификата, по умолчанию используют `User:CN=Metrics App, O=Confluent, C=GB` в качестве принцепипала.
- ❸ Добавляем цепочку клиентских сертификатов в хранилище ключей клиента.
- ❹ Хранилище доверия брокера должно содержать центры сертификации всех клиентов.

После создания хранилища ключей и доверия мы можем настроить протокол TLS для брокеров. Брокеры требуют хранилище доверия только в том случае, если протокол TLS используется для межброкерского взаимодействия или если включена проверка подлинности клиентов:

```
ssl.keystore.location=/path/to/server.ks.p12
ssl.keystore.password=server-ks-password
ssl.key.password=server-ks-password
ssl.keystore.type=PKCS12
ssl.truststore.location=/path/to/server.ts.p12
ssl.truststore.password=server-ts-password
ssl.truststore.type=PKCS12
ssl.client.auth=required
```

Клиенты настраиваются с помощью созданного хранилища доверия. Хранилище ключей должно быть настроено для клиентов, если требуется аутентификация клиента.

```
ssl.truststore.location=/path/to/client.ts.p12
ssl.truststore.password=client-ts-password
ssl.truststore.type=PKCS12
ssl.keystore.location=/path/to/client.ks.p12
```



```
ssl.keystore.password=client-ks-password  
ssl.key.password=client-ks-password  
ssl.keystore.type=PKCS12
```



Хранилища доверия

Конфигурация хранилища доверия может быть пропущена как в брокерах, так и в клиентах при использовании сертификатов, подписанных хорошо известными доверенными органами. В этом случае для установления доверия достаточно будет хранилищ доверия по умолчанию в установке Java. Этапы установки описаны в главе 2.

Хранилища ключей и хранилища доверия должны периодически обновляться до истечения срока действия сертификатов, чтобы избежать сбоев при TLS-рукопожатии. Хранилища SSL брокера можно динамически обновлять, изменяя один и тот же файл или устанавливая параметр конфигурации на новый файл с новой версией. В обоих случаях для запуска обновления можно использовать API Admin или инструмент `Kafka configs`. Следующий пример обновляет хранилище ключей для внешнего приемника брокера с идентификатором брокера 0 с помощью инструмента `configs`:

```
$ bin/kafka-configs.sh --bootstrap-server localhost:9092 \
  --command-config admin.props \
  --entity-type brokers --entity-name 0 --alter --add-config \
  'listener.name.external.ssl.keystore.location=/path/to/server.ks.p12'
```

Соображения безопасности

Протокол TLS широко применяется для обеспечения безопасности транспортного уровня для нескольких протоколов, включая HTTPS. Как и в случае с любым протоколом безопасности, при использовании протокола для критически важных приложений важно понимать потенциальные угрозы и стратегии устранения их последствий. По умолчанию Kafka включает только новые протоколы TLSv1.2 и TLSv1.3, поскольку более старые протоколы, такие как TLSv1.1, имеют известные уязвимости. Из-за проблем с небезопасным повторным согласованием Kafka не поддерживает пересогласование для соединений TLS. Проверка имени хоста включена по умолчанию для предотвращения атак типа «человек посередине». Безопасность можно усилить за счет ограничения наборов шифров. Надежные шифры с размером ключа шифрования не менее 256 бит защищают от криптографических атак и обеспечивают целостность данных при передаче их по защищенной сети. Некоторые организации требуют ограничения протокола TLS и шифров для соответствия стандартам безопасности, таким как FIPS 140-2.

Поскольку хранилища ключей, содержащие закрытые ключи, по умолчанию хранятся в файловой системе, крайне важно ограничить доступ к файлам хранилища

ключей с помощью разрешений файловой системы. Стандартные функции Java TLS могут быть использованы для обеспечения возможности отзыва сертификата в случае компрометации закрытого ключа. Для уменьшения риска в этом случае можно применять ключи с коротким сроком действия.

Рукопожатие TLS является дорогостоящим и занимает значительное время в сетевых потоках брокеров. Приемники, задействующие TLS в незащищенных сетях, должны быть защищены от атак типа «отказ в обслуживании» с помощью квот и ограничений на количество соединений для обеспечения доступности брокеров. Параметр конфигурации брокера `connection.failed.authentication.delay.ms` может быть использован для задержки ответа при сбоях аутентификации, чтобы снизить частоту повторных попыток аутентификации клиентами.

SASL

Протокол Kafka поддерживает аутентификацию с помощью SASL и имеет встроенную поддержку нескольких часто используемых механизмов SASL. SASL можно комбинировать с TLS в качестве транспортного уровня для обеспечения безопасного канала с аутентификацией и шифрованием. Аутентификация SASL выполняется через последовательность запросов сервера и ответов клиента, где механизм SASL определяет последовательность и формат передачи запросов и ответов. Брокеры Kafka поддерживают следующие механизмы SASL «из коробки» с настраиваемыми обратными вызовами для интеграции с существующей инфраструктурой безопасности.

- *GSSAPI*. Аутентификация Kerberos поддерживается с помощью SASL/GSSAPI и может быть использована для интеграции с серверами Kerberos, такими как Active Directory или OpenLDAP.
- *PLAIN*. Аутентификация по имени пользователя/паролю, которая обычно применяется с настраиваемым обратным вызовом на стороне сервера для проверки паролей из внешнего хранилища паролей.
- *SCRAM-SHA-256* и *SCRAM-SHA-512*. Аутентификация по имени пользователя/паролю, доступна в Kafka по умолчанию и не требует дополнительных хранилищ паролей.
- *OAUTHBEARER*. Аутентификация с помощью токенов на предъявителя OAuth, которая обычно используется с настраиваемыми обратными вызовами для получения и проверки токенов, предоставляемых стандартными серверами OAuth.

Один или несколько механизмов SASL могут быть включены в каждом приемнике с поддержкой протокола SASL в брокере путем настройки параметра

`sasl.enabled.mechanisms` для этого приемника. Клиенты могут выбрать любой из включенных механизмов, настроив параметр `sasl.mechanism`.

Kafka использует сервис аутентификации и авторизации Java (JAAS) для настройки SASL. Параметр конфигурации `sasl.jaas.config` содержит единственную запись конфигурации JAAS, которая определяет модуль входа в систему и его параметры. Брокеры применяют префиксы `listener` и `mechanism` при настройке параметра `sasl.jaas.config`. Например, `listener.name.external.gssapi.sasl.jaas.config` настраивает запись конфигурации JAAS для SASL/GSSAPI в приемнике с именем `EXTERNAL`. Процесс входа в систему в брокерах и клиентах использует конфигурацию JAAS для определения общедоступных и частных учетных данных, применяемых для аутентификации.



Файл конфигурации JAAS

Конфигурация JAAS может быть указана также в конфигурационных файлах с помощью системного свойства Java `java.security.auth.login.config`. Однако рекомендуется использовать параметр Kafka `sasl.jaas.config`, поскольку он поддерживает защиту паролем и отдельную настройку для каждого механизма SASL, если в приемнике включено несколько механизмов.

Механизмы SASL, поддерживаемые Kafka, можно настроить для интеграции со сторонними серверами аутентификации с помощью обработчиков обратного вызова. Обработчик обратного вызова для входа в систему может быть предоставлен брокерам или клиентам для настройки процесса входа в систему — например, для получения учетных данных, которые будут использоваться для аутентификации. Обработчик обратного вызова сервера может быть предоставлен для выполнения аутентификации учетных данных клиента — например, для проверки паролей с помощью внешнего сервера паролей. Обработчик обратного вызова клиента может быть предоставлен для введения учетных данных клиента вместо включения их в конфигурацию JAAS.

В следующих подразделах мы более подробно рассмотрим механизмы SASL, поддерживаемые Kafka.

SASL/GSSAPI

Kerberos — это широко используемый протокол сетевой аутентификации, который использует надежную криптографию для поддержки безопасной взаимной аутентификации в незащищенной сети. Универсальный программный интерфейс службы безопасности (GSS-API) — это фреймворк для предоставления услуг безопасности приложениям, задействующим различные механизмы аутентификации. RFC-4752 (<https://oreil.ly/wxTZt>) представляет SASL-механизм GSSAPI для аутентификации с помощью механизма GSS-API Kerberos V5.

Доступность серверов Kerberos с открытым исходным кодом, а также коммерческих реализаций Kerberos корпоративного уровня сделала его популярным выбором для аутентификации во многих отраслях со строгими требованиями к безопасности. Kafka поддерживает аутентификацию Kerberos с использованием SASL/GSSAPI.

Настройка SASL/GSSAPI

Kafka использует поставщики безопасности GSSAPI, включенные в среду выполнения Java, для поддержки безопасной аутентификации с помощью Kerberos. Конфигурация JAAS для GSSAPI включает путь к файлу `keytab`, который содержит отображение принципов на их долгосрочные ключи в зашифрованном виде. Чтобы настроить GSSAPI для брокеров, создайте `keytab` для каждого брокера с принципом, который включает имя хоста брокера. Имена хостов брокеров проверяются клиентами для обеспечения подлинности сервера и предотвращения атак типа «человек посередине». Kerberos требует наличия защищенной службы DNS для поиска имени хоста во время аутентификации. В тех случаях, когда прямой и обратный поиск не совпадают, в конфигурационном файле Kerberos `krb5.conf` в клиентах можно установить значение `rdns=false`, чтобы отключить обратный поиск. Конфигурация JAAS для каждого брокера должна включать модуль входа Kerberos V5 из среды выполнения Java, имя файла `keytab` и полное имя администратора доступа брокера:

```
sasl.enabled.mechanisms=GSSAPI
listener.name.external.gssapi.sasl.jaas.config=\ ❶
    com.sun.security.auth.module.Krb5LoginModule required \
        useKeyTab=true storeKey=true          \
        keyTab="/path/to/broker1.keytab" \ ❷
        principal="kafka/broker1.example.com@EXAMPLE.COM"; ❸
```

❶ Мы используем параметр `sasl.jaas.config` с префиксом приемника, который содержит имя приемника и механизм SASL в нижнем регистре.

❷ Файлы `keytab` должны быть доступны для чтения процессу брокера.

❸ Принципал сервиса для брокеров должен включать имя хоста брокера.

Если SASL/GSSAPI используется для межброкерского взаимодействия, межброкерский механизм SASL и имя службы Kerberos также должны быть настроены для брокеров:

```
sasl.mechanism.inter.broker.protocol=GSSAPI
sasl.kerberos.service.name=kafka
```

Клиенты должны быть настроены с собственным файлом `keytab` и принципом в конфигурации `JAAS` и `sasl.kerberos.service.name` для указания имени сервиса, к которому они подключаются:

```
sasl.mechanism=GSSAPI
sasl.kerberos.service.name=kafka ❶
sasl.jaas.config=com.sun.security.auth.module.Krb5LoginModule required \
    useKeyTab=true storeKey=true \
    keyTab="/path/to/alice.keytab" \
    principal="Alice@EXAMPLE.COM"; ❷
```

- ❶ Для клиентов должно быть указано имя сервиса `Kafka`.
- ❷ Клиенты могут использовать принципалов без имени хоста.

По умолчанию в качестве идентификатора клиента применяется короткое имя принципала. Например, в примере `User:Alice` является клиентским принципалом, а `User:kafka` — принципалом брокера. Конфигурация брокера `sasl.kerberos.principal.to.local.rules` может быть использована для применения списка правил для преобразования полностью квалифицированного принципала в пользовательский.

Вопросы безопасности. Использовать `SASL_SSL` рекомендуется в производственных развертываниях с применением `Kerberos` для защиты потока аутентификации, а также трафика данных в соединении после аутентификации. Если протокол `TLS` не используется для обеспечения безопасного транспортного уровня, злоумышленники в сети могут получить достаточно информации для проведения атаки перебором по словарю или методом грубой силы с целью кражи учетных данных клиента. Безопаснее использовать для брокеров случайно сгенерированные ключи, а не ключи, сгенерированные на основе паролей, которые легче взломать. Следует избегать слабых алгоритмов шифрования, таких как `DES-MD5`, в пользу более сильных алгоритмов. Доступ к файлам `keytab` должен быть ограничен с помощью разрешений файловой системы, поскольку любой пользователь, владеющий файлом, сможет выдавать себя за указанного пользователя.

`SASL/GSSAPI` требует защищенной службы `DNS` для аутентификации сервера. Поскольку атаки типа «отказ в обслуживании» на сервис `KDC` или `DNS` могут привести к сбоям в аутентификации клиентов, необходимо отслеживать доступность этих служб. `Kerberos` также использует слабо синхронизированные часы с настраиваемой изменчивостью для обнаружения повторных атак. Важно обеспечить безопасность синхронизации часов.

SASL/PLAIN

RFC-4616 (<https://oreil.ly/wZrxB>) определяет простой механизм аутентификации по имени пользователя и паролю, который может применяться с протоколом TLS для обеспечения безопасной аутентификации. При этом клиент отправляет серверу имя пользователя и пароль, а сервер проверяет последний с помощью своего хранилища паролей. Kafka имеет встроенную поддержку SASL/PLAIN, которая может быть интегрирована с защищенной внешней базой данных паролей с помощью пользовательского обработчика обратного вызова.

Настройка SASL/PLAIN. Реализация SASL/PLAIN по умолчанию использует конфигурацию JAAS брокера в качестве хранилища паролей. Все имена пользователей и пароли клиентов включаются в качестве параметров входа, и брокер проверяет, соответствует ли пароль, предоставленный клиентом во время аутентификации, одной из этих записей. Имя пользователя и пароль брокера требуются только в том случае, если SASL/PLAIN применяется для межброкерского взаимодействия:

```
sasl.enabled.mechanisms=PLAIN
sasl.mechanism.inter.broker.protocol=PLAIN
listener.name.external.plain.sasl.jaas.config=\
    org.apache.kafka.common.security.plain.PlainLoginModule required \
        username="kafka" password="kafka-password" \ ❶
        user_kafka="kafka-password" \
        user_Alice="Alice-password"; ❷
```

❶ Имя пользователя и пароль, используемые для межброкерских соединений, инициированных брокером.

❷ Когда клиент Алисы подключается к брокеру, пароль, предоставленный ею, проверяется на соответствие этому паролю в конфигурации брокера.

Клиенты должны быть настроены с именем пользователя и паролем для аутентификации:

```
sasl.mechanism=PLAIN
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule \
    required username="Alice" password="Alice-password";
```

Встроенная реализация, которая хранит все пароли в конфигурации JAAS каждого брокера, небезопасная и не очень гибкая, поскольку для добавления или удаления пользователя потребуется перезапустить все брокеры. При использовании SASL/PLAIN в производстве для интеграции брокеров с защищенным сторонним сервером паролей можно применять пользовательский обработчик обратного вызова сервера. Пользовательские обработчики обратного вызова

могут быть задействованы также для поддержки смены паролей. На стороне сервера обработчик обратного вызова должен поддерживать как старые, так и новые пароли до тех пор, пока все клиенты не перейдут на новый пароль. В следующем примере показан обработчик обратного вызова, который проверяет зашифрованные пароли из файлов, созданных с помощью инструмента Apache `htpasswd`:

```
public class PasswordVerifier extends PlainServerCallbackHandler {

    private final List<String> passwdFiles = new ArrayList<>(); ❶

    @Override
    public void configure(Map<String, ?> configs, String mechanism,
        List<AppConfigurationEntry> jaasEntries) {
        Map<String, ?> loginOptions = jaasEntries.get(0).getOptions();
        String files = (String) loginOptions.get("password.files"); ❷
        Collections.addAll(passwdFiles, files.split(", "));
    }

    @Override
    protected boolean authenticate(String user, char[] password) {
        return passwdFiles.stream() ❸
            .anyMatch(file -> authenticate(file, user, password));
    }

    private boolean authenticate(String file, String user, char[] password) {
        try {
            String cmd = String.format("htpasswd -vb %s %s %s", ❹
                file, user, new String(password));
            return Runtime.getRuntime().exec(cmd).waitFor() == 0;
        } catch (Exception e) {
            return false;
        }
    }
}
```

❶ Мы используем несколько файлов паролей, чтобы поддерживать смену паролей.

❷ Передаем имена путей к файлам паролей в качестве параметра JAAS в конфигурации брокера. Также могут применяться пользовательские параметры конфигурации брокера.

❸ Мы проверяем, совпадает ли пароль в каком-либо из файлов, что позволяет использовать как старые, так и новые пароли в течение определенного времени.

❹ Для простоты мы используем `htpasswd`. Для производственных развертываний можно применять защищенную базу данных.

Брокеры настраиваются с помощью обработчика обратного вызова проверки пароля и его параметров:

```
listener.name.external.plain.sasl.jaas.config=\
    org.apache.kafka.common.security.plain.PlainLoginModule required \
    password.files="/path/to/htpassword.props,/path/to/oldhtpassword.props";
listener.name.external.plain.sasl.server.callback.handler.class=\
    com.example.PasswordVerifier
```

На стороне клиента обработчик обратного вызова клиента, реализующий `org.apache.kafka.common.security.auth.AuthenticateCallbackHandler`, может использоваться для динамической загрузки паролей во время выполнения при установлении соединения вместо статической загрузки из конфигурации JAAS во время запуска. Пароли могут быть загружены из зашифрованных файлов или с помощью внешнего защищенного сервера для повышения безопасности. В следующем примере пароли загружаются динамически из файла с помощью конфигурационных классов в Kafka:

```
@Override
public void handle(Callback[] callbacks) throws IOException {
    Properties props = Utils.loadProps(passwdFile);           ❶
    PasswordConfig config = new PasswordConfig(props);
    String user = config.getString("username");
    String password = config.getPassword("password").value(); ❷
    for (Callback callback: callbacks) {
        if (callback instanceof NameCallback)
            ((NameCallback) callback).setName(user);
        else if (callback instanceof PasswordCallback) {
            ((PasswordCallback) callback).setPassword(password.toCharArray());
        }
    }
}

private static class PasswordConfig extends AbstractConfig {
    static ConfigDef CONFIG = new ConfigDef()
        .define("username", STRING, HIGH, "User name")
        .define("password", PASSWORD, HIGH, "User password"); ❸
    PasswordConfig(Properties props) {
        super(CONFIG, props, false);
    }
}
```

❶ Мы загружаем файл конфигурации в обратном вызове, чтобы гарантировать использование последнего пароля для поддержки смены паролей.

❷ Базовая библиотека конфигурации возвращает фактическое значение пароля, даже если он внешний.

❸ Определяем конфигурации паролей с типом `PASSWORD`, чтобы гарантировать, что пароли не будут включены в записи журнала.

Клиенты, а также брокеры, которые используют SASL/PLAIN для межброкерского взаимодействия, могут быть настроены с помощью обратного вызова на стороне клиента:

```
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule \
    required file="/path/to/credentials.props";
sasl.client.callback.handler.class=com.example.PasswordProvider
```

Вопросы безопасности. Поскольку SASL/PLAIN передает пароли в открытом виде по сети, механизм PLAIN следует включать только при шифровании с использованием протокола SASL_SSL для обеспечения безопасного транспортного уровня. Пароли, хранящиеся в виде открытого текста в конфигурации JAAS брокеров и клиентов, небезопасны, поэтому следует рассмотреть возможность шифрования или внешнего хранения этих паролей в безопасном хранилище. Вместо встроенного хранилища паролей, которое хранит все пароли клиентов в конфигурации брокера JAAS, используйте безопасный внешний сервер паролей, который надежно хранит пароли и применяет строгие политики паролей.



Пароли в виде открытого текста

Избегайте применения паролей в виде открытого текста в конфигурационных файлах, даже если последние могут быть защищены с помощью разрешений файловой системы. Рассмотрите возможность внешнего размещения или шифрования паролей, чтобы исключить их случайное раскрытие. Функция защиты паролей в Kafka описана далее в этой главе.

SASL/SCRAM

RFC-5802 (<https://oreil.ly/dXe3y>) представляет безопасный механизм аутентификации по имени пользователя/пароля, который решает проблемы безопасности с помощью механизмов аутентификации по паролю, таких как SASL/PLAIN, которые передают пароли по сети. Механизм аутентификации с ответом на вызов с использованием соли (Salted Challenge Response Authentication Mechanism, SCRAM) позволяет избежать передачи паролей в открытом виде и сохраняет их в формате, который практически лишает возможности выдать себя за клиента. При использовании соли пароли объединяются с некоторыми случайными данными перед применением односторонней криптографической хеш-функции для безопасного хранения паролей. В Kafka имеется встроенный провайдер SCRAM, который можно использовать в развертываниях с безопасным ZooKeeper без необходимости в дополнительных серверах паролей. Механизмы SCRAM SCRAM-SHA-256 и SCRAM-SHA-512 поддерживаются поставщиком Kafka.

Настройка SASL/SCRAM. Начальный набор пользователей может быть создан после запуска ZooKeeper до запуска брокеров. При запуске брокеры загружают метаданные пользователей SCRAM в кэш-память, обеспечивая

успешную аутентификацию всех пользователей, включая пользователь брокера для межброкерского взаимодействия. Пользователи могут быть добавлены или удалены в любое время. Брокеры поддерживают кэш в актуальном состоянии с помощью уведомлений, основанных на наблюдателе ZooKeeper. В этом примере мы создаем пользователь с принципом `User:Alice` и паролем `Alice-password` для SASL механизма SCRAM-SHA-512:

```
$ bin/kafka-configs.sh --zookeeper localhost:2181 --alter --add-config \
  'SCRAM-SHA-512=[iterations=8192,password=Alice-password]' \
  --entity-type users --entity-name Alice
```

Один или несколько механизмов SCRAM могут быть включены на приемнике путем настройки механизмов в брокере. Имя пользователя и пароль требуются брокерам только в том случае, если приемник применяется для межброкерского взаимодействия:

```
sasl.enabled.mechanisms=SCRAM-SHA-512
sasl.mechanism.inter.broker.protocol=SCRAM-SHA-512
listener.name.external.scram-sha-512.sasl.jaas.config=\
  org.apache.kafka.common.security.scram.ScramLoginModule required \
    username="kafka" password="kafka-password"; ❶
```

❶ Имя пользователя и пароль для межброкерских соединений, инициированных брокером.

Клиенты должны быть настроены на применение одного из механизмов SASL, включенных в брокере, а конфигурация клиента JAAS должна включать имя пользователя и пароль:

```
sasl.mechanism=SCRAM-SHA-512
sasl.jaas.config=org.apache.kafka.common.security.scram.ScramLoginModule \
  required username="Alice" password="Alice-password";
```

Вы можете добавлять новые пользователи SCRAM с помощью параметра `--add-config` и удалять существующие с помощью параметра `--delete-config` инструмента конфигурации. Когда существующий пользователь удаляется, новые соединения для него не могут быть установлены, но существующие его подключения будут продолжать работать. Интервал повторной аутентификации может быть настроен для брокера, чтобы ограничить количество времени, в течение которого существующие соединения могут продолжать работать после удаления пользователя. В следующем примере удаляется конфигурация SCRAM-SHA-512 для Алисы, чтобы удалить ее учетные данные для этого механизма:

```
$ bin/kafka-configs.sh --zookeeper localhost:2181 --alter --delete-config \
  'SCRAM-SHA-512' --entity-type users --entity-name Alice
```

Вопросы безопасности. SCRAM применяет одностороннюю криптографическую функцию хеширования пароля в сочетании со случайной солью, чтобы избежать передачи фактического пароля по сети или хранения в базе данных.

Однако любая система на основе паролей надежна лишь настолько, насколько надежны сами пароли. Для защиты системы от атак методом грубой силы или перебором по словарю необходимо применять политики надежных паролей. Kafka обеспечивает защиту, поддерживая только сильные алгоритмы хеширования SHA-256 и SHA-512 и избегая более слабых алгоритмов, таких как SHA-1. Это в сочетании с высоким числом итераций по умолчанию, равным 4 096, и уникальными случайными солями для каждого сохраняемого ключа ограничивает воздействие в случае нарушения безопасности ZooKeeper.

Необходимо принять дополнительные меры предосторожности для защиты ключей, передаваемых во время подтверждения установления связи, и ключей, хранящихся в ZooKeeper, для защиты от атак методом перебора. SCRAM должен использоваться с SASL_SSL в качестве протокола безопасности, чтобы предотвратить доступ злоумышленников к хешированным ключам во время аутентификации. ZooKeeper также должен поддерживать SSL, а данные ZooKeeper должны быть защищены с помощью шифрования диска, чтобы гарантировать, что сохраненные ключи не могут быть получены даже в случае взлома хранилища. В развертываниях без защищенного ZooKeeper для интеграции с безопасным внешним хранилищем учетных данных можно использовать обратные вызовы SCRAM.

SASL/OAUTHBEARER

OAuth — это механизм авторизации, который позволяет приложениям получать ограниченный доступ к сервисам HTTP. RFC-7628 (<https://oreil.ly/SPBfv>) определяет механизм OAUTHBEARER SASL, который дает возможность учетным данным, полученным с помощью OAuth 2.0, получать доступ к защищенным ресурсам в протоколах, не относящихся к HTTP. OAUTHBEARER позволяет избежать уязвимостей в механизмах безопасности, использующих долгосрочные пароли, за счет применения токенов OAuth 2.0 на предъявителя с меньшим сроком действия и ограниченным доступом к ресурсам. Kafka поддерживает SASL/OAUTHBEARER для аутентификации клиентов, что позволяет интегрироваться со сторонними OAuth-серверами. Встроенная реализация OAUTHBEARER использует незащищенные веб-токены JSON Web Tokens (JWTs) и не подходит для производственного применения. Пользовательские обратные вызовы могут быть добавлены для интеграции со стандартными OAuth-серверами для обеспечения безопасной аутентификации с помощью механизма OAUTHBEARER в производственных развертываниях.

Настройка SASL/OAUTHBEARER

Встроенная реализация SASL/OAUTHBEARER в Kafka не проверяет токены и, следовательно, требует только указания модуля входа в конфигурацию JAAS. Если приемник используется для межброкерского взаимодействия, необходимо

также указать детали токена, применяемого для клиентских соединений, инициированных брокерами. Параметр `unsecuredLoginStringClaim_sub` является утверждением субъекта, которое по умолчанию определяет `KafkaPrincipal` для соединения:

```
sasl.enabled.mechanisms=OAUTHBEARER
sasl.mechanism.inter.broker.protocol=OAUTHBEARER
listener.name.external.oauthbearer.sasl.jaas.config=\
    org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule \
        required unsecuredLoginStringClaim_sub="kafka"; ❶
```

❶ Утверждение субъекта для токена, используемого для межброкерских соединений.

Клиенты должны быть настроены с параметром утверждения субъекта `unsecuredLoginStringClaim_sub`. Также могут быть настроены другие утверждения и время жизни токена:

```
sasl.mechanism=OAUTHBEARER
sasl.jaas.config=\
    org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule \
        required unsecuredLoginStringClaim_sub="Alice"; ❶
```

❶ `User:Alice` является `KafkaPrincipal` по умолчанию для соединений, использующих эту конфигурацию.

Для интеграции Kafka со сторонними OAuth-серверами с целью использования токенов на предъявителя в производственной среде клиенты Kafka должны быть настроены с параметром `sasl.login.callback.handler.class` для получения токенов от сервера OAuth с помощью долгосрочного пароля или токена обновления. Если OAUTHBEARER применяется для межброкерского взаимодействия, брокеры также должны быть настроены с обработчиком обратного вызова логина для получения токенов для клиентских соединений, созданных брокером для межброкерского взаимодействия:

```
@Override
public void handle(Callback[] callbacks) throws UnsupportedCallbackException {
    OAuthBearerToken token = null;
    for (Callback callback : callbacks) {
        if (callback instanceof OAuthBearerTokenCallback) {
            token = acquireToken(); ❶
            ((OAuthBearerTokenCallback) callback).token(token);
        } else if (callback instanceof SaslExtensionsCallback) { ❷
            ((SaslExtensionsCallback) callback).extensions(processExtensions(token));
        } else
            throw new UnsupportedCallbackException(callback);
    }
}
```

❶ Клиенты должны получить токен от OAuth-сервера и установить действительный токен в обратном вызове.

❷ Клиент может также включать дополнительные расширения.

Брокеры должны быть настроены с обработчиком обратного вызова сервера с помощью `listener.name.<listener-name>.oauthbearer.sasl.server.callback.handler.class` для проверки токенов, предоставленных клиентом:

```
@Override
public void handle(Callback[] callbacks) throws UnsupportedCallbackException {
    for (Callback callback : callbacks) {
        if (callback instanceof OAuthBearerValidatorCallback) {
            OAuthBearerValidatorCallback cb = (OAuthBearerValidatorCallback)
                callback;
            try {
                cb.token(validatedToken(cb.tokenValue())); ❶
            } catch (OAuthBearerIllegalTokenException e) {
                OAuthBearerValidationResult r = e.reason();
                cb.error(errorStatus(r), r.failureScope(),
                    r.failureOpenIdConfig());
            }
        } else if (callback instanceof OAuthBearerExtensionsValidatorCallback) {
            OAuthBearerExtensionsValidatorCallback ecb =
                (OAuthBearerExtensionsValidatorCallback) callback;
            ecb.inputExtensions().map().forEach((k, v) ->
                ecb.valid(validateExtension(k, v))); ❷
        } else {
            throw new UnsupportedCallbackException(callback);
        }
    }
}
```

❶ Параметр `OAuthBearerValidatorCallback` содержит токен от клиента. Брокеры проверяют его.

❷ Брокеры проверяют любые дополнительные расширения от клиента.

Вопросы безопасности. Поскольку клиенты SASL/OAUTHBEARER отправляют токены предъявителя OAuth 2.0 по сети и эти токены могут быть использованы для выдачи себя за клиента, протокол TLS должен быть включен для шифрования трафика аутентификации. Для ограничения риска в случае компрометации токенов можно применять токены с коротким сроком действия. Повторная аутентификация может быть включена для брокеров, чтобы предотвратить устаревание соединений, используемых для аутентификации. Интервал повторной аутентификации, настроенный в брокерах, в сочетании с поддержкой отзыва токенов ограничивает количество времени, в течение которого существующее соединение может продолжать использовать токен после отзыва.

Токены делегирования

Токены делегирования — это совместно используемые секретные ключи между брокерами Kafka и клиентами, которые обеспечивают упрощенный механизм настройки без необходимости распространять хранилища SSL-ключей или таблиц ключей Kerberos среди клиентских приложений. Токены делегирования могут задействоваться для снижения нагрузки на серверы аутентификации, такие как центр распределения ключей Kerberos (KDC). Такие фреймворки, как Kafka Connect, могут использовать токены делегирования для упрощения настройки безопасности для исполнителей. Клиент, прошедший аутентификацию в брокерах Kafka, может создавать токены делегирования для одного и того же принципа пользователя и распространять их среди рабочих приложений, которые затем могут проходить аутентификацию непосредственно с помощью брокеров Kafka. Каждый токен делегирования состоит из идентификатора токена и хеш-кода аутентификации сообщений (HMAC), служащего совместно используемым секретным ключом. Аутентификация клиента с помощью токенов делегирования выполняется с применением SASL/SCRAM с идентификатором токена в качестве имени пользователя и HMAC в качестве пароля.

Токены делегирования могут быть созданы или обновлены с помощью API администратора Kafka или команды `delegation-tokens`. Чтобы создать токены делегирования для принципа `User:Alice`, клиент должен пройти аутентификацию с использованием учетных данных Алисы по любому протоколу аутентификации, отличному от токенов делегирования. Клиенты, прошедшие аутентификацию с помощью токенов делегирования, не могут создавать другие токены делегирования:

```
$ bin/kafka-delegation-tokens.sh --bootstrap-server localhost:9092 \
  --command-config admin.props --create --max-life-time-period -1 \
  --renewer-principal User:Bob ❶
$ bin/kafka-delegation-tokens.sh --bootstrap-server localhost:9092 \ ❷
  --command-config admin.props --renew --renew-time-period -1 --hmac c2VjcGV0
```

❶ Если Алиса выполнит эту команду, сгенерированный токен может быть использован для выдачи себя за Алису. Владелец этого токена является `User:Alice`. Мы также настраиваем пользователь `User:Bob` в качестве обновителя токена.

❷ Команда обновления может быть запущена владельцем токена (Алиса) или обновляющим токен (Боб).

Настройка токенов делегирования. Для создания и проверки токенов делегирования все брокеры должны быть настроены на один и тот же главный ключ с помощью параметра конфигурации `delegation.token.master.key`. Этот ключ

может быть изменен только путем перезапуска всех брокеров. Все существующие токены должны быть удалены перед обновлением главного ключа, поскольку они больше не могут быть использованы, а новые токены должны быть созданы после обновления ключа во всех брокерах.

Для поддержки аутентификации с использованием токенов делегирования в брокерах должен быть включен хотя бы один из механизмов SASL/SCRAM. Клиенты должны быть настроены на применение SCRAM с идентификатором токена в качестве имени пользователя и HMAC токена в качестве пароля. Принципал Kafka для соединений, использующих эту конфигурацию, будет исходным принципалом, связанным с токеном, например `User:Alice`:

```
sasl.mechanism=SCRAM-SHA-512
sasl.jaas.config=org.apache.kafka.common.security.scram.ScramLoginModule \
    required tokenauth="true" username="MTIz" password="c2VjcmV0"; ❶
```

❶ Конфигурация SCRAM с `tokenauth` используется для настройки токенов делегирования.

Вопросы безопасности. Как и встроенная реализация SCRAM, токены делегирования подходят и для производственных целей только в тех развертываниях, где ZooKeeper защищен. Все соображения безопасности, описанные в разделе SCRAM, применимы и к токенам делегирования.

Главный ключ, используемый брокерами для генерации токенов, должен быть защищен с помощью шифрования или посредством внешнего хранения ключа в защищенном хранилище паролей. Для ограничения риска в случае компрометации токена можно применять токены делегирования с коротким сроком действия. Повторная аутентификация может быть включена в брокерах для предотвращения соединений, работающих с токенами с истекшим сроком действия, и для ограничения времени, в течение которого существующие соединения могут продолжать работать после удаления токена.

Повторная аутентификация

Как мы видели ранее, брокеры Kafka выполняют аутентификацию клиента, когда тот устанавливает соединение. Учетные данные клиента проверяются брокерами, и соединение успешно аутентифицируется, если учетные данные действительны на данный момент. Некоторые механизмы безопасности, такие как Kerberos и OAuth, используют учетные данные с ограниченным сроком действия. Kafka задействует фоновый поток входа в систему для получения новых учетных данных до истечения срока действия старых, но новые учетные данные по умолчанию применяются только для аутентификации новых соединений. Существующие соединения, которые были аутентифицированы

с помощью старых учетных данных, продолжают обрабатывать запросы до тех пор, пока не произойдет отключение из-за истечения тайм-аута запроса, простоя или сетевых ошибок. Долговременные соединения могут продолжать обрабатывать запросы еще долго после истечения срока действия учетных данных, использованных для аутентификации соединений. Брокеры Kafka поддерживают повторную аутентификацию для соединений, аутентифицированных с помощью механизма SASL, применяя параметр конфигурации `connections.max.reauth.ms`. Если для этого параметра установлено целое положительное число, брокеры Kafka определяют продолжительность сеанса для соединений SASL и информируют о ней клиентов во время рукопожатия SASL. Продолжительность сессии равна меньшему из значений времени жизни учетной записи или значения `connections.max.reauth.ms`. Любое соединение, которое не проходит повторную аутентификацию в течение этого интервала, разрывается брокером. Клиенты выполняют повторную аутентификацию, используя последние учетные данные, полученные фоновым потоком входа в систему или введенные с помощью пользовательских обратных вызовов. Повторная аутентификация может быть применена для усиления безопасности в нескольких сценариях.

- Для механизмов SASL, таких как GSSAPI и OAUTHBEARER, которые используют учетные данные с ограниченным сроком действия, повторная аутентификация гарантирует, что все активные соединения связаны с действительными учетными данными. Учетные данные с коротким сроком действия ограничивают риск уязвимости в случае компрометации учетных данных.
- Механизмы SASL на основе паролей, такие как PLAIN и SCRAM, могут поддерживать смену паролей путем добавления периодического входа в систему. Повторная аутентификация ограничивает время обработки запросов в соединениях, аутентифицированных с помощью старого пароля. Пользовательский обратный вызов сервера, который позволяет использовать как старые, так и новые пароли в течение определенного периода времени, может быть задействован для предотвращения сбоя до тех пор, пока все клиенты не перейдут на новый пароль.
- Параметр `connections.max.reauth.ms` инициирует повторную аутентификацию во всех механизмах SASL, в том числе с учетными данными с неистекшим сроком действия. Это ограничивает время, в течение которого учетные данные могут быть связаны с активным соединением после их отзыва.
- Соединения от клиентов, не поддерживающих повторную аутентификацию SASL, прерываются по истечении срока действия сессии, заставляя клиентов заново подключаться и проходить аутентификацию, что обеспечивает те же гарантии безопасности для просроченных или отозванных учетных данных.



Скомпрометированные пользователи

Если пользователь скомпрометирован, необходимо как можно скорее удалить его из системы. После удаления пользователя с сервера аутентификации все новые соединения не смогут пройти аутентификацию в брокерах Kafka. Существующие соединения будут продолжать обрабатывать запросы до следующего тайм-аута повторной аутентификации. Если значение параметра `connections.max.reauth.ms` не настроено, тайм-аут не применяется и существующие соединения могут продолжать использовать идентификатор скомпрометированного пользователя в течение длительного времени. Kafka не поддерживает повторное согласование SSL из-за известных уязвимостей при повторном согласовании в старых протоколах SSL. Более новые протоколы, такие как TLSv1.3, не поддерживают повторное согласование. Таким образом, существующие SSL-соединения могут продолжать использовать отозванные или просроченные сертификаты. Запретить этим соединениям выполнять какие-либо операции можно с помощью запрещающих списков управления доступом для администратора доступа пользователя. Поскольку изменения списков управления доступом применяются с очень малой задержкой во всех брокерах, это самый быстрый способ запретить доступ скомпрометированным пользователям.

Обновления системы безопасности без простоя

Развертывания Kafka нуждаются в регулярном обслуживании для смены совместно используемых секретных ключей, применения исправлений безопасности и обновления до последних протоколов безопасности. Многие из этих задач обслуживания выполняются с помощью текущих обновлений (rolling update), когда брокеры один за другим отключаются и перезапускаются с обновленной конфигурацией. Некоторые задачи, такие как обновление хранилищ ключей SSL и хранилищ доверия, могут быть выполнены с помощью динамических обновлений конфигурации без перезапуска брокеров.

При добавлении нового протокола безопасности в существующее развертывание можно добавить новый приемник в брокеры с новым протоколом, сохранив при этом старый приемник со старым протоколом, чтобы клиентские приложения могли продолжать работать со старым приемником во время обновления. Например, для перехода с PLAINTEXT на SASL_SSL в существующем развертывании можно использовать такую последовательность действий.

1. Добавьте новый приемник на новый порт для каждого брокера с помощью инструмента конфигурации Kafka. Используйте одну команду обновления настроек для обновления `listeners` и `advertised.listeners`, чтобы включить старый и новый приемники и предоставьте все параметры конфигурации для нового приемника SASL_SSL с префиксом `listener`.

2. Измените все клиентские приложения для использования нового приемника SASL_SSL.
3. Если межброкерское взаимодействие обновляется для применения нового приемника SASL_SSL, выполните текущее обновление брокеров с новым `inter.broker.listener.name`.
4. Используйте инструмент конфигурации, чтобы удалить старый приемник из `listeners` и `advertised.listeners` и удалить все неиспользуемые параметры конфигурации старого приемника.

Механизмы SASL можно добавлять или удалять из существующих приемников SASL без простоев, используя текущие обновления на одном и том же порте приемника. Описанная далее последовательность действий приводит к переключению механизма с PLAIN на SCRAM-SHA-256.

1. Добавьте всех существующих пользователей в хранилище SCRAM с помощью инструмента конфигурации Kafka.
2. Установите значение параметра `sasl.enabled.mechanisms=PLAIN,SCRAM-SHA-256`, настройте `listener.name.<_listener-name>.scram-sha-256.sasl.jaas.config` для приемника и выполните скользящее обновление брокеров.
3. Измените все клиентские приложения, чтобы они использовали `sasl.mechanism=SCRAM-SHA-256`, и обновите `sasl.jaas.config` для использования SCRAM.
4. Если приемник применяется для межброкерского взаимодействия, используйте текущее обновление брокеров, чтобы установить `sasl.mechanism.inter.broker.protocol=SCRAM-SHA-256`.
5. Выполните еще одно текущее обновление брокеров, чтобы удалить механизм PLAIN. Установите значение параметра `sasl.enabled.mechanisms=SCRAM-SHA-256` и удалите `listener.name.<listener-name>.plain.sasl.jaas.config` и любые другие параметры конфигурации для PLAIN.

Шифрование

С помощью шифрования обеспечивается сохранение конфиденциальности и целостности данных. Как мы обсуждали ранее, приемники Kafka, использующие протоколы безопасности SSL и SASL_SSL, применяют протокол TLS в качестве транспортного уровня, обеспечивая безопасные зашифрованные каналы, которые защищают данные, передаваемые по незащищенной сети. Наборы шифров TLS могут быть ограничены для усиления безопасности и соответствия требованиям безопасности, таким как Федеральный стандарт обработки информации (FIPS).

Дополнительные меры должны быть приняты для защиты данных в состоянии покоя, чтобы гарантировать, что конфиденциальные данные не могут быть извлечены даже пользователями, имеющими физический доступ к диску, на котором хранятся журналы Kafka. Чтобы избежать нарушений безопасности даже в случае кражи диска, физическое хранилище можно зашифровать с помощью шифрования всего диска или тома.

Хотя шифрование транспортного уровня и хранения данных способно обеспечить адекватную защиту во многих развертываниях, может потребоваться дополнительная защита, чтобы избежать предоставления автоматического доступа к данным администраторам платформы. Незашифрованные данные, находящиеся в памяти брокера, могут появиться в дампах кучи, и администраторы, имеющие прямой доступ к диску, смогут получить доступ к ним, а также к журналам Kafka, содержащим потенциально конфиденциальные данные. В развертываниях с высокочувствительными данными или персонально идентифицируемой информацией (PII) требуются дополнительные меры для сохранения конфиденциальности данных. Для соблюдения нормативных требований, особенно в облачных развертываниях, необходимо гарантировать, что конфиденциальные данные не могут быть доступны администраторам платформы или поставщикам облачных услуг никаким способом. Пользовательские поставщики шифрования могут быть подключены к клиентам Kafka для реализации сквозного шифрования, которое гарантирует, что весь поток данных будет зашифрован.

Сквозное шифрование

В главе 3, посвященной производителям Kafka, мы видели, что *сериализаторы* используются для преобразования сообщений в массив байтов, хранящихся в журналах Kafka, а в главе 4, посвященной потребителям Kafka, — что *десериализаторы* преобразуют массив байтов обратно в сообщение. Сериализаторы и десериализаторы могут быть интегрированы с библиотекой шифрования для выполнения шифрования сообщения в ходе сериализации и дешифрования во время десериализации. Шифрование сообщений обычно выполняется с помощью алгоритмов симметричного шифрования, таких как AES. Общий ключ шифрования, хранящийся в системе управления ключами (KMS), позволяет производителям шифровать сообщение, а потребителям — расшифровывать его. Брокерам не требуется доступ к ключу шифрования, и они никогда не видят незашифрованное содержимое сообщения, что делает этот подход безопасным для использования в облачных средах. Параметры шифрования, необходимые для расшифровки сообщения, могут храниться в заголовках сообщений или в их полезной нагрузке, если доступ к ним требуется более старым потребителям без

поддержки заголовков. Цифровая подпись также может быть включена в заголовки сообщений для проверки целостности сообщения.

На рис. 11.2 показан поток данных Kafka со сквозным шифрованием.

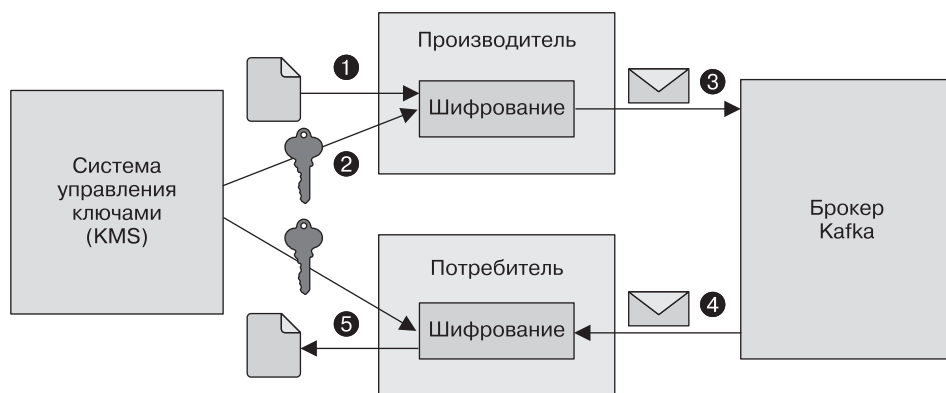


Рис. 11.2. Сквозное шифрование

1. Мы отправляем сообщение с помощью производителя Kafka.
2. Производитель использует ключ шифрования из KMS для шифрования сообщения.
3. Зашифрованное сообщение отправляется брокеру. Он сохраняет зашифрованное сообщение в журнале разделов.
4. Брокер отправляет зашифрованное сообщение потребителю.
5. Потребитель использует ключ шифрования из KMS для расшифровки сообщения.

Производители и потребители должны быть настроены с учетными данными для получения общих ключей от KMS. Для усиления безопасности рекомендуется периодически менять ключи, поскольку частая смена ограничивает количество скомпрометированных сообщений в случае взлома и защищает от атак методом перебора. В течение периода хранения сообщений, зашифрованных старым ключом, должна поддерживаться работа как со старым, так и с новым ключом. Многие системы KMS поддерживают постепенную смену ключей по умолчанию для симметричного шифрования, не требуя какой-либо специальной обработки в клиентах Kafka. В сжатых топиках сообщения, зашифрованные старыми ключами, способны храниться долго, и может возникнуть необходимость повторного шифрования старых сообщений. Чтобы избежать помех для новых сообщений, производители и потребители должны находиться в автономном режиме во время этого процесса.



Сжатие зашифрованных сообщений

Сжатие сообщений после шифрования вряд ли даст какие-либо преимущества с точки зрения сокращения пространства по сравнению со сжатием до шифрования. Сериализаторы могут быть настроены на выполнение сжатия перед шифрованием сообщения либо приложения могут быть настроены на выполнение сжатия перед созданием сообщений. В любом случае лучше отключить сжатие в Kafka, поскольку оно увеличивает накладные расходы, не давая дополнительных преимуществ. Для сообщений, передаваемых через незащищенный транспортный уровень, необходимо также учитывать известные уязвимости безопасности сжатых зашифрованных сообщений.

Во многих средах, особенно при использовании TLS в качестве транспортного уровня, ключи сообщений не требуют шифрования, поскольку они обычно не содержат конфиденциальных данных, таких как полезная нагрузка сообщения. Однако в некоторых случаях ключи с открытым текстом могут не соответствовать нормативным требованиям. Поскольку ключи сообщений применяются для разделения и сжатия, при передаче ключей должна сохраняться требуемая эквивалентность хеш-функции, чтобы гарантировать, что ключ сохраняет то же хеш-значение даже при изменении параметров шифрования. Один из подходов заключается в хранении защищенного хеша исходного ключа в качестве ключа сообщения и хранении зашифрованного ключа сообщения в его полезной нагрузке или заголовке. Поскольку Kafka сериализует ключ и значение сообщения независимо друг от друга, для выполнения этого преобразования можно использовать перехватчик производителя.

Авторизация

Авторизация — это процесс, определяющий, какие операции вам разрешено выполнять над какими ресурсами. Брокеры Kafka управляют контролем доступа с помощью настраиваемого авторизатора. Ранее мы видели, что каждый раз при установлении соединения между клиентом и брокером последний проверяет подлинность клиента и связывает `KafkaPrincipal`, который представляет личность клиента, с соединением. Когда запрос обрабатывается, брокер убеждается, что принципал, связанный с соединением, авторизован выполнять этот запрос. Например, когда производитель Алисы пытается добавить новую запись заказа клиента в тему `customerOrders`, брокер проверяет, авторизован ли `User:Alice` для записи в этот топик.

Kafka имеет встроенный авторизатор `AclAuthorizer`, который можно включить, настроив имя класса авторизатора следующим образом:

```
authorizer.class.name=kafka.security.authorizer.AclAuthorizer
```



SimpleAclAuthorizer

AclAuthorizer был представлен в Apache Kafka 2.3. В более ранних версиях, начиная с версии 0.9.0.0, был встроенный авторизатор `kafka.security.auth.SimpleAclAuthorizer`. Он устарел, но все еще поддерживается.

AclAuthorizer

Авторизатор **AclAuthorizer** поддерживает детальное управление доступом к ресурсам Kafka с помощью списков управления доступом (ACL). Списки управления доступом хранятся в ZooKeeper и кэшируются в памяти каждого брокера для обеспечения высокопроизводительного поиска при авторизации запросов. Списки управления доступом загружаются в кэш при запуске брокера, и кэш поддерживается в актуальном состоянии с помощью уведомлений, основанных на наблюдателе ZooKeeper. Каждый запрос Kafka авторизуется путем проверки, имеет ли `KafkaPrincipal`, связанный с соединением, разрешения на выполнение запрашиваемой операции с запрашиваемыми ресурсами.

Каждая привязка списков управления доступом состоит из следующих элементов:

- тип ресурса: `Cluster|Topic|Group|TransactionalId|DelegationToken` (Кластер|Топик|Группа|Идентификатор транзакции|Токен делегирования);
- тип шаблона: `Literal|Prefixed` (Буквальный|Префиксный);
- имя ресурса: имя ресурса, или префикс, или подстановочный знак *;
- операция: `Describe|Create|Delete|Alter|Read|Write|DescribeConfigs|AlterConfigs` (Описать|Создать|Удалить|Изменить|Читать|Записать|Описать настройки|Изменить настройки);
- тип разрешения: `Allow|Deny` (Разрешить|Запретить). `Deny` (Запретить) имеет более высокий приоритет;
- принципал: принципал Kafka, представленный как `<principalType>:<principalName>`, например, `ser:Bob` или `Group:Sales`. Списки управления доступом могут использовать `User:*` для предоставления доступа всем пользователям;
- хост: исходный IP-адрес клиентского соединения или *, если разрешены все хосты.

Например, список управления доступом может указывать:

```
User:Alice has Allow permission for Write to Prefixed Topic:customer from
192.168.0.1
```

AclAuthorizer авторизует действие, если нет запрещающих списков управления доступом `Deny`, соответствующих действию, и есть хотя бы один разрешающий

список управления доступом **Allow**, соответствующий действию. Разрешение **Describe** предоставляется неявно, если предоставлено разрешение **Read**, **Write**, **Alter** или **Delete**. Разрешение **Describe Configs** предоставляется неявно, если предоставлено разрешение **AlterConfigs**.



Списки управления доступом с подстановочными знаками

Списки управления доступом с типом шаблона **Literal** и именем ресурса * используются в качестве списков управления доступом с подстановочными знаками, которые соответствуют всем именам ресурсов определенного типа.

Брокерам должен быть предоставлен доступ **Cluster:ClusterAction** для авторизации запросов контроллера и запросов на выборку реплик. Производители требуют **Topic:Write** для производства в топик. Для идемпотентных производителей без транзакций производителем также должен быть предоставлен доступ **Cluster:IdempotentWrite**. Транзакционные производители требуют **TransactionalId:Write** для доступа к IS транзакции и **Group:Read** для групп потребителей для фиксации смещений. Потребителям требуется **Topic:Read** для потребления из топика и **Group:Read** для группы потребителей, если используется управление группами или управление смещениями. Административные операции требуют соответствующего доступа **Create**, **Delete**, **Describe**, **Alter**, **DescribeConfigs** или **AlterConfigs**. В табл. 11.1 показаны запросы **Kafka**, к которым применяется каждый список управления доступом.

Таблица 11.1. Доступ, предоставляемый для каждого списка управления доступом **Kafka**

Список управления доступом	Запросы Kafka	Примечания
Cluster:ClusterAction	Межброкерские запросы, включая запросы контроллера и запросы на получение выборки последователей для репликации	Должны предоставляться только брокерам
Cluster:Create	CreateTopics и автоматическое создание топиков	Используйте Topic:Create для детального контроля доступа к созданию определенных топиков
Cluster:Alter	CreateAcls , DeleteAcls , AlterReplicaLogDirs , ElectReplicaLeader , AlterPartitionReassignments	
Cluster:AlterConfigs	AlterConfigs и IncrementalAlterConfigs для брокера и регистратора брокера, AlterClient Quotas	

Таблица 11.1 (продолжение)

Список управления доступом	Запросы Kafka	Примечания
Cluster:Describe	DescribeAcls, DescribeLogDirs, ListGroups, ListPartitionReassignments, описывающие авторизованные операции для кластера в запросе метаданных	Используйте Group:Describe для детального контроля доступа для групп списков ListGroups
Cluster:DescribeConfigs	DescribeConfigs для брокера и регистратора брокера, DescribeClientQuotas	
Cluster:IdempotentWrite	Идемпотентные запросы InitProducerId и Produce	Требуются только для нетранзакционных идемпотентных производителей
Topic:Create	CreateTopics и автоматическое создание топиков	
Topic:Delete	DeleteTopics, DeleteRecords	
Topic:Alter	CreatePartitions	
Topic:AlterConfigs	AlterConfigs и IncrementalAlterConfigs для топиков	
Topic:Describe	Запрос метаданных для топика, OffsetForLeaderEpoch, ListOffset, OffsetFetch	
Topic:DescribeConfigs	DescribeConfigs для топиков для возврата конфигурации в ответе CreateTopics	
Topic:Read	Consumer Fetch, OffsetCommit, TxnOffsetCommit, OffsetDelete	Должны предоставляться потребителям
Topic:Write	Produce, AddPartitionToTxn	Должны предоставляться производителям
Group:Read	JoinGroup, SyncGroup, LeaveGroup, Heartbeat, OffsetCommit, AddOffsetsToTxn, TxnOffsetCommit	Требуется для потребителей, использующих управление группами потребителей или управление смещениями на основе Kafka. Также нужен транзакционным производителям для фиксации смещений в рамках транзакции
Group:Describe	FindCoordinator, DescribeGroup, ListGroups, OffsetFetch	
Group:Delete	DeleteGroups, OffsetDelete	
TransactionalId:Write	Produce и InitProducerId с транзакциями, AddPartitionToTxn, AddOffsetsToTxn, TxnOffsetCommit, EndTxn	Требуется для производителей транзакций
TransactionalId:Describe	FindCoordinator для координатора транзакции	
DelegationToken:Describe	DescribeTokens	

Kafka предоставляет инструмент для управления списками управления доступом с помощью авторизатора, настроенного в брокерах. Списки управления доступом можно создавать и непосредственно в ZooKeeper. Это удобно для создания списков управления доступом брокеров перед запуском брокеров:

```
$ bin/kafka-acls.sh --add --cluster --operation ClusterAction \
  --authorizer-properties zookeeper.connect=localhost:2181 \ ❶
  --allow-principal User:kafka
$ bin/kafka-acls.sh --bootstrap-server localhost:9092 \
  --command-config admin.props --add --topic customerOrders \ ❷
  --producer --allow-principal User:Alice
$ bin/kafka-acls.sh --bootstrap-server localhost:9092 \
  --command-config admin.props --add --resource-pattern-type PREFIXED \ ❸
  --topic customer --operation Read --allow-principal User:Bob
```

❶ Списки управления доступом для пользователя брокера создаются непосредственно в ZooKeeper.

❷ По умолчанию команда списков управления доступом предоставляет буквенные списки управления доступом. `User:Alice` предоставляется доступ для записи в топик `customerOrders`.

❸ Список управления доступом с префиксом дает Бобу разрешение читать все топики, начинающиеся с `customer`.

`AclAuthorizer` имеет два варианта конфигурации для предоставления широкого доступа к ресурсам или администраторам доступа, чтобы упростить управление списками управления доступом, особенно при первом добавлении авторизации в существующие кластеры:

```
super.users=User:Carol;User:Admin
allow.everyone.if.no.acl.found=true
```

Суперпользователям предоставляется доступ ко всем операциям на всех ресурсах без каких-либо ограничений, и им нельзя запретить доступ с помощью запрещающих списков управления доступом `Deny`. Если учетные данные Кэрол скомпрометированы, она должна быть удалена из списка `super.users`, а брокеры должны быть перезапущены для того, чтобы изменения вступили в силу. Безопаснее предоставлять определенный доступ пользователям в производственных системах с помощью списков управления доступом, чтобы при необходимости обеспечить возможность легкой отмены доступа.



Разделитель суперпользователей

В отличие от других конфигураций списков в Kafka, которые разделяются запятыми, `super.users` разделяются точкой с запятой, поскольку принципалы пользователей, такие как отличительные имена из SSL-сертификатов, часто содержат запятые.

Если включен параметр `allow.everybody.if.no.acl.found`, всем пользователям предоставляется доступ к ресурсам без каких-либо списков управления доступом. Этот параметр может быть полезен при первом включении авторизации в кластере или во время разработки, но не подходит для производственного применения, поскольку доступ к новым ресурсам может быть предоставлен непреднамеренно. Доступ также может быть неожиданно удален при добавлении списка управления доступом для соответствующего префикса или подстановочного знака, если условие `no.acl.found` больше не применяется.

Настройка авторизации

Авторизация может быть настроена в Kafka для реализации дополнительных ограничений или добавления новых типов контроля доступа, например управления доступом на основе ролей.

Следующий пользовательский авторизатор ограничивает использование некоторых запросов только внутренним приемником. Для простоты запросы и имя приемника здесь жестко закодированы, но для большей гибкости их можно настроить с помощью пользовательских свойств авторизатора:

```
public class CustomAuthorizer extends AclAuthorizer {
    private static final Set<Short> internalOps =
        Utils.mkSet(CREATE_ACLS.id, DELETE_ACLS.id);
    private static final String internalListener = "INTERNAL";

    @Override
    public List<AuthorizationResult> authorize(
        AuthorizableRequestContext context, List<Action> actions) {
        if (!context.listenerName().equals(internalListener) && ❶
            internalOps.contains((short) context.requestType()))
            return Collections.nCopies(actions.size(), DENIED);
        else
            return super.authorize(context, actions); ❷
    }
}
```

❶ Авторизаторам предоставляется контекст запроса с метаданными, включающими имена приемников, протокол безопасности, типы запросов и т. д., что позволяет настраиваемым авторизаторам добавлять или снимать ограничения на основе контекста.

❷ Мы повторно используем функциональность встроенного авторизатора Kafka, применяя общедоступный API.

Авторизатор Kafka может быть интегрирован с внешними системами для поддержки управления доступом на основе групп или ролей. Различные типы принципалов могут быть использованы для создания списка управления доступом

для принципов групп или принципов ролей. Например, роли и группы с сервера LDAP могут быть использованы для периодического заполнения групп и ролей в приведенном ниже классе Scala для поддержки разрешенных списков управления доступом на разных уровнях:

```
class RbacAuthorizer extends AclAuthorizer {

  @volatile private var groups = Map.empty[KafkaPrincipal, Set[KafkaPrincipal]]
    .withDefaultValue(Set.empty) ❶
  @volatile private var roles = Map.empty[KafkaPrincipal, Set[KafkaPrincipal]]
    .withDefaultValue(Set.empty) ❷

  override def authorize(context: AuthorizableRequestContext,
    actions: util.List[Action]): util.List[AuthorizationResult] = {
    val principals = groups(context.principal) + context.principal
    val allPrincipals = principals.flatMap(roles) ++ principals ❸
    val contexts = allPrincipals.map(authorizeContext(context, _))
    actions.asScala.map { action =>
      val authorized = contexts.exists(
        super.authorize(_, List(action).asJava).get(0) == ALLOWED
      ) if (authorized) ALLOWED else DENIED ❹
    }.asJava
  }

  private def authorizeContext(context: AuthorizableRequestContext,
    contextPrincipal: KafkaPrincipal): AuthorizableRequestContext = {
    new AuthorizableRequestContext { ❺
      override def principal() = contextPrincipal
      override def clientId() = context.clientId
      override def requestType() = context.requestType
      override def requestVersion() = context.requestVersion
      override def correlationId() = context.correlationId
      override def securityProtocol() = context.securityProtocol
      override def listenerName() = context.listenerName
      override def clientAddress() = context.clientAddress
    }
  }
}
```

❶ Группы, к которым принадлежит каждый пользователь, заполняются из внешнего источника, например LDAP.

❷ Роли, связанные с каждым пользователем, заполняются из внешнего источника, например LDAP.

❸ Мы выполняем авторизацию для пользователя, а также для всех групп и ролей пользователя.

❹ Если любой из контекстов авторизован, возвращаем ALLOWED. Обратите внимание на то, что в этом примере не поддерживаются запрещающие списки управления доступом Deny для групп или ролей.

❸ Мы создаем контекст авторизации для каждого принципа с теми же метаданными, что и исходный контекст.

Списки управления доступом могут быть назначены для группы **Sales** или роли **Operator** с помощью стандартного инструмента **Kafka ACL**:

```
$ bin/kafka-acls.sh --bootstrap-server localhost:9092 \
  --command-config admin.props --add --topic customer --producer \
  --resource-pattern-type PREFIXED --allow-principal Group:Sales ❶
$ bin/kafka-acls.sh --bootstrap-server localhost:9092 \
  --command-config admin.props --add --cluster --operation Alter \
  --allow-principal=Role:Operator ❷
```

❶ Мы используем принципал **Group:Sales** с пользовательским типом принципа **Group** для создания списка управления доступом, который применяется к пользователям, принадлежащим к группе **Sales**.

❷ Мы используем принципал **Role:Operator** с пользовательским типом принципа **Role**, чтобы создать список управления доступом, который применяется к пользователям с ролью **Operator**.

Вопросы безопасности

Поскольку **AclAuthorizer** хранит списки управления доступом в **ZooKeeper**, доступ к **ZooKeeper** должен быть ограничен. Развертывания без защищенного **ZooKeeper** могут реализовать пользовательские авторизаторы для хранения списков управления доступом в защищенной внешней базе данных.

В крупных организациях с большим количеством пользователей управление списками доступа для отдельных ресурсов может стать очень громоздким. Резервирование различных префиксов ресурсов для разных отделов позволяет использовать списки управления доступом с префиксами, которые сводят к минимуму количество необходимых списков управления доступом. Их можно комбинировать со списками управления доступом на основе групп или ролей, как показано в предыдущем примере, для дальнейшего упрощения контроля доступа в больших развертываниях.

Ограничение доступа пользователей с помощью принципа наименьших привилегий может ограничить уязвимость в случае компрометации пользователя. Это означает предоставление доступа только к тем ресурсам, которые необходимы каждому пользователю для выполнения своих операций, и удаление списков управления доступом, когда они больше не требуются. Списки управления доступом должны быть удалены немедленно, когда аккаунт перестает использоваться, например, когда человек покидает организацию. Чтобы избежать каких-

либо сбоях при увольнении сотрудников, можно настроить долго работающие приложения с применением учетных данных службы, а не учетных данных, связанных с конкретным пользователем. Поскольку долгоживущие приложения с основным пользователем могут продолжать обрабатывать запросы даже после удаления пользователя из системы, можно использовать запрещающие списки управления доступом **Deny**, чтобы гарантировать, что основному пользователю не будет непреднамеренно предоставлен доступ через список управления доступом с подстановочными принципами. Повторного использования принципов следует избегать, если это возможно, чтобы предотвратить предоставление доступа соединениям, использующим более старую версию принципа.

Аудит

Брокеры Kafka могут быть настроены на генерацию полных журналов **log4j** для аудита и отладки. Уровень ведения журнала, а также аппендеры (**appender**), используемые для этого, и параметры их конфигурации можно указать в файле **log4j.properties**. Экземпляры журналов **kafka.authorizer.logger**, применяемый для регистрации авторизации, и **kafka.request.logger**, используемый для регистрации запросов, могут быть заданы независимо друг от друга, чтобы настроить уровень журнала и срок хранения журналов для ведения журнала аудита. В производственных системах для анализа и визуализации этих журналов можно использовать такие фреймворки, как **Elastic Stack**.

Авторизаторы генерируют записи журнала на информационном уровне **INFO** для каждой попытки выполнения операции, для которой доступ был запрещен, и записи журнала на отладочном уровне **DEBUG** для каждой операции, для которой он был разрешен, например:

```
DEBUG Principal = User:Alice is Allowed Operation = Write from host = 127.0.0.1
on resource = Topic:LITERAL:customerOrders for request = Produce with
resourceRefCount
= 1 (kafka.authorizer.logger)
INFO Principal = User:Mallory is Denied Operation = Describe from host =
10.0.0.13 on resource = Topic:LITERAL:customerOrders for request = Metadata
with resourceRefCount = 1 (kafka.authorizer.logger)
```

Журнал запросов, создаваемый на отладочном уровне **DEBUG**, включает также сведения о пользователе и клиентском узле. Полная информация о запросе включается, если регистратор запросов настроен на регистрацию на трассировочном уровне **TRACE**, например:

```
DEBUG Completed request:RequestHeader(apiKey=PRODUCE, apiVersion=8,
clientId=producer-1, correlationId=6) --
```

```
{acks=-1,timeout=30000,partitionSizes=[customerOrders-0=15514]},response:
{responses=[{topic=customerOrders,partition_responses=[{partition=0,error_code=0
,base_offset=13,log_append_time=-1,log_start_offset=0,record_errors=[],error_mes
sage=null}]}]},throttle_time_ms=0} from connection
127.0.0.1:9094-127.0.0.1:61040-0;totalTime:2.42,requestQueueTime:0.112,local-
Time:2.15,remoteTime:0.0,throttleTime:0,responseQueueTime:0.04,sendTime:
0.118,securityProtocol:SASL_SSL,principal:User:Alice,listener:SASL_SSL,clientInf
ormation:ClientInformation(softwareName=apache-kafka-java,
softwareVersion=2.7.0-SNAPSHOT) (kafka.request.logger)
```

Журналы авторизатора и запросов можно проанализировать для выявления подозрительных действий. Показатели, отслеживающие сбои аутентификации, а также журналы сбоев авторизации могут оказаться чрезвычайно полезными для аудита и предоставлять ценную информацию в случае атаки или несанкционированного доступа. Для обеспечения сквозного аудита и отслеживания сообщений метаданные аудита могут быть включены в заголовки сообщений при их создании. Для защиты целостности этих метаданных можно использовать сквозное шифрование.

Обеспечение безопасности ZooKeeper

ZooKeeper хранит метаданные Kafka, критически важные для поддержания доступности кластеров Kafka, поэтому очень важно обеспечить безопасность ZooKeeper в дополнение к безопасности Kafka. ZooKeeper поддерживает аутентификацию с помощью SASL/GSSAPI для аутентификации Kerberos и SASL/DIGEST-MD5 для аутентификации по имени пользователя/паролю. В версии 3.5.0 ZooKeeper также добавлена поддержка TLS, обеспечивающая взаимную аутентификацию, а также шифрование данных при передаче. Обратите внимание на то, что SASL/DIGEST-MD5 следует использовать только с шифрованием TLS и он не подходит для использования в производственной среде из-за известных уязвимостей в безопасности.

SASL

Конфигурация SASL для ZooKeeper обеспечивается с помощью системного свойства Java `java.security.auth.login.config`. Оно должно быть установлено в конфигурационный файл JAAS, который содержит раздел `login` с соответствующим модулем `login` и его параметрами для сервера ZooKeeper. Брокеры Kafka должны быть настроены с секцией `login` на стороне клиента, чтобы клиенты ZooKeeper могли взаимодействовать с серверами ZooKeeper с поддержкой SASL. В следующей секции `Server` представлена конфигурация JAAS для сервера ZooKeeper, чтобы включить проверку подлинности Kerberos:

```
Server {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true storeKey=true
    keyTab="/path/to/zk.keytab"
    principal="zookeeper/zk1.example.com@EXAMPLE.COM";
};
```

Чтобы включить аутентификацию SASL на серверах ZooKeeper, настройте поставщики аутентификации в файле конфигурации ZooKeeper:

```
authProvider.sasl=org.apache.zookeeper.server.auth.SASLAuthenticationProvider
kerberos.removeHostFromPrincipal=true
kerberos.removeRealmFromPrincipal=true
```



Принципал брокера

По умолчанию ZooKeeper использует полный принципал Kerberos, например `kafka/broker1.example.com@EXAMPLE.COM`, в качестве идентификатора клиента. Если для авторизации ZooKeeper включены списки управления доступом, серверы ZooKeeper должны быть настроены со значениями `kerberos.removeHostFromPrincipal=true` и `kerberos.removeRealmFromPrincipal=true`, чтобы гарантировать, что все брокеры имеют одинаковый принципал.

Брокеры Kafka должны быть настроены на аутентификацию в ZooKeeper с использованием SASL с помощью конфигурационного файла JAAS, который предоставляет учетные данные клиента для брокера:

```
Client {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true storeKey=true
    keyTab="/path/to/broker1.keytab"
    principal="kafka/broker1.example.com@EXAMPLE.COM";
};
```

SSL

SSL может быть включен в любой конечной точке ZooKeeper, в том числе тех, которые используют аутентификацию SASL. Как и в Kafka, SSL может быть настроен для включения клиентской аутентификации, но, в отличие от Kafka, соединения как с клиентской аутентификацией SASL, так и с SSL аутентифицируются с помощью обоих протоколов и связывают несколько принципалов с соединением. Авторизатор ZooKeeper предоставляет доступ к ресурсу, если какой-либо из принципалов, связанных с соединением, этот доступ имеет.

Для настройки SSL на сервере ZooKeeper необходимо настроить хранилище ключей с именем хоста сервера или подстановочного хоста. Если включена

аутентификация клиентов, требуется также хранилище доверия для проверки клиентских сертификатов:

```
secureClientPort=2181
serverCnxnFactory=org.apache.zookeeper.server.NettyServerCnxnFactory
authProvider.x509=org.apache.zookeeper.server.auth.X509AuthenticationProvider
ssl.keyStore.location=/path/to/zk.ks.p12
ssl.keyStore.password=zk-ks-password
ssl.keyStore.type=PKCS12
ssl.trustStore.location=/path/to/zk.ts.p12
ssl.trustStore.password=zk-ts-password
ssl.trustStore.type=PKCS12
```

Чтобы настроить SSL для соединений Kafka с ZooKeeper, брокеры должны быть сконфигурированы с хранилищем доверенных сертификатов для проверки сертификатов ZooKeeper. Если включена аутентификация клиента, необходимо также хранилище ключей:

```
zookeeper.ssl.client.enable=true
zookeeper.clientCnxnSocket=org.apache.zookeeper.ClientCnxnSocketNetty
zookeeper.ssl.keystore.location=/path/to/zkclient.ks.p12
zookeeper.ssl.keystore.password=zkclient-ks-password
zookeeper.ssl.keystore.type=PKCS12
zookeeper.ssl.truststore.location=/path/to/zkclient.ts.p12
zookeeper.ssl.truststore.password=zkclient-ts-password
zookeeper.ssl.truststore.type=PKCS12
```

Авторизация

Авторизация может быть включена для узлов ZooKeeper посредством установки списка управления доступом для пути. Когда брокеры настроены с помощью параметра `zookeeper.set.acl=true`, брокер устанавливает список управления доступом для узлов ZooKeeper при создании узла. По умолчанию узлы метаданных доступны для чтения всем, но изменять их могут только брокеры. При необходимости могут быть добавлены дополнительные списки управления доступом для внутренних пользователей-администраторов, которым может понадобиться обновлять метаданные непосредственно в ZooKeeper. Конфиденциальные пути, такие как узлы, содержащие учетные данные SCRAM, по умолчанию недоступны для чтения всем.

Обеспечение безопасности платформы

В предыдущих разделах мы обсудили варианты блокировки доступа к Kafka и ZooKeeper для обеспечения безопасности развертывания Kafka. При проектировании системы безопасности для производственной системы следует использовать модель угроз, которая рассматривает угрозы безопасности не только

для отдельных компонентов, но и для системы в целом. Модели угроз создают абстракцию системы и определяют потенциальные угрозы и связанные с ними риски. После того как угрозы оценены, задокументированы и расставлены по приоритетам на основе рисков, необходимо реализовать стратегии снижения рисков для каждой потенциальной угрозы, чтобы обеспечить защиту всей системы. При оценке потенциальных угроз важно учитывать как внешние, так и внутренние угрозы. Для систем, хранящих идентифицирующие личность данные (ПИ) или другие конфиденциальные сведения, необходимо также реализовать дополнительные меры по соблюдению нормативной политики. Подробное обсуждение стандартных методов моделирования угроз выходит за рамки данной главы.

Помимо защиты данных в Kafka и метаданных в ZooKeeper с помощью безопасной аутентификации, авторизации и шифрования, необходимо предпринять дополнительные шаги для обеспечения безопасности платформы. Средства защиты могут включать в себя сетевые брандмауэры для защиты сети и шифрование для защиты физического хранилища. Хранилища ключей, хранилища доверенных сертификатов и файлы ключей Kerberos, содержащие учетные данные, используемые для аутентификации, должны быть защищены с помощью разрешений файловой системы. Доступ к файлам конфигурации, содержащим критическую для безопасности информацию, такую как учетные данные, должен быть ограничен. Поскольку пароли, хранящиеся в виде открытого текста в файлах конфигурации, небезопасны даже при ограничении доступа, Kafka поддерживает внешнее хранение паролей в защищенном хранилище.

Защита паролей

Настраиваемые поставщики конфигурации могут быть настроены для брокеров и клиентов Kafka на получение паролей из защищенного хранилища паролей сторонних производителей. Пароли также могут храниться в зашифрованном виде в файлах конфигурации с помощью настраиваемых поставщиков конфигурации, которые выполняют дешифровку.

Следующий поставщик пользовательской конфигурации использует инструмент `gpg` для расшифровки свойств брокера или клиента, хранящихся в файле:

```
public class GpgProvider implements ConfigProvider {  
  
    @Override  
    public void configure(Map<String, ?> configs) {}  
  
    @Override  
    public ConfigData get(String path) {  
        try {  
            String passphrase = System.getenv("PASSPHRASE"); ❶  
            String data = Shell.execCommand(  
                "gpg", "--decrypt", "--passphrase", passphrase, path); ❷  
            Properties props = new Properties();
```

```

        props.load(new StringReader(data));
        Map<String, String> map = new HashMap<>();
        for (String name : props.stringPropertyNames())
            map.put(name, props.getProperty(name));
        return new ConfigData(map);
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}

@Override
public ConfigData get(String path, Set<String> keys) {
    ConfigData configData = get(path);
    Map<String, String> data = configData.data().entrySet()
        .stream().filter(e -> keys.contains(e.getKey()))
        .collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue));
    return new ConfigData(data, configData.ttl());
}

@Override
public void close() {}
}

```

❶ Мы предоставляем кодовую фразу для расшифровки паролей процессу в переменной среды `PASSPHRASE`.

❷ Расшифровываем настройки с помощью `gpg`. Возвращаемое значение содержит полный набор расшифрованных настроек.

❸ Мы интерпретируем настройки в `data` как свойства (properties) Java.

❹ При возникновении ошибки быстро завершаем работу с исключением `RuntimeException`.

❺ Вызывающая сторона может запросить подмножество ключей из пути. Здесь мы получаем все значения и возвращаем запрошенное подмножество.

Возможно, вы помните, что в разделе, посвященном SASL/PLAIN, мы использовали стандартные классы конфигурации Kafka для загрузки учетных данных из внешнего файла. Теперь можем зашифровать этот файл с помощью `gpg`:

```

gpg --symmetric --output credentials.props.gpg \
    --passphrase "$PASSPHRASE" credentials.props

```

Далее мы добавляем косвенные настройки и параметры поставщика настроек в исходный файл свойств, чтобы клиенты Kafka загружали свои учетные данные из зашифрованного файла:

```

username=${gpg:/path/to/credentials.props.gpg:username}
password=${gpg:/path/to/credentials.props.gpg:password}
config.providers=gpg
config.providers.gpg.class=com.example.GpgProvider

```

Конфиденциальные параметры конфигурации брокера также можно хранить в зашифрованном виде в ZooKeeper с помощью инструмента конфигурации Kafka без подключения пользовательских поставщиков. Следующая команда может быть выполнена перед запуском брокеров для хранения в ZooKeeper зашифрованных паролей хранилища ключей SSL для брокеров. Секрет кодировщика паролей должен быть настроен в файле конфигурации каждого брокера для расшифровки значения:

```
$ bin/kafka-configs.sh --zookeeper localhost:2181 --alter \
  --entity-type brokers --entity-name 0 --add-config \
  'listener.name.external.ssl.keystore.password=server-ks-
  password,password.encoder.secret=encoder-secret'
```

Резюме

В течение последнего десятилетия частота и масштабы утечек данных увеличиваются, поскольку кибератаки становятся все более изощренными. Помимо значительных затрат на выявление и устранение нарушений, а также затрат на перерывы в работе до тех пор, пока не будут применены исправления безопасности, утечки данных могут также привести к штрафным санкциям со стороны регулирующих органов и долгосрочному ущербу для репутации бренда. В этой главе мы рассмотрели широкий спектр возможностей, доступных для обеспечения конфиденциальности, целостности и доступности данных, хранящихся в Kafka.

Возвращаясь к примеру потока данных в начале этой главы, мы рассмотрели варианты, доступные для различных аспектов безопасности на протяжении всего потока.

- *Подлинность клиента.* Когда клиент Алисы устанавливает соединение с брокером Kafka, приемник, использующий протокол SASL или SSL с аутентификацией клиента, может убедиться, что соединение действительно происходит от Алисы, а не от самозванца. Повторная аутентификация может быть настроена для ограничения доступа в случае компрометации пользователя.
- *Подлинность сервера.* Клиент Алисы может убедиться, что его соединение установлено с настоящим брокером, используя протокол SSL с проверкой имени хоста или механизмы SASL с взаимной аутентификацией, такие как Kerberos или SCRAM.
- *Конфиденциальность данных.* Использование механизма SSL для шифрования данных при передаче защищает их от перехватчиков. Шифрование дисков или томов защищает данные в состоянии покоя, даже если диск украден. Для особо важных данных сквозное шифрование обеспечивает детальный контроль доступа и гарантирует, что поставщики облачных услуг

и администраторы платформы, имеющие физический доступ к сети и дискам, не смогут получить доступ к данным.

- *Целостность данных.* Механизм SSL можно использовать для обнаружения несанкционированного доступа к данным в незащищенной сети. Цифровые подписи могут быть включены в сообщения для проверки целостности с помощью сквозного шифрования.
- *Контроль доступа.* Каждая операция, выполняемая Алисой, Бобом и даже брокерами, авторизуется с помощью настраиваемого авторизатора. В Kafka есть встроенный авторизатор, который позволяет осуществлять детальный контроль доступа с помощью списков управления доступом.
- *Контролируемость.* Журналы авторизатора и журналы запросов могут использоваться для отслеживания операций и попыток выполнения операций для аудита и обнаружения аномалий.
- *Доступность.* Для защиты брокеров от атак типа «отказ в обслуживании» можно использовать комбинацию квот и параметров конфигурации для управления соединениями. ZooKeeper может быть защищен с помощью SSL, SASL и ACL, чтобы обеспечить безопасность метаданных, необходимых для обеспечения доступности брокеров Kafka.

Из-за большого выбора доступных вариантов обеспечения безопасности выбор подходящих вариантов для каждого сценария использования может оказаться непростой задачей. Мы рассмотрели проблемы безопасности, которые необходимо учитывать для каждого механизма безопасности, а также средства управления и политики, которые можно использовать для ограничения потенциальной поверхности атаки. Мы также рассмотрели дополнительные меры, необходимые для блокировки ZooKeeper и остальной части платформы. Стандартные технологии безопасности, поддерживаемые Kafka, и различные точки расширения для интеграции с существующей инфраструктурой безопасности в вашей организации позволяют создавать согласованные решения безопасности для защиты всей платформы.

Администрирование Kafka

Для управления кластером Kafka требуются дополнительные инструменты для выполнения административных изменений в топиках, конфигурациях и многом другом. Kafka предоставляет пользователям несколько утилит командной строки (CLI), удобных для администрирования кластеров. Они реализованы в виде классов Java, для правильного вызова которых изначально имеются наборы сценариев. Эти утилиты позволяют выполнять простейшие действия, более сложные операции реализовать с их помощью нельзя, или они будут слишком громоздкими для использования в больших масштабах. В этой главе мы опишем только доступные в проекте Apache Kafka основные утилиты. Дополнительную информацию о более продвинутых утилитах, созданных сообществом разработчиков вне рамок основного проекта Kafka, можно найти на сайте Apache Kafka (<https://kafka.apache.org>).



Авторизация административных операций

В Apache Kafka реализованы аутентификация и авторизация для управления операциями с топиками, конфигурации по умолчанию не ограничивают применение этих инструментов. Это значит, что вышеупомянутые утилиты командной строки можно использовать без всякой аутентификации, то есть такие операции, как изменение топика, можно выполнять без какой-либо проверки на безопасность или аудита. Всегда следите за тем, чтобы доступ к этому инструментарию в ваших развертываниях был ограничен только администраторами, чтобы предотвратить несанкционированные изменения.

Операции с топиками

Утилита `kafka-topics.sh` позволяет легко выполнить большинство операций с топиками. Она дает возможность создавать, менять, удалять и выводить информацию об имеющихся в кластере топиках. Хотя некоторые настройки топиков возможны с помощью этой команды, они устарели, и для их изменения рекомендуется использовать более надежный метод с помощью инструмента `kafka-config.sh`. Чтобы использовать команду `kafka-topics.sh`, вы должны указать

строку подключения кластера и порт с помощью параметра `--bootstrap-server`. В следующих примерах строка подключения к кластеру выполняется локально на одном из хостов кластера Kafka, и мы будем использовать `local host:9092`.

На протяжении всей этой главы все инструменты будут находиться в каталоге `/usr/local/kafka/bin/`. Примеры команд в этом разделе будут предполагать, что вы находитесь в этом каталоге или добавили его в свой `$PATH`.



Проверьте версию

Корректная работа многих инструментов командной строки для Kafka зависит от версии Kafka. К ним относятся некоторые команды, которые могут хранить данные в ZooKeeper, а не подключаться к брокерам. Поэтому важно проверять соответствие версии используемых утилит версии брокеров в кластере. Безопаснее всего запускать версии этих утилит, установленные на самих брокерах Kafka.

Создание нового топика

При создании нового топика с помощью команды `--create` существует несколько обязательных аргументов. Их следует указывать, несмотря на то что для некоторых из них могут быть заданы значения по умолчанию на уровне брокера. Дополнительные аргументы и переопределение конфигурации возможны и в настоящее время с помощью параметра `--config`, но они рассматриваются позже в этой главе. Далее приведен список трех обязательных аргументов:

- `--topic` — название создаваемого топика;
- `--replication-factor` — число реплик топика в кластере;
- `--partitions` — число создаваемых для данного топика разделов.



Хорошие практики именования топиков

Названия топиков могут содержать алфавитно-цифровые символы, символы подчеркивания, тире и точки, однако не рекомендуется использовать точки в именах топиков. Внутренние показатели в Kafka преобразуют символы точек в символы подчеркивания (например, `topic.1` становится `topic_1` при расчете показателей), что может привести к конфликтам в названиях топиков.

Еще одна рекомендация — избегать двойного подчеркивания в начале названия топика. По соглашению об именовании топика для внутренних операций Kafka создаются с двойным подчеркиванием в названии (например, топик `__consumer_offsets`, который отслеживает хранение смещений групп потребителей). Поэтому не рекомендуется использовать названия топиков, начинающиеся с двойного подчеркивания, чтобы избежать путаницы.

Создать новый топик очень просто. Запустите сценарий `kafka-topics.sh` следующим образом:

```
kafka-topics.sh --zookeeper <zookeeper connect> --create --topic <string>
--replication-factor <integer> --partitions <integer>
#
```

В результате кластер создаст топик с заданными названием и числом разделов. Для каждого раздела он подберет заданное число подходящих реплик. Это значит, что если кластер настроен для распределения реплик с учетом стоек, то реплики разделов будут находиться в отдельных стойках. Если же такое поведение нежелательно, укажите аргумент командной строки `--disable-rack-aware`.

Например, создадим топик `my-topic` с восемью разделами, в каждом из которых по две реплики:

```
# kafka-topics.sh --zookeeper zoo1.example.com:2181/kafka-cluster --create
--topic my-topic --replication-factor 2 --partitions 8
Created topic "my-topic".
#
```



Правильное использование аргументов `if-exists` и `if-not-exists`

При использовании сценария `kafka-topics.sh` для автоматизации может оказаться полезным аргумент `--if-not-exists`, для которого при создании новых топиков не будет возвращаться ошибка, если топик уже существует.

Хотя аргумент `--if-exists` предусмотрен для команды `--alter`, использовать его не рекомендуется. Использование этого аргумента приведет к тому, что команда не вернет ошибку, если изменяемого топика не существует. Это может маскировать проблемы, когда не существует топика, который должен был быть создан.

Вывод списка всех топиков в кластере

Команда `--list` выводит список всех топиков в кластере. Список форматируется по одному топiku в строке без определенного порядка, что полезно для создания полного списка топиков.

Вот пример команды `--list`, которая выводит список всех топиков в кластере:

```
# kafka-topics.sh --bootstrap-server localhost:9092 --list
__consumer_offsets
my-topic
other-topic
```

Вы можете заметить, что здесь указан внутренний топик `consumer_offsets`. Выполнение команды с параметром `--exclude-internal` удалит из списка все топик, начинающиеся с двойного подчеркивания, о котором говорилось ранее, что может быть полезно.

Подробное описание топиков

Можно также получить подробную информацию по одному или нескольким топикам кластера. Выводимая информация включает число разделов, переопределения настроек топиков и список разделов с распределением реплик. Можно ограничиться информацией по одному топик, указав для команды аргумент `--topic`.

Например, выведем описание недавно созданного в кластере топика `my-topic`:

```
# kafka-topics.sh --bootstrap-server localhost:9092 --describe --topic my-topic
Topic: my-topic PartitionCount: 8      ReplicationFactor: 2      Configs: seg
ment.bytes=1073741824
    Topic: my-topic Partition: 0      Leader: 1      Replicas: 1,0      Isr: 1,0
    Topic: my-topic Partition: 1      Leader: 0      Replicas: 0,1      Isr: 0,1
    Topic: my-topic Partition: 2      Leader: 1      Replicas: 1,0      Isr: 1,0
    Topic: my-topic Partition: 3      Leader: 0      Replicas: 0,1      Isr: 0,1
    Topic: my-topic Partition: 4      Leader: 1      Replicas: 1,0      Isr: 1,0
    Topic: my-topic Partition: 5      Leader: 0      Replicas: 0,1      Isr: 0,1
    Topic: my-topic Partition: 6      Leader: 1      Replicas: 1,0      Isr: 1,0
    Topic: my-topic Partition: 7      Leader: 0      Replicas: 0,1      Isr: 0,1
#
```

У команды `describe` есть несколько полезных параметров для фильтрации выводимой информации. Они могут пригодиться при диагностике проблем с кластером. В случае их использования не задавайте аргумент `-topic`, поскольку смысл состоит в том, чтобы найти все топик или разделы кластера, соответствующие заданному критерию. Эти параметры не работают с командой `list`, описанной в предыдущем разделе. Вот список полезных аргументов.

- `--topics-with-overrides`. В этом аргументе будут описаны только те топик, чьи настройки отличаются от настроек кластера по умолчанию.
- `--exclude-internal`. Вышеупомянутая команда удалит из списка все топик, имена которых начинаются с двойного подчеркивания.

Следующие команды используются для поиска разделов топиков, в которых могут возникнуть проблемы.

- `--under-replicated-partitions`. Этот аргумент выведет все разделы, в которых одна или несколько реплик не синхронизированы с лидером. Это не обязательно плохо, поскольку обслуживание, развертывание и перебалансировка

кластера могут привести к появлению недостаточно реплицированных разделов (или URP), но это то, о чем следует знать.

- **--at-min-isr-partitions.** Этот аргумент выводит все разделы, в которых количество реплик, включая лидера, точно соответствует настройке для минимального количества синхронизированных реплик (ISR). Эти топика по-прежнему доступны для клиентов производителей или потребителей, но вся избыточность утрачена и они могут стать недоступными.
- **--under-min-isr-partitions.** Этот аргумент показывает все разделы, в которых количество ISR меньше настроенного минимума для успешного выполнения действий по производству. Эти разделы фактически находятся в режиме только для чтения, и в них нельзя выполнять производство.
- **--unavailable-partitions.** Этот аргумент выводит все разделы, у которых нет ведущей реплики. Это серьезная проблема, означающая, что раздел в настоящий момент находится в автономном режиме и недоступен для клиентов-потребителей и клиентов-производителей.

Вот пример поиска топиков с минимальными настройками ISR. В этом примере топик настроен на минимальное значение ISR, равное 1, и имеет коэффициент репликации (RF), равный 2. Хост 0 находится в режиме онлайн, а хост 1 отключен для обслуживания:

```
# kafka-topics.sh --bootstrap-server localhost:9092 --describe --at-min-isr-
partitions
    Topic: my-topic Partition: 0      Leader: 0      Replicas: 0,1  Isr: 0
    Topic: my-topic Partition: 1      Leader: 0      Replicas: 0,1  Isr: 0
    Topic: my-topic Partition: 2      Leader: 0      Replicas: 0,1  Isr: 0
    Topic: my-topic Partition: 3      Leader: 0      Replicas: 0,1  Isr: 0
    Topic: my-topic Partition: 4      Leader: 0      Replicas: 0,1  Isr: 0
    Topic: my-topic Partition: 5      Leader: 0      Replicas: 0,1  Isr: 0
    Topic: my-topic Partition: 6      Leader: 0      Replicas: 0,1  Isr: 0
    Topic: my-topic Partition: 7      Leader: 0      Replicas: 0,1  Isr: 0
#
```

Добавление разделов

Иногда оказывается нужно увеличить количество разделов топика. Топики масштабируются и реплицируются в кластере посредством разделов. Наиболее распространенной причиной увеличения количества разделов является горизонтальное масштабирование топика на большее количество брокеров за счет уменьшения пропускной способности для одного раздела. Можно также увеличивать топика, если требуется несколько экземпляров одного и того же потребителя в рамках одной группы потребителей, поскольку только один участник группы потребителей может читать один раздел.

Далее приведен пример увеличения количества разделов для топика `my-topic` до 16 с помощью команды `--alter`, а также проверка того, что это сработало:

```
# kafka-topics.sh --bootstrap-server localhost:9092
--alter --topic my-topic --partitions 16
# kafka-topics.sh --bootstrap-server localhost:9092 --describe --topic my-topic
Topic: my-topic PartitionCount: 16      ReplicationFactor: 2    Configs: seg
ment.bytes=1073741824
    Topic: my-topic Partition: 0      Leader: 1      Replicas: 1,0  Isr: 1,0
    Topic: my-topic Partition: 1      Leader: 0      Replicas: 0,1  Isr: 0,1
    Topic: my-topic Partition: 2      Leader: 1      Replicas: 1,0  Isr: 1,0
    Topic: my-topic Partition: 3      Leader: 0      Replicas: 0,1  Isr: 0,1
    Topic: my-topic Partition: 4      Leader: 1      Replicas: 1,0  Isr: 1,0
    Topic: my-topic Partition: 5      Leader: 0      Replicas: 0,1  Isr: 0,1
    Topic: my-topic Partition: 6      Leader: 1      Replicas: 1,0  Isr: 1,0
    Topic: my-topic Partition: 7      Leader: 0      Replicas: 0,1  Isr: 0,1
    Topic: my-topic Partition: 8      Leader: 1      Replicas: 1,0  Isr: 1,0
    Topic: my-topic Partition: 9      Leader: 0      Replicas: 0,1  Isr: 0,1
    Topic: my-topic Partition: 10     Leader: 1      Replicas: 1,0  Isr: 1,0
    Topic: my-topic Partition: 11     Leader: 0      Replicas: 0,1  Isr: 0,1
    Topic: my-topic Partition: 12     Leader: 1      Replicas: 1,0  Isr: 1,0
    Topic: my-topic Partition: 13     Leader: 0      Replicas: 0,1  Isr: 0,1
    Topic: my-topic Partition: 14     Leader: 1      Replicas: 1,0  Isr: 1,0
    Topic: my-topic Partition: 15     Leader: 0      Replicas: 0,1  Isr: 0,1
#
```



Тонкая настройка топиков с ключами

С точки зрения потребителей, добавление разделов в топик, содержащие сообщения с ключами, может оказаться весьма непростой задачей. Дело в том, что соответствие ключей разделам может меняться при смене числа разделов. Поэтому рекомендуется задавать число разделов для топиков, содержащих сообщения с ключами, однократно при их создании и стараться не менять их размер.

Уменьшение количества разделов

Уменьшить количество разделов топика невозможно. Удаление раздела из топика привело бы к удалению части его данных и несогласованности с точки зрения клиента. Кроме того, попытка перераспределения данных по оставшимся разделам — задача непростая, чреватая неправильным упорядочением сообщений. При необходимости уменьшить число разделов рекомендуется удалить топик и создать его заново или, если удаление невозможно, создать новую версию существующего топика и переместить весь трафик производства в новый топик, например `my-topic-v2`.

Удаление топика

Даже не содержащий сообщений топик расходует ресурсы кластера, такие как дисковое пространство, открытые дескрипторы файлов и оперативная память. У контроллера также есть ненужные метаданные, информацию о которых он должен сохранять, что может снизить производительность в больших масштабах. Если топик больше не нужен, следует его удалить, чтобы освободить эти ресурсы. Для выполнения этого действия параметр конфигурации брокеров кластера `delete.topic.enable` должен быть равен `true`. Если же этот параметр установлен в `false`, запрос на удаление топиков будет проигнорирован и не выполнится.

Удаление топика — асинхронная операция. Это означает, что ее выполнение пометит топик для удаления, но удаление может произойти не сразу, в зависимости от объема данных и необходимой очистки. Контроллер уведомит брокеры о предстоящем удалении как можно скорее (после завершения существующих задач контроллера), после чего брокеры аннулируют метаданные топика и удалят файлы с диска. Настоятельно рекомендуется, чтобы операторы не удаляли более одного или двух топиков одновременно и давали им достаточно времени для завершения перед удалением других топиков из-за ограничений в способе выполнения этих операций контроллером. В небольшом кластере, показанном в примерах в этой книге, удаление топиков произойдет почти сразу, но в больших кластерах это может занять больше времени.



Данные будут утеряны

Удаление топика также приводит к удалению всех его сообщений. Эта операция необратима. Выполняйте ее осторожно.

Вот пример удаления топика `my-topic` с использованием аргумента `--delete`. В зависимости от версии `Kafka` появится примечание, сообщающее вам, что аргумент не будет работать, если не задана другая конфигурация:

```
# kafka-topics.sh --bootstrap-server localhost:9092
--delete --topic my-topic
```

```
Note: This will have no impact if delete.topic.enable is not set
to true.
#
```

Как видите, что нет никакого явного сообщения о том, успешно ли было завершено удаление топика. Убедитесь, что удаление прошло успешно, запустив `--list` или `--describe`, чтобы увидеть, что топик больше не находится в кластере.

Группы потребителей

Группы потребителей — это скоординированные группы потребителей Kafka, потребляющих данные из топиков или нескольких разделов одного топика. Инструмент `kafka-consumer-groups.sh` помогает получать представление о группах потребителей, которые потребляют данные из топиков в кластере, и управлять ими. Его можно использовать для вывода списка групп потребителей, описания конкретных групп, удаления групп потребителей или информации о конкретных группах, а также для сброса информации о смещении групп потребителей.



Группы потребителей в ZooKeeper

В более старых версиях Kafka управлять группами потребителей и поддерживать их можно было в ZooKeeper. Это поведение устарело в версиях 0.11.0.* и более поздних, и старые группы потребителей больше не используются. Некоторые версии предоставленных сценариев все еще могут отображать устаревшие команды строки подключения `--zookeeper`, но их не рекомендуется использовать, если только у вас нет старой среды с некоторыми группами потребителей, которые не обновились до более поздних версий Kafka.

Вывод списка и описание групп

Для вывода списка групп потребителей воспользуйтесь параметрами `--bootstrap-server` и `--list`. Специальные потребители, использующие сценарий `kafka-consumer-groups.sh`, будут отображаться в списке потребителей как `console-consumer-<сгенерированный_id>`:

```
# kafka-consumer-groups.sh --bootstrap-server localhost:9092 --list
console-consumer-95554
console-consumer-9581
my-consumer
#
```

Более подробное описание любой из перечисленных групп можно получить, заменив параметр `--list` на `--describe` и добавив параметр `--group`. В результате этого будут выведены все топика и разделы, которые читает группа, а также дополнительная информация, такая как смещения для всех разделов топиков. В табл. 12.1 приведено полное описание всех полей, представленных в выходных данных.

Например, выведем подробную информацию о специальной группе потребителей с названием `my-consumer`:

```
# kafka-consumer-groups.sh --bootstrap-server localhost:9092
--describe --group my-consumer
```

```

GROUP          TOPIC          PARTITION  CURRENT-OFFSET  LOG-END-OFFSET
LAG            CONSUMER-ID
HOST
my-consumer    my-topic          0           2                4
2              consumer-1-029af89c-873c-4751-a720-cefd41a669d6 /
127.0.0.1      consumer-1
my-consumer    my-topic          1           2                3
1              consumer-1-029af89c-873c-4751-a720-cefd41a669d6 /
127.0.0.1      consumer-1
my-consumer    my-topic          2           2                3
1              consumer-2-42c1abd4-e3b2-425d-a8bb-e1ea49b29bb2 /
127.0.0.1      consumer-2
#

```

Таблица 12.1. Поля вывода информации о группе потребителей с названием my-consumer

Поле	Описание
GROUP	Название группы потребителей
TOPIC	Название читаемого топика
PARTITION	Идентификатор читаемого раздела
CURRENT-OFFSET	Следующее смещение, которое будет потреблено группой потребителей для данного раздела топика. Представляет собой позицию потребителя в разделе
LOG-END-OFFSET	Текущее максимальное смещение для данного раздела топика из имеющихся в брокере. Это смещение следующего сообщения, которое будет выдано этому разделу
LAG	Разница между Current-Offset потребителя и Log-End-Offset брокера для данного раздела топика
CONSUMER-ID	Сгенерированный уникальный идентификатор потребителя на основе предоставленного идентификатора клиента
HOST	Адрес хоста, с которого читает данные группа потребителей
CLIENT-ID	Строка, предоставляемая клиентом, идентифицирующая клиента, который потребляет из группы

Удаление группы

Удаление групп потребителей может быть выполнено с помощью аргумента `--delete`. Это действие приводит к удалению всей группы, включая все сохраненные смещения для всех потребляемых группой топиков. Чтобы выполнить удаление, необходимо прекратить работу всех потребителей группы, поскольку в ней не должно быть активных членов. Если вы попытаетесь удалить не пустую группу, будет выдана ошибка «Группа не пустая» (The group is not empty) и ничего не произойдет. Ту же самую команду можно использовать и для удаления смещений для отдельного читаемого группой топика, не удаляя всю группу, добавив аргумент `--topic` и указав, какие смещения топика нужно удалить.

Далее приведен пример удаления целой группы потребителей под названием `my-consumer`:

```
# kafka-consumer-groups.sh --bootstrap-server localhost:9092 --delete -group
my-consumer
Deletion of requested consumer groups ('my-consumer') was successful.
#
```

Управление смещениями

Помимо отображения и удаления смещений групп потребителей, существует возможность извлечения этих смещений и сохранения их в пакете. Это может пригодиться для сброса значения смещений конкретного потребителя в случае возникновения проблемы, из-за которой нужно будет перечитать сообщения или перескочить через проблемное сообщение, например плохо отформатированное, обработать которое у потребителя не получается.

Экспорт смещений

Для экспорта смещений из группы потребителей в файл CSV используйте аргумент `--reset-offsets` с параметром `--dry-run`. Это позволит сформировать экспорт текущих смещений в формате файла, который позже можно повторно использовать для импорта или отката смещений. Экспортированный файл в формате CSV будет иметь следующую структуру:

<название топика>, <номер раздела>, <смещение>

Выполнение той же команды без параметра `--dry-run` приведет к полному сбросу смещений, поэтому будьте осторожны.

Вот пример экспорта смещений для топика `my-topic`, который потребляется группой потребителей с именем `my-consumer`, в файл `offsets.csv`:

```
# kafka-consumer-groups.sh --bootstrap-server localhost:9092
--export --group my-consumer --topic my-topic
--reset-offsets --to-current --dry-run > offsets.csv

# cat offsets.csv
my-topic,0,8905
my-topic,1,8915
my-topic,2,9845
my-topic,3,8072
my-topic,4,8008
my-topic,5,8319
my-topic,6,8102
my-topic,7,12739
#
```

Импорт смещений

Утилита импорта смещений представляет собой противоположность утилиты экспорта. Она принимает на входе файл, полученный в результате работы утилиты экспорта из предыдущего раздела, и устанавливает на его основе текущие смещения группы потребителей. Общепринятая практика — экспортировать текущие смещения группы потребителей, чтобы сделать резервную копию файла, и вносить в файл изменения, задавая для смещений нужные значения.



Предварительно остановите работу потребителей

Перед выполнением этого шага нужно остановить все потребители группы. Они не будут читать новые смещения, записываемые в ходе работы группы потребителей, а просто переключаются на импортированные.

В следующем примере мы импортируем смещения группы потребителей `my-consumer` из созданного в предыдущем примере файла с названием `offsets.csv`:

```
# kafka-consumer-groups.sh --bootstrap-server localhost:9092
--reset-offsets --group my-consumer
--from-file offsets.csv --execute
```

TOPIC	PARTITION	NEW-OFFSET
my-topic	0	8905
my-topic	1	8915
my-topic	2	9845
my-topic	3	8072
my-topic	4	8008
my-topic	5	8319
my-topic	6	8102
my-topic	7	12739

```
#
```

Динамические изменения конфигурации

Существует множество конфигураций для топиков, клиентов, брокеров и много другого, которые можно динамически обновлять во время работы без необходимости отключения или повторного развертывания кластера. `kafka-configs.sh` является основным инструментом для изменения этих конфигураций. В настоящее время существует четыре основных категории, или типа сущностей, динамических конфигураций, которые можно изменять: *топики*, *брокеры*, *пользователи* и *клиенты*. Для каждого типа сущностей существуют определенные конфигурации, которые могут быть переопределены. Новые динамические конфигурации добавляются с каждым выпуском Kafka, поэтому полезно убедиться, что у вас есть версия этого инструмента, соответствующая

версии Kafka, которую вы используете. Для упрощения последовательной настройки этих конфигураций с помощью автоматизации аргумент `--add-config-file` можно использовать с предварительно отформатированным файлом всех конфигураций, которыми вы хотите управлять и которые хотите обновлять.

Переопределение значений настроек топиков по умолчанию

Существует множество конфигураций, которые устанавливаются по умолчанию для топиков, определенных в файлах конфигурации статического брокера (например, политика времени хранения). С помощью динамических конфигураций мы можем переопределить значения по умолчанию на уровне кластера для отдельных топиков, чтобы сочетать различные сценарии использования в одном кластере. В табл. 12.2 показаны допустимые ключи конфигурации для топиков, которые могут быть изменены динамически.

Формат команды изменения настроек топика:

```
kafka-configs.sh --bootstrap-server localhost:9092
--alter --entity-type topics --entity-name <topic-name>
--add-config <key>=<value>[,<key>=<value>...]
```

Вот пример установки длительности хранения для топика `my-topic` на 1 час (3 600 000 мс):

```
# kafka-configs.sh --bootstrap-server localhost:9092
--alter --entity-type topics --entity-name my-topic
--add-config retention.ms=3600000
Updated config for topic: "my-topic".
#
```

Таблица 12.2. Допустимые ключи (конфигурации) топиков

Ключ конфигурации	Описание
<code>cleanup.policy</code>	При значении <code>compact</code> сообщения топика будут отбрасываться и из сообщений с заданным ключом будет сохраняться только самое последнее (сжатые журналы)
<code>compression.type</code>	Тип сжатия, используемый брокером при записи на диск пакетов сообщений для данного топика
<code>delete.retention.ms</code>	Длительность (в миллисекундах) хранения отметок об удалении для данного топика. Имеет смысл только для топиков со сжатием журналов

Ключ конфигурации	Описание
file.delete.delay.ms	Длительность (в миллисекундах) ожидания перед удалением сегментов журнала и индексов для данного топика с диска
flush.messages	Количество сообщений, которое может быть получено, прежде чем будет выполнен принудительный сброс сообщений данного топика на диск
flush.ms	Промежуток времени (в миллисекундах) перед принудительным сбросом сообщений данного топика на диск
follower.replication.throttled.replicas	Список реплик, для которых репликация журнала должна регулироваться подписчиком
index.interval.bytes	Допустимое количество байтов сообщений, генерируемых между записями индекса сегмента журнала
leader.replication.throttled.replica	Список реплик, для которых репликация журнала должна регулироваться лидером
max.compaction.lag.ms	Максимальное время, в течение которого сообщение не будет сжиматься в журнале
max.message.bytes	Максимальный размер отдельного сообщения данного топика (в байтах)
message.downconversion.enable	Позволяет преобразовывать версию формата сообщения к предыдущей версии, если это разрешено, с некоторыми накладными расходами
message.format.version	Используемая брокером при записи сообщений на диск версия формата сообщений. Должна представлять собой допустимую версию API
message.timestamp.difference.max.ms	Максимально допустимая разница (в миллисекундах) между меткой даты/времени отправки сообщения и меткой даты/времени брокера о его получении. Допустимо только в случае, если ключ message.timestamp.type равен CreateTime
message.timestamp.type	Используемая при записи сообщений на диск метка даты/времени. Текущие значения — CreateTime для задаваемой клиентом метки даты/времени и LogAppendTime при записи сообщения в раздел брокером
min.cleanable.dirty.ratio	Частота попыток сжатия разделов данного топика утилитой сжатия журналов (в виде отношения числа несжатых сегментов журнала к общему числу сегментов). Имеет смысл только для топиков со сжатием журналов
min.compaction.lag.ms	Минимальное время, в течение которого сообщение будет оставаться в журнале в несжатом виде
min.insync.replicas	Минимальное число согласованных реплик, необходимое для того, чтобы раздел топика считался доступным

Таблица 12.2 (продолжение)

Ключ конфигурации	Описание
<code>preallocate</code>	При установке в <code>true</code> место под сегменты журналов для этого топика будет выделяться заранее, при создании нового сегмента
<code>retention.bytes</code>	Объем хранимых сообщений этого топика (в байтах)
<code>retention.ms</code>	Длительность (в миллисекундах) хранения сообщений данного топика
<code>segment.bytes</code>	Объем сообщений (в байтах), записываемый в отдельный сегмент журнала в разделе
<code>segment.index.bytes</code>	Максимальный размер (в байтах) отдельного индекса сегмента журнала
<code>segment.jitter.ms</code>	Максимальное число миллисекунд, задаваемых случайным образом и добавляемых к ключу <code>segment.ms</code> при создании сегментов журналов
<code>segment.ms</code>	Частота (в миллисекундах) чередования сегментов журналов для каждого из разделов
<code>unclean.leader.election.enable</code>	При установке в <code>false</code> для данного топика будет запрещен «нечистый» выбор ведущей реплики

Переопределение настроек клиентов и пользователей по умолчанию

Для клиентов и пользователей Kafka существует всего несколько конфигураций, которые можно переопределить, и все они, по сути, являются разновидностями квот. Две наиболее часто изменяемые конфигурации — это скорость передачи (байт/с), разрешенная для производителей и потребителей с указанным идентификатором клиента на основе каждого брокера. Полный список общих конфигураций, которые могут быть изменены как для пользователей, так и для клиентов, приведен в табл. 12.3.



Неравномерное регулирование в плохо сбалансированных кластерах

Поскольку регулирование происходит для каждого брокера, равномерный баланс лидерства разделов в кластере становится особенно важным для его правильного применения. Если вы задаете квоту производителя 10 Мбайт/с на клиент для кластера с пятью брокерами, то этот клиент сможет генерировать 10 Мбайт/с данных на каждом из брокеров, что в итоге составит 50 Мбайт/с, предполагая сбалансированное лидерство на всех пяти хостах. Однако, если лидерство для каждого раздела принадлежит брокеру 1, один и тот же производитель сможет производить не более 10 Мбайт/с.

Таблица 12.3. Настройки (ключи конфигурации) клиентов

Ключ конфигурации	Описание
consumer_bytes_rate	Допустимый объем сообщений, потребляемых из одного брокера в секунду по отдельному идентификатору клиента (в байтах)
producer_bytes_rate	Допустимый объем сообщений, генерируемых для одного брокера в секунду по отдельному идентификатору клиента (в байтах)
controller_mutations_rate	Скорость принятия изменений для запросов на создание топиков, создание разделов и удаление топиков. Скорость суммируется по количеству созданных или удаленных разделов
request_percentage	Процентное соотношение для каждого окна квоты (из общего количества $(\text{num.io.threads} + \text{num.network.threads}) \times 100 \%$) для запросов от пользователя или клиента



Идентификаторы клиентов и группы потребителей

Идентификаторы клиентов — далеко не всегда то же самое, что название группы потребителей. Потребители могут задавать собственные идентификаторы клиентов, так что вполне возможно существование в разных группах нескольких потребителей с одинаковым идентификатором клиента. Рекомендуется задавать для каждой группы потребителей уникальный идентификатор клиента, причем каким-то образом идентифицирующий ее. Благодаря этому группа потребителей сможет использовать квоту совместно, а поиск в журналах сведений о том, какая группа отвечает за запрос, значительно упростится.

Совместимые изменения конфигурации пользователя и клиента могут быть указаны вместе для совместимых конфигураций, которые применяются к обоим. Вот пример команды для изменения скорости мутации контроллера для пользователя и клиента за один шаг конфигурации:

```
# kafka-configs.sh --bootstrap-server localhost:9092
--alter --add-config "controller_mutations_rate=10"
--entity-type clients --entity-name <client ID>
--entity-type users --entity-name <user ID>
#
```

Переопределение настроек конфигурации брокера по умолчанию

Конфигурации на уровне брокера и кластера в основном задаются статически в конфигурационных файлах кластера, но существует множество конфигураций, которые могут быть переопределены во время выполнения без необходимости повторного развертывания Kafka. Более 80 переопределений можно изменить

для брокеров с помощью `kafka-configs.sh`. Мы не будем перечислять все в этой книге, их можно найти в команде `--help` или в документации с открытым исходным кодом (<https://oreil.ly/R8hhb>). Вот несколько важных конфигураций, на которые стоит обратить особое внимание:

- `min.insync.replicas`. Настраивает минимальное количество реплик, которые должны подтвердить запись, чтобы запрос на производство был успешным, если производители установили параметру `acks` значение `all` (или `-1`);
- `unclean.leader.election.enable`. Позволяет репликам быть избранными в качестве лидера, даже если это приводит к потере данных. Это полезно, когда допустимо иметь некоторые потери данных или включать на короткое время, чтобы отключить кластер Kafka, если невозможно избежать безвозвратной потери данных;
- `max.connections`. Максимальное количество подключений к брокеру, разрешенных в любой момент времени. Мы также можем использовать параметры `max.connections.per.ip` и `max.connections.per.ip.overrides` для более точной настройки регулирования.

Описание переопределений настроек

С помощью утилиты `kafka-config.sh` можно вывести список всех переопределений настроек и просмотреть конкретные настройки топика, брокера или клиента. Как и в остальных утилитах, здесь для этого используется команда `--describe`.

В следующем примере мы можем получить все переопределения настроек для топика `my-topic`, которые, по нашим наблюдениям, являются только временем хранения:

```
# kafka-configs.sh --bootstrap-server localhost:9092
--describe --entity-type topics --entity-name my-topic
Configs for topics:my-topic are
retention.ms=3600000
#
```



Только переопределения настроек топика

Описание конфигурации выводит только переопределения и не включает настройки кластера по умолчанию. Не существует способа динамически выяснить настройки самих брокеров. Это значит, что в случае использования в ходе автоматизации вышеупомянутой утилиты для выяснения настроек топика или клиента необходимо отдельно предоставить пользователю информацию о настройках кластера по умолчанию.

Удаление переопределений настроек

Можно полностью удалить динамические настройки, в результате чего объект возвратится к настройкам кластера по умолчанию. Для удаления переопределений настроек используется команда `--alter` с параметром `--delete-config`.

Например, удалим переопределения настроек ключа `retention.ms` для `my-topic`:

```
# kafka-configs.sh --bootstrap-server localhost:9092
--alter --entity-type topics --entity-name my-topic
--delete-config retention.ms
Updated config for topic: "my-topic".
#
```

Производство и потребление

Работая с Kafka, вы часто будете сталкиваться с необходимостью вручную создавать или потреблять некоторые примеры сообщений, чтобы проверить, что происходит с вашими приложениями. Для этого предусмотрены две утилиты, `kafka-console-consumer.sh` и `kafka-console-producer.sh`, которые были кратко рассмотрены в главе 2 для проверки нашей установки. Эти утилиты являются обертками основных клиентских библиотек Java, которые позволяют вам взаимодействовать с топиками Kafka без необходимости писать для этого целое приложение.



Передача выходных данных другому приложению

Хотя можно писать приложения, которые будут обертками консоли потребителя или производителя (например, потреблять сообщения и передавать их по конвейеру другому приложению для обработки), такой тип приложений является довольно хрупким и его следует избегать. Трудно взаимодействовать с консольным потребителем таким образом, чтобы не терялись сообщения. Аналогично консольный производитель не позволяет использовать все возможности, а правильная отправка байтов — сложная задача. Лучше всего применять либо непосредственно клиентские библиотеки Java, либо клиентские библиотеки сторонних разработчиков для других языков, которые используют протокол Kafka напрямую.

Консольный производитель

Утилиту `kafka-console-producer.sh` можно использовать для записи сообщений в топик Kafka в вашем кластере. По умолчанию сообщения читаются по одному в строке, с символом табуляции, разделяющим ключ и значение (если символ

табуляции отсутствует, то ключ равен нулю). Как и в случае с консольным потребителем, производитель считывает и создает необработанные байты при помощи сериализатора по умолчанию, то есть `DefaultEncoder`.

Консольный производитель требует предоставления как минимум двух аргументов, чтобы знать, к какому кластеру Kafka подключаться и в какой топик в нем производить. Первый — это обычная строка подключения к серверу `--bootstrap-server`, которую мы привыкли использовать. Когда закончите производство, отправьте символ конца файла (EOF), чтобы закрыть клиент. В большинстве обычных терминалов это делается нажатием `Ctrl+D`.

Здесь мы видим пример создания четырех сообщений для топика `my-topic`:

```
# kafka-console-producer.sh --bootstrap-server localhost:9092 --topic my-topic
>Message 1
>Test Message 2
>Test Message 3
>Message 4
>^D
#
```

Использование параметров конфигурации производителя

Можно передавать обычные параметры конфигурации производителя также консольному производителю. Это можно сделать двумя способами в зависимости от того, сколько параметров вам нужно передать и как вы предпочитаете это делать. Первый способ — предоставить файл конфигурации производителя, указав `--producer.config<файл_конфигурации>`, где `<файл_конфигурации>` — полный путь к файлу, содержащему параметры конфигурации. Другой способ — указать параметры в командной строке с одним или несколькими аргументами в виде `--producer-property <ключ>=<значение>`, где `<ключ>` — имя параметра конфигурации, а `<значение>` — значение, которое нужно установить. Это может быть полезно для параметров производителя, таких как настройки пакетной обработки сообщений, например `linger.ms` или `batch.size`.



Путаница с параметрами командной строки

Параметр командной строки `--property` доступен как для консольного производителя, так и для консольного потребителя, но его не следует путать с параметрами `--producer-property` или `--consumer-property` соответственно. Параметр `--property` используется только для передачи настроек в программу форматирования сообщений, а не самому клиенту.

Консольный производитель имеет множество аргументов командной строки, доступных для использования с параметром `--producer-property` для настройки его поведения. Вот некоторые из наиболее полезных параметров.

- `--batch-size`. Определяет количество сообщений, отправляемых в одном пакете, если они не отправляются синхронно.
- `--timeout`. Если производитель работает в асинхронном режиме, задает максимальное время ожидания размера пакета перед производством, чтобы избежать длительного ожидания в топиках с низкой производительностью.
- `--compression-codec <string>`. Указывает тип сжатия, который будет использоваться при производстве сообщений. В качестве допустимого типа может быть один из следующих: `none`, `gzip`, `snappy`, `zstd` или `lz4`. По умолчанию берется `gzip`.
- `--sync`. Создает сообщения синхронно, ожидая подтверждения каждого сообщения перед отправкой следующего.

Параметры считывателя строк

Класс `kafka.tools.ConsoleProducer$LineMessageReader`, который отвечает за чтение стандартного ввода и создание записей производителя, также имеет несколько полезных параметров, которые можно передать консольному производителю с помощью параметра командной строки `--property`.

- `ignore.error`. Устанавливается в значение `false`, чтобы вызвать исключение, если параметру `parse.key` установлено значение `true`, а разделитель ключей отсутствует. По умолчанию установлено значение `true`.
- `parse.key`. Устанавливается значение `false`, чтобы всегда присваивать ключу нулевое значение `null`. По умолчанию установлено значение `true`.
- `key.separator`. Указывает символ-разделитель, который будет использоваться между ключом сообщения и его значением при чтении. По умолчанию применяется символ табуляции.



Изменение режима чтения строк

Вы можете предоставить Kafka собственный класс для настраиваемых методов чтения строк. Созданный вами класс должен расширять `kafka.common.MessageReader` и будет отвечать за создание `ProducerRecord`. Укажите свой класс в командной строке с помощью параметра `--line-reader` и убедитесь, что JAR-файл, содержащий ваш класс, находится в пути к классам. По умолчанию используется `kafka.tools.Console Producer$LineMessageReader`.

При создании сообщений `LineMessageReader` будет разделять входные данные на первом экземпляре `key.separator`. Если после этого не останется символов, значение сообщения будет пустым. Если в строке нет символа-разделителя ключа или если `parse.key` установлено значение `false`, ключ будет равен `null`.

Консольный потребитель

Утилита `kafka-console-consumer.sh` предоставляет средства для потребления сообщений из одного или нескольких топиков в вашем кластере Kafka. Сообщения выводятся в стандартный вывод и разделяются новой строкой. По умолчанию он выводит необработанные байты в сообщении без ключа и без форматирования, используя `DefaultFormatter`. Как и в случае с производителем, для начала работы требуются несколько основных параметров: строка подключения к кластеру, топик, из которого вы хотите потреблять, и временной интервал, который хотите использовать.



Проверка версий инструментов

Очень важно использовать потребитель той же версии, что и ваш кластер Kafka. Более старые консольные потребители могут повредить кластер, взаимодействуя с кластером или ZooKeeper некорректными способами.

Как и в других командах, строкой подключения к кластеру будет параметр `--bootstrap-server`, однако вы можете выбрать один из двух вариантов топиков для потребления.

- `--topic` — указывает один топик для потребления.
- `--whitelist` — регулярное выражение, сопоставляющее все топики для потребления (не забудьте правильно экранировать регулярное выражение, чтобы оно не было неправильно обработано оболочкой).

Следует выбрать и использовать только один из предыдущих параметров. После запуска консольного потребителя инструмент будет продолжать попытки потребления до тех пор, пока не будет дана команда завершения оболочки (в данном случае `Ctrl+C`). Вот пример потребления всех топиков из нашего кластера, которые соответствуют префиксу `my` (в данном примере есть только один топик, `my-topic`):

```
# kafka-console-consumer.sh --bootstrap-server localhost:9092
--whitelist 'my.*' --from-beginning
Message 1
Test Message 2
Test Message 3
Message 4
^C
#
```


Использование параметров конфигурации потребителя

В дополнение к основным параметрам командной строки можно передать потребителю консоли также обычные параметры конфигурации потребителя. Как и в утилите `kafka-console-producer.sh`, это можно сделать двумя способами в зависимости от того, сколько параметров нужно передать и как вы предпочитаете это делать. Первый — предоставить файл конфигурации потребителя, указав `--consumer.config <файл_конфигурации>`, где `<файл_конфигурации>` — это полный путь к файлу, содержащему параметры конфигурации. Другой способ — указать параметры в командной строке с одним или несколькими аргументами вида `--consumer-property <ключ>=<значение>`, где `<ключ>` — имя параметра конфигурации, а `<значение>` — значение, которое нужно установить.

Есть еще несколько часто используемых параметров для потребителя консоли, которые полезно знать и с которыми нужно быть знакомыми.

- `--formatter <имя_класса>`. Указывает класс программы форматирования сообщений, который будет использоваться для декодирования сообщений. По умолчанию применяется `kafka.tools.DefaultMessageFormatter`.
- `--from-beginning`. Потребляет сообщения в указанном топике (топиках) с самого старого смещения. В противном случае потребление начинается с самого позднего смещения.
- `--max-messages <int>`. Максимальное количество сообщений, которое необходимо потреблять перед завершением работы.
- `--partition <int>`. Потребляет только из раздела с указанным идентификатором.
- `--offset`. Идентификатор смещения, с которого будет производиться потребление, если он указан (`<int>`). Другими допустимыми параметрами являются `earliest` — потребление с самого начала и `latest` — потребление с самого последнего смещения.
- `--skip-message-on-error`. Пропускает сообщение, если при обработке возникла ошибка, вместо того чтобы остановить процесс. Полезно для отладки.

Параметры программы форматирования сообщений

Наряду со стандартным можно использовать три средства форматирования сообщений.

- `kafka.tools.LoggingMessageFormatter`. Выводит сообщения, используя логгер, а не стандартный вывод. Сообщения выводятся на информационном уровне INFO и включают метку времени, ключ и значение.

- `kafka.tools.ChecksumMessageFormatter`. Выводит только контрольные суммы сообщений.
- `kafka.tools.NoOpMessageFormatter`. Потребляет сообщения, но не выводит их.

Далее приведен пример потребления тех же сообщений, что и раньше, но с использованием `kafka.tools.ChecksumMessageFormatter`, а не по умолчанию:

```
# kafka-console-consumer.sh --bootstrap-server localhost:9092
--whitelist 'my.*' --from-beginning
--formatter kafka.tools.ChecksumMessageFormatter
checksum:0
checksum:0
checksum:0
checksum:0
#
```

У `kafka.tools.DefaultMessageFormatter` также есть несколько полезных параметров, которые можно передать с помощью параметра командной строки `--property`, как показано в табл. 12.4.

Таблица 12.4. Свойства программы форматирования сообщений

Свойство	Описание
<code>print.timestamp</code>	Устанавливает значение <code>true</code> для отображения временной метки каждого сообщения (если доступно)
<code>print.key</code>	Устанавливает значение <code>true</code> для отображения ключа сообщения в дополнение к значению
<code>print.offset</code>	Устанавливает значение <code>true</code> для отображения смещения сообщения в дополнение к значению
<code>print.partition</code>	Устанавливает значение <code>true</code> для отображения раздела топика, из которого потребляется сообщение
<code>key.separator</code>	Указывает символ-разделитель, который будет использоваться между ключом сообщения и значением сообщения при печати
<code>line.separator</code>	Указывает символ-разделитель, который будет ставиться между сообщениями
<code>key.deserializer</code>	Указывает имя класса, который используется для десериализации ключа сообщения перед печатью
<code>value.deserializer</code>	Указывает имя класса, который применяется для десериализации значения сообщения перед печатью

Классы десериализаторов должны реализовывать `org.apache.kafka.common.serialization.Deserializer`, и консольный потребитель будет вызывать метод `toString`, чтобы отобразить выходные данные. Обычно вы реализуете эти десериализаторы в виде класса Java, который вставляете в путь к классам для

консольного потребителя, установив переменную окружения `CLASSPATH` перед выполнением утилиты `kafka_console_consumer.sh`.

Потребление топиков смещений

Иногда полезно посмотреть, какие смещения фиксируются для групп потребителей кластера. Вы можете захотеть узнать, фиксирует ли определенная группа смещения вообще или как часто фиксируются смещения. Это можно сделать, используя консольный потребитель для потребления специального внутреннего топика под названием `consumer_offsets`. Все смещения потребителей записываются в него в виде сообщений. Чтобы декодировать сообщения в этом топике, необходимо использовать класс средства форматирования `kafka.coordinator.group.GroupMetadataManager$OffsetsMessageFormatter`.

Объединив все, что мы узнали, далее приводим пример потребления самого раннего сообщения из топика `consumer_offsets`:

```
# kafka-console-consumer.sh --bootstrap-server localhost:9092
--topic __consumer_offsets --from-beginning --max-messages 1
--formatter "kafka.coordinator.group.GroupMetadataManager$OffsetsMessageFormatter"
--consumer-property exclude.internal.topics=false
[my-group-name,my-topic,0]::[OffsetMetadata[1,NO_METADATA]
CommitTime 1623034799990 ExpirationTime 1623639599990]
Processed a total of 1 messages
#
```

Управление разделами

Стандартная установка Kafka содержит несколько сценариев для управления разделами. Один из них предназначен для повторного выбора ведущих реплик, другой представляет собой низкоуровневую утилиту распределения разделов по брокерам. Вместе эти инструменты могут помочь в ситуациях, когда требуется более практический подход к балансировке трафика сообщений в кластере брокеров Kafka.

Выбор предпочтительной ведущей реплики

Как обсуждалось в главе 7, у разделов может быть несколько реплик для большей надежности. Важно понимать, что только одна из них может быть ведущей репликой раздела в любой момент времени и вся генерация и потребление выполняются в соответствующем брокере. Поддержание баланса между тем, какие реплики раздела имеют лидерство в каком брокере, необходимо для того, чтобы обеспечить распределение нагрузки по всему кластеру Kafka.

Лидером в Kafka становится первая синхронизированная реплика в списке реплик. Однако в случае, когда брокер останавливается или теряет связь с остальным кластером, лидерство передается другой синхронизированной реплике, а первоначальная реплика не становится автоматически ведущей для каких-либо разделов. Это может привести к крайне неэффективной балансировке после развертывания всего кластера, если не включено автоматическое переназначение ведущих реплик. Поэтому рекомендуется убедиться, что этот параметр включен, или использовать другие инструменты с открытым исходным кодом, такие как Cruise Control, чтобы обеспечить постоянное поддержание хорошего баланса.

Если вы обнаружите, что ваш кластер Kafka плохо сбалансирован, можете выполнить легкую, обычно не оказывающую влияния процедуру, называемую *выбором предпочтительного лидера*. Она приказывает контроллеру кластера выбирать идеальные ведущие реплики для разделов. У клиентов есть возможность отслеживать смену ведущей реплики автоматически, поэтому они смогут перейти к новому брокеру в кластере, в котором передается лидерство. Эту операцию можно запустить вручную с помощью утилиты `kafka-leader-election.sh`. Более старая версия этой утилиты под названием `kafka-preferred-replica-election.sh` также доступна, но она устарела — есть новая утилита, которая позволяет выполнять большее количество настроек, например указывать, хотим ли мы получить «предпочтительный» или «нечистый» тип выборов.

Например, запустить выбор предпочтительного лидера для всех топиков кластера можно с помощью следующей команды:

```
# kafka-leader-election.sh --bootstrap-server localhost:9092
--election-type PREFERRED --all-topic-partitions
#
```

Также можно запускать выборы в определенных разделах или топиках. Это можно сделать, передав имя топика с параметром `--topic` и напрямую раздел с параметром `--partition`. Также можно передать список из нескольких разделов, которые должны быть избраны. Это делается настройкой файла JSON, который мы будем называть `partitions.json`:

```
{
  "partitions": [
    {
      "partition": 1,
      "topic": "my-topic"
    },
    {
      "partition": 2,
      "topic": "foo"
    }
  ]
}
```

В этом примере мы инициируем выбор предпочтительной ведущей реплики с заданным списком разделов, находящимся в файле `partitions.json`:

```
# kafka-leader-election.sh --bootstrap-server localhost:9092
--election-type PREFERRED --path-to-json-file partitions.json
#
```

Изменение реплик раздела

Периодически возникает необходимость вручную изменить назначения реплик для раздела. Вот несколько примеров, когда такая необходимость возникает.

- Существует неравномерная нагрузка на брокеров, которую автоматическое распределение лидеров обрабатывает неправильно.
- Один из брокеров отключился, и раздел недостаточно реплицирован.
- Был добавлен новый брокер, и мы хотим быстрее сбалансировать новые разделы в нем.
- Вы хотите настроить коэффициент репликации топика.

Для выполнения этой операции можно воспользоваться утилитой `kafka-reassign-partitions.sh`. Это многоэтапный процесс создания набора перемещений и последующего выполнения в соответствии с предоставленным предложением набора перемещений. Сначала на основе списка брокеров и топиков мы генерируем предложение для набора перемещений. Для этого нам потребуется создать файл JSON со списком топиков, которые необходимо предоставить. На следующем этапе выполняются перемещения, которые были сгенерированы предыдущим предложением. Наконец, инструмент можно использовать вместе с созданным списком для отслеживания и проверки хода или завершения переназначений разделов.

Давайте создадим гипотетический сценарий, в котором у вас есть кластер Kafka с четырьмя брокерами. Недавно вы добавили два новых брокера, доведя общее количество до шести, и хотите переместить два своих топика в брокеры 5 и 6.

Для генерации набора перемещений разделов необходимо сначала создать файл с JSON-объектом, содержащим список топиков. Формат JSON-объекта следующий (номер версии в настоящий момент всегда равен 1):

```
{
  "topics": [
    {
      "topic": "foo1"
    },
  ],
}
```

```

    {
        "topic": "foo2"
    }
],
"version": 1
}

```

После того как мы определили JSON-файл, можем использовать его, чтобы сгенерировать перемещения разделов для переноса перечисленных в файле `topics.json` разделов в брокеры с идентификаторами 5 и 6:

```

# kafka-reassign-partitions.sh --bootstrap-server localhost:9092
--topics-to-move-json-file topics.json
--broker-list 5,6 --generate
{
  "version":1,
  "partitions":[{"topic":"foo1","partition":2,"replicas":[1,2]},
                {"topic":"foo1","partition":0,"replicas":[3,4]},
                {"topic":"foo2","partition":2,"replicas":[1,2]},
                {"topic":"foo2","partition":0,"replicas":[3,4]},
                {"topic":"foo1","partition":1,"replicas":[2,3]},
                {"topic":"foo2","partition":1,"replicas":[2,3]}]
}

```

Proposed partition reassignment configuration

```

{"version":1,
 "partitions":[{"topic":"foo1","partition":2,"replicas":[5,6]},
               {"topic":"foo1","partition":0,"replicas":[5,6]},
               {"topic":"foo2","partition":2,"replicas":[5,6]},
               {"topic":"foo2","partition":0,"replicas":[5,6]},
               {"topic":"foo1","partition":1,"replicas":[5,6]},
               {"topic":"foo2","partition":1,"replicas":[5,6]}]
}
#

```

Предлагаемые здесь выходные данные отформатированы правильно, в них можно сохранить два новых файла JSON, которые мы назовем `revert-reassignment.json` и `expand-cluster-reassignment.json`. Первый файл можно использовать для перемещения разделов обратно в исходное положение, если по каким-то причинам вам потребуется выполнить откат. Второй файл может быть использован для следующего шага, поскольку это всего лишь предложение и пока еще ничего не выполняется. Вы можете заметить, что в выходных данных нет хорошего баланса лидерства, так как в результате предложения все лидеры переместятся к брокеру 5. Мы пока проигнорируем это и предположим, что в кластере включена автоматическая балансировка лидерства, что поможет распределить его позже. Следует отметить, что первый шаг можно пропустить, если вы точно знаете, куда хотите переместить разделы, и вручную создаете JSON для перемещения разделов.

Чтобы реализовать предлагаемое переназначение разделов из файла `expand-cluster-reassignment.json`, выполните следующую команду:

```
# kafka-reassign-partitions.sh --bootstrap-server localhost:9092
--reassignment-json-file expand-cluster-reassignment.json
--execute
Current partition replica assignment

{"version":1,
 "partitions":[{"topic":"foo1","partition":2,"replicas":[1,2]},
               {"topic":"foo1","partition":0,"replicas":[3,4]},
               {"topic":"foo2","partition":2,"replicas":[1,2]},
               {"topic":"foo2","partition":0,"replicas":[3,4]},
               {"topic":"foo1","partition":1,"replicas":[2,3]},
               {"topic":"foo2","partition":1,"replicas":[2,3]]}

Save this to use as the --reassignment-json-file option during rollback
Successfully started reassignment of partitions
{"version":1,
 "partitions":[{"topic":"foo1","partition":2,"replicas":[5,6]},
               {"topic":"foo1","partition":0,"replicas":[5,6]},
               {"topic":"foo2","partition":2,"replicas":[5,6]},
               {"topic":"foo2","partition":0,"replicas":[5,6]},
               {"topic":"foo1","partition":1,"replicas":[5,6]},
               {"topic":"foo2","partition":1,"replicas":[5,6]]}
}
#
```

Эта команда запустит переназначение заданных реплик разделов по новым брокерам. На выходе получается то же самое, что и при проверке сгенерированного предложения. Контроллер кластера выполняет переназначение посредством добавления новых реплик в список реплик каждого из разделов этих топиков, что временно повысит коэффициент репликации. После этого новые реплики копируют все существующие сообщения для всех разделов с текущей ведущей реплики. В зависимости от размера разделов на диске эта операция может занять немало времени, так как возникают затраты на копирование данных по сети в новые реплики. После завершения репликации контроллер удаляет старые реплики из списка реплик, уменьшая коэффициент репликации до первоначального значения.

Вот несколько других полезных функций команды, которыми вы можете воспользоваться.

- **--additional.** Эта команда позволит вам добавлять новые переназначения к уже существующим, чтобы они продолжали выполняться без перерыва и не ожидая завершения исходных перемещений, чтобы начать новую партию.

- **--disable-rack-aware**. Бывают случаи, когда из-за настроек осведомленности о стойке конечное состояние предложения может оказаться невозможным. При необходимости это можно отменить с помощью данной команды.
- **--throttle**. Это значение выражается в байтах в секунду (байт/с). Переназначения разделов серьезно влияют на производительность кластера, поскольку они вызывают изменения в согласованности страничного кэша памяти и используют сеть и дисковые операции ввода/вывода. Регулирование перемещения разделов может быть полезно для предотвращения этой проблемы. Его можно сочетать с тегом **--additional** для ограничения уже запущенного процесса переназначения, который может вызывать проблемы.



Улучшение использования сети при переназначении реплик

При удалении многих разделов из одного брокера, например, если он удаляется из кластера, может быть полезно сначала удалить все лидеры из брокера. Это можно сделать, вручную переместив лидеры из брокера, однако использовать перечисленные ранее инструменты для этого довольно затруднительно. Другие инструменты с открытым исходным кодом, например Cruise Control, включают в себя такие функции, как «понижение» брокера, которая безопасно перемещает лидеров из брокера и, вероятно, является самым простым способом сделать это.

Однако если у вас нет доступа к таким инструментам, достаточно простого перезапуска брокера. По мере того как брокер готовится к выключению, все лидеры для разделов в этом конкретном брокере перейдут к другим брокерам в кластерах. Это может значительно повысить производительность переназначений и снизить воздействие на кластер, поскольку трафик репликации будет распределен между многими брокерами. Однако если автоматическое переназначение лидеров включено после отказа брокера, лидеры могут вернуться к этому брокеру, поэтому может быть полезно временно отключить эту функцию.

Чтобы проверить ход перемещения разделов, можно использовать инструмент для проверки статуса переназначения. Он покажет, какие переназначения в настоящее время выполняются, какие завершены и, если произошла ошибка, какие завершились неудачно. Для этого у вас должен быть файл с объектом JSON, который использовался на этапе выполнения.

Вот пример возможных результатов с использованием параметра **--verify** при выполнении предыдущего переназначения разделов из файла **expand-cluster-reassignment.json**:

```
# kafka-reassign-partitions.sh --bootstrap-server localhost:9092
--reassignment-json-file expand-cluster-reassignment.json
--verify
Status of partition reassignment:
```



```
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
#
```

Изменение коэффициента репликации

Утилиту `kafka-reassign-partitions.sh` также можно использовать для увеличения или уменьшения коэффициента репликации (RF) для раздела. Это может потребоваться в ситуациях, когда раздел был создан с неправильным коэффициентом репликации, вы хотите увеличить избыточность при расширении кластера или уменьшить избыточность для экономии средств. Одним из наглядных примеров является то, что при изменении настройки коэффициента репликации по умолчанию кластера существующие топика не будут автоматически увеличены. Эту утилиту можно использовать для увеличения коэффициента репликации в существующих разделах.

Например, если мы хотим увеличить топик `foo1` из предыдущего примера с `RF = 2` до `RF = 3`, то можем создать JSON, аналогичный предложению о выполнении, которое использовали ранее, за исключением того, что добавим дополнительный идентификатор брокера в набор реплик. Например, мы можем создать JSON под названием `increase-foo1-RF.json`, в котором добавим брокер 4 к уже существующему набору из брокеров 5 и 6:

```
{
  {"version":1,
   "partitions":[{"topic":"foo1","partition":1,"replicas":[5,6,4]},
                 {"topic":"foo1","partition":2,"replicas":[5,6,4]},
                 {"topic":"foo1","partition":3,"replicas":[5,6,4]},
               ]
}
```

Затем мы используем команды, показанные ранее, для выполнения этого предложения. Когда они завершатся, мы можем убедиться, что коэффициент репликации был увеличен, с помощью либо флага `--verify`, либо сценария `kafka-topics.sh` для описания топика:

```
# kafka-topics.sh --bootstrap-server localhost:9092 --topic foo1 --describe
Topic:foo1    PartitionCount:3      ReplicationFactor:3    Configs:
  Topic: foo1 Partition: 0    Leader: 5      Replicas: 5,6,4 Isr: 5,6,4
  Topic: foo1 Partition: 1    Leader: 5      Replicas: 5,6,4 Isr: 5,6,4
  Topic: foo1 Partition: 2    Leader: 5      Replicas: 5,6,4 Isr: 5,6,4
#
```

Отмена переназначения реплик

В прошлом отмена переназначения реплик была опасным процессом, который требовал небезопасных ручных манипуляций с узлами ZooKeeper (или z-узлами) путем удаления узла `/admin/reassign_partitions`. К счастью, сейчас это не так. Сценарий `kafka-reassign-partitions.sh` (а также `AdminClient`, для которого он является оболочкой) теперь поддерживает параметр `--cancel`, который отменяет активные переназначения, выполняемые в кластере. При остановке текущего перемещения разделов команда `--cancel` предназначена для восстановления набора реплик до того состояния, в котором он находился до начала переназначения. Таким образом, если реплики удаляются из неработающего или перегруженного брокера, это может привести к тому, что кластер окажется в нежелательном состоянии. Также нет гарантии, что переназначенный набор реплик будет располагаться в том же порядке, что и ранее.

Сброс на диск сегментов журнала

В некоторых случаях вам может понадобиться прочитать сообщение с конкретным содержимым, например «отравленную таблетку» — поврежденное сообщение в вашем топике, которое потребитель не может обработать. Утилита `kafka-dump-log.sh` предоставляется для декодирования сегментов журнала раздела. Она позволяет просматривать отдельные сообщения, не потребляя и не декодируя их. В качестве аргумента эта утилита принимает разделенный запятыми список файлов сегментов журнала и может выводить сводную или развернутую информацию по сообщению.

В этом примере мы выгрузим журналы из образца топика `my-topic`, который представляет собой новый топик, содержащий всего четыре сообщения. Сначала просто декодируем файл сегмента журнала с названием `000000000000000000000000.log` и извлечем основную информацию о метаданных каждого сообщения без фактической печати содержимого сообщения. В примере установки Kafka каталог данных Kafka находится в `/tmp/kafka-logs`. Таким образом, нашей директорией для поиска сегментов журнала будет `/tmp/kafka-logs/<topic-name>-<partition>`, в данном случае `/tmp/kafka-logs/my-topic-0/`:

```
# kafka-dump-log.sh --files /tmp/kafka-logs/my-topic-0/00000000000000000000.log
Dumping /tmp/kafka-logs/my-topic-0/00000000000000000000.log
Starting offset: 0
baseOffset: 0 lastOffset: 0 count: 1 baseSequence: -1 lastSequence: -1
  producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0
  isTransactional: false isControl: false position: 0
  CreateTime: 1623034799990 size: 77 magic: 2
  compressCodec: NONE crc: 1773642166 isValid: true
baseOffset: 1 lastOffset: 1 count: 1 baseSequence: -1 lastSequence: -1
```

```

producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0
isTransactional: false isControl: false position: 77
CreateTime: 1623034803631 size: 82 magic: 2
compresscodec: NONE crc: 1638234280 invalid: true
baseOffset: 2 lastOffset: 2 count: 1 baseSequence: -1 lastSequence: -1
producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0
isTransactional: false isControl: false position: 159
CreateTime: 1623034808233 size: 82 magic: 2
compresscodec: NONE crc: 4143814684 invalid: true
baseOffset: 3 lastOffset: 3 count: 1 baseSequence: -1 lastSequence: -1
producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0
isTransactional: false isControl: false position: 241
CreateTime: 1623034811837 size: 77 magic: 2
compresscodec: NONE crc: 3096928182 invalid: true
#

```

В следующем примере добавим параметр `--print-data-log`, который предоставит нам информацию о фактической полезной нагрузке и многое другое:

```

# kafka-dump-log.sh --files /tmp/kafka-logs/my-topic-0/00000000000000000000.log
--print-data-log
Dumping /tmp/kafka-logs/my-topic-0/00000000000000000000.log
Starting offset: 0
baseOffset: 0 lastOffset: 0 count: 1 baseSequence: -1 lastSequence: -1
producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0
isTransactional: false isControl: false position: 0
CreateTime: 1623034799990 size: 77 magic: 2
compresscodec: NONE crc: 1773642166 invalid: true
| offset: 0 CreateTime: 1623034799990 keysize: -1 valuesize: 9
  sequence: -1 headerKeys: [] payload: Message 1
baseOffset: 1 lastOffset: 1 count: 1 baseSequence: -1 lastSequence: -1
producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0
isTransactional: false isControl: false position: 77
CreateTime: 1623034803631 size: 82 magic: 2
compresscodec: NONE crc: 1638234280 invalid: true
| offset: 1 CreateTime: 1623034803631 keysize: -1 valuesize: 14
  sequence: -1 headerKeys: [] payload: Test Message 2
baseOffset: 2 lastOffset: 2 count: 1 baseSequence: -1 lastSequence: -1
producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0
isTransactional: false isControl: false position: 159
CreateTime: 1623034808233 size: 82 magic: 2
compresscodec: NONE crc: 4143814684 invalid: true
| offset: 2 CreateTime: 1623034808233 keysize: -1 valuesize: 14
  sequence: -1 headerKeys: [] payload: Test Message 3
baseOffset: 3 lastOffset: 3 count: 1 baseSequence: -1 lastSequence: -1
producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0
isTransactional: false isControl: false position: 241
CreateTime: 1623034811837 size: 77 magic: 2
compresscodec: NONE crc: 3096928182 invalid: true
| offset: 3 CreateTime: 1623034811837 keysize: -1 valuesize: 9
  sequence: -1 headerKeys: [] payload: Message 4
#

```

Утилита также содержит несколько других полезных параметров, таких как проверка файлов индексов, сопутствующих сегментам журналов. Индексы применяются для поиска сообщений в сегменте журнала, их порча может вызвать ошибки при потреблении. Проверка всегда выполняется, если брокер запускается в «нечистом» состоянии (то есть не был остановлен штатным образом), но ее можно запустить и вручную. Существует два параметра для проверки индексов, применяемых в зависимости от желаемой тщательности процесса. Параметр `--index-sanity-check` проверяет только пригодность индекса к использованию, а `--verify-index-only` — наличие неточностей в индексе без вывода всех его записей. Другой полезный параметр, `--value-decoder-class`, позволяет десериализовывать сериализованные сообщения путем передачи в декодер.

Проверка реплик

Репликация разделов функционирует аналогично обычному клиенту-потребителю Kafka: ведомый брокер начинает репликацию с самого старого смещения и периодически записывает в контрольные точки данные о текущем смещении на диск. При останове и перезапуске репликация возобновляется с последней контрольной точки. Ранее реплицированные сегменты журналов могут удаляться из брокера, в этом случае ведомый брокер не будет заполнять промежутки.

Чтобы проверить согласованность всех реплик разделов топика в кластере, можно воспользоваться утилитой `kafka-replica-verification.sh`. Она извлекает сообщения из всех реплик заданного набора разделов топика, проверяет наличие всех сообщений во всех репликах и выводит максимальную задержку для заданных разделов. Этот процесс станет работать непрерывно в цикле до тех пор, пока не будет отменен. Для этого вы должны предоставить явный список брокеров, разделенных запятыми, к которым необходимо подключиться. По умолчанию проверяются все топика, однако вы также можете задать инструменту регулярное выражение, соответствующее всем топикам, которые нужно проверить. Если его не указать, то будут проверяться все топика.



Осторожно: влияет на производительность кластера

Утилита проверки реплик влияет на производительность кластера так же, как и переназначение разделов, поскольку для проверки реплик читает все смещения, начиная с самого старого. Кроме того, она читает данные из всех реплик раздела параллельно, так что будьте осторожны при ее использовании.

Например, проверим реплики для топиков, название которых начинается с `my`, в брокерах Kafka 1 и 2, которые содержат раздел `my-topic`:

```
# kafka-replica-verification.sh --broker-list kafka.host1.domain.com:
9092,kafka.host2.domain.com:9092
--topic-white-list 'my.*'

2021-06-07 03:28:21,829: verification process is started.
2021-06-07 03:28:51,949: max lag is 0 for partition my-topic-0 at offset 4
among 1 partitions
2021-06-07 03:29:22,039: max lag is 0 for partition my-topic-0 at offset 4
among 1 partitions
...
#
```

Другие утилиты

В дистрибутив Kafka включены еще несколько утилит (они не рассматриваются подробно в этой книге), которые могут быть полезны при администрировании кластера Kafka для конкретных сценариев использования. Дополнительную информацию о них можно найти на сайте Apache Kafka (<https://kafka.apache.org>).

- *Списки управления доступом клиентов.* Утилита командной строки `kafka-acls.sh` предназначена для взаимодействия с механизмом управления доступом для клиентов Kafka. Она включает в себя полный набор функций для свойств авторизатора, настройку принципов запрета или разрешения, ограничения на уровне кластера или топика, конфигурацию TLS-файла ZooKeeper и многое другое.
- *Облегченный скрипт MirrorMaker.* Для зеркального копирования данных доступен облегченный скрипт `kafka-mirror-maker.sh`. Более подробно репликация рассматривается в главе 10.
- *Инструменты тестирования.* Есть еще несколько сценариев, используемых для тестирования Kafka или для обновления функций. Утилита `kafka-broker-api-versions.sh` помогает легко идентифицировать различные версии элементов API, используемых при переходе с одной версии Kafka на другую, и проверить их на совместимость. Предусмотрены сценарии для тестирования производительности производителей и потребителей. Есть также несколько сценариев для администрирования ZooKeeper. Существует также утилита `trogdor.sh`, которая представляет собой тестовый фреймворк, предназначенный для запуска бенчмарков и других рабочих нагрузок, чтобы попробовать провести стресс-тестирование системы.

Небезопасные операции

Среди административных задач есть и такие, которые технически выполнить возможно, но лучше этого не делать, разве что в самом крайнем случае, например при диагностике проблем, когда все остальные варианты исчерпаны, или при поиске обходного пути для исправления конкретной ошибки. Обычно эти операции не документированы и не поддерживаются, а их выполнение несет некоторые риски для приложения.

Несколько наиболее распространенных из них описаны далее, чтобы в непредвиденной ситуации у вас был лишний вариант восстановления. При нормальном функционировании кластера использовать их не рекомендуется, так что, прежде чем их запустить, стоит дважды подумать.



Опасность: неизведанная территория

Операции из этого раздела часто предполагают работу напрямую с хранящимися в ZooKeeper метаданными. Это чрезвычайно опасная операция, так что будьте осторожны и не модифицируйте напрямую информацию в ZooKeeper, за исключением рассмотренных далее случаев.

Перенос контроллера кластера

У каждого кластера Kafka есть один брокер, который назначен контроллером. У контроллера есть специальный поток выполнения, который, помимо обычной работы брокера, отвечает за управление операциями кластера. Обычно выборы контроллера происходят автоматически путем эфемерного мониторинга узла ZooKeeper. Когда контроллер отключается или становится недоступным, другие брокеры выдвигают свои кандидатуры как можно скорее, поскольку после включения контроллера z-узел удаляется.

В некоторых случаях при устранении неполадок в кластере или брокере может быть полезно принудительно переместить контроллер на другой брокер без выключения хоста. В качестве примера можно привести ситуацию, когда в контроллере возникло исключение или какая-либо другая проблема, в результате чего он продолжает работать, но уже не выполняет свои функции. Перенос контроллера в подобных случаях обычно не представляет особого риска, но так как это нестандартная операция, то и выполнять ее регулярно не стоит.

Чтобы принудительно переместить контроллер, удалите вручную узел ZooKeeper по адресу `/admin/controller`, это приведет к выходу текущего контроллера из состава кластера, и кластер случайным образом выберет новый контроллер. В настоящее время не существует способа указать конкретный брокер в качестве контроллера в Apache Kafka.

Отмена удаления топиков

При попытке удаления топика в Kafka узел ZooKeeper запрашивает удаление. Как только каждая реплика завершит удаление топика и подтвердит, что процедура завершена, z-узел будет удален. При нормальных обстоятельствах кластер выполняет это очень быстро. Однако иногда этот процесс может пойти не так. Вот несколько сценариев, при которых запрос на удаление может «залипнуть».

1. Запрашивающий не имеет возможности узнать, включено ли удаление топика в кластере, и может запросить удаление топика из кластера, в котором удаление отключено.
2. Запрашивается удаление очень большого топика, но прежде, чем запрос будет обработан, один или несколько наборов реплик переходят в автономный режим из-за сбоев оборудования, и удаление не может быть завершено, поскольку контроллер не может подтвердить, что это сделано успешно.

Чтобы «разлепить» удаление темы, сначала удалите узел `/admin/delete_topic/<topic>`. Удаление узлов ZooKeeper топика (но не родительского узла `/admin/delete_topic`) приведет к удалению ожидающих запросов. Если удаление происходит по кэшированным запросам в контроллере, может также потребоваться принудительное перемещение контроллера, как показано ранее, сразу после удаления узла топика, чтобы убедиться, что в контроллере нет кэшированных запросов.

Удаление топиков вручную

Если удаление топиков в вашем кластере отключено или вам понадобилось убрать какие-либо топика вне нормального технологического процесса, существует возможность вручную удалять их из кластера. Однако для этого необходимо остановить все работающие брокеры кластера.



Сначала остановите брокеры

Модификация метаданных кластера в ZooKeeper во время его (кластера) работы — очень опасная операция, которая может привести к нестабильности кластера. Никогда не пытайтесь удалять или модифицировать метаданные топика в ZooKeeper во время работы кластера.

Для удаления топика из кластера сделайте следующее.

1. Остановите все брокеры кластера.
2. Удалите каталог ZooKeeper `/brokers/topics/<topic>` из пути кластера Kafka. Обратите внимание на то, что сначала необходимо удалить его дочерние узлы.

3. Удалите каталоги разделов из каталогов журналов всех брокеров. Они называются `<topic>-<int>`, где `<int>` — идентификаторы разделов.
4. Перезапустите все брокеры.

Резюме

Эксплуатация кластера Kafka — непростая задача, поскольку имеется множество настроек и задач по сопровождению, необходимых для обеспечения максимальной производительности. В этой главе мы обсудили многие задачи, которые часто требуется выполнять регулярно, например администрирование настроек топиков и клиентов. Рассмотрели также более нетривиальные операции, порой необходимые для отладки, например просмотр сегментов журналов. Наконец описали несколько небезопасных нерегулярных операций, которые иногда приходится выполнять, чтобы выйти из проблемных ситуаций. Все вместе эти утилиты чрезвычайно полезны для управления кластером Kafka. Когда вы начнете масштабировать свои кластеры Kafka, даже использование этих утилит может стать трудным и сложным в управлении. Настоятельно рекомендуется взаимодействовать с сообществом разработчиков Kafka с открытым исходным кодом и использовать преимущества других проектов с открытым исходным кодом в экосистеме, которые помогут автоматизировать многие из задач, описанных в этой главе.

Теперь, когда мы уверены в инструментах, необходимых для администрирования нашего кластера и управления им, это все еще невозможно без должного мониторинга. В главе 13 мы обсудим способы мониторинга работоспособности и функционирования брокеров и кластера, благодаря которым вы сможете быть уверены, что Kafka работает должным образом, или будете знать, что это не так. Опишем также рекомендуемые практики мониторинга ваших клиентов — как производителей, так и потребителей.

Мониторинг Kafka

У приложений Kafka множество показателей для отслеживания функционирования. Их столько, что можно легко запутаться, что важно отслеживать, а что можно не учитывать, от простых показателей общей интенсивности трафика до подробных показателей хронометража для всех типов запросов, в том числе по отдельным топикам и разделам. Благодаря им у вас будет подробная информация обо всех производимых в брокере операциях, но они же могут стать настоящим проклятием ответственных за мониторинг системы.

В этой главе мы подробно опишем важнейшие показатели, которые следует контролировать постоянно, и расскажем, как реагировать на их изменения. Мы также рассмотрим некоторые из показателей, которые могут пригодиться при отладке. Конечно, наш список отнюдь не исчерпывающий, поскольку перечень доступных показателей часто меняется и многие из них имеют смысл только для разработчиков ядра Kafka.

Основы показателей

Прежде чем углубиться в рассмотрение показателей брокера и клиентов Kafka, поговорим об основах мониторинга приложений Java и практиках, рекомендуемых к использованию при мониторинге и уведомлении пользователей. Этот фундамент позволит вам понять, как выполнять мониторинг своих приложений и почему мы считаем наиболее важными именно те показатели, которые обсуждаются далее в этой главе.

Как получить доступ к показателям

Ко всем показателям Kafka можно обращаться через интерфейс расширений Java для управления (Java Management Extensions, JMX). Удобнее всего использовать их во внешней системе посредством подсоединения к процессу Kafka агента-сборщика, предоставляемого вашей системой мониторинга. Он может

быть отдельным системным процессом, который подключается к интерфейсу JMX, как это делают плагины `check_jmx` системы мониторинга Nagios XI или `jmxtrans`. Можно также воспользоваться JMX-агентом, запускаемым непосредственно внутри процесса Kafka для доступа к показателям по протоколу HTTP, например Jolokia или MX4J.

Детальное обсуждение настройки агентов мониторинга выходит за рамки данной главы, и вариантов слишком много, чтобы охватить все. Если ваша компания пока не занималась мониторингом приложений Java, возможно, наилучшим решением будет мониторинг как сервис. Существует множество компаний, предлагающих агенты мониторинга, точки сбора показателей, их хранение, графическое отображение и уведомление о проблемах в составе пакета сервисов. Они же помогут вам и в настройке нужных агентов мониторинга.



Как найти порт JMX

Чтобы упростить настройку приложений, подключающихся к JMX брокера Kafka напрямую, следует указать настроенный JMX-порт в настройках брокера, хранимых в ZooKeeper. Z-узел `/brokers/ids/<ID>` содержит данные брокера в формате JSON, включая ключи `hostname` и `jmx_port`. Однако следует отметить, что удаленный JMX по умолчанию отключен в Kafka по соображениям безопасности. Если вы собираетесь включить его, то должны правильно настроить безопасность для порта. Это связано с тем, что JMX позволяет не только просматривать состояние приложения, но и выполнять код. Настоятельно рекомендуется использовать агент показателя JMX, загруженный в приложение.

Неприкладные показатели

Не все показатели будут поступать из самой Kafka. Существует пять основных групп источников, откуда вы можете получать свои показатели. В табл. 13.1 описаны категории показателей, по которым мы проводим мониторинг брокеров Kafka.

Журналы, генерируемые Kafka, обсуждаются далее в этой главе, как и показатели клиента. Мы также вкратце коснемся синтетических показателей. Однако показатели инфраструктуры зависят от вашей конкретной среды и выходят за рамки данного обсуждения. Чем дальше вы продвигаетесь в своем путешествии по Kafka, тем более важными будут эти источники показателей для полного понимания работы ваших приложений, поскольку чем ниже в списке они находятся, тем более объективное представление о Kafka дают. Например, на начальном этапе вам будет достаточно показателей от ваших брокеров, но позже вы захотите получить более объективное представление о том, как они работают. Один из традиционных примеров ценности внешних показателей —

мониторинг состояния веб-сайта. Веб-сервер работает нормально, и все его показатели говорят, что веб-сайт функционирует. Однако существует проблема с сетью между веб-сервером и внешними пользователями, из-за чего веб-сервер недоступен пользователям. Синтетический клиент, выполняемый за пределами сети, с помощью которого проверяют доступность веб-сайта, может обнаружить подобную проблему и уведомить вас о ситуации.

Таблица 13.1. Источники показателей

Категория	Описание
Показатели приложения	Это показатели, которые вы получаете из самой Kafka, из интерфейса JMX
Журналы	Еще один тип данных мониторинга, которые поступают из самой Kafka. Поскольку это не просто число, а некая форма текстовых или структурированных данных, они требуют немного большей обработки
Показатели инфраструктуры	Эти показатели поступают из систем, которые находятся перед Kafka, но все еще в пределах пути запроса и под вашим контролем. Примером может служить балансировщик нагрузки
Синтетические клиенты	Это данные от инструментов, которые являются внешними по отношению к вашему развертыванию Kafka, как и клиент, но находятся под вашим непосредственным контролем и обычно не выполняют ту же работу, что и ваши клиенты. Внешний монитор, такой как Kafka Monitor, относится к этой категории
Показатели клиентов	Это показатели, которые предоставляются клиентами Kafka, подключающимися к вашему кластеру

Какие показатели нам нужны

Какие конкретно показатели важны для вас — вопрос почти такой же сложный, как и вопрос о том, какой редактор лучше использовать. Он будет существенно зависеть от того, что вы собираетесь с ними делать, какие у вас есть инструменты для сбора данных, как далеко вы продвинулись в применении Kafka и сколько времени можете потратить на создание инфраструктуры вокруг Kafka. У разработчика внутренних компонентов брокера будут совершенно иные потребности, чем у инженера по надежности сайта, который занимается развертыванием Kafka.

Оповещение или отладка?

Первый вопрос, который вы должны задать себе, заключается в том, является ли вашей основной целью предупреждение о возникновении проблем с Kafka или отладка возникающих проблем. Ответ обычно включает в себя немного и того

и другого, но знание того, для чего предназначен показатель, позволит вам по-разному относиться к нему после того, как будут собраны данные.

Показатель, предназначенный для оповещения, будет полезен в течение очень короткого периода времени — как правило, ненамного дольше, чем время, необходимое для реагирования на проблему. Это могут быть часы или, возможно, дни. Эти показатели будут использоваться средствами автоматизации, которые реагируют на известные вам проблемы, а также людьми-операторами в тех случаях, когда автоматизации еще не существует. Обычно важно, чтобы эти показатели были более объективными, поскольку проблема, не влияющая на клиентов, гораздо менее критична, чем та, которая влияет.

Данные, предназначенные в первую очередь для отладки, имеют более длительный временной горизонт, поскольку вы часто диагностируете проблемы, которые существуют уже в течение некоторого времени, или углубленно изучаете более сложную проблему. Эти данные должны оставаться доступными в течение нескольких дней или недель после того, как они были собраны. Кроме того, обычно это будут более субъективные измерения или данные из самого приложения Kafka. Следует помнить, что не всегда необходимо собирать эти данные в систему мониторинга. Если показатели используются для отладки проблем на месте, достаточно, чтобы они были доступны, когда это необходимо. Вам не нужно перегружать систему мониторинга, постоянно собирая десятки тысяч значений.



Исторические показатели

Существует третий тип данных, которые вам в конечном итоге понадобятся, — исторические данные о вашем приложении. Чаще всего они применяются для управления пропускными способностями, поэтому включают информацию об используемых ресурсах, в том числе о вычислительных ресурсах, хранилище и сети. Эти показатели необходимо будет хранить в течение очень длительного периода времени, измеряемого годами. Вам также может потребоваться собрать дополнительные метаданные, чтобы вписать показатели в контекст, например, когда брокеры были добавлены в кластер или удалены из него.

Автоматизация или люди?

Еще один вопрос, который следует рассмотреть: кто будет потребителем показателей? Если показатели будут использоваться средствами автоматизации, они должны быть очень конкретными. Вполне нормально иметь большое количество показателей, каждый из которых описывает мелкие детали, потому что компьютеры существуют именно для этого — для обработки большого количества данных. Чем конкретнее данные, тем легче создать автоматизиро-

ванный процесс, который будет действовать на их основе, потому что такие данные не оставляют возможности для интерпретации их значений. Но если показатели будут потребляться людьми, им окажется трудно справиться с представлением большого их количества. Это становится еще более важным при определении предупреждений на основе этих измерений. Слишком легко устать от оповещений, когда их появляется так много, что трудно понять, насколько серьезна проблема. Также трудно правильно определить пороговые значения для каждого показателя и поддерживать их в актуальном состоянии. Когда оповещений становится слишком много или они часто неверны, мы начинаем сомневаться в том, что оповещения правильно описывают состояние наших приложений.

Это можно сравнить с работой автомобиля. Чтобы правильно отрегулировать соотношение воздуха и топлива во время движения автомобиля, компьютеру необходимо провести ряд измерений плотности воздуха, топлива, выхлопных газов и других мелочей, связанных с работой двигателя. Однако эти измерения были бы непосильны для человека, управляющего автомобилем. Вместо этого у нас есть индикатор «проверьте двигатель». Один индикатор говорит вам о наличии проблемы, и есть способ получить более подробную информацию, которая точно скажет вам, в чем проблема. На протяжении данной главы мы будем отмечать показатели с максимальным охватом, позволяющие упростить систему оповещения.

Контроль состояния приложения

Вне зависимости от способа сбора показателей Kafka необходимо также контролировать общее состояние процесса приложения с помощью простой проверки рабочего состояния. Сделать это можно двумя способами:

- с помощью внешнего процесса, который сообщает, работает брокер или отключен (проверка состояния);
- посредством того, что брокер Kafka оповещает об отсутствии показателей (которые иногда называют *устаревшими показателями*).

Хотя второй метод работает, при его использовании трудно отличить сбой брокера Kafka от сбоя самой системы мониторинга.

Для брокера Kafka такой контроль состояния можно выполнить, просто подключившись к внешнему порту (тому самому, который используют для подключения к брокеру клиенты) и проверив, отвечает ли брокер. Для клиентских приложений задача усложняется и может варьироваться от простой проверки того, работает ли процесс, до написания внутреннего метода, который определяет состояние приложения.

Цели на уровне обслуживания

Одной из областей мониторинга, которая особенно важна для инфраструктурных сервисов, таких как Kafka, является определение целей на уровне обслуживания, или SLO. Так мы сообщаем нашим клиентам, какого уровня обслуживания они могут ожидать от инфраструктурного сервиса. Клиенты хотят иметь возможность относиться к таким сервисам, как Kafka, как к непрозрачной системе: они не хотят и не должны понимать внутреннюю суть того, как она работает, — им нужен только интерфейс, который они используют, и уверенность, что он будет делать то, что им нужно.

Определения уровня сервиса

Прежде чем обсуждать цели на уровне обслуживания в Kafka, необходимо договориться относительно применяемой терминологии. Часто можно услышать, как инженеры, менеджеры, руководители и все остальные неправильно используют термины в области «уровня обслуживания», что приводит к путанице в понимании того, о чем, собственно, идет речь.

Индикатор уровня обслуживания (service level indicators, SLI) — это показатель, описывающий один из аспектов надежности сервиса. Он должен быть тесно связан с опытом вашего клиента, поэтому обычно верно то, что чем объективнее эти измерения, тем они лучше. В системе обработки запросов, такой как Kafka, обычно лучше всего выражать эти измерения как отношение между количеством хороших событий и общим количеством событий — например, доля запросов к веб-серверу, которые возвращают ответ 2xx, 3xx или 4xx.

Целевой уровень обслуживания (service level objectives, SLO), которую также можно назвать *порогом уровня обслуживания* (service-level threshold, SLT), объединяет SLI с целевым значением. Обычно цель выражается числом девяток (99,9 % — это «три девятки»), хотя это отнюдь не обязательно. Целевой уровень обслуживания должен включать также временные рамки, в течение которых она измеряется, часто в масштабе дней. Например, 99 % запросов к веб-серверу должны возвращать ответ 2xx, 3xx или 4xx в течение семи дней.

Соглашение об уровне обслуживания (service level agreements, SLA) — это контракт между поставщиком услуг и клиентом. Оно обычно включает в себя несколько целевых уровней обслуживания, а также подробную информацию о том, как они измеряются и сообщаются, как клиент обращается за поддержкой к поставщику услуг и каковы штрафные санкции, которые будут применены к поставщику услуг в случае невыполнения им соглашения об уровне обслуживания. Например, в соглашении об уровне обслуживания для предыдущего целевого уровня

обслуживания может быть указано, что если поставщик услуг не работает в рамках цели уровня обслуживания, то он вернет клиенту всю плату за период, в течение которого услуга не соответствовала целевому уровню обслуживания.



Соглашение на оперативном уровне

Термин «операционное соглашение об уровнях поддержки» (operational-level agreement, OLA) используется реже. Он описывает соглашения между несколькими внутренними службами или поставщиками услуг в рамках общего выполнения соглашения об уровне обслуживания. Цель заключается в обеспечении того, чтобы многочисленные виды деятельности, необходимые для выполнения соглашения об уровне обслуживания, были должным образом описаны и учтены в повседневных операциях.

Очень часто можно услышать, как люди говорят о соглашении об уровне обслуживания, хотя на самом деле они имеют в виду целевой уровень обслуживания. Хотя те, кто предоставляет услуги платным клиентам, могут иметь соглашения об уровне обслуживания с этими клиентами, очень редко инженеры, запускающие приложения, отвечают за что-то большее, чем производительность этой услуги в рамках целевого уровня обслуживания. Кроме того, те, кто имеет только внутренних клиентов (то есть использует Kafka в качестве внутренней инфраструктуры данных для гораздо более крупного сервиса), обычно не имеют соглашений об уровне обслуживания с этими внутренними клиентами. Однако это не должно мешать вам устанавливать и сообщать целевые уровни обслуживания, так как это приведет к уменьшению предположений клиентов о том, как, по их мнению, должна работать Kafka.

Какие показатели являются хорошими индикаторами уровня обслуживания

В целом показатели для ваших индикаторов уровня обслуживания должны собираться с помощью чего-то внешнего по отношению к брокерам Kafka. Причина этого в том, что целевые уровни обслуживания должны описывать, доволен или нет типичный пользователь вашего сервиса, и вы не можете измерить это субъективно. Ваших клиентов не волнует, считаете ли вы, что ваш сервис работает правильно, — важен их опыт (в совокупности). Это означает, что показатели инфраструктуры в порядке, комплексные клиенты в порядке, а показатели на стороне клиента, вероятно, являются лучшими для большинства ваших индикаторов уровня обслуживания.

Наиболее распространенные индикаторы уровня обслуживания, используемые в системах запроса/ответа и хранения данных, приведены в табл. 13.2 (это далеко не исчерпывающий список).



Клиенты всегда хотят большего

Есть некоторые целевые уровни обслуживания, которые могут интересовать ваших клиентов и которые важны для них, но не находятся под вашим контролем. Например, их может волновать корректность или новизна данных, поступающих в Kafka. Не соглашайтесь поддерживать целевые уровни обслуживания, за которые вы не несете ответственности, так как это приведет лишь к тому, что вы возьмете на себя работу, которая снизит качество основной задачи по обеспечению корректной работы Kafka. Убедитесь, что вы передали их соответствующему отделу, чтобы установить взаимопонимание и соглашения в отношении этих дополнительных требований.

Таблица 13.2. Типы индикаторов уровня обслуживания

Доступность	Может ли клиент сделать запрос и получить ответ?
Задержка	Как быстро возвращается ответ?
Качество	Надлежащий ли ответ?
Безопасность	Защищены ли запрос и ответ соответствующим образом, будь то авторизация или шифрование?
Пропускная способность	Может ли клиент получить достаточно данных и достаточно быстро?

Имейте в виду, что обычно лучше, чтобы ваши индикаторы уровня обслуживания были основаны на счетчике событий, которые попадают в пороговые значения целевого уровня обслуживания. Это означает, что в идеале каждое событие следует проверять в индивидуальном порядке, чтобы увидеть, соответствует ли оно пороговому значению цели уровня обслуживания. Это исключает квантильные показатели как хорошие индикаторы уровня обслуживания, поскольку они будут только сообщать вам о том, что 90 % ваших событий были ниже заданного значения, не позволяя контролировать это значение. Тем не менее объединение значений в сегменты (например, «менее 10 мс», «10–50 мс», «50–100 мс» и т. д.) может быть полезным при работе с целевыми уровнями обслуживания, особенно если вы еще не уверены в том, что такое хороший порог. Это даст вам представление о распределении событий в диапазоне целевых уровней обслуживания, и вы сможете настроить сегменты так, чтобы границы были разумными для порогового значения целевого уровня обслуживания.

Использование целевого уровня обслуживания для оповещений

Одним словом, целевые уровни обслуживания должны стать базой для ваших основных оповещений. Причина этого в том, что целевые уровни обслуживания описывают проблемы с точки зрения ваших клиентов и именно они должны волновать вас в первую очередь. По большому счету, если проблема

не затрагивает ваших клиентов, она не должна будить вас по ночам. Целевые уровни обслуживания также расскажут вам о проблемах, которые вы не знаете, как обнаружить, потому что никогда не сталкивались с ними раньше. Они не скажут вам, в чем заключаются эти проблемы, но проинформируют о том, что те существуют.

Проблема заключается в том, что очень сложно использовать целевые уровни обслуживания непосредственно в качестве оповещения. Целевые уровни обслуживания лучше всего подходят для длительных периодов времени, например недели, поскольку мы хотим сообщать о них руководству и клиентам таким образом, чтобы их можно было использовать. Кроме того, к тому времени, когда срабатывает предупреждение целевого уровня обслуживания, будет уже слишком поздно — вы уже работаете за пределами целевого уровня обслуживания. Некоторые применяют производное значение для раннего предупреждения, но лучший способ использовать целевые уровни обслуживания для оповещения — это наблюдать за тем, с какой скоростью они выгорают за определенный период времени.

В качестве примера предположим, что ваш кластер Kafka получает 1 миллион запросов в неделю и у вас есть целевой уровень обслуживания, в котором указано, что 99,9 % запросов должны отправлять первый байт ответа в течение 10 мс. Это означает, что в течение недели у вас может быть до 1000 запросов, которым для ответа требуется больше времени, и все равно все будет в порядке. Обычно вы видите один такой запрос каждый час, что составляет около 168 плохих запросов в неделю, выявляемых с воскресенья по субботу. У вас есть показатель «скорость выгорания целевого уровня обслуживания», и один запрос в час при миллионе запросов в неделю — это скорость выгорания 0,1 % в час.

Во вторник в 10:00 показатель изменяется и теперь говорит о том, что коэффициент выгорания составляет 0,4 % в час. Это не очень хорошо, но все же не проблема, потому что к концу недели вы будете находиться в пределах целевого уровня обслуживания. Вы открываете заявку, чтобы рассмотреть проблему, но возвращаетесь к какой-то более приоритетной работе. В среду в 14:00 скорость выгорания возрастает до 2 % в час, и у вас срабатывают предупреждения. Вы знаете, что при таком темпе нарушите целевой уровень обслуживания к обеду в пятницу. Бросив все дела, вы диагностируете проблему и примерно через 4 часа выгорания снова скорость выгорания до 0,4 % в час, и она остается на этом уровне до конца недели. Используя коэффициент скорости выгорания, вы смогли избежать нарушения целевого уровня обслуживания за эту неделю.

Получить более подробную информацию об использовании целевого уровня обслуживания и коэффициента выгорания для оповещения вы можете из превосходных книг под редакцией Бетси Бейер (Betsy Beyer) и других «Site Reliability Engineering. Надежность и безотказность как в Google» (Питер, 2019) и «Site Reliability Workbook: практическое применение» (Питер, 2021).

Показатели брокеров Kafka

Существует множество показателей брокеров Kafka. Многие из них представляют собой низкоуровневые показатели, добавленные разработчиками при поиске причин конкретных проблем или ради возможности получения отладочной информации в будущем. Имеются показатели практически по каждой функции в брокере, но чаще всего используются те, которые дают необходимую для ежедневной работы Kafka информацию.



Кто наблюдает за наблюдателями

Множество компаний используют Kafka для сбора показателей приложений, системных показателей и журналов для дальнейшей отправки в централизованную систему мониторинга. Это отличный способ отделения приложений от системы мониторинга, но существует нюанс, связанный с Kafka. Если использовать эту же систему для мониторинга самой Kafka, то очень вероятно, что вы не узнаете о сбое в ее функционировании, поскольку поток данных системы мониторинга тоже будет прерван.

Существует множество путей решения этой проблемы. Один из них — воспользоваться для Kafka отдельной, независимой от нее системой мониторинга. Другой способ: при наличии нескольких ЦОД сделать так, чтобы показатели кластера Kafka в ЦОД А отправлялись в ЦОД В и наоборот. Какой бы способ вы ни выбрали, главное, чтобы мониторинг и оповещение о проблемах Kafka не зависели от ее функционирования.

Этот раздел мы начнем с обсуждения высокоуровневого рабочего процесса для диагностики проблем с вашим кластером Kafka, ссылаясь на полезные показатели. Эти и другие показатели будут более подробно описаны далее в этой главе. Конечно, это отнюдь не исчерпывающий список показателей брокеров, а лишь некоторые, совершенно необходимые для проверки состояния брокера и кластера. Завершим эту тему обсуждением вопроса журналирования, после чего перейдем к показателям клиентов.

Диагностика проблем с кластером

Когда речь идет о проблемах с кластером Kafka, можно выделить три основные категории:

- проблемы с одним брокером;
- перегруженные кластеры;
- проблемы с контроллером.

Безусловно, легче всего диагностировать проблемы с отдельными брокерами и реагировать на них. Они проявляются как отклонения в показателях кластера

и часто связаны с медленными или неработающими устройствами хранения данных или ограничениями вычислений со стороны других приложений в системе. Чтобы их обнаружить, убедитесь, что вы отслеживаете доступность отдельных серверов, а также состояние устройств хранения данных, используя показатели операционной системы (ОС).

Однако если проблема не выявлена на уровне операционной системы или аппаратного обеспечения, причиной почти всегда является дисбаланс нагрузки на кластер Kafka. Хотя Kafka пытается обеспечить равномерное распределение данных внутри кластера между всеми брокерами, это не означает, что доступ клиентов к этим данным распределяется равномерно. Он также не обнаруживает такие проблемы, как «горячие» разделы. Настоятельно рекомендуется использовать внешний инструмент для постоянного поддержания сбалансированности кластера. Одним из таких инструментов является Cruise Control (<https://oreil.ly/rLybu>) — приложение, которое постоянно следит за кластером и восстанавливает баланс разделов внутри него. Оно также предоставляет ряд других административных функций, таких как добавление и удаление брокеров.



Выборы предпочтительных реплик

Прежде чем пытаться диагностировать проблему дальше, необходимо убедиться, что вы недавно проводили выборы предпочтительных реплик (см. главу 12). Брокеры Kafka не становятся автоматически вновь ведущими для разделов (если не включена автоматическая перебалансировка ведущих реплик) после потери статуса ведущей реплики (например, когда брокер вышел из строя или был выключен). Это означает, что ведущие реплики очень легко могут стать несбалансированными в кластере. Выборы предпочтительных реплик безопасны и просты, поэтому рекомендуется сначала сделать это и посмотреть, исчезнет ли проблема.

Перегруженные кластеры — еще одна проблема, которую легко обнаружить. Если кластер сбалансирован, но многие брокеры показывают повышенную задержку запросов или низкий коэффициент простоя пула обработчиков запросов, вы достигли предела возможностей ваших брокеров по обслуживанию трафика для этого кластера. При более глубокой проверке вы можете обнаружить, что у вас есть клиент, который изменил свой шаблон запросов и теперь вызывает проблемы. Однако, даже если это произойдет, вы мало что сможете сделать для изменения клиента. Доступные вам решения заключаются в том, чтобы либо снизить нагрузку на кластер, либо увеличить количество брокеров.

Проблемы с контроллером в кластере Kafka гораздо сложнее диагностировать, и они часто попадают в категорию ошибок в самой Kafka. Эти проблемы проявляются в виде рассинхронизации метаданных брокера, неработающих реплик, когда брокеры, казалось бы, в порядке, и неправильного выполнения действий по управлению топиками, таких как создание. Если вы ломаете голову над

проблемой в кластере и говорите: «Это действительно странно», есть большая вероятность того, что это происходит, потому что контроллер сделал что-то непредсказуемое и некорректное. Существует не так уж много способов мониторинга контроллера, но мониторинг количества активных контроллеров, а также размера очереди контроллеров даст вам высокоуровневый индикатор наличия проблемы.

Искусство недореплицированных разделов

Одним из наиболее популярных показателей, используемых при мониторинге Kafka, являются недореплицированные разделы. Этот показатель, взятый для каждого брокера кластера, представляет собой число разделов, для которых этот брокер является ведущей репликой (ведомые реплики отстают). Он позволяет охватить взглядом множество проблем с кластером Kafka от аварийного останова брокера до исчерпания доступных ресурсов. Учитывая разнообразие проблем, на которые может указывать отличное от нуля значение этого показателя, имеет смысл выяснить подробнее, как реагировать на него. Многие из показателей, используемых при диагностике подобных проблем, будут описаны далее в этой главе. В табл. 13.3 приведена более подробная информация по поводу недореплицированных разделов.

Таблица 13.3. Показатели и соответствующие им недореплицированные разделы

Показатель	Недореплицированные разделы
Управляемый компонент (MBean) JMX	<code>kafka.server:type=ReplicaManager,name=UnderReplicatedPartitions</code>
Диапазон значений	Целое число, равное нулю или больше его



Ловушка оповещения URP

В предыдущем издании этой книги, а также во многих выступлениях на конференциях авторы подробно говорили о том, что показатель недореплицированных разделов (URP) должен быть вашим основным показателем оповещения из-за того, как много проблем он описывает. Этот подход порождает значительное количество проблем, не последней из которых является то, что показатель URP часто может быть отличным от нуля по безобидным причинам. Это означает, что вы, управляя кластером Kafka, будете получать ложные предупреждения, что приведет к их игнорированию. Кроме того, чтобы понять, о чем говорит показатель, требуется значительный объем знаний. По этой причине мы больше не рекомендуем использовать URP для оповещения. Вместо этого для обнаружения неизвестных проблем следует полагаться на оповещения на основе SLO.

Если многие брокеры кластера сообщают о постоянном (не меняющемся) числе недореплицированных разделов, то обычно это значит, что один из брокеров кластера отключен. Число недореплицированных разделов в масштабе всего кластера будет равно числу распределенных на этот брокер разделов, и отказавший брокер не будет отправлять показатели. В этом случае необходимо выяснить, что случилось с брокером, и решить проблему. Зачастую причиной становится отказ аппаратного обеспечения либо проблема с операционной системой или Java.

Если число недореплицированных разделов меняется или оно постоянно, но отключенных брокеров нет, то дело обычно в проблеме с производительностью кластера. Искать причины подобных проблем довольно сложно из-за их разнообразия, но существует алгоритм из нескольких шагов, с помощью которого можно сузить список возможных до наиболее вероятных. Первый шаг — попытаться выяснить, связана проблема с отдельным брокером или со всем кластером. Иногда ответить на этот вопрос непросто. Если недореплицированные разделы находятся в одном брокере, как показано в следующем примере, то обычно причина проблемы именно в нем и ошибка указывает на то, что у других брокеров возникают затруднения при репликации сообщений из него.

Если недореплицированные разделы есть на нескольких брокерах, дело может быть в проблеме с кластером или с отдельным брокером, поскольку у одного из брокеров могут возникнуть затруднения с репликацией сообщений из всех других мест и придется выяснять, о каком именно брокере идет речь. Для этого можно, например, получить список недореплицированных разделов кластера и посмотреть, не относятся ли все недореплицированные разделы к одному брокеру. А вывести список недореплицированных разделов можно с помощью утилиты `kafka-topics.sh`, которую мы подробно обсуждали в главе 12.

Например, выведем список недореплицированных разделов кластера:

```
# kafka-topics.sh --bootstrap-server kafka1.example.com:9092/kafka-cluster
--describe --under-replicated
  Topic: topicOne   Partition: 5   Leader: 1   Replicas: 1,2   Isr: 1
  Topic: topicOne   Partition: 6   Leader: 3   Replicas: 2,3   Isr: 3
  Topic: topicTwo   Partition: 3   Leader: 4   Replicas: 2,4   Isr: 4
  Topic: topicTwo   Partition: 7   Leader: 5   Replicas: 5,2   Isr: 5
  Topic: topicSix   Partition: 1   Leader: 3   Replicas: 2,3   Isr: 3
  Topic: topicSix   Partition: 2   Leader: 1   Replicas: 1,2   Isr: 1
  Topic: topicSix   Partition: 5   Leader: 6   Replicas: 2,6   Isr: 6
  Topic: topicSix   Partition: 7   Leader: 7   Replicas: 7,2   Isr: 7
  Topic: topicNine  Partition: 1   Leader: 1   Replicas: 1,2   Isr: 1
  Topic: topicNine  Partition: 3   Leader: 3   Replicas: 2,3   Isr: 3
  Topic: topicNine  Partition: 4   Leader: 3   Replicas: 3,2   Isr: 3
  Topic: topicNine  Partition: 7   Leader: 3   Replicas: 2,3   Isr: 3
  Topic: topicNine  Partition: 0   Leader: 3   Replicas: 2,3   Isr: 3
  Topic: topicNine  Partition: 5   Leader: 6   Replicas: 6,2   Isr: 6
#
```

В этом примере общий для всех брокер под номером 2. Это указывает на то, что у него есть проблемы с репликацией сообщений, поэтому имеет смысл сосредоточиться на нем. Если общего брокера в списке не видно, то проблема, вероятнее всего, с кластером в целом.

Проблемы уровня кластера

Проблемы с кластером обычно относятся к одной из двух таких категорий, как:

- дисбаланс нагрузки;
- исчерпание ресурсов.

Источник первой из них — дисбаланс разделов или ведущих реплик — найти нетрудно, но решить ее может оказаться сложно. Для диагностики вам понадобятся от брокеров кластера следующие данные:

- число разделов;
- число ведущих разделов;
- суммарная частота входящих сообщений по всем топикам;
- суммарная входящая скорость передачи данных по всем топикам (байт/с);
- суммарная исходящая скорость передачи данных по всем топикам (байт/с).

Изучите эти показатели. В идеально сбалансированном кластере они будут примерно одинаковыми для всех брокеров кластера, как в табл. 13.4.

Таблица 13.4. Показатели использования кластера

Брокер	Раздел	Ведущая реплика	Входящие сообщения, сообщений/с	Входящих байтов, Мбайт/с	Исходящих байтов, Мбайт/с
1	100	50	13 130	3,56	9,45
2	101	49	12 842	3,66	9,25
3	100	50	13 086	3,23	9,82

Как видите, все брокеры получают примерно одинаковый объем входящего трафика. В предположении, что вы уже выбрали предпочтительную реплику, сильное отклонение указывает на дисбаланс трафика в кластере. Для решения этой проблемы необходимо переместить разделы из более нагруженных брокеров в менее нагруженные. Сделать это можно с помощью утилиты `kafka-reassign-partitions.sh`, описанной в главе 12.



Вспомогательные утилиты для балансировки кластера

Сами брокеры Kafka не позволяют автоматически переназначать разделы в кластере. Это значит, что балансировка нагрузки в нем превращается в изнурительный процесс просмотра длинных списков показателей в ручном режиме и попыток найти удачное распределение реплик. Чтобы упростить его, в некоторых организациях были разработаны специальные автоматизированные утилиты. Одна из них — утилита `kafka-assigner`, размещенная компанией LinkedIn в репозитории с открытым исходным кодом `kafka-tools` на GitHub (<https://oreil.ly/8ilPw>). Эту возможность содержат и некоторые коммерческие предложения по поддержке Kafka.

Еще одна распространенная проблема с производительностью кластера — превышение пределов возможностей брокеров по обслуживанию запросов. Замедлить работу могут различные узкие места, среди которых наиболее часто встречаются CPU, дисковый ввод/вывод, пропускная способность сети. К ним не относится переполнение дисков, поскольку брокеры работают нормально вплоть до заполнения диска, после чего происходит внезапный отказ. Для диагностики подобных проблем существует множество показателей, которые можно отслеживать на уровне операционной системы, в том числе:

- использование CPU;
- пропускная способность сети на входе;
- пропускная способность сети на выходе;
- среднее время ожидания диска;
- процент использования диска.

Исчерпание любого из этих ресурсов будет проявляться одинаково — в виде недореплицированных разделов. Важно не забывать, что процесс репликации брокеров работает точно так же, как и другие клиенты Kafka. В случае проблем с репликацией у вашего кластера Kafka становятся неизбежными проблемы с потреблением и генерацией сообщений у заказчиков. Имеет смысл выработать эталонное значение этих показателей, при котором кластер работает должным образом, после чего задать пороговые значения, которые указывали бы на возникновение проблемы задолго до исчерпания ресурсов. Не помешает также понаблюдать за тенденциями их изменения при росте поступающего в кластер трафика. Если говорить о показателях брокеров Kafka, то суммарная по всем топикам исходящая скорость передачи данных в байтах (`All Topics Bytes In Rate`) отлично иллюстрирует использование кластера.

Проблемы уровня хоста

Если проблемами с производительностью Kafka охвачен не весь кластер — они возникают в одном или двух брокерах, то имеет смысл взглянуть на соответствующий сервер и разобраться, чем он отличается от остального кластера. Подобные проблемы делятся на следующие общие категории:

- отказы аппаратного обеспечения;
- неполадки в сетевом взаимодействии;
- конфликты между процессами;
- различия локальных настроек.



Типичные серверы и проблемы

Сервер и его операционная система — сложный механизм из тысяч компонентов, в любом из которых может возникнуть проблема, приводящая к полному отказу или просто к снижению производительности. Охватить в нашей книге все возможные сбои нереально — на эту тему уже написано множество огромных томов и все время создаются новые. Но в наших силах обсудить некоторые наиболее распространенные из них. В этом разделе мы сосредоточимся на проблемах с типичным сервером под управлением операционной системы Linux.

Сбой аппаратного обеспечения — вещь очевидная, при этом сервер просто перестает работать, а снижение производительности бывает вызвано менее очевидными проблемами. Обычно они представляют собой случайным образом возникающие ошибки, при которых система продолжает работать, но менее эффективно. В их числе сбойные участки памяти, обнаруженные системой и требующие обхода, вследствие чего снижается общий доступный объем памяти. Аналогичная ситуация может возникнуть с CPU. Для решения подобных проблем следует использовать возможности, предоставляемые аппаратным обеспечением, например интеллектуальный интерфейс управления платформой (intelligent platform management interface, IPMI) для мониторинга состояния аппаратного обеспечения. При наличии проблемы вы сможете с помощью утилиты `dmesg`, отображающей буфер ядра, увидеть поступающие в консоль системы журнальные сообщения.

Более распространенный тип аппаратного сбоя, приводящий к снижению производительности Kafka, — отказ диска. Apache Kafka необходимы диски для сохранения сообщений, так что производительность производителей напрямую связана со скоростью фиксации дисками этих операций записи. Любые отклонения в работе дисков выражаются в проблемах с производительностью

производителей и потоков извлечения данных из реплик. Именно последнее обстоятельство приводит к образованию недореплицированных разделов. Поэтому важно постоянно отслеживать состояние дисков и быстро решать возникающие проблемы.



Одна паршивая овца

Отказ одного-единственного диска в одном-единственном брокере может свести на нет производительность всего кластера. Дело в том, что клиенты-производители подключаются ко всем брокерам, в которых располагаются ведущие разделы для топика, а если вы следовали рекомендациям, то эти разделы будут равномерно распределены по всему кластеру. Ухудшение работы одного-единственного брокера и замедление запросов производителей приведет к отрицательному обратному воздействию на производители и замедлению запросов ко всем брокерам.

Прежде всего отслеживайте информацию о состоянии дисков с помощью IPMI или другого интерфейса аппаратного обеспечения. Кроме того, запустите в операционной системе утилиты SMART (self-monitoring, analysis and reporting technology — технология автоматического мониторинга, анализа и оповещения) для мониторинга и регулярного тестирования дисков. Благодаря им вы заранее узнаете о надвигающихся отказах. Важно также следить за контроллером диска, особенно если у него есть функциональность RAID, вне зависимости от того, используете вы аппаратный RAID или нет. У многих контроллеров имеется встроенный кэш, используемый только при нормальном состоянии контроллера и работающей резервной батарее (battery backup unit, BBU). Отказ BBU может привести к отключению кэша и снижению производительности диска.

Передача данных по сети — еще одна сфера, в которой частичные сбои могут вызывать проблемы. Некоторые из них обусловлены неполадками в аппаратном обеспечении, например испорченным сетевым кабелем или коннектором. Некоторые связаны с настройками на стороне сервера или ближе по конвейеру, в сетевом аппаратном обеспечении. Проблемы настройки сети могут выразиться также в проблемах операционной системы, например в недостаточном размере сетевых буферов или в ситуации, когда слишком много сетевых подключений требуют слишком большой доли общего объема памяти. Один из ключевых индикаторов в этой сфере — число зафиксированных на сетевых интерфейсах ошибок. Если это число растет, то, вероятно, имеется нерешенная проблема.

Если аппаратных проблем нет, то часто имеет смысл поискать работающее в той же системе другое приложение, которое потребляет ресурсы и затрудняет работу брокера Kafka. Это может быть установленное по ошибке приложение или процесс, например, мониторинговый агент, который, как предполагается,

работает, но на деле имеет какие-то проблемы. Воспользуйтесь системными утилитами, например `top`, для поиска процессов, которые задействуют больше процессорного времени или оперативной памяти, чем ожидается.

Если все возможности исчерпаны, а вы так и не нашли причину ненормальной работы конкретного хоста, то, вероятно, существует разница в настройках по сравнению или с брокером, или с самой системой. Учитывая количество приложений, работающих на любом сервере, и количество настроек каждого из них, поиск различий — поистине титаническая работа. Поэтому так важно использовать системы управления настройками, такие как Chef (<https://www.chef.io>) или Puppet (<https://puppet.com>), для поддержания согласованности настроек во всех ваших операционных системах и приложениях, включая Kafka.

Показатели брокеров

Помимо недореплицированных разделов, есть и другие показатели брокера в целом, которые желательно отслеживать. Хотя не обязательно задавать пороговые значения оповещения для всех них, они служат источником ценной информации о брокерах и кластере. Их желательно включать во все создаваемые вами панели инструментов мониторинга.

Признак текущего контроллера

Показатель *«признак текущего контроллера»* (active controller count) указывает, является ли данный брокер текущим контроллером кластера. Он может принимать значения 0 и 1, где 1 указывает на то, что данный брокер сейчас является контроллером. В любой момент контроллером может быть только один брокер, и, наоборот, какой-то один брокер всегда обязан быть контроллером кластера. Если два брокера утверждают, что являются текущим контроллером кластера, то имеется проблема: поток выполнения контроллера не завершил работу как полагается, а завис. Вследствие этого может оказаться невозможно выполнять административные задачи, например перемещения разделов, должным образом. Чтобы исправить ситуацию, необходимо как минимум перезапустить оба брокера. Однако в случае появления лишнего контроллера в кластере при безопасном выключении брокеров зачастую возникают проблемы, и вам придется принудительно остановить брокер. Подробности относительно признака текущего контроллера смотрите в табл. 13.5.

Если ни один из брокеров не претендует на звание контроллера кластера, последний не сможет должным образом реагировать на изменения состояния, включая создание топиков/разделов и сбой брокеров. В подобном случае для выяснения того, почему потоки выполнения контроллеров не работают как полагается, необходимо провести расследование. К такой ситуации может привести, например,

нарушение связности сети с кластером ZooKeeper. После исправления лежащей в его основе проблемы имеет смысл перезапустить все брокеры кластера для сброса состояния потоков выполнения контроллеров.

Таблица 13.5. Подробная информация о показателе «признак текущего контроллера»

Показатель	Признак текущего контроллера
Управляемый компонент (MBean) JMX	<code>kafka.controller:type=KafkaController, name=ActiveControllerCount</code>
Диапазон значений	0 или 1

Размер очереди контроллера

Показатель размера очереди контроллера показывает, обработки скольких запросов от брокеров контроллер ожидает в данный момент. Показатель будет равен 0 или больше, при этом значение часто колеблется по мере поступления новых запросов от брокеров и выполнения административных действий, таких как создание либо перемещение разделов и обработка изменений лидера. Следует ожидать скачков показателя, но если это значение постоянно увеличивается или остается стабильно высоким и не падает, то контроллер может зависнуть. Это может вызвать проблемы, связанные с невозможностью корректного выполнения административных задач. Чтобы исправить ситуацию, необходимо переместить контроллер в другой брокер, что требует выключения брокера, в котором в данный момент находится контроллер. Однако, когда контроллер зависает, часто возникают проблемы с выполнением контролируемого выключения любого брокера. Более подробная информация о размере очереди контроллера указана в табл. 13.6.

Таблица 13.6. Подробная информация о показателе размера очереди контроллера

Название показателя	Размер очереди контроллера
Управляемый компонент (MBean) JMX	<code>kafka.controller:type=ControllerEventManager,name=EventQueueSize</code>
Диапазон значений	Целое число, 0 или более

Коэффициент простоя обработчиков запросов

Kafka использует два пула потоков выполнения для обработки всех запросов клиентов: *сетевые потоки* и *потоки обработчиков запросов* (также называемые *потоками ввода/вывода*). Сетевые потоки отвечают за чтение данных и их запись в клиенты по сети. Это не требует больших вычислительных затрат, то есть вероятность исчерпать ресурсы сетевых потоков невелика. Потоки же

обработчиков запросов отвечают за обслуживание самих запросов клиентов, к которому относятся чтение сообщений с диска или их запись на диск. Следовательно, при повышении загруженности брокеров влияние на этот пул потоков существенно возрастает (подробности о коэффициенте простоя обработчиков запросов см. в табл. 13.7).

Таблица 13.7. Подробная информация о коэффициенте простоя обработчиков запросов

Показатель	Признак текущего контроллера
Управляемый компонент (MBean) JMX	<code>kafka.server:type=KafkaRequestHandlerPool, name=RequestHandlerAvgIdlePercent</code>
Диапазон значений	Число с плавающей запятой между 0 и 1 включительно



Разумное использование потоков

Может показаться, что вам понадобятся сотни потоков обработчиков запросов, однако на деле нет необходимости задавать в настройках больше потоков, чем процессоров в брокере. Apache Kafka весьма разумно использует обработчики запросов, выгружая в буфер-чистилище запросы, обработка которых займет много времени. Это применяется, в частности, при ограничениях на запросы в виде квот или в случае, когда требуется более одного подтверждения запроса производителя.

Показатель «коэффициент простоя обработчиков запросов» отражает долю времени (в процентах), в течение которого обработчики запросов не используются. Чем меньше это значение, тем сильнее загружен брокер. По нашему опыту, коэффициент простоя меньше 20 % указывает на потенциальную проблему, а меньше 10 % — на возникшую проблему с производительностью. Помимо слишком маломощного кластера, существует две возможные причины повышенного коэффициента использования потоков пула. Первая — в пуле недостаточно потоков. Вообще говоря, число потоков обработчиков запросов должно быть равно числу процессоров системы, включая процессоры с технологией *hyper-threading*.

Вторая часто встречающаяся причина — выполнение потоками ненужной работы для каждого запроса. До версии Kafka 0.10 поток обработчика запросов отвечал за распаковку пакетов всех входящих запросов, проверку сообщений и назначение смещений, а также дальнейшую упаковку пакета сообщения со смещениями перед записью на диск. Осложняли ситуацию синхронные блокировки всех методов сжатия. В версии 0.10 появился новый формат сообщений с относительными смещениями в пакетах сообщений. Это значит, что производители новых версий задают относительные смещения перед отправкой пакетов сообщений, благодаря чему брокер может пропустить шаг распаковки

пакета сообщений. Одно из важнейших усовершенствований, которое вы можете внести в свою систему, — обеспечение поддержки клиентами производителей и потребителей формата сообщений 0.10 и изменение версии формата сообщений в брокерах тоже на 0.10. Это приведет к колоссальному снижению использования потоков обработчиков запросов.

Суммарная входящая скорость передачи данных

Суммарная по всем топикам входящая скорость передачи данных, выраженная в байтах в секунду, может оказаться полезной в качестве показателя количества сообщений, получаемых брокерами от клиентов-производителей. Этот показатель удобен при определении того, когда нужно расширять кластер или выполнять другие работы, связанные с масштабированием. Не помешает вычислить его и в случае, когда один из брокеров кластера получает больше трафика, чем другие, что может говорить о необходимости переназначения разделов кластера (подробности — в табл. 13.8).

Таблица 13.8. Подробная информация о показателе «суммарная по всем топикам входящая скорость передачи данных»

Показатель	Входящая скорость, байт/с
Управляемый компонент (MBean) JMX	<code>kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec</code>
Диапазон значений	Скорость — число с двойной точностью, количество — целочисленное значение

Поскольку это первый из обсуждаемых нами показателей скорости/частоты передачи данных, имеет смысл кратко остановиться на атрибутах подобных показателей. У всех них есть семь атрибутов, выбираемых в зависимости от того, какой тип показателя требуется. Благодаря им вы можете получить отдельное количество событий, а также среднее количество событий за разные периоды времени. Используйте эти показатели правильно, иначе ваше представление о состоянии брокера может оказаться неточным.

Первые два атрибута не показатели, но они полезны для понимания:

- **EventType** — единица измерения для всех атрибутов, в данном случае байты;
- **RateUnit** — для атрибутов скорости/частоты представляет собой период времени, за которое рассчитывается скорость, в данном случае секунды.

Эти два описательных атрибута сообщают, что скорость вне зависимости от промежутка времени, за который производится усреднение, указывается

в байтах в секунду. Существует четыре атрибута скорости с различным шагом детализации:

- **OneMinuteRate** — среднее значение за предыдущую минуту;
- **FiveMinuteRate** — среднее значение за предыдущие 5 минут;
- **FifteenMinuteRate** — среднее значение за предыдущие 15 минут;
- **MeanRate** — среднее значение за все время, прошедшее с момента запуска брокера.

Атрибут **OneMinuteRate** быстро меняется и дает скорее сиюминутное представление о показателе. Это может быть полезно для отслеживания кратковременных пиков трафика. **MeanRate** практически не будет меняться, он отображает общие тенденции. Хотя у **MeanRate** есть своя область применения, вероятно, это не тот показатель, о котором вы хотели бы получать уведомления. **FiveMinuteRate** и **FifteenMinuteRate** представляют собой компромисс между двумя предыдущими показателями.

Помимо этих атрибутов, существует также атрибут **Count**. Он представляет собой значение, постоянно нарастающее с момента запуска брокера. Для данного показателя — суммарной входящей скорости передачи данных — атрибут **Count** отражает общее число байтов, отправленных брокеру с момента запуска процесса. Система показателей, поддерживающая показатели-счетчики, позволяет получить с его помощью абсолютные значения вместо усредненных.

Суммарная исходящая скорость передачи данных

Как и суммарная входящая скорость передачи данных, *суммарная исходящая скорость передачи данных* — обобщенный показатель масштабирования. В данном случае он отражает исходящую скорость чтения потребителями данных. Исходящая скорость передачи может масштабироваться не так, как входящая, благодаря способности Kafka с легкостью работать с несколькими потребителями. Существует множество примеров использования платформы, в которых исходящая скорость может в шесть раз превышать входящую! Именно поэтому так важно учитывать и отслеживать исходящую скорость отдельно (подробности — в табл. 13.9).



Учет потоков извлечения данных из реплик

Исходящая скорость включает и трафик реплик. Это значит, что если коэффициент репликации всех топиков в настройках равен 2, то при отсутствии клиентов-потребителей исходящая скорость передачи данных будет равна входящей. При чтении всех сообщений кластера одним клиентом-потребителем исходящая скорость будет вдвое превышать входящую. Это может озадачить наблюдателя, который не знает, что именно подсчитывается.

Таблица 13.9. Подробная информация о показателе «суммарная по всем топикам исходящая скорость передачи данных»

Показатель	Исходящая скорость, байт/с
Управляемый компонент (MBean) JMX	kafka.server:type=BrokerTopicMetrics,name=BytesOutPerSec
Диапазон значений	Скорость — число с двойной точностью, количество — целочисленное значение

Суммарное по всем топикам число входящих сообщений

Скорости передачи данных отражают трафик брокера в абсолютных показателях — в байтах, в то время как показатель *входящих сообщений* отражает количество отдельных сгенерированных в секунду входящих сообщений вне зависимости от их размера. Он полезен в качестве дополнительного показателя трафика производителей. Можно использовать его также в сочетании с числом входящих байтов для определения среднего размера сообщения. Как и скорость передачи входящих данных, он позволяет заметить дисбаланс брокеров, тем самым давая вам понять, что необходимо техобслуживание (подробности — в табл. 13.10).

Таблица 13.10. Подробная информация о показателе «суммарное по всем топикам число входящих сообщений»

Показатель	Сообщений в секунду
Управляемый компонент (MBean) JMX	kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec
Диапазон значений	Скорость — число с двойной точностью, количество — целочисленное значение



А почему не исходящих сообщений?

Нас часто спрашивают: почему для брокеров Kafka нет показателя числа исходящих сообщений? Причина в том, что при потреблении сообщений брокер просто отправляет следующий пакет потребителю, не распаковывая его и не получая информации о том, сколько сообщений в нем содержится. Следовательно, брокер фактически не знает, сколько было отправлено сообщений. Единственный возможный в такой ситуации показатель — число извлечений в секунду — представляет собой частоту выполнения запросов, а не число сообщений.

Число разделов

Показатель *количества разделов* для брокера обычно незначительно меняется с течением времени, ведь он представляет собой общее число разделов, назначенных данному брокеру. Он включает все реплики в брокере, неважно, ведущие или ведомые. Мониторинг этого показателя более интересен в случае кластера, в котором включено автоматическое создание топиков, поскольку владелец кластера не всегда контролирует этот процесс (подробности — в табл. 13.11).

Таблица 13.11. Подробная информация о показателе «число разделов»

Показатель	Число разделов
Управляемый компонент (MBean) JMX	kafka.server:type=ReplicaManager,name=PartitionCount
Диапазон значений	Целое число, равное нулю или больше его

Число ведущих реплик

Число *ведущих реплик* отражает количество разделов, для которых данный брокер в настоящий момент является ведущей репликой. Как и большинство других показателей, он должен быть примерно одинаков для всех брокеров кластера. Чрезвычайно важно регулярно проверять число ведущих реплик, возможно, даже настроить на его основе уведомление, поскольку оно отражает несбалансированность кластера даже в случае идеального баланса реплик по числу и размеру. Дело в том, что брокер может перестать быть ведущим для реплики по многим причинам, среди которых, например, истечение срока сеанса ZooKeeper, и после восстановления не станет снова ведущим автоматически (разве что вы задали в настройках автоматическое переназначение ведущих реплик). В этом случае данный показатель покажет меньшее число (или даже 0) ведущих реплик, указывая на необходимость запуска выбора предпочтительной реплики для переназначения ведущих реплик кластера (подробности — в табл. 13.12).

Таблица 13.12. Подробная информация о показателе «число ведущих реплик»

Показатель	Число ведущих реплик
Управляемый компонент (MBean) JMX	kafka.server:type=ReplicaManager,name=LeaderCount
Диапазон значений	Целое число, равное нулю или больше его

Удобно использовать этот показатель совместно с числом разделов для отображения процента разделов, для которых данный брокер является ведущим. В хорошо сбалансированном кластере с коэффициентом репликации 2 все брокеры должны быть ведущими примерно для 50 % своих разделов. Если используется коэффициент репликации 3, это соотношение снижается до 33 %.

Отключенные разделы

Наряду с мониторингом числа недореплицированных разделов критически важен мониторинг *числа отключенных разделов* (табл. 13.13). Этот показатель предоставляет только брокер — контроллер кластера (все остальные брокеры будут возвращать 0), причем он показывает число разделов кластера, у которых сейчас нет ведущей реплики. Раздел без ведущей реплики может возникнуть по двум причинам.

- Останов всех брокеров, на которых находятся реплики данного раздела.
- Ни одна согласованная реплика не может стать ведущей из-за расхождения числа сообщений в случае, когда отключен «нечистый» выбор ведущей реплики.

Таблица 13.13. Подробная информация о показателе «число автономных разделов»

Показатель	Число ведущих реплик
Управляемый компонент (MBean) JMX	<code>kafka.controller:type=KafkaController,name=OfflinePartitionsCount</code>
Диапазон значений	Целое число, равное нулю или больше его

При промышленной эксплуатации кластера Kafka автономные разделы могут влиять на клиентов-производителей, приводя к потере сообщений или вызывая обратное давление в приложении. Чаще всего это заканчивается тем, что сайт становится неработоспособным и требует немедленного вмешательства.

Показатели запросов

В протоколе Kafka, описанном в главе 6, есть множество различных типов запросов. Начиная с версии 2.5.0, с помощью определенных показателей контролируют функционирование типов запросов, приведенных в табл. 13.14.

Таблица 13.14. Названия показателей запроса

AddOffsetsToTxn	AddPartitionsToTxn	AlterConfigs
AlterPartitionReassignments	AlterReplicaLogDirs	ApiVersions
ControlledShutdown	CreateAcls	CreateDelegationToken
CreatePartitions	CreateTopics	DeleteAcls
DeleteGroups	DeleteRecords	DeleteTopics
DescribeAcls	DescribeConfigs	DescribeDelegationToken
DescribeGroups	DescribeLogDirs	ElectLeaders
EndTxn	ExpireDelegationToken	Fetch
FetchConsumer	FetchFollower	FindCoordinator
Heartbeat	IncrementalAlterConfigs	InitProducerId
JoinGroup	LeaderAndIsr	LeaveGroup
ListGroups	ListOffsets	ListPartitionReassignments
Metadata	OffsetCommit	OffsetDelete
OffsetFetch	OffsetsForLeaderEpoch	Produce
RenewDelegationToken	SaslAuthenticate	SaslHandshake
StopReplica	SyncGroup	TxnOffsetCommit
UpdateMetadata	WriteTxnMarkers	

Для каждого из них существует восемь показателей, позволяющих отслеживать обработку каждой из фаз запроса. Например, показатели для запроса `Fetch` перечислены в табл. 13.15.

Таблица 13.15. Показатели для запроса `Fetch`

Название	Управляемый компонент (MBean) JMX
Общее время	<code>kafka.network:type=RequestMetrics,name=TotalTimeMs,request=Fetch</code>
Время нахождения запроса в очереди	<code>kafka.network:type=RequestMetrics,name=RequestQueueTimeMs,request=Fetch</code>
Локальное время	<code>kafka.network:type=RequestMetrics,name=LocalTimeMs,request=Fetch</code>
Удаленное время	<code>kafka.network:type=RequestMetrics,name=RemoteTimeMs,request=Fetch</code>
Длительность притормаживания	<code>kafka.network:type=RequestMetrics,name=ThrottleTimeMs,request=Fetch</code>
Время нахождения ответа в очереди	<code>kafka.network:type=RequestMetrics,name=ResponseQueueTimeMs,request=Fetch</code>

Название	Управляемый компонент (MBean) JMX
Длительность отправки запроса	<code>kafka.network:type=RequestMetrics,name=ResponseSendTimeMs,request=Fetch</code>
Запросов в секунду	<code>kafka.network:type=RequestMetrics,name=RequestsPerSec,request=Fetch</code>

Число запросов в секунду, как говорилось ранее, — частотный показатель, отражающий общее количество запросов данного типа, полученных и обработанных за единицу времени. Он позволяет узнать частоту выполнения каждого из запросов, хотя следует отметить, что многие из них, например `StopReplica` и `UpdateMetadata`, выполняются нерегулярно.

Каждый из семи *временных* показателей предоставляет для запросов набор процентилей, а также дискретный атрибут `Count`, аналогичный показателям частоты/скорости. Показатели подсчитываются с момента запуска брокера, так что при обнаружении долго не изменяющихся значений имейте в виду, что чем дольше работает брокер, тем меньше будут меняться значения. Они отражают следующие характеристики обработки запроса.

- *Общее время* — общее количество времени обработки запроса брокером от получения до отправки ответа.
- *Время нахождения запроса в очереди* — длительность времени, проведенного запросом в очереди после его получения, но до начала обработки.
- *Локальное время* — количество потраченного ведущей репликой на обработку запроса времени, включая отправку его на диск (но, возможно, не учитывая фактический сброс на диск).
- *Удаленное время* — время, которое пришлось потратить на ожидание ведомых реплик, прежде чем стало возможно завершить обработку запроса.
- *Длительность притормаживания* — промежуток времени, на который необходимо придержать ответ, чтобы замедлить запрашивающий клиент настолько, что будут удовлетворены настройки квот клиентов.
- *Время нахождения ответа в очереди* — количество времени, проводимого запросом в очереди перед отправкой запрашивающему клиенту.
- *Длительность отправки запроса* — количество времени, фактически затраченного на отправку запроса.

У всех показателей имеются следующие атрибуты:

- `Count` — фактическое количество запросов с момента запуска процесса;
- `Min` — минимальное значение по всем запросам;
- `Max` — максимальное значение по всем запросам;

- **Mean** — среднее значение по всем запросам;
- **StdDev** — стандартное отклонение показателей хронометража запросов в совокупности;
- процентиля — **50thPercentile**, **75thPercentile**, **95thPercentile**, **98thPercentile**, **99thPercentile**, **999thPercentile**.



Что такое процентиль

Процентили — распространенный способ представления показателей хронометража. Девяносто девятый процентиль означает, что 99 % значений выборки (в данном случае значений хронометража запросов) меньше значения показателя. А 1 % — больше заданного значения. Чаще всего используют среднее значение, а также 99-й и 99,9-й процентиля. При этом становится понятно, как происходит обработка среднестатистического запроса, а как — аномальных запросов.

Какие же из этих показателей и атрибутов запросов необходимо отслеживать? Как минимум следует собирать данные о среднем значении и один из верхних процентилей (99-й или 99,9-й) для показателя общего времени, а также число запросов в секунду для каждого из типов запросов. Это даст вам общее представление о производительности выполнения запросов к брокеру Kafka. По возможности следует также собирать сведения и об остальных шести показателях хронометража для каждого из типов запросов, что позволит сузить область поиска проблем с производительностью до конкретной фазы обработки запроса.

Задавать пороговые значения для оповещения на основе показателей хронометража — непростая задача. Временные показатели выполнения запроса **Fetch**, например, очень сильно варьируются в зависимости от множества факторов, включая настройки длительности ожидания сообщений на стороне клиента, степень загруженности конкретного топика, а также скорость сетевого соединения между клиентом и брокером. Имеет смысл, однако, выработать эталонное значение 99-го процентиля, по крайней мере для показателя общего времени, особенно для запросов типа **Produce**, и настроить по нему оповещение. Аналогично показателю числа недореплицированных разделов резкое повышение 99-го процентиля для запросов типа **Produce** может указывать на разнообразные проблемы с производительностью.

Показатели топиков и разделов

Помимо множества доступных в брокерах показателей, описывающих функционирование брокеров Kafka в целом, существуют и показатели уровня топиков и разделов. В больших кластерах их может быть очень много, так что не получит-

ся систематизировать их все в процессе обычного функционирования. Однако они очень удобны при отладке конкретных проблем с клиентами. Например, показатели уровня топиков можно использовать для поиска конкретного топика, который вызывает большой прирост объема трафика кластера. Не помешает также сделать так, чтобы пользователи Kafka (клиенты-производители и клиенты-потребители) имели к ним доступ. Вне зависимости от того, есть ли у вас возможность регулярно собирать эти показатели, следует знать, какую пользу они могут принести.

Во всех примерах из табл. 13.16 мы используем *имя_топика* и раздел 0. При обращении к описанным показателям не забудьте поменять имя топика и номер раздела на соответствующие значения для вашего кластера.

Таблица 13.16. Показатели каждого топика

Название	Управляемый компонент (MBean) JMX
Скорость входящей передачи данных, байт/с	<code>kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec,topic=имя_топика</code>
Скорость исходящей передачи данных, байт/с	<code>kafka.server:type=BrokerTopicMetrics,name=BytesOutPerSec,topic=имя_топика</code>
Частота неудачного извлечения данных	<code>kafka.server:type=BrokerTopicMetrics,name=FailedFetchRequestsPerSec,topic=имя_топика</code>
Частота неудачной генерации сообщений	<code>kafka.server:type=BrokerTopicMetrics,name=FailedProduceRequestsPerSec,topic=имя_топика</code>
Частота входящих сообщений	<code>kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec,topic=имя_топика</code>
Частота запросов на извлечение данных	<code>kafka.server:type=BrokerTopicMetrics,name=TotalFetchRequestsPerSec,topic=имя_топика</code>
Частота запросов от производителей	<code>kafka.server:type=BrokerTopicMetrics,name=TotalProduceRequestsPerSec,topic=имя_топика</code>

Показатели уровня топика

Показатели уровня топика очень схожи с описанными ранее показателями брокеров. Фактически единственное отличие состоит в указании названия топика, к которому будут относиться показатели. С учетом числа доступных показателей и в зависимости от количества топиков в вашем кластере мониторинг и оповещение по ним почти наверняка будут излишними. Однако они могут пригодиться клиентам для оценки и настройки использования Kafka.

Показатели уровня раздела

Показатели уровня раздела обычно менее полезны в текущей работе, чем показатели уровня топика. Кроме того, они слишком многочисленны, ведь в сотнях топиков легко можно насчитать тысячи разделов. Тем не менее в отдельных ситуациях они могут оказаться полезными. В частности, показатели уровня раздела отражают объем данных (в байтах), хранящийся в настоящий момент на диске для определенного раздела (табл. 13.17). Если их суммировать, можно узнать объем хранимых данных отдельного топика, что удобно при выделении ресурсов для отдельных клиентов Kafka. Расхождение между размерами двух разделов одного и того же топика может указывать на проблему с неравномерным распределением сообщений по использовавшемуся при генерации ключу. Показатель числа сегментов журнала отражает количество файлов сегментов журнала на диске для данного раздела. Он удобен для отслеживания ресурсов, как и показатель размера разделов.

Таблица 13.17. Показатели каждого раздела

Название	Управляемый компонент (MBean) JMX
Размер раздела	<code>kafka.log:type=Log,name=Size,topic=имя_моника,partition=0</code>
Количество сегментов журнала	<code>kafka.log:type=Log,name=NumLogSegments,topic=имя_моника,partition=0</code>
Начальное смещение журнала	<code>kafka.log:type=Log,name=LogEndOffset,topic=имя_моника,partition=0</code>
Конечное смещение журнала	<code>kafka.log:type=Log,name=LogStartOffset,topic=имя_моника,partition=0</code>

Показатели конечного и начального смещений журнала представляют собой максимальное и минимальное смещения сообщений в данном разделе соответственно. Следует отметить, однако, что разница между этими числами не обязательно соответствует числу сообщений в разделе, поскольку сжатие журналов может вызвать появление пропущенных смещений, удаленных из раздела в результате поступления более новых сообщений с тем же ключом. При некоторых конфигурациях для разделов может оказаться полезен мониторинг этих смещений. Один из подобных сценариев использования — обеспечение более точного соответствия меток даты/времени смещениям, благодаря чему клиенты-потребители легко могут откатывать смещения на конкретный момент (хотя благодаря появившемуся в Kafka 0.10.1 индексному поиску по времени это утратило свое значение).



Показатели недореплицированных разделов

Существует показатель уровня раздела, говорящий о том, является ли раздел недореплицированным. В целом он не слишком полезен в повседневной работе из-за большого числа требующих отслеживания параметров. Намного легче контролировать число недореплицированных разделов в масштабах брокера, используя утилиты командной строки (описаны в главе 12) для выявления конкретных недореплицированных разделов.

Мониторинг JVM

Помимо показателей для брокера Kafka, следует контролировать стандартный набор показателей для всех серверов и самой виртуальной машины Java (JVM). Благодаря этому вы сможете получить оповещение о возникших проблемных ситуациях, например о росте активности сборщика мусора, отрицательно влияющей на производительность брокера. Благодаря им вы также сможете понять причину изменения показателей далее по конвейеру, в брокере.

Сборка мусора

Для JVM критически важно наблюдать за процессом сборки мусора (GC). Какие именно Java-компоненты необходимо отслеживать для получения этой информации, очень зависит от используемого JRE (Java Runtime Environment), а также конкретных настроек GC. В табл. 13.18 показано, какие компоненты использовать в случае JRE Oracle Java 1.8 со сборкой мусора G1.

Таблица 13.18. Показатели сборки мусора G1

Название	Управляемый компонент (MBean) JMX
Полных циклов GC	<code>java.lang:type=GarbageCollector,name=G1 Old Generation</code>
Молодых циклов GC	<code>java.lang:type=GarbageCollector,name=G1 Young Generation</code>

Обратите внимание на то, что в терминах сборки мусора старый (Old) и полный (Full) — одно и то же. Для каждого из этих показателей необходимо отслеживать два атрибута — `CollectionCount` и `CollectionTime`. `CollectionCount` представляет собой число циклов GC соответствующего типа (полных или молодых) с момента запуска JVM. `CollectionTime` — это продолжительность времени (в миллисекундах), потраченного на этот тип сборки мусора с момента запуска JVM. Будучи по своей сути счетчиками, эти показатели могут использоваться системой показателей для вывода конкретного числа циклов и времени,

потраченных на сборку мусора за единицу времени. Их можно применять также для получения средней длительности цикла GC, хотя при нормальном функционировании это особой пользы не приносит.

У каждого из этих показателей есть атрибут `LastGcInfo`. Это составное значение, имеющее пять полей и содержащее информацию о последнем цикле GC для описанного компонентом типа сборки мусора. Самое важное из этих полей — значение `duration`, указывающее длительность последнего цикла GC (в миллисекундах). Остальные значения атрибута (`GcThreadCount`, `id`, `startTime` и `endTime`) носят информационный характер и не особо полезны. Важно отметить, что увидеть длительность всех циклов GC с помощью этого атрибута нельзя в силу нерегулярности, в частности, молодых циклов GC.

Мониторинг операционной системы из Java

JVM может предоставить некоторую информацию об операционной системе с помощью компонента `java.lang:type=OperatingSystem`. Однако это неполная информация, не отражающая все, что необходимо знать о системе, в которой работает брокер. В ее составе есть два полезных атрибута, данные по которым тут можно собрать, но непросто получить в операционной системе: `MaxFileDescriptorCount` и `OpenFileDescriptorCount`. `MaxFileDescriptorCount` представляет собой максимальное допустимое для JVM число открытых дескрипторов файлов. `OpenFileDescriptorCount` соответствует числу открытых в данный момент дескрипторов. Дескрипторы будут открываться для каждого сегмента журнала и сетевого подключения, и их число станет быстро расти. Если сетевые подключения не закрываются должным образом, то их количество, разрешенное для брокера, будет быстро исчерпано.

Мониторинг ОС

JVM не способна предоставить всю необходимую нам информацию об операционной системе. Поэтому необходимо собирать показатели не только от брокера, но и от самой операционной системы. Большинство систем мониторинга предоставляют агенты, способные собрать намного больше информации об операционной системе, чем вам может теоретически понадобиться. Основное, что нужно отслеживать, — использование ресурсов CPU, оперативной памяти, дисков, дисковых операций ввода/вывода и сети.

Что касается использования CPU, то необходимо как минимум следить за усредненной загрузкой системы. Этот показатель представляет собой отдельное значение, отражающее относительную загрузку процессоров. Кроме того, он может пригодиться для получения загрузки CPU, разбитой по типам. В зависимости от метода сбора данных и конкретной операционной системы вам могут

быть доступны все или часть процентных отношений следующих категорий загрузки CPU (с помощью приведенных аббревиатур):

- **us** — процессорное время, потраченное в пользовательском адресном пространстве;
- **sy** — процессорное время, потраченное в адресном пространстве ядра;
- **ni** — процессорное время, потраченное на фоновые процессы;
- **id** — время бездействия процессора;
- **wa** — время ожидания процессором дисков;
- **hi** — процессорное время, потраченное на обработку аппаратных прерываний;
- **si** — процессорное время, потраченное на обработку программных прерываний;
- **st** — процессорное время, потраченное на ожидание гипервизора.



Что такое системная нагрузка

Многим известно, что системная нагрузка — это степень использования CPU системы, но немногие знают, как она измеряется. Средняя нагрузка — число работоспособных процессов, ожидающих выполнения процессором. Linux включает в их число также потоки выполнения, находящиеся в состоянии непрерываемого сна (uninterruptible sleep state), например ожидающие выполнения дисковых операций. Нагрузка представлена в виде трех чисел: среднего количества за последнюю минуту, последние 5 минут и 15 минут. В системе с одним CPU значение 1 означает, что система загружена на 100 % и какой-нибудь поток всегда ожидает выполнения. В системе с несколькими CPU соответствующее 100%-ной нагрузке значение будет равно числу CPU в системе. Например, если в системе 24 процессора, то средняя нагрузка, равная 24, соответствует 100%-ной нагрузке.

Брокер Kafka использует для обработки запросов значительные ресурсы CPU. Поэтому при мониторинге Kafka важно отслеживать загрузку CPU. Учет оперативной памяти менее важен для самого брокера, поскольку Kafka обычно запускается с относительно небольшим размером кучи JVM. Он использует для функций сжатия относительно небольшое количество памяти вне кучи, но большая часть системной памяти оставляется для кэша. Тем не менее лучше отслеживать использование оперативной памяти, чтобы другие приложения не мешали брокеру. Стоит также наблюдать за объемом общей и свободной памяти подкачки, чтобы убедиться, что эта память не задействуется.

Что касается Kafka, диск, безусловно, важнейшая подсистема. Все сообщения сохраняются на него, так что производительность Kafka сильно зависит от производительности дисков. Очень важно отслеживать использование пространства

и индексных дескрипторов (*индексные дескрипторы* — это объекты метаданных файлов и каталогов в файловых системах Unix) на дисках, чтобы не исчерпать дисковое пространство. Это особенно важно для разделов, в которых хранятся данные Kafka. Необходимо также следить за статистикой операций дискового ввода/вывода, чтобы гарантировать эффективное использование дисков. Следите по крайней мере за статистикой дисков, на которых хранятся данные Kafka: числом операций записи и чтения в секунду, средними размерами очередей на чтение и запись, средним временем ожидания и эффективностью использования диска в процентах.

Наконец, следите за использованием сети на брокерах. Этот показатель представляет собой просто входящий и исходящий сетевой трафик, обычно указываемый в битах в секунду. Не забывайте, что каждый входящий в брокер Kafka бит означает соответствующее коэффициенту репликации топика число битов исходящих данных при отсутствии потребителей. В зависимости от числа потребителей исходящий сетевой трафик может оказаться на порядки больше входящего. Не забывайте об этом при задании пороговых значений для оповещения.

Журналирование

Любое обсуждение мониторинга будет неполным без упоминания журналирования. Подобно множеству приложений, брокер Kafka мгновенно забывает диск журнальными сообщениями, если только ему разрешить. Чтобы извлечь из журналов полезную информацию, необходимо включить правильные механизмы журналирования на нужных уровнях. Простая запись всех сообщений на уровне INFO даст множество важной информации о состоянии брокера. Полезно будет, однако, отделить несколько механизмов журналирования, чтобы получить более компактный набор файлов журналов.

Существует два механизма журналирования, записывающих информацию в отдельные файлы на диске. Первый — `kafka.controller` на уровне INFO, он служит для получения информации конкретно о контроллере кластера. В каждый момент времени только один брокер может быть контроллером, следовательно, записывать в эти журналы данные будет всегда только один брокер. Его информация включает данные о создании и изменении топика, изменении состояния брокера, а также такие операции кластера, как выбор предпочтительной реплики и перемещения разделов. Второй механизм журналирования — `kafka.server.ClientQuotaManager`, тоже уровня INFO. Он используется для отображения сообщений, связанных с квотами на операции генерации и потребления. Это полезная информация, но в главном файле журнала брокера она будет только отвлекать.

Не помешает также занести в журнал информацию о состоянии потоков сжатия журналов. Не существует отдельного показателя, отражающего состояние этих

потоков, и сбой сжатия одного раздела может полностью застопорить потоки сжатия журналов, причем пользователь не получит никакого оповещения об этом. Для вывода информации о состоянии этих потоков необходимо включить механизмы журналирования `kafka.log.LogCleaner`, `kafka.log.Cleaner` и `kafka.log.LogCleanerManager` на уровне `DEBUG`. Эта информация включает данные о сжатии каждого из разделов, включая размер и число сообщений. При обычном функционировании сведений не так уж много, так что можно включить это журналирование по умолчанию, не опасаясь утонуть в информации.

Будет полезно включить еще некоторые виды журналирования при отладке проблем с Kafka. Например, `kafka.request.logger` на уровне `DEBUG` или `TRACE`. Данный журнал заносит в журнал информацию обо всех отправленных брокеру запросах. На уровне `DEBUG` он включает конечные точки соединений, хронометраж запросов и сводную информацию. На уровне `TRACE` — также информацию о топике и разделе — практически всю информацию запроса, за исключением содержимого самого сообщения. На любом из этих уровней он генерирует значительный объем данных, так что включать его рекомендуется только для отладки.

Мониторинг клиентов

Все приложения требуют мониторинга. У приложений, реализующих клиенты Kafka, производители или потребители, имеются соответствующие показатели. В этом разделе мы будем говорить об официальных клиентских библиотеках Java, хотя и в других реализациях должны быть доступны свои показатели.

Показатели производителя

Клиент-производитель Kafka существенно повысил компактность имеющихся показателей, сделав их доступными в виде атрибутов небольшого числа управляемых компонентов (JMX MBeans). А предыдущая версия клиента-производителя (более не поддерживаемая) предоставляла более подробную информацию по большинству показателей за счет большего числа управляемых компонентов (в ней было больше процентных показателей и различных скользящих средних). В результате суммарно охватывалась большая «площадь поверхности», но поиск аномальных значений был затруднен.

Все показатели производителя содержат идентификатор клиента-производителя в названии компонента. В приведенных примерах он заменен на *ID_клиента*. В случаях, когда название компонента содержит идентификатор брокера, этот идентификатор заменен на *ID_брокера*. Названия топиков заменены на *имя_топика*. Пример приведен в табл. 13.19.

Таблица 13.19. Управляемые компоненты показателей производителя Kafka

Название	Управляемый компонент (MBean) JMX
В целом по производителю	<code>kafka.producer:type=producer-metrics,client-id=ID_клиента</code>
Для отдельного брокера	<code>kafka.producer:type=producer-node-metrics,client-id=ID_клиента,node-id=node-ID_брокера</code>
Для отдельного топика	<code>kafka.producer:type=producer-topic-metrics,client-id=ID_клиента,topic=имя_топика</code>

У каждого из компонентов показателей в табл. 13.19 есть несколько атрибутов, предназначенных для описания состояния производителя. Конкретные наиболее полезные атрибуты рассмотрены в следующем разделе. Прежде чем двигаться дальше, убедитесь, что хорошо понимаете семантику работы производителя, о которой шла речь в главе 3.

Показатели производителя в целом

Компоненты показателей производителей в целом имеют атрибуты, описывающие все, начиная с размеров пакетов сообщений и заканчивая использованием буферов памяти. Хотя все эти показатели применяются при отладке, лишь немногие из них — регулярно и лишь пара из этих немногих заслуживает мониторинга и настройки оповещений. Обратите внимание: хотя мы будем обсуждать показатели, представляющие собой средние значения (заканчиваются на **-avg**), существуют также максимальные значения всех показателей (заканчиваются на **-max**), полезные лишь в некоторых ситуациях.

Определенно имеет смысл настроить оповещение для атрибута **record-error-rate**. Этот показатель всегда должен быть равен 0, а если он больше 0, значит, производитель отменяет сообщения, которые пытается отправить брокерам Kafka. Для производителя задаются число попыток повтора и пауза между ними, по истечении которых сообщения, называемые тут *записями*, будут отменяться. Можно также отслеживать атрибут **record-retry-rate**, но он не так важен, как частота ошибок, поскольку повторы отправки свидетельствуют о нормальном функционировании.

Еще один показатель, для которого стоит настроить оповещение, — **request-latency-avg**. Он представляет собой среднюю длительность отправки запроса от производителя. Вам лучше определить эталонное значение этого параметра при нормальном функционировании и установить оповещение при его превышении. Увеличение времени задержки запроса означает замедление запросов от производителей. Причина может быть в проблемах с сетью или проблемах на брокерах. В любом случае речь идет о проблеме с производительностью, вызывающей приостановки и другие неполадки в приложении-производителе.

Помимо этих важных показателей, не помешает знать объемы трафика отправляемых производителем сообщений. Эту информацию можно получить в трех различных разрезах с помощью трех атрибутов. Атрибут `outgoing-byte-rate` говорит о трафике сообщений в байтах в секунду. `record-send-rate` описывает трафик в терминах числа сгенерированных сообщений в секунду. Наконец, `request-rate` позволяет узнать число отправленных производителями брокерам запросов в секунду. И конечно, каждое сообщение состоит из определенного числа байтов. Эти показатели отнюдь не станут лишними на инструментальной панели вашего приложения.

Существуют также показатели, описывающие размеры записей, запросов и пакетов. С помощью `request-size-avg` можно получить средний размер запросов, отправляемых брокерам производителями, в байтах. С помощью `batch-size-avg` — средний размер отдельного пакета сообщений, состоящего по умолчанию из сообщений, предназначенных для отдельного раздела топика, в байтах. `record-size-avg` показывает средний размер отдельной записи в байтах. В случае применения производителя с одним топиком эти показатели предоставляют полезную информацию о сгенерированных сообщениях. Если используются производители с несколькими топиками, например `MirrorMaker`, их информативность снижается. Помимо этих трех показателей, существует `records-per-request-avg`, описывающий среднее число сообщений в отдельном запросе от производителя.

Последний из рекомендуемых атрибутов общих показателей производителей — `record-queue-time-avg`. Он представляет собой среднее время (в миллисекундах), которое отдельному сообщению приходится ожидать в производителе после отправки его приложением и до фактической генерации его для `Kafka`. После того как приложение посредством метода `send` вызовет клиент-производитель для отправки сообщения, производитель будет ждать, пока не произойдет одно из двух событий:

- наберется такое количество сообщений, которого будет достаточно для заполнения пакета, в соответствии с параметром конфигурации `batch.size`;
- с момента отправки прошлого пакета пройдет время, соответствующее параметру конфигурации `linger.ms`.

Любое из этих двух событий приведет к закрытию клиентом-производителем формируемого в текущий момент пакета и отправке его брокерам. Проще всего сформулировать это можно следующим образом: для загруженных топиков будет применяться первое условие, а для медленных топиков — второе. Показатель `record-queue-time-avg` дает информацию о том, сколько времени занимает генерация сообщений, следовательно, он будет полезен при настройке этих двух параметров конфигурации с целью удовлетворения требований вашего приложения к времени задержки.

Показатели уровня брокера и топика

Помимо общих показателей производителей, существуют компоненты показателей с ограниченным набором атрибутов для подключения к отдельным брокерам Kafka, а также для каждого топика, для которого генерируется сообщение. Эти показатели в некоторых случаях удобно использовать при отладке, но вряд ли их следует отслеживать постоянно. Все атрибуты этих компонентов аналогичны описанным ранее атрибутам компонентов общих показателей производителей, и смысл их точно такой же, за исключением того, что они относятся к отдельному брокеру или топика.

Наиболее полезный из показателей производителей, относящихся к отдельным брокерам, — `request-latency-avg`. Дело в том, что значение этого показателя практически всегда постоянно (при стабильной работе пакетной отправки сообщений), параметр может отражать проблемы с подключением к конкретным брокерам. Остальные атрибуты, например `outgoing-byte-rate` и `request-latency-avg`, меняются в зависимости от разделов, для которых данный брокер является ведущим. Это значит, что «должное» значение этих показателей в каждый момент времени может быть различным в зависимости от состояния кластера Kafka.

Показатели топиков интереснее, чем показатели брокеров, но они могут принести пользу только при использовании производителей, работающих более чем с одним топиком. Кроме того, применять их на постоянной основе можно, только если производитель не работает с большим числом топиков. Например, MirrorMaker может генерировать сотни, если не тысячи топиков. Отслеживать все их показатели очень трудно, а задать для каждого разумное пороговое значение для оповещения практически нереально. Как и показатели для отдельных брокеров, показатели для отдельных топиков лучше всего использовать при поиске причин конкретной проблемы. Атрибуты `record-send-rate` и `record-error-rate`, например, можно использовать для выяснения того, к какому топика относятся отмененные сообщения (или подтверждения того, что они имеются во всех топиках). Кроме того, существует показатель `byte-rate` — общая частота сообщений топика (в байтах в секунду).

Показатели потребителей

Аналогично клиенту-производителю клиент-потребитель в Kafka объединяет множество показателей в атрибуты всего лишь нескольких компонентов показателей. Из этих показателей, как и для клиента-производителя, исключены процентные показатели для задержки и скользящие средние скорости/частоты, которые были представлены в устаревшем потребителе Scala. В потребителе

в силу того, что логика потребления сообщений сложнее простой их отправки брокерам Kafka, есть несколько дополнительных показателей (табл. 13.20), с которыми нужно иметь дело.

Таблица 13.20. Управляемые компоненты показателей потребителя Kafka

Название	Управляемый компонент (MBean) JMX
В целом по потребителю	<code>kafka.consumer:type=consumer-metrics,client-id=ID_клиента</code>
Диспетчер извлечения	<code>kafka.consumer:type=consumer-fetch-manager-metrics,client-id=ID_клиента</code>
Для отдельного топика	<code>kafka.consumer:type=consumer-fetch-manager-metrics,client-id=ID_клиента,topic=имя_топика</code>
Для отдельного брокера	<code>kafka.consumer:type=consumer-node-metrics,client-id=ID_клиента,node-id=node-ID_брокера</code>
Координатор	<code>kafka.consumer:type=consumer-coordinator-metrics,client-id=ID_клиента</code>

Показатели диспетчера извлечения

В клиенте-потребителе компонент показателя по потребителю в целом приносит меньше пользы, поскольку интересующие нас данные расположены не там, а в *компонентах диспетчера извлечения*. В нем есть показатели, относящиеся к низкоуровневым операциям сети, а в компоненте диспетчера извлечения — показатели скоростей в байтах, а также частот запросов и записей. В отличие от клиента-производителя предоставляемые потребителем показатели полезны для изучения, но по ним не имеет смысла устанавливать оповещения.

Один из атрибутов показателей диспетчера извлечения, по которому имеет смысл настроить мониторинг и оповещение, — `fetch-latency-avg`. Как и с помощью аналогичного `request-latency-avg` в клиенте-производителе, с его помощью можно выяснить, сколько времени занимает выполнение запросов к брокерам на извлечение. Проблема с оповещением на основе этого показателя состоит в том, что длительность задержки определяется параметрами `fetch.min.bytes` и `fetch.max.wait.ms` конфигурации потребителя. У медленного топика время задержки будет хаотически меняться, так как иногда брокер будет отвечать быстро (когда сообщения доступны), а иногда не будет отвечать в течение `fetch.max.wait.ms` (когда доступных сообщений нет). При потреблении топиков с более постоянным и насыщенным трафиком сообщений может оказаться полезно отслеживать этот показатель.



Постойте-ка! Никакого отставания?

Лучший совет по поводу потребителей — отслеживать их отставание. Так почему же мы не рекомендуем мониторинг атрибута `records-lag-max` компонента диспетчера извлечения? Этот показатель показывает текущее отставание (разницу между смещением потребителя и смещением лога брокера) для наиболее отстающего раздела.

Здесь наблюдается двойная проблема: указанный атрибут показывает отставание только для одного раздела и зависит от правильного функционирования потребителя. Если другого выхода нет, можно воспользоваться им для отслеживания отставания и настроить на его основе оповещение. Но рекомендуется применять внешние средства мониторинга отставания, как описывается в разделе «Мониторинг отставания» далее в этой главе.

Чтобы узнать объемы обрабатываемого клиентом-потребителем трафика сообщений, необходимо отслеживать показатели `bytes-consumed-rate` или `records-consumed-rate`, а лучше и тот и другой. Они описывают потребляемый данным экземпляром клиента трафик сообщений в байтах в секунду и сообщениях в секунду соответственно. Некоторые пользователи задают оповещение при минимальных пороговых значениях этих показателей, чтобы получать уведомление о выполнении потребителем недостаточного объема работ. Однако делать это следует осторожно. Kafka нацелена на разделение клиентов-потребителей и клиентов-производителей и предоставляет им возможность работать независимо друг от друга. Скорость чтения сообщений потребителем зачастую зависит от того, работает ли производитель должным образом, так что отслеживание их на потребителе означает определенные допущения относительно его состояния. Это может привести к ложным оповещениям в клиентах-потребителях.

Не помешает также хорошо представлять себе соотношения байтов сообщений и запросов, и диспетчер извлечения предоставляет данные для этого. Показатель `fetch-rate` сообщает число выполняемых потребителем запросов на извлечение в секунду. Показатель `fetch-size-avg` — средний размер этих запросов на извлечение в байтах. Наконец, показатель `records-per-request-avg` дает среднее число сообщений в каждом запросе на извлечение. Обратите внимание на то, что у потребителя нет аналога показателя `record-size-avg` производителя, с помощью которого можно узнать средний размер сообщения. Если для вас это важно, его можно вычислить на основе других доступных показателей или перехватить в вашем приложении после получения сообщений от клиентской библиотеки потребителя.

Показатели уровня брокера и топика

Показатели, предоставляемые клиентом-потребителем по каждому из соединений брокера и каждому из потребляемых топиков, как и в случае с клиентом-производителем, удобны для отладки проблем с потреблением, но

отслеживать их регулярно, вероятно, смысла нет. Как и в случае с диспетчером извлечения, атрибут `request-latency-avg` компонентов для показателей уровня брокера пригоден лишь в некоторых ситуациях в зависимости от объема трафика сообщений потребляемых топиков. Показатели `incoming-byte-rate` и `request-rate` представляют собой разбиение показателей диспетчера извлечения, относящихся к потребленным сообщениям, на показатели, выраженные в байтах в секунду и запросах в секунду соответственно. Их можно использовать для поиска причин проблем соединения потребителя с конкретным брокером.

Предоставляемые клиентом-потребителем показатели уровня топика оказываются полезны при чтении более чем одного топика. В противном случае они ничем не будут отличаться от показателей диспетчера извлечения и окажутся просто избыточными. В то же время, если клиент потребляет очень много топиков (Kafka MirrorMaker, например), изучать данные этих показателей будет непросто. Если вы решите их собирать, то наиболее важные из них — `bytes-consumed-rate`, `records-consumed-rate` и `fetch-size-avg`. Показатель `bytes-consumed-rate` отражает объемы прочитанных из конкретного топика сообщений в байтах в секунду, а `records-consumed-rate` — ту же информацию в виде числа сообщений. `fetch-size-avg` представляет собой средний размер запроса на извлечение для данного топика в байтах.

Показатели координатора потребителя

Как описывалось в главе 4, клиенты-потребители обычно работают в составе группы потребителей. Она выполняет определенные координационные действия, например присоединение новых участников и отправку брокерам контрольных сигналов для поддержания состояния членства в группе. Координатор потребителя представляет собой часть клиента-потребителя, отвечающую за эти действия, и у него есть свои показатели. Как и других показателей, их очень много, но отслеживать на постоянной основе имеет смысл лишь несколько ключевых.

Главная проблема, с которой сталкиваются потребители в результате выполнения действий по координации, — приостановка потребления на время синхронизации группы потребителей. Так происходит, когда экземпляры потребителей группы договариваются о том, кто какие разделы будет потреблять. Время, которое это может занять, зависит от числа потребляемых разделов. Координатор предоставляет атрибут показателя `sync-time-avg` — среднее время согласования в миллисекундах. Не помешает также информация из атрибута `sync-rate`, представляющего собой число операций согласования группы в секунду. При постоянном составе группы потребителей он практически всегда будет равен нулю.

Потребителю для записи данных о ходе потребления сообщений в контрольных точках приходится фиксировать смещения автоматически через равные промежутки времени или вручную, иницилируя создание контрольных точек из кода приложения. По сути, такая фиксация представляет собой запросы на генерацию (хотя тип запроса у них свой), поскольку фиксация смещения — это просто сообщение, генерируемое в специальный топик. У координатора потребителя имеется атрибут `commit-latency-avg` — показатель среднего времени, расходуемого на фиксацию смещения. Рекомендуется отслеживать это значение так же, как отслеживается время задержки запроса в производителе. При желании можно выработать для себя эталонное ожидаемое значение этих показателей и задать разумные пороговые значения, чтобы получить уведомление при их превышении.

Еще один полезный показатель координатора — `assigned-partitions`. Он представляет собой число разделов, назначенных для потребления клиенту-потребителю как отдельному участнику группы потребителей. Сравнение значений этого показателя от различных клиентов-потребителей данной группы позволяет оценить распределение нагрузки по всей группе. Его можно применить для поиска перекосов, вызванных проблемами в алгоритме, используемом координатором для распределения разделов по участникам группы.

Квоты

Apache Kafka умеет притормаживать запросы клиентов ради предотвращения ситуации, когда один клиент перегружает весь кластер. Эту возможность, выражаемую в терминах объема трафика, который клиент с конкретным идентификатором может отправлять конкретному брокеру (в байтах в секунду), можно настроить как для клиентов-потребителей, так и для клиентов-производителей. Существует параметр конфигурации брокера, задающий значение по умолчанию для всех клиентов, а также возможность динамического переопределения его на уровне клиентов. Когда брокер решает, что клиент превысил квоту, он замедляет его посредством задержки ответа ему на достаточное время.

Брокеры Kafka не используют в ответах коды ошибок для индикации того, что клиент притормаживается. Это значит, что суть происходящего не будет понятна приложению, если оно не отслеживает длительности притормаживания клиентов. Список показателей, которые необходимо отслеживать, приведен в табл. 13.21.

По умолчанию квоты в брокерах Kafka отключены, но мониторинг указанных показателей вполне допустим независимо от того, используете вы сейчас квоты или нет. Их мониторинг — рекомендуемая практика, поскольку в какой-то момент в будущем они могут оказаться включены и легче сразу начать их мониторинг, а не добавлять показатели потом.

Таблица 13.21. Показатели, требующие отслеживания

Клиент	Название компонента
Потребитель	bean kafka.consumer:type=consumer-fetch-manager-metrics,client-id=ID_клиента,attribute fetch-throttle-time-avg
Производитель	bean kafka.producer:type=producer-metrics,client-id=ID_клиента,attribute produce-throttle-time-avg

Мониторинг отставания

Самое важное, что требуется отслеживать потребителям Kafka, — это отставание потребителя. Оно измеряется количеством сообщений и представляет собой разницу между последним сгенерированным в конкретный раздел сообщением и последним сообщением, обработанным потребителем. Обычно оно оценивается на предыдущем этапе, при мониторинге клиента-потребителя, но это один из тех случаев, когда возможности внешнего мониторинга намного превышают возможности самого клиента. Как упоминалось ранее, в клиенте-потребителе существует показатель отставания, но использовать его неудобно. Он отражает данные только по одному разделу — тому, который отстает больше всего, так что не дает информации о том, насколько на самом деле отстает потребитель. Кроме того, он требует нормального функционирования потребителя, поскольку тот сам вычисляет его при каждом запросе на извлечение. Если потребитель работает некорректно или отключен, показатель будет неточен или вообще недоступен.

Предпочтительный метод мониторинга отставания потребителя — использование внешнего процесса, который может отслеживать как состояние раздела в брокере (наблюдая за последним сгенерированным сообщением), так и состояние потребителя (наблюдая за последним смещением, зафиксированным группой потребителей для данного раздела). Это дает объективную картину, своевременно обновляемую вне зависимости от состояния самого потребителя. Подобная проверка должна проводиться для всех разделов, которые читает группа потребителей. Если потребители большие, например MirrorMaker, это может означать десятки тысяч разделов.

В главе 12 приводилась информация об использовании утилит командной строки для получения информации о группах, включая зафиксированные смещения и отставание. Подобный мониторинг отставания, однако, связан с рядом проблем. Во-первых, необходимо знать, каков допустимый уровень отставания для каждого раздела. Для топика, получающего 100 сообщений в час, необходимо иное пороговое значение, чем для топика, получающего 10 000 сообщений

в секунду. Во-вторых, вы должны иметь возможность считывать все показатели отставания в систему мониторинга и устанавливать по ним оповещения. Если ваша группа потребителей потребляет 100 000 разделов из 1500 топиков, задача будет не из легких.

Один из способов снижения сложности мониторинга групп потребителей — использование Burrow (<https://oreil.ly/supY1>). Это приложение с открытым исходным кодом, разработанное компанией LinkedIn, обеспечивает мониторинг состояния потребителей путем сбора информации об отставании для всех групп потребителей кластера и подсчета единого показателя состояния для каждой группы, информирующего, функционирует ли она должным образом, отстает или вообще остановила работу. Для этого ему не требуются пороговые значения, полученные при мониторинге хода обработки сообщений группой потребителей, хотя можно получить отставание по сообщениям в виде конкретного числа. В технологическом блоге LinkedIn (<http://www.bit.ly/2sankZb>) приводится обстоятельное обсуждение обоснований и методологии работы Burrow. Развертывание Burrow — простой способ обеспечения мониторинга всех потребителей кластера или нескольких кластеров, его можно легко интегрировать с уже имеющимися у вас системами мониторинга и оповещения.

Если же других вариантов нет, показатель `records-lag-max` позволит получить хотя бы неполную картину состояния потребителя. Однако мы настоятельно рекомендуем использовать внешнюю систему мониторинга, например Burrow.

Сквозной мониторинг

Еще одна рекомендуемая для применения разновидность мониторинга, помогающая выяснить, нормально ли функционирует кластер Kafka, — сквозной мониторинг. Он позволяет взглянуть на состояние кластера Kafka с точки зрения клиента. У клиентов-производителей и клиентов-потребителей есть показатели, говорящие о наличии проблем с кластером Kafka, но можно только догадываться, из-за чего увеличилась задержка — из-за проблем с клиентом, сетью или самой Kafka. Кроме того, если вы отвечаете за работу только кластера Kafka, а не клиентов, вам придется отслеживать и функционирование клиентов тоже. На самом деле вам нужно знать:

- можно ли генерировать сообщения для кластера Kafka;
- можно ли потреблять сообщения из кластера Kafka.

В идеальном мире можно было бы отслеживать все это для каждого топика отдельно. Однако в большинстве случаев неразумно раздувать трафик топиков за счет искусственной добавки. Можно, однако, по крайней мере получить ответы

на данные вопросы для каждого из брокеров кластера, и именно это делает Xinfra Monitor (ранее известная как Kafka Monitor) (<https://oreil.ly/QqXD9>). Эта утилита с открытым исходным кодом, созданная командой разработчиков Kafka компании LinkedIn, непрерывно генерирует и потребляет данные из топика, распределенного по всем брокерам кластера. Она оценивает доступность каждого из брокеров для запросов как на потребление, так и на генерацию, а также общую сквозную задержку. Ценность такого мониторинга очень высока, поскольку он способен подтвердить, что кластер Kafka работает должным образом, ведь, как и в случае мониторинга отставания потребителя, брокер Kafka не может дать ответ на вопрос, есть ли у клиентов возможность использовать кластер так, как предполагается.

Резюме

Мониторинг — ключевой аспект правильной эксплуатации кластера Kafka. Именно поэтому многие команды разработчиков тратят значительную долю времени на отладку этой функциональности. Во многих компаниях Kafka используется для работы с петабайтными потоками данных. Одно из важнейших бизнес-требований — поток данных не должен прерываться, а сообщения не должны теряться. Мы должны также помогать пользователям с мониторингом применения Kafka их приложениями, обеспечивая необходимые для этого показатели.

В этой главе мы рассмотрели основы мониторинга Java-приложений, а именно приложений Kafka. Изучили небольшую часть многочисленных показателей, доступных в брокерах Kafka, коснулись вопросов мониторинга Java и операционной системы, а также журналирования. Далее мы подробно обсудили возможности мониторинга, имеющиеся у клиентских библиотек Kafka, включая мониторинг квот. Наконец, поговорили об использовании внешних систем мониторинга с целью отслеживания отставания потребителей и сквозной доступности кластера. Эта глава, хоть и не претендует на звание исчерпывающего списка доступных показателей, охватывает наиболее важные из них, требующие постоянного отслеживания.

Потоковая обработка

Кафка традиционно рассматривают как мощную шину сообщений, которая может доставлять потоки событий, но не имеет возможности обработать или преобразовать их. Надежность потоковой доставки делает Kafka идеальным источником данных для систем потоковой обработки. Apache Storm, Apache Spark Streaming, Apache Flink, Apache Samza и многие другие системы потоковой обработки зачастую проектируются в расчете на Kafka в качестве единственного надежного источника данных.

По мере роста популярности Apache Kafka сначала в качестве простой шины сообщений, а потом системы интеграции данных во многих компаниях начали появляться системы, содержащие множество потоков интересных данных, хранящихся в течение длительных промежутков времени и прекрасно упорядоченных, которые только и ждут, когда появится какой-нибудь потоковый фреймворк для их обработки. Другими словами, аналогично тому, как до изобретения баз данных обработка данных была гораздо более сложной задачей, потоковую обработку сдерживало отсутствие соответствующей платформы.

Начиная с версии 0.10.0, Kafka не просто обеспечивает надежный источник потоков данных для практически любого популярного фреймворка потоковой обработки. Теперь она включает в свой набор клиентских библиотек мощную библиотеку потоковой обработки, называемую Kafka Streams (или иногда Streams API). Благодаря этому разработчики могут потреблять, обрабатывать и генерировать события в своих приложениях и им не нужно использовать внешний фреймворк для обработки.

Начнем эту главу с объяснения того, что мы понимаем под потоковой обработкой, поскольку этот термин часто понимают неправильно, затем обсудим некоторые основные понятия потоковой обработки и паттерны проектирования, общие для всех систем потоковой обработки. Затем займемся библиотекой потоковой обработки Apache Kafka — ее задачами и архитектурой. Мы приведем небольшой пример использования библиотеки Kafka Streams для подсчета скользящего среднего цен акций. Затем обсудим другие примеры удачных

сценариев применения потоковой обработки и завершим главу коротким перечнем критериев выбора фреймворка потоковой обработки для использования совместно с Apache Kafka.

Эта глава задумана как краткое введение в большой и увлекательный мир потоковой обработки и Kafka Streams. На эти темы написаны целые книги.

Некоторые издания охватывают основные концепции потоковой обработки с точки зрения архитектуры данных.

- В книге «Понимание потоковой обработки» (<https://oreil.ly/omhmk>) Мартина Клеппманна (Martin Kleppmann) (O'Reilly) обсуждаются преимущества переосмысления приложений как приложений для обработки потоков и способы переориентации архитектуры данных на идею потоков событий.
- «Потоковые системы» (<https://oreil.ly/vcBBF>) Тайлера Акидау (Tyler Akidau), Славы Черняка (Slava Chernyak) и Реувена Лакса (Reuven Lax) (O'Reilly) — это отличное общее введение в тему потоковой обработки и некоторые основные идеи в этой области.
- «Архитектуры потоков» (<https://oreil.ly/ajOTG>) Джеймса Уркхарта (James Urquhart) (O'Reilly) ориентирована на технических директоров и обсуждает влияние потоковой обработки на бизнес.

Конкретным фреймворкам посвящены другие книги:

- «Kafka Streams и ksqlDB: данные в реальном времени» (<https://oreil.ly/5Ijpx>), автор Митч Сеймур (Mitch Seymour) (Питер, 2023);
- «Kafka Streams в действии. Приложения и микросервисы для работы в реальном времени» (<https://oreil.ly/TfUxs>), автор Уильям П. Беджек-младший (William P. Bejeck Jr.) (Питер, 2019);
- «Потоковая обработка событий с помощью Kafka Streams и ksqlDB» (<https://oreil.ly/EK06e>), автор Уильям П. Беджек-младший (Manning);
- «Обработка потоков с помощью Apache Flink» (<https://oreil.ly/ransF>), авторы Фабиан Хуэске и Василики Калаври (O'Reilly);
- «Обработка потоков с помощью Apache Spark» (<https://oreil.ly/B0ODf>), авторы Жерар Маас (Gerard Maas) и Франсуа Гарильо (Francois Garillot) (O'Reilly).

Наконец, Kafka Streams — это все еще фреймворк в стадии разработки. В каждом основном выпуске API объявляются устаревшими, а семантика изменяется. В этой главе описаны API и семантика для Apache Kafka версии 2.8. Мы избегали использования API, которые планировалось объявить устаревшими в версии 3.0, но наше обсуждение семантики соединений и обработки временных меток не включает ни одного из изменений, запланированных в этой версии.

Что такое потоковая обработка

Вокруг термина «потоковая обработка» существует большая путаница. Во многих определениях смешаны в кучу детали реализации, требования к производительности, модели данных и многие другие аспекты инженерии разработки ПО. Подобная ситуация произошла в сфере реляционных баз данных — абстрактные определения реляционной модели вечно теряются в деталях реализации и конкретных ограничениях распространенных движков баз данных.

Мир потоковой обработки активно развивается, и детали функционирования или конкретные ограничения какой-либо популярной реализации не означают, что эти особенности являются неотъемлемой частью обработки потоков данных.

Начнем с начала: что такое *поток данных* (data stream), называемый также *потоком событий* (event stream) или *потоковыми данными* (streaming data)? Прежде всего *поток данных* — это абстрактное представление неограниченного набора данных. *Неограниченность* (unbounded stream) означает его потенциально бесконечный размер и непрерывный рост. Набор данных является неограниченным, потому что с течением времени в него продолжают поступать все новые записи. Это определение применяется компаниями Google, Amazon, да и практически всеми остальными.

Обратите внимание на то, что эту простую модель (поток событий) можно использовать для представления практически любой бизнес-операции, которую только имеет смысл анализировать. Это может быть поток транзакций платежных карт, операций на фондовой бирже, доставки почты, проходящей через сетевой коммутатор, событий сети, событий от датчиков в промышленном оборудовании, отправленных сообщений электронной почты, шагов в игре и т. п. Список примеров бесконечен, поскольку практически все что угодно можно рассматривать как последовательность событий.

Вот еще несколько характерных черт модели потоков событий в дополнение к ее неограниченности.

- *Упорядоченность потоков событий.* Неотъемлемой частью потоков событий является информация о том, какие события произошли раньше, а какие — позже других. Наиболее ясно это в случае финансовых событий. Последовательность событий, при которой вы сначала кладете деньги на счет в банке, а затем их тратите, сильно отличается от последовательности, при которой вы сначала тратите деньги, а затем гасите долг путем помещения денег на счет. Второй вариант влечет за собой необходимость уплаты комиссионного сбора за перерасход средств, а первый — нет. Отметим, что в этом состоит

одно из различий между потоком событий и таблицей базы данных: записи в таблице всегда рассматриваются как неупорядоченные, а предложение `order by` оператора SQL не является частью реляционной модели, оно было добавлено для упрощения создания отчетности.

- *Неизменяемость записей данных.* Уже произошедшие события не могут меняться. Отмененная финансовая транзакция не исчезает. Вместо этого в поток записывается дополнительное событие, фиксирующее отмену предыдущей транзакции. При возврате покупателем товаров в магазин мы не удаляем факт продажи ему товаров, а записываем возврат в виде дополнительного события. Между потоком данных и таблицей базы данных есть и еще одно различие: мы можем удалять или модифицировать записи в таблице, но эти изменения представляют собой дополнительные транзакции базы данных, которые требуют записи в потоке событий, фиксирующей выполнение всех транзакций. Если вы знакомы с такими понятиями, как двоичные журналы, журналы упреждающей записи (`write-ahead log`, WAL) или журналы повтора (`redo log`), то знаете, что, если вставить запись в таблицу, а затем удалить ее, в таблице больше этой записи не будет, зато в журнале повтора будут сохраняться две транзакции — вставки и удаления.
- *Повторяемость потоков событий.* Это свойство желательно, но не обязательно. Хотя можно легко представить себе неповторяющиеся потоки событий (потоки TCP-пакетов, проходящих через сокет, обычно не повторяются), для большинства бизнес-приложений критически важно иметь возможность повтора необработанного потока событий, произошедших месяцы или даже годы тому назад. Это необходимо для исправления ошибок, экспериментов с новыми методами анализа и выполнения аудита. Kafka настолько усовершенствовала потоковую обработку в современных условиях именно потому, что обеспечивает возможность захвата и повтора потока событий. Без этого потоковая обработка была бы не более чем лабораторной игрушкой для исследователей данных.

Следует отметить, что ни в определении потока событий, ни в атрибутах, которые мы перечислим далее, ничего не говорится ни о содержащихся в событиях данных, ни о числе событий в секунду. В разных системах данные различаются — события могут быть крошечными (иногда всего лишь несколько байтов) или огромными (XML-сообщения с множеством заголовков), они могут быть совершенно не структурированными, парами «ключ/значение», полуструктурированным JSON или структурированными сообщениями Avro или Protobuf. Хотя потоки данных часто по умолчанию считаются большими данными, включающими миллионы событий в секунду, обсуждаемые нами методики так же успешно, а иногда и еще лучше подойдут для меньших потоков событий — с несколькими событиями в секунду или даже в минуту.

Теперь, когда мы разобрались, что такое потоки событий, самое время поговорить о потоковой обработке. Поточковая обработка означает непрерывную обработку одного или нескольких потоков событий. Поточковая обработка — парадигма программирования, как и парадигма «запрос/ответ» и пакетная обработка. Сравним различные парадигмы программирования, чтобы лучше понять место потоковой обработки в архитектурах программного обеспечения.

- *Запрос/ответ.* Это парадигма с минимальной задержкой, при которой время ответа колеблется от субмиллисекунд до нескольких миллисекунд, причем обычно ожидается, что время ответа всегда практически одинаковое. Обработка в большинстве случаев происходит с блокировкой — приложение отправляет запрос, после чего ждет ответа от системы обработки. В базах данных эта парадигма известна под названием *обработки транзакций в режиме реального времени* (online transaction processing, OLTP). POS-системы, обработка платежей по кредитным картам и системы учета рабочего времени обычно придерживаются этой парадигмы.
- *Пакетная обработка.* Этот вариант отличается длительной задержкой и высокой пропускной способностью. Система обработки активируется в заданное время, например каждый час или в 2:00 каждый день, и читает нужные входные данные (все появившиеся после прошлого выполнения данные, все данные с начала месяца и т. д.), записывает все требуемые результаты и прекращает работу до следующего запланированного времени запуска. Длительность обработки варьируется от минут до часов, и пользователи готовы к тому, что результаты могут оказаться устаревшими. В мире баз данных существуют хранилища данных и системы бизнес-аналитики, в которые один раз в день громадными пакетами загружаются данные, генерируются отчеты, и пользователи видят одни и те же отчеты до момента следующей загрузки данных. Эта парадигма отличается высокой эффективностью и экономичностью, но в последние годы для повышения своевременности и эффективности принятия решений в коммерческой деятельности данные требуются в более сжатые сроки. Это сильно затрудняет работу систем, написанных в расчете на экономичность обработки больших объемов данных, а не на получение отчетности практически без задержки.
- *Потоковая обработка.* Этот вариант отличается непрерывной обработкой без блокировки. Поточковая обработка заполняет пробел между вселенной «запрос/ответ», где приходится ждать событий, а обработка занимает всего 2 мс, и вселенной пакетной обработки, где данные обрабатываются раз в день и это занимает 8 часов. Большинство бизнес-процессов не требуют немедленного ответа в течение нескольких миллисекунд, но в то же время не могут ждать до следующего дня. Большинство бизнес-процессов происходит непрерывно,

и обработка может продолжаться, не ожидая конкретного ответа в течение нескольких миллисекунд, до тех пор, пока бизнес-отчеты обновляются непрерывно и линейка бизнес-приложений может реагировать непрерывно. Такие бизнес-процессы, как оповещение о подозрительных кредитных транзакциях или сетевых операциях, тонкая подстройка цен на основе спроса и предложения или отслеживание почтовой доставки, естественным образом подходят для непрерывной обработки без блокировок.

Важно отметить, что это определение не навязывает нам конкретного фреймворка, API или каких-либо функциональных возможностей. Раз мы непрерывно читаем данные из неограниченного набора данных, что-то с ними делаем и выводим результаты, значит, мы выполняем потоковую обработку. Но она должна быть непрерывной и постоянной. Запускаемый ежедневно в 2 часа ночи процесс, который читает 500 записей из потока, выводит результат и завершает работу, под определение потоковой обработки не подходит.

Основные понятия потоковой обработки

Потоковая обработка очень похожа на остальные виды обработки данных: мы пишем код, который получает данные, делает с ними что-либо — несколько преобразований, группировок и т. д. — и выводит куда-то результаты. Однако есть несколько специфичных для потоковой обработки понятий, часто сбивающих с толку тех, кто на основе своего опыта обработки данных из других сфер пытается писать приложения потоковой обработки. Рассмотрим некоторые из них.

Топология

Приложение для обработки потоков включает одну или несколько топологий обработки. Топология обработки начинается с одного или нескольких исходных потоков, которые проходят через граф потоковых процессоров, соединенных через потоки событий, пока результаты не будут записаны в один или несколько потоков-приемников. Каждый потоковый процессор представляет собой вычислительный шаг, применяемый к потоку событий для преобразования последних. Примерами некоторых потоковых процессоров, которые мы будем использовать в своих примерах, являются фильтрация, подсчет, группировка и левое соединение. Мы часто визуализируем приложения обработки потоков, рисуя узлы обработки и соединяя их стрелками, чтобы показать, как события передаются от одного узла к другому в процессе обработки данных приложением.

Время

Время, вероятно, важнейшее из понятий потокковой обработки, а заодно и наиболее запутанное. Если вы хотите получить представление о том, насколько сложным может быть время в сфере распределенных систем, рекомендуем взглянуть в превосходную статью Джастина Шийи (Justin Sheehy) «Настоящего не существует» (There is No Now) (<http://www.bit.ly/2rXXdLr>). В контексте потокковой обработки единое представление о времени критически важно, поскольку большинство потоковых приложений выполняют операции в соответствии с временными окнами. Например, потоковое приложение может вычислять скользящее пятиминутное среднее цен на акции. В этом случае нужно знать, что делать, если один из производителей отключается на два часа из-за проблем с сетью и возвращается в строй с данными за два часа, в основном относящимися к тем пятиминутным временным окнам, которые давным-давно прошли и для которых результаты уже подсчитаны и сохранены.

Системы потокковой обработки обычно используют следующие виды времени.

- *Время события.* Момент времени, когда произошло отслеживаемое событие и создана запись, — время, когда был измерен показатель, продан товар в магазине, пользователь открыл страницу веб-сайта и т. д. В версиях 0.10.0 и более поздних Kafka при создании в записи производителя автоматически добавляет текущее время. Если это не соответствует представлению приложения о *времени события*, например при создании записей Kafka на основе записи базы данных через какое-либо время после фактического события, мы рекомендуем добавить время события в виде поля самой записи, чтобы обе временные метки были доступны для последующей обработки. При обработке потоковых данных основное значение имеет именно время события.
- *Время добавления информации в журнал.* Время поступления события в брокер Kafka и сохранения его там, также называемое *временем приема*. В версиях 0.10.0 и более поздних брокеры Kafka автоматически добавляют его в получаемые записи, если Kafka настроена соответствующим образом или записи поступили от производителей более старых версий и не содержат меток даты/времени. В потокковой обработке такое понимание времени обычно не используется, поскольку при этом нас обычно интересует момент, когда произошло событие. Например, при подсчете числа произведенных за день устройств нас интересует число устройств, которые действительно были произведены в соответствующий день, даже если из-за проблем с сетью событие поступило в Kafka только на следующий день. Однако в случаях, когда настоящее время события не было зафиксировано, можно без потери согласованности воспользоваться временем добавления

информации в журнал: поскольку оно не меняется после создания записи и при отсутствии задержек в конвейере, это может быть разумным приближением времени события.

- *Время обработки.* Это момент времени, в который приложение потоковой обработки получило событие для выполнения каких-либо вычислений. Этот момент может отстоять на миллисекунды, часы или дни от того момента, когда произошло событие. При этом представлении о времени одному и тому же событию присваиваются различные метки даты/времени в зависимости от момента прочтения этого события каждым приложением потоковой обработки. Оно может различаться даже для двух потоков выполнения одного приложения! Следовательно, такое представление времени крайне ненадежно и лучше его избегать.

Kafka Streams присваивает время каждому событию на основе интерфейса `TimestampExtractor`. Разработчики приложений Kafka Streams могут использовать различные реализации этого интерфейса, которые могут использовать одну из трех временных семантик, описанных ранее, или совершенно иначе выбирать временную метку, в том числе извлекать ее из содержимого самого события.

Когда Kafka Streams записывает выходные данные в топик Kafka, она присваивает метку времени каждому событию на основе следующих правил.

- Когда выходная запись сопоставлена непосредственно с входной записью, выходная запись будет использовать ту же временную метку, что и входная.
- Когда выходная запись является результатом агрегации, временная метка выходной записи будет максимальной временной меткой, используемой в агрегации.
- Когда выходная запись является результатом объединения двух потоков, временная метка выходной записи будет наибольшей из двух объединяемых записей. При объединении потока и таблицы используется временная метка из записи потока.
- Наконец, если выходная запись была создана функцией Kafka Streams, которая генерирует данные по определенному расписанию независимо от входных данных, например `punctuate()`, метка времени вывода будет зависеть от текущего внутреннего времени приложения обработки потока.

Если используется API обработки нижнего уровня Kafka Streams, а не DSL, Kafka Streams включает API для манипулирования временными метками записей напрямую, поэтому разработчики могут реализовать семантику временных меток, соответствующую требуемой бизнес-логике приложения.



Не забывайте о часовых поясах

Работая с временем, важно помнить о часовых поясах. Весь конвейер данных должен работать с единым часовым поясом, иначе результаты потоковых операций окажутся запутанными, а зачастую бессмысленными. Если вам нужно работать с потоками данных в различных часовых поясах, убедитесь, что можете преобразовать события к одному часовому поясу, прежде чем выполнять операции с временными окнами. Часто это означает необходимость сохранения часового пояса в самой записи.

Состояние

До тех пор пока нам требуется обрабатывать события по отдельности, потокковая обработка — вещь очень простая. Например, для простого чтения потока транзакций о покупках в интернет-магазине из Kafka, поиска среди них транзакций на сумму более 10 000 долларов и отправки по электронной почте сообщения о них соответствующему торговцу нам достаточно нескольких строк кода с использованием потребителя Kafka и SMTP-библиотеки.

Наиболее интересной потокковая обработка становится при необходимости выполнения операций с несколькими событиями: подсчета числа событий по типам, вычисления скользящих средних, объединения двух потоков данных для обогащения потока информации и т. д. В подобных случаях недостаточно рассматривать события по отдельности. Необходимо отслеживать дополнительную информацию, например, сколько событий каждого типа встретилось нам за час, хранить список всех требующих объединения событий, сумм, средних значений и т. д. Мы будем называть эту информацию *состоянием* (state).

Заманчиво было бы хранить состояние в локальных переменных приложения потокковой обработки, например хранить скользящие средние в простой хеш-таблице. Однако такой подход к хранению состояния при потокковой обработке ненадежен, поскольку при остановке или выходе из строя приложения потокковой обработки состояние сбрасывается, что приводит к изменению результатов. Обычно это нежелательно, так что не забывайте сохранять последнее состояние и восстанавливать его при запуске приложения.

В потокковой обработке используются несколько типов состояния.

- *Локальное (внутреннее) состояние.* Состояние, доступное только конкретному экземпляру приложения потокковой обработки. Обычно хранится и контролируется встроенной базой данных в оперативной памяти, работающей внутри приложения. Преимущество локального состояния — исключительная быстрота работы с ним. Недостаток — ваши возможности ограничены объемом доступной памяти. В результате многие паттерны

проектирования в сфере потоковой обработки нацелены на разбиение данных на субпотоки, допускающие обработку при ограниченном размере локального состояния.

- *Внешнее состояние.* Состояние, хранимое во внешнем хранилище данных, обычно в NoSQL-системе наподобие Cassandra. Преимущества внешнего состояния — практически полное отсутствие ограничений размера и возможность доступа к нему из различных экземпляров приложения или даже различных приложений. Недостатки — увеличение времени задержки и приносимая еще одной системой дополнительная сложность, а также то, что приложение должно учитывать вероятность недоступности внешней системы. Большинство приложений потоковой обработки стараются избегать работы с внешним хранилищем или по крайней мере ограничивать накладные расходы из-за задержки за счет кэширования информации в локальном состоянии и взаимодействовать с внешним хранилищем как можно реже.

Таблично-поточковый дуализм

Все знают, что такое таблица базы данных. *Таблица* — это набор записей, идентифицируемых по первичному ключу и содержащих набор заданных схемой атрибутов. Записи таблицы изменяемые, то есть в таблицах разрешены операции обновления и удаления. С помощью запроса к таблице можно узнать состояние данных на конкретный момент времени. Например, при запросе к таблице `CUSTOMERS_CONTACTS` базы данных мы ожидаем, что получим подробные актуальные контактные данные всех наших покупателей. Если речь не идет о специально созданной «исторической» таблице, то предыдущих контактных данных в ней не будет.

В отличие от таблиц в потоках содержится история изменений. *Поток* представляет собой последовательность событий, в которой каждое событие является причиной изменения данных. Из этого описания очевидно, что потоки и таблицы — две стороны одной монеты: мир непрерывно меняется, и иногда нас интересуют вызвавшие изменения события, а иногда — текущее состояние. Возможности систем, которые позволяют нам перемещаться между двумя представлениями данных, шире возможностей систем, поддерживающих лишь одно представление.

Для преобразования потока в таблицу необходимо фиксировать вызывающие ее модификацию события. Следует сохранить все события `insert`, `update` и `delete` в таблице. Большинство СУБД с этой целью предоставляют утилиты для сбора данных об изменениях (change data capture, CDC). Кроме того, существует множество коннекторов Kafka для конвейерной передачи этих изменений в Kafka и дальнейшей их потоковой обработки.

Для преобразования потока данных в таблицу необходимо применить все содержащиеся в этом потоке изменения. Этот процесс называется *материализацией* (materializing) потока данных. Создается таблица в оперативной памяти, внутреннем хранилище состояний или внешней базе данных, после чего мы проходим по всем событиям из потока данных, от начала до конца, изменяя состояние по мере продвижения. По окончании у нас будет пригодная для использования таблица, отражающая состояние на конкретный момент времени.

Допустим, у нас есть обувной магазин. Потокковое представление розничных продаж может представлять собой поток следующих событий.

- «Прибыла партия красных, синих и зеленых туфель».
- «Проданы синие туфли».
- «Проданы красные туфли».
- «Покупатель вернул синие туфли».
- «Проданы зеленые туфли».

Чтобы узнать, что находится на складе в настоящий момент или сколько денег мы уже заработали, необходимо материализовать представление. Как видно из рис. 14.1, сейчас у нас есть 299 пар красных туфель. Чтобы увидеть, насколько загружен магазин, можно просмотреть весь поток данных и увидеть, что сегодня произошли четыре события, связанных с клиентами. Возможно, нам захочется также выяснить, почему вернули синие туфли.

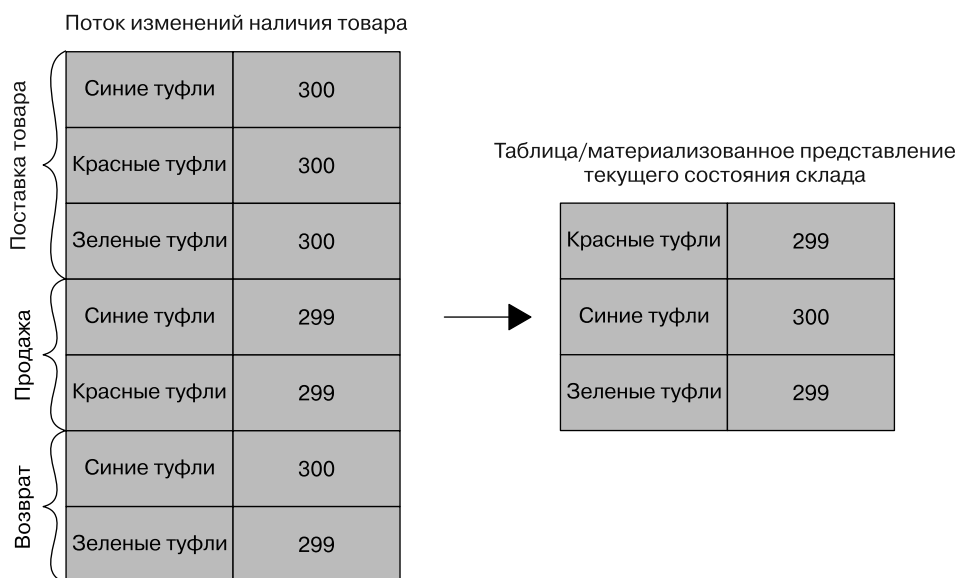


Рис. 14.1. Материализация изменений товарных остатков

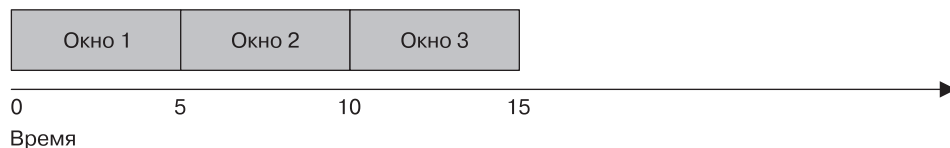
Временные окна

Большинство операций над потоками данных — оконные, то есть оперирующие временными интервалами: скользящие средние, самые продаваемые товары за неделю, 99-й процентиль нагрузки на систему и т. д. Операции объединения двух потоков данных также носят оконный характер — при этом объединяются события, произошедшие в один промежуток времени. Очень немногие люди останавливаются хоть на секунду, чтобы задуматься, какой именно тип временного окна им требуется. Например, при вычислении скользящих средних необходимо знать следующее.

- *Размер окна*: нужно вычислить среднее значение по всем событиям из каждого пятиминутного окна? Каждого 15-минутного окна? Или за целый день? Чем больше окно, тем лучше сглаживание, но и больше отставание — чтобы заметить увеличение цены, понадобится больше времени, чем при меньшем окне. Kafka Streams включает в себя также *окно сессии* (session window), где размер окна определяется периодом бездействия. Разработчик определяет промежуток между сессиями, и все события, которые непрерывно поступают с интервалами меньшими, чем определенный промежуток между сессиями, относятся к одной и той же сессии. Разрыв в поступлениях определяет новую сессию, и все события, поступающие после разрыва, но до следующего разрыва, будут принадлежать новой сессии.
- *Насколько часто окно сдвигается (интервал опережения, advance interval)*: обновлять ли пятиминутные средние значения каждую минуту, секунду или при каждом поступлении нового события? Окна, для которых размер является фиксированным временным интервалом, называются *прыгающими окнами* (hopping windows). Окно, размер которого равен его *интервалу опережения*, иногда называют *кувыркающимся* (tumbling window).
- *В течение какого времени сохраняется возможность обновления окна (льготный период, grace period)*: допустим, что пятиминутное скользящее среднее подсчитывается для окна 00:00–00:05. А через час мы получаем еще несколько входных данных, относящихся к 00:02. Обновлять ли результаты для периода 00:00–00:05? Или что было, то прошло? Оптимально было бы задавать определенный промежуток времени, в течение которого события могут добавляться к соответствующему временному срезу. Например, если они наступили не позднее чем через четыре часа, необходимо пересчитать и обновить результаты. Если же позже, то их можно игнорировать.

Можно выравнивать окна по показаниям часов, то есть первым срезом пятиминутного окна, перемещающегося каждую минуту, будет 00:00–00:05, а вторым — 00:01–00:06. Или можно не выравнивать, а просто начинать окно с момента запуска приложения, так что первым срезом будет, например, 3:17–3:22. Различия между этими двумя типами окон показаны на рис. 14.2.

Кувыркающееся окно: пятиминутное окно, перемещается каждые 5 минут



Прыгающее окно: пятиминутное окно, перемещается каждую минуту.

Окна перекрываются, так что одно и то же событие может относиться к нескольким окнам

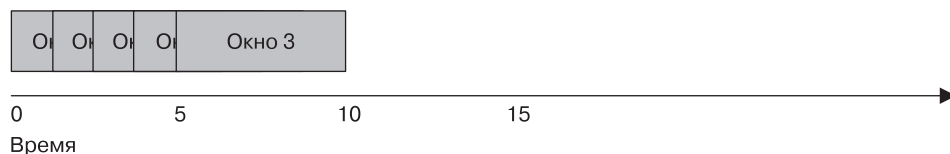


Рис. 14.2. Кувыркающиеся и прыгающие окна

Гарантии обработки

Ключевым требованием для приложений потоковой обработки является возможность обработки каждой записи ровно один раз независимо от сбоев. Без гарантий обработки только один раз потоковая обработка не может применяться в случаях, когда требуются точные результаты. Как подробно рассматривалось в главе 8, Apache Kafka поддерживает семантику «только один раз» с транзакционным и идемпотентным производителем. Kafka Streams использует транзакции Kafka для реализации гарантий «только один раз» для приложений потоковой обработки. Каждое приложение, использующее библиотеку Kafka Streams, может включить гарантии «только один раз», установив параметр `processing.guarantee` в значение `exactly_once`. Kafka Streams версии 2.6 или более поздней включает более эффективную реализацию «только один раз», которая требует наличия брокеров Kafka версии 2.5 или более поздней. Эту эффективную реализацию можно включить, установив параметру `processing.guarantee` значение `exactly_once_beta`.

Паттерны проектирования потоковой обработки

Между собой различаются все системы потоковой обработки, от простых сочетаний потребителя, логики обработки и производителя до таких сложных кластеров, как Spark Streaming с его библиотеками машинного обучения, включая множество промежуточных вариантов. Но существуют базовые паттерны проектирования, разработанные для удовлетворения часто встречающихся требований архитектур потоковой обработки. Рассмотрим несколько широко известных паттернов и покажем примеры их применения.

Обработка событий по отдельности

Простейший паттерн потоковой обработки — обработка каждого события по отдельности. Он известен также под названием *паттерна отображения/фильтрации*, поскольку часто используется для фильтрации ненужных событий из потока или преобразования событий. (Термин «отображение» (*map*) ведет начало от паттерна отображения/свертки (*map/reduce*), в котором события преобразуются на этапе отображения, после чего агрегируются на этапе свертки.)

В этом паттерне приложение потоковой обработки читает события из потока, модифицирует каждое из них, после чего генерирует события в другой поток. В качестве примера можно привести приложение, читающее журнальные сообщения из потока данных и записывающее события **ERROR** в поток с максимальным приоритетом, а остальные — в поток с минимальным приоритетом. Еще один пример — приложение, читающее события из потока данных и меняющее их формат с JSON на Avro. Подобным приложениям не нужно хранить внутри себя состояние, поскольку события могут обрабатываться по отдельности. Это значит, что восстановление после сбоев или балансировка нагрузки чрезвычайно упрощаются, ведь восстанавливать состояние не нужно, мы можем просто делегировать обработку событий другому экземпляру приложения.

Для этого паттерна вполне достаточно простого производителя и потребителя (рис. 14.3).

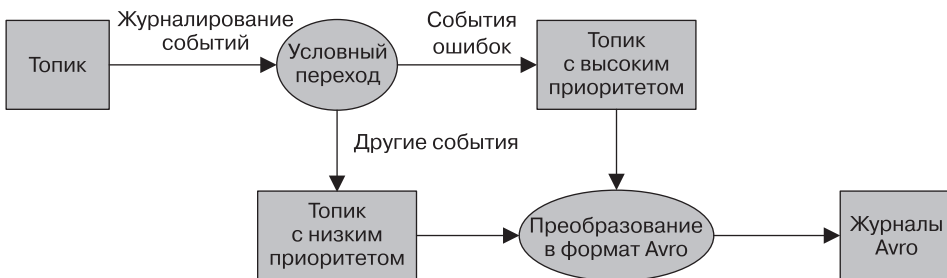


Рис. 14.3. Топология обработки событий по отдельности

Обработка с использованием локального состояния

Для большинства приложений потоковой обработки важную роль играет агрегирование информации, особенно по окнам. Примером этого может служить поиск минимальной и максимальной цены акций для каждого дня торгов и вычисление скользящего среднего.

Подобное агрегирование требует сохранения *состояния*. В нашем примере для вычисления минимальной и средней цены акций за день необходимо

хранить встречавшиеся до сих пор минимальное значение, сумму и количество записей.

Для этого можно использовать *локальное* (неразделяемое) состояние, поскольку все операции в примере представляют собой агрегирование типа `group by`, то есть производящееся по каждому символу акции, а не по рынку акций в целом. Чтобы гарантировать запись событий с одним символом акции в один раздел, воспользуемся объектом `Partitioner` Kafka. Далее каждый экземпляр приложения получит все события из назначенных ему разделов (это гарантирует потребитель Kafka). Это значит, что каждый экземпляр приложения может хранить состояние для подмножества символов акций, записанных в соответствующие ему разделы (рис. 14.4).

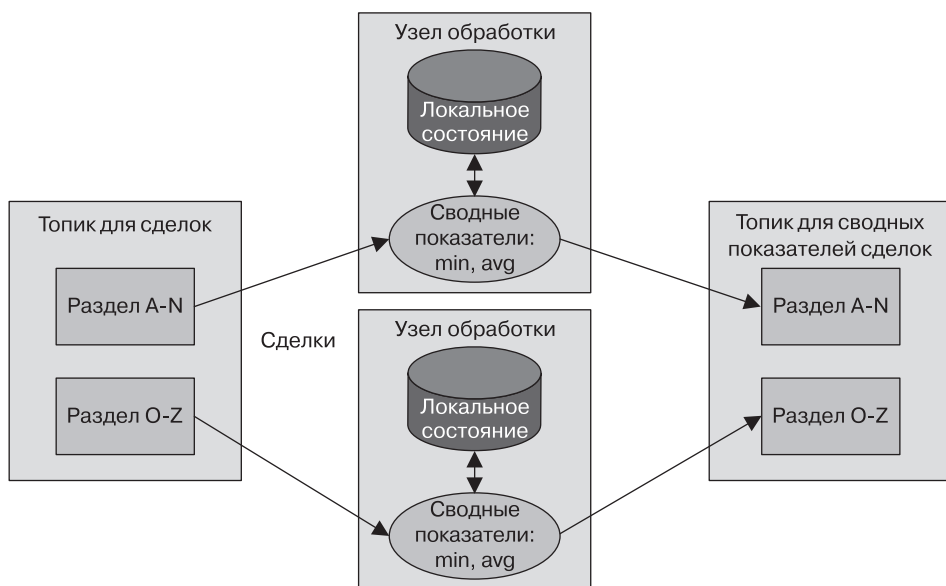


Рис. 14.4. Топология обработки событий с применением локального состояния

Приложения потоковой обработки существенно усложняются при наличии у приложения локального состояния. Возникает несколько проблем, которые должно решить такое приложение.

- *Использование памяти.* Локальное состояние идеально помещается в доступной экземпляру приложения оперативной памяти. Некоторые локальные хранилища позволяют выполнять выгрузку на диск, но это существенно влияет на производительность.
- *Сохраняемость.* Необходимо гарантировать, что состояние не будет утрачено при останове экземпляров приложения и есть возможность восстановить его при повторном их запуске или замене на другой экземпляр. С этим отлично

справляется библиотека Kafka Streams — локальное состояние сохраняется в оперативной памяти с помощью встроенной базы RocksDB, сохраняющей также данные на диск для быстрого восстановления после перезапуска. Но все изменения в локальном состоянии отправляются и в топик Kafka. В случае останова узла потока данных локальное состояние не утрачивается — его можно легко восстановить путем повторного чтения событий из топика Kafka. Например, если локальное состояние содержало текущий минимум для акций IBM = 167,19, оно сохраняется в Kafka, так что позднее можно будет повторно заполнить локальный кэш на основе этих данных. Kafka применяет для топиков сжатие журналов, чтобы гарантировать, что они не будут расти до бесконечности, и иметь возможность восстановить состояние.

- *Переназначение.* Иногда разделы переназначаются другому потребителю. При этом экземпляр, у которого «отобрали» раздел, должен сохранить последнее рабочее состояние, а экземпляр, получивший раздел, — восстановить нужное состояние.

Фреймворки потоковой обработки в разной степени обеспечивают разработчикам возможность администрирования нужного им локального состояния. Если вашему приложению требуется хранить локальное состояние, нам надо проверить, какие гарантии обеспечивает используемый фреймворк. В конце главы мы приведем краткое сравнительное руководство по ним, но, как всем известно, программное обеспечение меняется очень быстро, особенно фреймворки потоковой обработки.

Многоэтапная обработка/повторное разделение на разделы

Локальное состояние — отличная вещь, если требуется агрегирование типа group by. Но что, если результаты должны использовать всю доступную информацию? Например, допустим, что нужно каждый день публиковать десять самых быстро растущих ценных бумаг — десять ценных бумаг, стоимость которых сильнее всего выросла за день торгов (с открытия до закрытия биржи). Конечно, никаких локальных действий на отдельном экземпляре приложения не будет для этого достаточно, поскольку все десять нужных ценных бумаг могут находиться в разделах, относящихся к другим экземплярам. Нам понадобится двухэтапный подход. Сначала нужно вычислить ежедневный рост/падение цены для каждого из символов акций. Это можно сделать в каждом из отдельных экземпляров с помощью локального состояния. Затем следует записать результаты в новый топик из одного раздела. Далее отдельный экземпляр приложения читает этот раздел и находит там десять наиболее быстро растущих в цене акций. Второй топик, содержащий лишь сводные показатели по символам акций, очевидно, будет намного меньше (как и объем его трафика), чем топики, содержащие саму информацию о сделках, а значит, его сможет обработать один экземпляр приложения. Иногда для получения результата необходимо больше шагов (рис. 14.5).

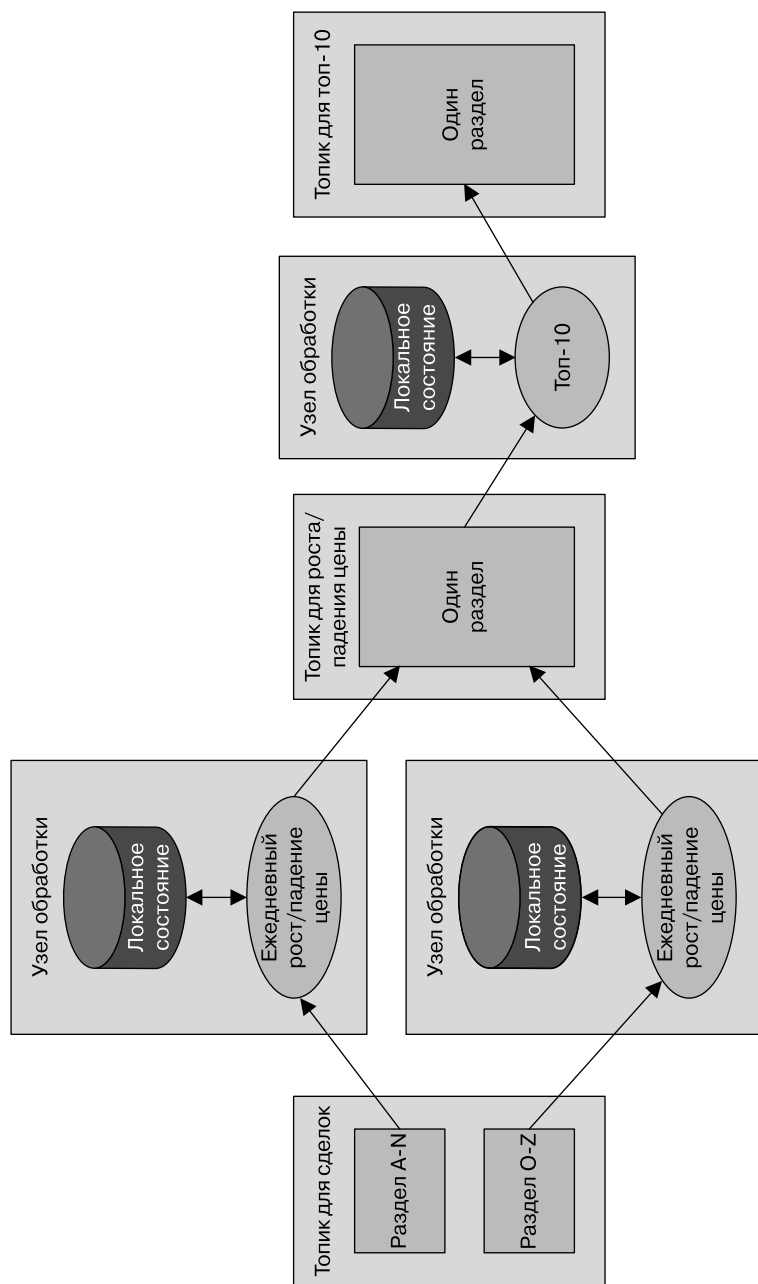


Рис. 14.5. Топология, включающая как использование локального состояния, так и повторное секционирование

Такая разновидность многоэтапной обработки хорошо знакома тем, кому случалось писать код отображения/свертки (MapReduce), в котором часто приходится прибегать к нескольким этапам свертки. Если вы хотя бы раз писали такой код, то помните, что для каждого этапа свертки необходимо отдельное приложение. В отличие от MapReduce при использовании большинства фреймворков потоковой обработки можно включить все этапы в одно приложение, в то время как фреймворк возьмет на себя распределение выполнения этапов по экземплярам или исполнителям приложения.

Обработка с применением внешнего справочника: соединение потока данных с таблицей

Иногда для потоковой обработки необходима интеграция с внешним по отношению к потоку производителем данных. Это нужно, например, для проверки соответствия транзакций набору хранимых в базе данных правил или для обогащения данных о маршрутах перемещения по веб-сайту пользователей информацией о них.

Очевидный вариант задействования внешнего справочника для обогащения данных выглядит примерно так: при каждом встреченном в потоке событии перехода пользователя по ссылке находить соответствующего пользователя в базе данных профилей и записывать в другой топик событие, включающее первоначальный щелчок на ссылке плюс возраст и пол пользователя (рис. 14.6).

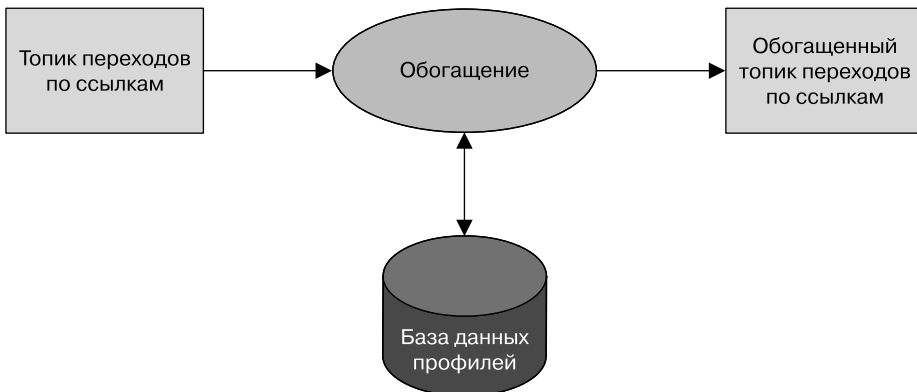


Рис. 14.6. Потоковая обработка с применением внешнего источника данных

Проблема с этим напрашивающимся вариантом состоит в том, что внешний справочник существенно увеличивает время обработки каждой записи — обычно на 5 или 15 мс. Во многих случаях это недопустимо. Зачастую неприемлема

и возникающая дополнительная нагрузка на внешнее хранилище — системы потоковой обработки обычно способны обрабатывать 100–500 тысяч событий в секунду, а базы данных, вероятно, лишь 10 тысяч событий в секунду при сносной производительности. Также появляется дополнительная сложность, связанная с доступностью, — наше приложение должно будет обрабатывать ситуации, когда внешняя база данных недоступна.

Чтобы обеспечить хорошую производительность и доступность, необходимо кэшировать информацию из базы данных в приложении потоковой обработки. Однако управление кэшем может оказаться непростой задачей: как предотвратить устаревание информации в нем? Если слишком часто обновлять события, то нагрузка на базу данных все равно будет большой и кэш особо не поможет. Если же получать новые события слишком редко, то потоковая обработка будет выполняться на основе устаревшей информации.

Но если бы мы смогли захватывать все происходящие с таблицей базы данных изменения в поток событий, то можно было бы организовать прослушивание этого потока заданием, которое выполняет потоковую обработку, и обновлять кэш в соответствии с событиями изменения базы данных. Процесс захвата вносимых в базу данных изменений в виде событий потока данных носит название *«сбор данных об изменениях»* (change data capture, CDC). В Kafka Connect есть множество коннекторов, предназначенных для выполнения CDC и преобразования таблиц базы данных в поток событий изменения. Благодаря этому мы сможем хранить собственную копию таблицы, обновляя ее соответствующим образом при получении уведомления о каждом событии изменения базы данных (рис. 14.7).

Далее при получении событий переходов пользователей по ссылкам мы сможем найти `user_id` в локальном состоянии и выполнить обогащение события. Благодаря применению локального состояния такое решение масштабируется намного лучше и не оказывает негативного влияния на базу данных и другие использующие ее приложения.

Мы будем называть этот вариант *соединением потока данных с таблицей* (stream-table join), поскольку один из потоков отражает изменения в кэшируемой локально таблице.

Соединение таблицы с таблицей

В предыдущем разделе мы обсудили, насколько эквивалентны таблица и поток событий обновления. Подробно рассмотрели, как это работает при соединении потока и таблицы. Нет никаких причин, по которым мы не можем иметь материализованные таблицы на обеих сторонах операции соединения.

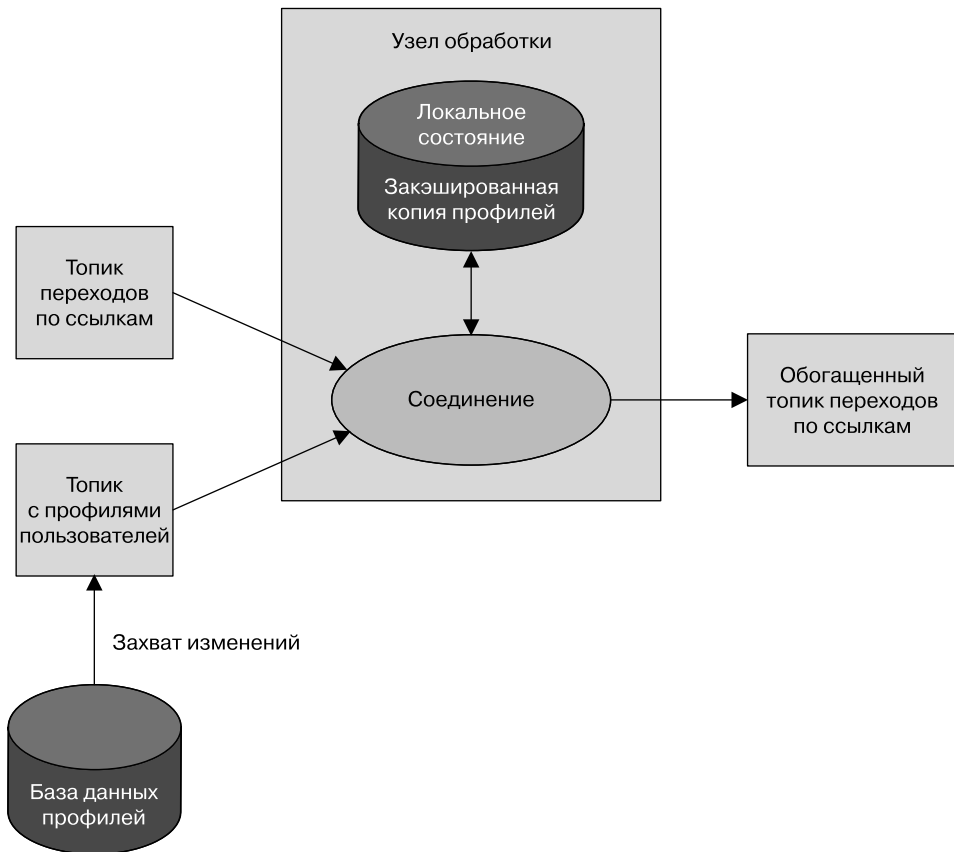


Рис. 14.7. Топология соединения таблицы и потока событий, благодаря которой нет необходимости использовать внешний источник данных при потоковой обработке

Соединение двух таблиц всегда выполняется без окна и позволяет увидеть текущее состояние обеих таблиц на момент выполнения операции. С помощью Kafka Streams мы можем выполнить эквисоединение, при котором обе таблицы имеют одинаковый ключ, который разделен одинаковым образом, и поэтому операция соединения может быть эффективно распределена между большим количеством экземпляров приложений и машин.

Kafka Streams также поддерживает соединение двух таблиц по внешнему ключу — ключ одного потока или таблицы соединяется с произвольным полем из другого потока или таблицы. Подробнее о том, как это работает, вы можете узнать из выступления «Пересечение потоков» (<https://oreil.ly/f34U6>), сделанного на саммите Kafka Summit 2020, или из более подробной статьи в блоге (<https://oreil.ly/hlKNz>).

Соединение потоков

Иногда бывает нужно соединить два потока событий, а не поток с таблицей. Благодаря чему поток данных становится настоящим? Как вы помните из обсуждения в начале главы, потоки неограниченны. При использовании потока для представления таблицы можно смело проигнорировать большую часть исторической информации из него, поскольку нас в этом случае интересует только текущее состояние. Но соединение двух потоков данных означает соединение полной истории событий и поиск соответствий событий из одного потока событиям из другого, относящимся к тем же временным окнам с такими же ключами. Поэтому соединение потоков называют также *оконным соединением* (windowed-join).

Например, имеется один поток данных с поисковыми запросами, которые пользователи вводили на нашем веб-сайте, а второй — со сделанными ими щелчками кнопкой мыши на ссылках, в том числе на результатах запросов. Нужно найти соответствия поисковых запросов результатам, на которых щелкнули пользователи, чтобы выяснить, какие результаты наиболее популярны при каком запросе. Разумеется, нам хотелось бы найти соответствия результатов по ключевым словам, но только соответствия в пределах определенного временного окна. Мы предполагаем, что пользователь выполняет щелчок на результате поиска в течение нескольких секунд после ввода запроса в поисковую систему. Так что имеет смысл использовать для каждого потока окна небольшие, длиной несколько секунд, и искать соответствие результатов для каждого из них (рис. 14.8).

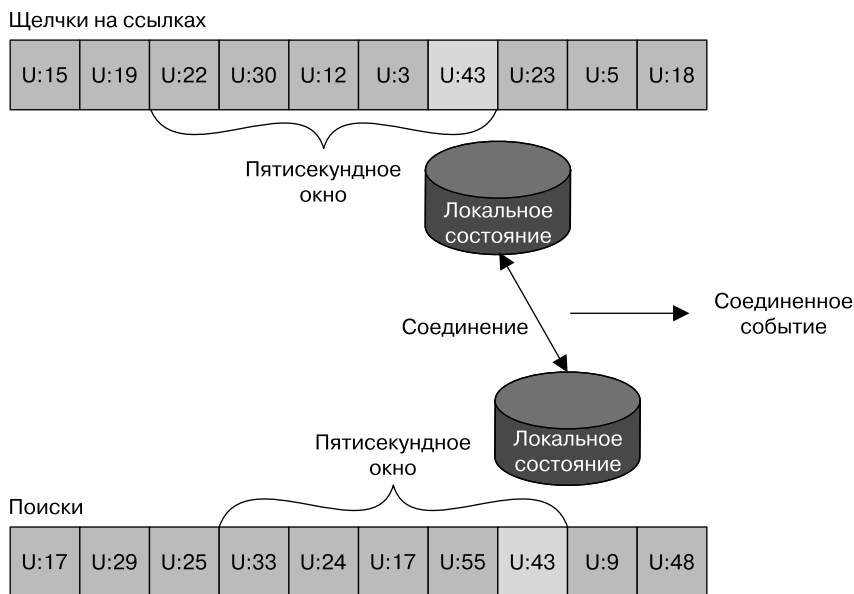


Рис. 14.8. Соединение двух потоков событий. В таких случаях всегда применяется временное окно

Kafka Streams поддерживает эквисоединение, в котором потоки данных, запросы и щелчки на ссылках секционируются по одним и тем же ключам, представляющим собой и ключи соединения. При этом все события щелчков от пользователя `user_id:42` попадают в раздел 5 топики событий щелчков, а все события поиска для `user_id:42` — в раздел 5 топики событий поиска. После этого Kafka Streams обеспечивает назначение раздела 5 обоих топиков одной задаче Kafka, так что ей оказываются доступны все соответствующие события для пользователя `user_id:42`. Она хранит во встроенном хранилище данных RocksDB временное окно соединения для обоих топиков и благодаря этому может выполнить соединение.

Внеочередные события

Обработка событий, поступивших в поток несвоевременно, — непростая задача не только в потоковой обработке, но и в традиционных ETL-системах. Внеочередные события — довольно часто встречающееся обыденное явление в сценариях Интернета вещей (IoT) (рис. 14.9). Например, мобильное устройство может потерять сигнал Wi-Fi на несколько часов и отправить данные за это время после восстановления соединения. Подобное случается и при мониторинге сетевого оборудования (сбойный сетевой коммутатор не отправляет диагностических сигналов о своем состоянии, пока не будет починен) или в машиностроении (печально известны своей нестабильностью сетевые соединения на фабриках, особенно в развивающихся странах).

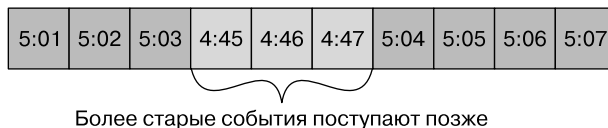


Рис. 14.9. Внеочередные события

Наши потоковые приложения должны корректно работать при подобных сценариях. Обычно это означает, что такое приложение должно:

- распознать несвоевременное поступление события, для чего прочитать время события и определить, что оно меньше текущего;
- определиться с интервалом времени, в течение которого оно будет пытаться синхронизировать внеочередные события. Скажем, при задержке в три часа событие можно синхронизировать, а события трехнедельной давности можно отбросить;
- обладать достаточными возможностями для синхронизации данного события. Именно в этом и состоит основное различие между потоковыми

приложениями и пакетными заданиями. Если несколько событий поступили после завершения ежедневного пакетного задания, можно просто запустить вчерашнее задание повторно и обновить события. В случае же потоковой обработки возможности запустить вчерашнее задание повторно нет — один и тот же непрерывный процесс должен обрабатывать как старые, так и новые события;

- иметь возможность обновить результаты. Если результаты потоковой обработки записываются в базу данных, для их обновления достаточно команды `put` или `update`. В случае отправки потоковым приложением результатов по электронной почте выполнение обновлений может оказаться более сложной задачей.

В нескольких фреймворках потоковой обработки, в том числе Dataflow компании Google и Kafka Streams, есть встроенная поддержка независимого от времени обработки (основного времени) представления времени, а также возможность обработки событий, время которых больше или меньше текущего основного времени. Обычно для этого в локальном состоянии хранятся несколько доступных для обновления окон агрегирования, причем разработчики могут настраивать промежуток времени, в течение которого они доступны для обновления. Конечно, чем больше этот промежуток, тем больше памяти необходимо для хранения локального состояния.

API Kafka Streams всегда записывает результаты агрегирования в топики результатов. Обычно они представляют собой сжатые топики, то есть для каждого ключа сохраняется только последнее значение. При необходимости обновления результатов окна агрегирования вследствие поступления запоздавшего события Kafka Streams просто записывает новый результат для данного окна агрегирования, фактически заменяя предыдущий результат.

Повторная обработка

Последний из важных паттернов — повторная обработка событий. Существует два его варианта.

- У нас появилась новая версия приложения потоковой обработки, и нужно организовать обработку этой версией того же потока событий, который обрабатывает старая, получить новый поток результатов, не замещающий первой версии, сравнить две версии результатов и в какой-то момент перевести клиентов на использование новых результатов вместо существующих.
- В наше приложение потоковой обработки вкралась программная ошибка. Мы ее исправили и хотели бы заново обработать поток событий и вычислить новые результаты.

Первый сценарий основан на том, что Apache Kafka в течение длительного времени хранит потоки событий целиком в масштабируемом хранилище данных. Это значит, что для работы двух версий приложения потоковой обработки, записывающих два потока результатов, достаточно выполнить следующее.

- Развернуть новую версию приложения в качестве новой группы потребителей.
- Настроить новую версию так, чтобы она начала обработку с первого смещения исходных топиков (а значит, у нее была своя копия всех событий из входных потоков).
- Продолжить работу нового приложения и переключить клиентские приложения на новый поток результатов, после того как новая версия выполняющего обработку задания наверстает отставание.

Второй сценарий сложнее — он требует перенастроить существующее приложение так, чтобы начать обработку с начала входных потоков данных, сбросить локальное состояние (чтобы не смешались результаты, полученные от двух версий приложения) и, возможно, очистить предыдущий выходной поток. Хотя в составе библиотеки Kafka Streams есть утилита для сброса состояния приложения потоковой обработки, мы рекомендуем использовать первый вариант во всех случаях, когда есть ресурсы для запуска двух копий приложения и генерирования двух потоков результатов. Первый метод намного безопаснее — он позволяет переключаться между несколькими версиями и сравнивать их результаты, не рискуя потерять критически важные данные или внести ошибки в процессе очистки.

Интерактивные запросы

Как обсуждалось ранее, приложения потоковой обработки имеют состояние, которое может быть распределено между многими экземплярами приложения. В большинстве случаев пользователи приложений потоковой обработки получают результаты обработки, считывая их из топика вывода. В некоторых случаях, однако, желательно использовать самый короткий путь и считать результаты из самого хранилища состояния. Так часто бывает, когда результатом является таблица (например, топ-10 самых продаваемых книг), а поток результатов на самом деле представляет собой поток обновлений этой таблицы, — гораздо быстрее и проще прочитать таблицу непосредственно из состояния приложения потоковой обработки.

Kafka Streams включает в себя гибкие API для запроса состояния приложения обработки потоков (<https://oreil.ly/pCGeC>).

Kafka Streams в примерах

Приведем несколько примеров использования API фреймворка Apache Kafka Streams, чтобы продемонстрировать реализацию рассмотренных паттернов на практике. Мы берем именно этот конкретный API из-за простоты его применения, а также потому, что он поставляется вместе с уже имеющимся у нас Apache Kafka. Важно помнить, что эти паттерны можно реализовать в любом фреймворке потоковой обработки или библиотеке — сами паттерны универсальны, конкретны только примеры.

В Apache Kafka есть два потоковых API — низкоуровневый Processor API и высокоуровневый Streams DSL. Мы воспользуемся Kafka Streams DSL. Он позволяет задавать приложение потоковой обработки путем описания последовательности преобразований событий потока. Преобразования могут быть простыми, например фильтрами, или сложными, например соединениями потоков. Низкоуровневый API позволяет создавать собственные преобразования. Чтобы узнать больше о низкоуровневом API процессора, ознакомьтесь с руководством разработчика (<https://oreil.ly/bQ5nE>), которое содержит подробную информацию, а презентация «За пределами DSL» (<https://oreil.ly/4vson>) является отличным введением.

Создание приложения, задействующего API DSL, всегда начинается с формирования с помощью StreamsBuilder *топологии* обработки — ориентированного ациклического графа (DAG) преобразований, применяемых ко всем событиям потоков. Затем на основе топологии создается исполняемый объект `KafkaStreams`. При его запуске создается несколько потоков выполнения, каждый из которых использует топологию обработки к событиям из потоков. Обработка завершается по закрытию объекта `KafkaStreams`.

Мы рассмотрим несколько примеров применения Kafka Streams для реализации некоторых из обсуждавшихся ранее паттернов проектирования. Для демонстрации паттерна отображения/свертки и простых сводных показателей воспользуемся простым примером с подсчетом слов. Затем перейдем к примеру с вычислением различных сводных статистических показателей для рынка ценных бумаг, который позволит продемонстрировать сводные показатели по временным окнам. И наконец, проиллюстрируем соединение потоков на примере обогащения потока переходов по ссылкам.

Подсчет количества слов

Вкратце рассмотрим сокращенный вариант примера подсчета слов для Kafka Streams. Полный пример вы можете найти на GitHub (<http://www.bit.ly/2ri00gj>).

Прежде всего при создании приложения потоковой обработки необходимо настроить Kafka Streams. У него есть множество параметров, которые мы не ста-

нем тут обсуждать, так как их описание можно найти в документации (<http://www.bit.ly/2t7obPU>). Кроме того, можно настроить встроенные в Kafka Streams производитель и потребитель, добавив любые нужные настройки производителя или потребителя в объект `Properties`:

```
public class WordCountExample {

    public static void main(String[] args) throws Exception{

        Properties props = new Properties();
        props.put(StreamsConfig.APPLICATION_ID_CONFIG,
            "wordcount"); ❶
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
            "localhost:9092"); ❷
        props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
            Serdes.String().getClass().getName()); ❸
        props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
            Serdes.String().getClass().getName());
```

❶ У каждого приложения Kafka должен быть свой идентификатор приложения. Он используется для координации действий экземпляров приложения, а также для именования внутренних локальных хранилищ и относящихся к ним топиков. Среди приложений Kafka Streams, работающих в пределах одного кластера, идентификаторы не должны повторяться.

❷ Приложения Kafka Streams всегда читают данные из топиков Kafka и записывают результаты в топик Kafka. Как мы увидим далее, приложения Kafka Streams также применяют Kafka для координации своих действий. Так что лучше указать приложению, где искать Kafka.

❸ Приложение должно выполнять сериализацию и десериализацию при чтении и записи данных, поэтому мы указываем классы, наследующие интерфейс `Serde` для использования по умолчанию.

Задав настройки, можно перейти к построению топологии потоков:

```
StreamsBuilder builder = new StreamsBuilder(); ❶

KStream<String, String> source =
    builder.stream("wordcount-input");

final Pattern pattern = Pattern.compile("\\W+");

KStream<String, String> counts = source.flatMapValues(value->
    Arrays.asList(pattern.split(value.toLowerCase()))) ❷
    .map((key, value) -> new KeyValue<String,
        String>(value, value))
    .filter((key, value) -> (!value.equals("the"))) ❸
    .groupByKey() ❹
```

```

        .count().mapValues(value->
            Long.toString(value)).toStream(); ❸
counts.to("wordcount-output"); ❹

```

❶ Создаем объект класса `StreamsBuilder` и приступаем к описанию потока, передавая название входного топика.

❷ Все читаемые из топика-производителя события представляют собой строки слов. Мы разбиваем их с помощью регулярного выражения на последовательности отдельных слов. Затем вставляем каждое из слов — значение записи для какого-либо события — в ключ записи этого события для дальнейшего использования в операции группировки.

❸ Отфильтровываем слово `the` лишь для демонстрации того, как просто это делать.

❹ И группируем по ключу, получая наборы событий для каждого уникального слова.

❺ Подсчитываем количество событий в каждом наборе. Заносим результаты в значение типа `Long`. Преобразуем его в `String` для большей удобочитаемости результатов.

❻ Осталось только записать результаты обратно в `Kafka`.

После описания последовательности выполняемых приложением преобразований нам осталось только его запустить:

```

KafkaStreams streams = new KafkaStreams(builder.build(), props); ❶

streams.start(); ❷

// Обычно потоковое приложение работает
// постоянно, в данном же примере мы запустим
// его на некоторое время, а затем остановим,
// поскольку входные данные не бесконечны.
Thread.sleep(5000L);

streams.close(); ❸

```

❶ Описываем объект `KafkaStreams` на основе нашей топологии и заданных нами свойств.

❷ Запускаем `Kafka Streams`.

❸ Через некоторое время останавливаем.

Вот и все! Понадобилось всего несколько строк, чтобы реализовать паттерн обработки отдельных событий (выполнили отображение, а затем фильтрацию

событий). Мы заново разделили данные, добавив оператор `group-by`, после чего на основе простого локального состояния подсчитали число записей, в которых каждое уникальное слово является ключом, то есть количество вхождений каждого из слов.

Теперь рекомендуем запустить полный вариант примера. Инструкции по его выполнению найдете в файле `README` репозитория на GitHub (<http://www.bit.ly/2sOXzUN>).

Отметим, что для запуска всего примера не требуется устанавливать ничего, кроме самой Apache Kafka. Если в нашем входном топике несколько разделов, можно путем запуска нескольких экземпляров приложения `WordCount` (подобно запуску приложения в нескольких различных вкладках терминала) создать свой первый кластер обработки Kafka Streams. Экземпляры приложения `WordCount` при этом смогут взаимодействовать друг с другом и согласовывать свою работу. Один из главных порогов вхождения для некоторых фреймворков потоковой обработки — то, что использовать его в локальном режиме очень просто, но для запуска производственного кластера необходимо установить YARN или Mesos, после чего установить фреймворк обработки на всех машинах, а затем разобраться с запуском приложения в кластере. С помощью Kafka Streams API мы можем просто запустить несколько экземпляров приложения — и кластер готов. Одно и то же приложение работает на машине разработчика и при промышленной эксплуатации.

Сводные показатели фондовой биржи

Следующий пример сложнее — мы прочитаем поток событий биржевых операций, включающий символы акций, цену и величину предложения. В биржевых операциях *цена предложения* (*ask price*) — это то, сколько просит за акции продавец, а *цена заявки* (*bid price*) — то, что готов заплатить покупатель. *Величина предложения* (*ask size*) — число акций, которое продавец согласен продать по данной цене. Для упрощения примера мы полностью проигнорируем заявки. Не станем также включать в данные метки даты/времени, вместо этого воспользуемся временем события, передаваемым производителем Kafka.

Затем мы создадим выходные потоки данных, содержащие несколько сводных оконных показателей:

- наилучшую, то есть минимальную, цену предложения для каждого пятисекундного окна;
- число сделок для каждого пятисекундного окна;
- среднюю цену предложения для каждого пятисекундного окна.

Все сводные показатели будут обновляться каждую секунду.

Для простоты предположим, что на бирже торгуется лишь десять символов акций. Настройки очень похожи на те, которые мы ранее использовали в примере с подсчетом слов:

```
Properties props = new Properties();
props.put(StreamsConfig.APPLICATION_ID_CONFIG, "stockstat");
props.put(StreamsConfig.BootstrapServersConfig, Constants.BROKER);
props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
    Serdes.String().getClass().getName());
props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
    TradeSerde.class.getName());
```

Основное отличие — в использовании классов `Serde`. В примере подсчета количества слов как для ключей, так и для значений применялся строковый тип данных, а следовательно, в качестве сериализатора и десериализатора для обоих задействовался метод `Serdes.String()`. В данном же примере ключ — по-прежнему строка, но значение представляет собой объект класса `Trade`, содержащий символ акции, цену и величину предложения. Для сериализации и десериализации данного объекта, а также нескольких других объектов, применяемых в этом небольшом приложении, воспользуемся библиотекой `Gson` от компании `Google`, чтобы генерировать сериализатор и десериализатор `JSON` на основе `Java`-объекта. Затем создадим небольшой адаптер, формирующий из них объект `Serde`. Объект `Serde` создаем следующим образом:

```
static public final class TradeSerde extends WrapperSerde<Trade> {
    public TradeSerde() {
        super(new JsonSerializer<Trade>(),
            new JsonDeserializer<Trade>(Trade.class));
    }
}
```

Ничего особенного, но не забудьте, что для каждого объекта, который вы хотели бы хранить в `Kafka`, — входного, выходного, а в некоторых случаях и объектов для промежуточных результатов — вам понадобится передать объект класса `Serde`. Для упрощения рекомендуем генерировать объекты `Serde` с помощью таких библиотек, как `Gson`, `Avro`, `Protobufs`, или чего-то похожего.

Теперь, когда настройка завершена, можно заняться топологией:

```
KStream<Windowed<String>, TradeStats> stats = source
    .groupByKey() ❶
    .windowedBy(TimeWindows.of(Duration.ofMillis(windowSize))
        .advanceBy(Duration.ofSeconds(1))) ❷
    .aggregate( ❸
        () -> new TradeStats(),
        (k, v, tradestats) -> tradestats.add(v), ❹
        Materialized.<String, TradeStats, WindowStore<Bytes, byte[]>>
```

```

        as("trade-aggregates") ❸
        .withValueSerde(new TradeStatsSerde())) ❹
    .toStream() ❺
    .mapValues((trade) -> trade.computeAvgPrice()); ❻
stats.to("stockstats-output",
    Produced.keySerde(
        WindowedSerdes.timeWindowedSerdeFrom(String.class, windowSize))); ❼

```

❶ Мы начинаем с чтения событий из входного топика и выполнения операции `groupByKey()`. Несмотря на название, она ничего не группирует. Она обеспечивает разделение потока событий на основе ключа записи. А поскольку мы записываем данные в топик с ключами и не меняем последние до вызова `groupByKey()`, то данные остаются разделенными по ключам, так что этот метод в данном случае ничего не делает.

❷ Мы определяем окно — в данном случае окно 5 с, перемещающееся вперед каждую секунду.

❸ Обеспечив правильное секционирование и работу с окнами, секционирование и работу с окнами, приступаем к агрегированию. Выполнение метода `aggregate` приведет к разбивке потока данных на перекрывающиеся окна (пятисекундные окна с обновлением каждую секунду) с последующим применением агрегирующего метода ко всем событиям каждого окна. Первый параметр этого метода представляет собой новый объект, в который будут помещены результаты агрегирования, — в нашем случае объект класса `Tradestats`. Он был создан в качестве вместилища всех интересующих нас сводных показателей по каждому временному окну: минимальной цены, средней цены и числа сделок.

❹ Далее мы указываем метод для фактического агрегирования записей — в данном случае метод `add` объекта `Tradestats` используется для обновления значений минимальной цены, числа сделок и итоговых цен по окну при поступлении новой записи.

❺ Как упоминалось в разделе «Паттерны проектирования потоковой обработки», оконное агрегирование требует хранения состояния и локального хранилища, в котором его можно хранить. Последний параметр метода агрегирования как раз и представляет собой конфигурацию хранилища состояния. `Materialized` — это объект конфигурации магазина, и мы настраиваем название магазина как `trade-aggregates`. Им может быть любое уникальное название.

❻ В рамках конфигурации хранилища состояния мы также предоставляем объект `Serde` для сериализации и десериализации результатов агрегации (объект `Tradestats`).

❼ Результаты агрегирования представляют собой таблицу с символом акции, временным окном в качестве первичного ключа и результатом агрегирования в качестве значения. Мы возвращаем таблицу обратно в поток событий.

❸ Последний шаг — обновление средней цены. В настоящий момент результаты агрегирования включают сумму цен и число сделок. Мы проходим по этим записям и вычисляем на основе существующих сводных показателей среднюю цену, которую затем можно будет включить в выходной поток.

❹ И наконец, записываем результаты в поток `stockstats-output`. Поскольку результаты являются частью оконной операции, мы создаем `WindowedSerde`, который сохраняет результат в оконном формате данных, включающем временную метку окна. Размер окна передается как часть объекта `Serde`, даже если он не используется при сериализации (при десериализации требуется размер окна, поскольку в топике вывода сохраняется только время начала окна).

После определения потока выполнения можно воспользоваться им для генерации и выполнения объекта `KafkaStreams` подобно тому, как мы поступили в разделе «Подсчет количества слов».

Этот пример демонстрирует возможности выполнения операций оконного агрегирования над потоком данных — вероятно, самый часто встречающийся сценарий использования потоковой обработки. Стоит отметить, как просто хранить локальное состояние агрегирования — достаточно объекта `Serde` и названия хранилища состояния. Тем не менее это приложение способно масштабироваться на много экземпляров и автоматически восстанавливаться после сбоя отдельных экземпляров посредством делегирования обработки части разделов одному из продолжающих работать экземпляров. Мы подробнее рассмотрим эту процедуру в разделе «Kafka Streams: обзор архитектуры» далее.

Как обычно, вы можете найти полный пример, включая инструкции по запуску, на GitHub (<http://www.bit.ly/2r6BLm1>).

Обогащение потока событий перехода по ссылкам

Последний пример будет посвящен демонстрации соединений потоков путем обогащения потока событий перехода по ссылкам на веб-сайте. Мы сгенерируем поток имитационных щелчков на ссылках, поток обновлений таблицы базы данных с фиктивными профилями, а также поток операций поиска в Сети. Затем соединим все три потока, чтобы получить полный обзор деятельности всех пользователей. Что искали пользователи? По каким результатам поиска они переходили? Меняли ли они список интересов в своих профилях? Подобные соединения позволяют получить массу информации для анализа. На информации такого рода часто основаны рекомендации товаров: если пользователь искал велосипеды, щелкал на ссылках для слова *Trek*, значит, ему интересны велосипедные путешествия, так что можно рекламировать ему велосипеды *Trek*, шлемы и велотуры в экзотические места, например в штат Небраска.

Поскольку настройка приложения такая же, как в предыдущих примерах, пропустим этот этап и сразу перейдем к топологии соединения нескольких потоков:

```
KStream<Integer, PageView> views =
    builder.stream(Constants.PAGE_VIEW_TOPIC,
        Consumed.with(Serdes.Integer(), new PageViewSerde())); ❶
KStream<Integer, Search> searches =
    builder.stream(Constants.SEARCH_TOPIC,
        Consumed.with(Serdes.Integer(), new SearchSerde()));
KTable<Integer, UserProfile> profiles =
    builder.table(Constants.USER_PROFILE_TOPIC,
        Consumed.with(Serdes.Integer(), new ProfileSerde())); ❷

KStream<Integer, UserActivity> viewsWithProfile = views.leftJoin(profiles, ❸
    (page, profile) -> {
        if (profile != null)
            return new UserActivity(
                profile.getUserID(), profile.getUserName(),
                profile.getZipcode(), profile.getInterests(),
                "", page.getPage()); ❹
        else
            return new UserActivity(
                -1, "", "", null, "", page.getPage());
    });

KStream<Integer, UserActivity> userActivityKStream =
    viewsWithProfile.leftJoin(searches, ❺
    (userActivity, search) -> {
        if (search != null)
            userActivity.updateSearch(search.getSearchTerms()); ❻
        else
            userActivity.updateSearch("");
        return userActivity;
    },

    JoinWindows.of(Duration.ofSeconds(1)).before(Duration.ofSeconds(0)), ❼
    StreamJoined.with(Serdes.Integer(), ❽
        new UserActivitySerde(),
        new SearchSerde()));
```

❶ Прежде всего мы создаем объекты потоков для двух потоков, которые собираемся объединять, — переходов по ссылкам и операций поиска. Когда создаем объект потока, передаем входной топику, а также ключ и значение объекта *Serde*, которые будут использоваться при получении записей из топики и их десериализации, во входные объекты.

❷ Создаем также таблицу типа *KTable* для профилей пользователей. *KTable* представляет собой материализованное хранилище, обновляемое посредством потока изменений.

❸ Далее мы обогащаем поток переходов по ссылкам информацией о профилях пользователей, соединяя поток событий с таблицей профилей. При соединении

потока данных с таблицей каждое событие в потоке получает информацию из закешированной копии таблицы профилей. Мы выполняем левое внешнее соединение, так что в результаты попадут переходы по ссылкам, для которых выполнивший их пользователь неизвестен.

④ Это и есть метод, выполняющий соединение, — он принимает на входе два значения, одно из потока, а второе из записи, и возвращает третье значение. В отличие от баз данных мы должны сами решить, как эти два значения будут объединены в общий результат. В данном случае мы создали один объект `activity`, содержащий как информацию о пользователе, так и просмотренную им страницу.

⑤ Далее необходимо объединить информацию о переходах по ссылкам с информацией о выполненных соответствующим пользователем операциях поиска. Соединение остается левым, но теперь соединяются два потока, а не поток с таблицей.

⑥ Это и есть метод, выполняющий соединение, — мы просто добавляем ключевые слова поиска ко всем соответствующим просмотрам страниц.

⑦ А вот это самое интересное — *соединение потока с потоком* представляет собой соединение с временным окном. Соединение всех переходов по ссылкам с информацией об операциях поиска особого смысла не имеет — необходимо соединить каждую операцию поиска с соответствующими переходами по ссылкам, то есть щелчками на ссылках, выполненными в течение короткого промежутка времени после поиска. Так что мы зададим размер окна соединения, равный 1 с. Мы вызываем `of`, чтобы создать окно с интервалом 1 с до и после каждого поиска, а затем вызываем `before` с интервалом 0 с, чтобы убедиться, что мы соединяем только те щелчки, которые происходят через секунду после каждого поиска, а не до него. Результаты будут включать соответствующие щелчки, поисковые запросы и профиль пользователя. Благодаря этому появится возможность провести полный анализ операций поиска и их результатов.

⑧ Здесь мы определяем `Serde` результата объединения. Это включает в себя `Serde` для ключа, который является общим для обеих сторон соединения, и `Serde` для обоих значений, которые будут включены в результат соединения. В данном случае ключом является идентификатор пользователя, поэтому мы применим простой `Integer Serde`.

После завершения описания последовательности операций можно воспользоваться ею для генерирования и выполнения объекта `KafkaStreams`, подобно тому как мы поступили в разделе «Подсчет количества слов».

Этот пример демонстрирует, что в потоковой обработке возможны два различных паттерна соединений. Один относится к соединению потока с таблицей для обогащения всех событий потока информацией из таблицы. Он напоминает соединение таблицы фактов с измерением при выполнении запросов к базе данных.

Второй паттерн относится к соединению двух потоков на основе временного окна. Эта операция встречается только в сфере потоковой обработки.

Как обычно, вы найдете полный пример, включая инструкции по запуску, на GitHub (<http://www.bit.ly/2sq096i>).

Kafka Streams: обзор архитектуры

Примеры из предыдущего раздела демонстрируют использование API Kafka Streams для реализации нескольких широко известных паттернов проектирования потоковой обработки. Но, чтобы лучше понять, как библиотека Kafka Streams на самом деле работает и масштабируется, необходимо «заглянуть под капот» и разобраться с некоторыми базовыми принципами архитектуры этого API.

Построение топологии

Любое потоковое приложение реализует и выполняет одну *топологию*. Топология, называемая в других фреймворках потоковой обработки также DAG (directed acyclic graph — ориентированный ациклический граф), представляет собой набор операций и преобразований, через которые проходят все события на пути от входных данных до результатов. На рис. 14.10 показана топология для примера с подсчетом количества слов.

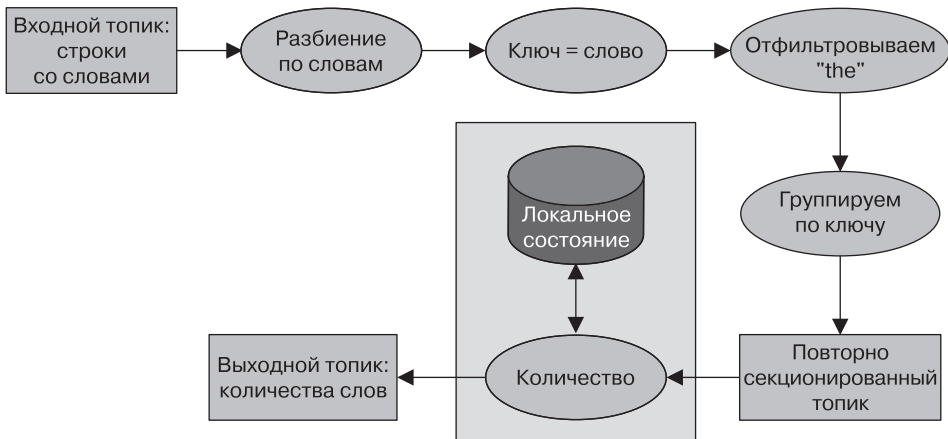


Рис. 14.10. Топология для примера подсчета числа слов с помощью потоковой обработки

Даже у простых приложений топология нетривиальна. Она состоит из узлов обработки — узлов графа топологии (на схеме они представлены овалами). Большинство узлов обработки реализуют операции над данными — фильтрацию,

отображение, агрегирование и т. п. Существуют также узлы обработки — источники, потребляющие данные из топиков и передающие их дальше, а также узлы обработки — приемники, получающие данные из предыдущих узлов обработки и генерирующие их в топик. Топология всегда начинается с одного или нескольких узлов обработки — производителей и заканчивается одним или несколькими узлами обработки — приемниками.

Оптимизация топологии

По умолчанию Kafka Streams выполняет приложения, созданные с помощью DSL API, отображая каждый метод DSL на эквивалент более низкого уровня. Если оценивать каждый метод DSL независимо, будут упущены возможности оптимизации общей результирующей топологии.

Однако обратите внимание на то, что выполнение приложения Kafka Streams представляет собой трехэтапный процесс.

1. Логическая топология определяется путем создания объектов `KStream` и `KTable` и выполнения над ними операций DSL, таких как `filter` и `join`.
2. Функция `StreamsBuilder.build()` создает физическую топологию на основе логической топологии.
3. Функция `KafkaStreams.start()` выполняет топологию — здесь происходят потребление, обработка и производство данных.

На втором этапе, когда физическая топология создается на основе логических определений, можно применить общую оптимизацию плана.

В настоящее время Apache Kafka содержит лишь несколько оптимизаций, в основном связанных с повторным использованием топиков, где это возможно. Их можно включить, установив параметр `StreamsConfig.TOPOLOGY_OPTIMIZATION` в значение `StreamsConfig.OPTIMIZE` и вызвав `build(props)`. Если вы вызываете только `build()` без передачи конфигурации, оптимизация все равно будет отключена. Рекомендуется тестировать приложения с оптимизацией и без нее, сравнивать время выполнения и объемы данных, записываемых в Kafka, и, конечно, проверять идентичность результатов в различных известных сценариях.

Тестирование топологии

По большому счету, мы хотим протестировать программное обеспечение, прежде чем использовать его в сценариях, где важно его успешное выполнение. Автоматизированное тестирование считается золотым стандартом. Повторяющиеся тесты, которые оцениваются каждый раз при внесении изменений в программное

приложение или библиотеку, позволяют быстро выполнять итерации и облегчают поиск и устранение неисправностей.

Мы хотим применить такую же методологию к нашим приложениям Kafka Streams. В дополнение к автоматизированным сквозным тестам, которые запускают приложение потоковой обработки в тестовой среде со сгенерированными данными, мы хотим включить более быстрые, легкие и простые в отладке модульные и интеграционные тесты.

Основным инструментом тестирования приложений Kafka Streams является `TopologyTestDriver`. С момента появления в версии 1.1.0 его API значительно улучшен, а версии, начиная с 2.4, удобны и просты в использовании. Эти тесты выглядят как обычные модульные тесты. Мы определяем входные данные, производим их для имитации входных топиков, запускаем топологию с помощью тестового драйвера, считываем результаты из макетных выходных топиков и проверяем результат, сравнивая его с ожидаемыми значениями.

Мы рекомендуем использовать `TopologyTestDriver` для тестирования приложений обработки потоков, но, поскольку он не имитирует режим кэширования Kafka Streams (оптимизация, не обсуждаемая в этой книге, совершенно не связанная с самим хранилищем состояний, которое моделируется этим фреймворком), существуют целые классы ошибок, которые он не обнаружит.

Модульные тесты обычно дополняются интеграционными тестами, и для Kafka Streams существует два популярных фреймворка интеграционных тестов: `EmbeddedKafkaCluster` и `Testcontainers`. Первый запускает брокеры Kafka внутри JVM, на которой выполняются тесты, а второй — контейнеры Docker с брокерами Kafka и многими другими компонентами, необходимыми для тестов. Рекомендуется использовать `Testcontainers`, поскольку применение Docker позволяет полностью изолировать Kafka, его зависимости и использование ресурсов от приложения, которое мы пытаемся протестировать.

Это всего лишь краткий обзор методологии тестирования потоков Kafka. Рекомендуем прочитать статью в блоге «Тестирование потоков Kafka — глубокое погружение» (<https://oreil.ly/RvTIA>), где имеется более детальное объяснение и приводятся подробные примеры кода топологий и тестов.

Масштабирование топологии

Kafka Streams масштабируется за счет того, что внутри одного экземпляра приложения могут работать несколько потоков выполнения, а также благодаря балансировке нагрузки между распределенными экземплярами приложения. Можно запустить приложение Kafka Streams на одной машине в многопоточном режиме или на нескольких машинах — в любом случае обработкой данных будут заниматься все активные потоки выполнения приложения.

Движок Streams распараллеливает выполнение топологии, разбивая ее на задачи. Число задач определяется движком Streams и зависит от числа разделов в обрабатываемых приложениях топиках. Каждая задача отвечает за какое-то подмножество разделов: она подписывается на эти разделы и читает из них события. Для каждого прочитанного события задача выполняет по порядку все подходящие для этого раздела шаги обработки, после чего записывает результаты в приемник. Эти задачи — базовая единица параллелизма в Kafka Streams, поскольку любую задачу можно выполнять независимо от остальных (рис. 14.11).

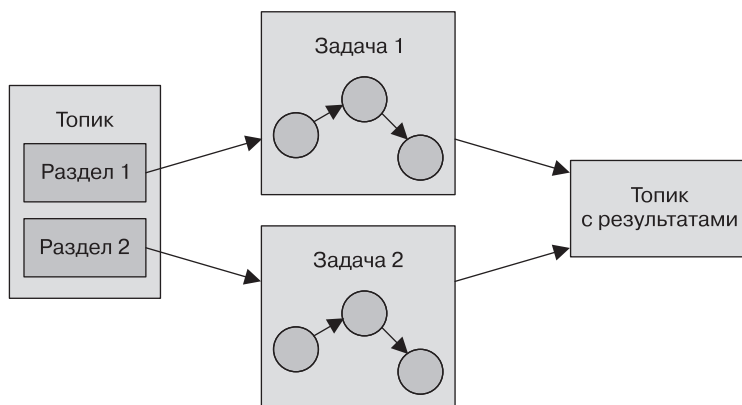


Рис. 14.11. Две задачи, реализующие одну топологию, — по одной для каждого раздела входного топика

У разработчика приложения есть возможность выбрать число потоков выполнения для каждого экземпляра приложения. При доступности нескольких потоков выполнения каждый из них будет выполнять часть создаваемых приложением задач. Если несколько экземпляров приложения работают на нескольких серверах, то в каждом потоке на каждом сервере будут выполняться различные задачи. Именно таким образом масштабируются потоковые приложения: задач будет столько, сколько имеется разделов в обрабатываемых топиках. Если нужно повысить скорость обработки, увеличьте число потоков выполнения. Если на сервере заканчиваются ресурсы, запустите еще один экземпляр приложения на другом сервере. Kafka автоматически координирует работу — каждой задаче будет назначаться свое подмножество разделов, события из которых она будет обрабатывать независимо от других задач, поддерживая собственное локальное состояние с соответствующими сводными показателями, если этого требует топология (рис. 14.12).

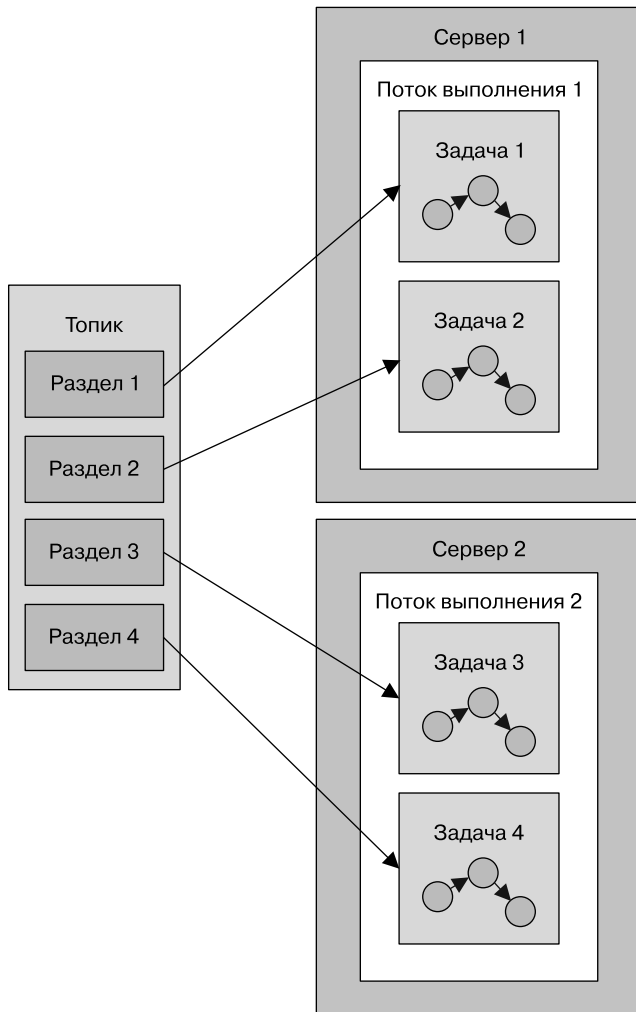


Рис. 14.12. Задачи потоковой обработки могут выполняться в нескольких потоках и на нескольких серверах

Иногда для шага обработки требуются входные данные из нескольких разделов, вследствие чего между задачами могут возникать зависимости. Например, при соединении двух потоков данных (как в примере из подраздела «Обогащение потока событий перехода по ссылкам» ранее в этой главе) для получения результата понадобятся данные из раздела каждого из потоков. Фреймворк Kafka Streams решает эту проблему за счет назначения всех необходимых для одного соединения разделов одной задаче, так что задачи могут читать данные из всех

нужных разделов и выполнять соединение независимо друг от друга. Именно поэтому для Kafka Streams требуется, чтобы во всех участвующих в операции соединения топиках было одинаковое число разделов и чтобы они были секционированы по ключу соединения.

Еще один пример возникновения зависимостей между задачами — случай, когда для приложения требуется повторное секционирование. Так, в примере с потоком событий переходов ключ всех событий — идентификатор пользователя. Но что, если нам понадобится сгенерировать сводные показатели по страницам? Или по почтовому индексу? В подобном случае потоки Kafka повторно разделяют данные по почтовому индексу и выполняют их агрегирование на основе новых разделов. Если задача 1 обрабатывает данные из раздела 1 и доходит до узла обработки, который секционирует данные повторно (операция `groupBy`), понадобится *перетасовать* (`shuffle`) или отправить события другим задачам для обработки. В отличие от других фреймворков потоковой обработки Kafka Streams выполняет повторное разделение путем записи событий в новый топик с новыми ключами и разделами. Далее иной набор задач читает эти события из нового топика и продолжает обработку. Шаг повторного разделения разбивает топологию на две субтопологии, каждая со своими задачами. Второй набор задач зависит от первого, поскольку обрабатывает результаты первой субтопологии. Однако первый и второй наборы задач все же можно запускать независимо друг от друга и параллельно, поскольку первый записывает данные в топик с одной скоростью, а второй читает и обрабатывает данные оттуда — с другой. Между их задачами нет никакого взаимодействия и разделения ресурсов, и они не обязаны работать в одних потоках выполнения на серверах. Это одна из самых полезных черт Kafka — снижение количества зависимостей между различными частями конвейера (рис. 14.13).

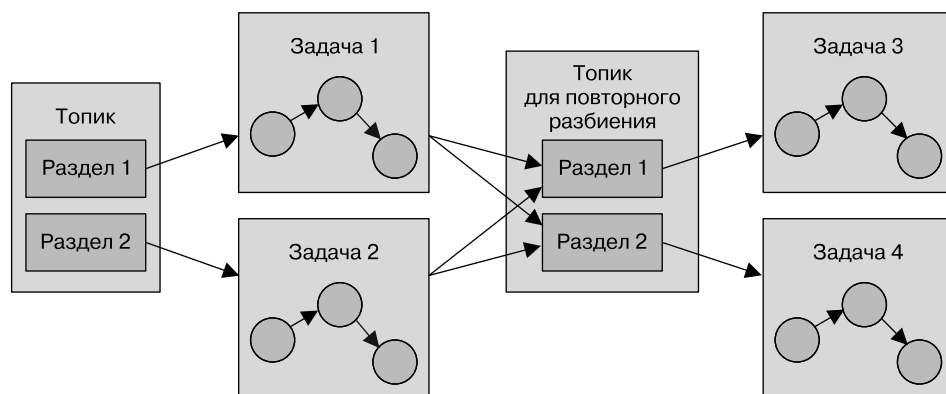


Рис. 14.13. Два набора задач, обрабатывающих события, с топиком для повторного разбиения на разделы событий между ними

Как пережить отказ

Та же модель, которая позволяет масштабировать приложение, дает возможность изящно справляться с отказами. Получить доступ к Kafka легко, следовательно, сохраняемые в ней данные также высокодоступны. Так что приложение в случае сбоя и необходимости перезапуска может узнать из Kafka свою последнюю позицию в потоке и продолжить обработку с последнего зафиксированного ею перед сбоем смещения. Отметим, что в случае утраты хранилища локального состояния (например, при необходимости замены сервера, на котором оно находилось) потоковое приложение всегда может создать его заново на основе хранящегося в Kafka журнала изменений.

Фреймворк Kafka Streams также использует предоставляемую Kafka координацию потребителей с целью обеспечения высокой доступности для задач. Если задача завершилась неудачей, но есть другие активные потоки или экземпляры потокового приложения, ее можно перезапустить в одном из доступных потоков. Это напоминает то, как группа потребителей справляется с отказом одного из потребителей группы посредством переназначения его разделов одному из оставшихся потребителей. Kafka Streams выиграла от улучшений в протоколе координации групп потребителей Kafka, таких как статическое членство в группах и совместная перебалансировка (описаны в главе 4), а также из улучшений семантики Kafka «только один раз» (рассмотрена в главе 8).

Хотя описанные здесь методы обеспечения высокой доступности хорошо работают в теории, на практике возникают некоторые сложности. Одной из важных проблем является скорость восстановления. Когда поток должен начать обработку задачи, которая раньше выполнялась в отказавшем потоке, ему сначала нужно восстановить свое сохраненное состояние — например, текущие окна агрегации. Часто это делается перечитыванием внутренних топиков из Kafka, чтобы разогреть хранилища состояния Kafka Streams. В течение времени, необходимого для восстановления состояния отказавшей задачи, задание обработки потока не будет работать с этим подмножеством своих данных, что приведет к снижению доступности и появлению устаревших данных.

Таким образом, сокращение времени восстановления часто сводится к уменьшению времени, необходимого для восстановления состояния. Ключевой прием заключается в том, чтобы убедиться, что все топики Kafka Streams настроены на агрессивное сжатие: установив низкое значение `min.compaction.lag.ms` и настроив размер сегмента на 100 Мбайт вместо 1 Гбайт по умолчанию (напомним, что последний сегмент в каждом разделе — активный — не сжимается).

Для еще более быстрого восстановления мы рекомендуем настроить резервные реплики — задачи, которые просто являются тенью активных задач в приложении обработки потоков и сохраняют текущее состояние на другом сервере.

Когда происходит восстановление после отказа, они уже имеют самое актуальное состояние и готовы продолжить обработку практически без простоя.

Более подробную информацию как о масштабируемости, так и о высокой доступности потоков Kafka можно найти в статье в блоге (<https://oreil.ly/mj9Ca>) и в докладе на саммите Kafka по этой теме (<https://oreil.ly/cUvKa>).

Сценарии использования потоковой обработки

На протяжении главы мы изучали потоковую обработку — от основных понятий и паттернов до конкретных примеров применения Kafka Streams. На данном этапе имеет смысл взглянуть на часто встречающиеся сценарии использования потоковой обработки. Как объяснялось в начале главы, потоковая, она же непрерывная, обработка полезна в тех случаях, когда нужно обрабатывать события одно за другим, а не ждать часами следующего пакета, но когда ответ также не ожидается в течение миллисекунд. Все это справедливо, однако слишком расплывчато. Рассмотрим несколько настоящих задач, решаемых с помощью потоковой обработки.

- *Обслуживание клиентов.* Допустим, мы только что забронировали номер в большой сети отелей и ожидаем получения подтверждения по электронной почте и платежной квитанции. Через несколько минут после бронирования, когда подтверждение все еще не прибыло, мы звоним в отдел по обслуживанию клиентов для подтверждения брони. Представьте, что менеджер по обслуживанию клиентов говорит вам: «Я не вижу заказа в системе, но пакетное задание, загружающее данные из системы бронирования в систему отеля и систему обслуживания клиентов, выполняется только раз в сутки, так что перезвоните завтра, пожалуйста. Подтверждение по электронной почте придет вам в течение 2–3 рабочих дней». Создается впечатление, что сервис здесь не слишком хорош, но у нас не раз случались такие разговоры с отелями из крупных сетей. Желательно, чтобы обновленные данные о бронировании в течение нескольких секунд или минут после него получала каждая система в сети отелей, в том числе отдел по обслуживанию клиентов, сам отель, система отправки подтверждений по электронной почте, сайт и т. д. Желательно также, чтобы отдел по обслуживанию клиентов мог сразу же извлечь из системы все подробности ваших предыдущих визитов в любой из отелей сети, а на стойке администратора в отеле знали, что вы постоянный клиент, и сделали на этом основании вам скидку. Создание всех этих систем на основе приложений потоковой обработки позволяет организовать получение и обработку обновлений в режиме псевдореального времени, благодаря чему ощущения клиентов от

сервиса значительно улучшатся. С подобной системой клиент получал бы подтверждение по электронной почте в течение нескольких минут, деньги с кредитной карты снимали бы вовремя, платежную квитанцию присылали своевременно, а служба поддержки могла бы немедленно ответить на любые вопросы относительно брони.

- *Интернет вещей.* Интернет вещей может означать самые разные сущности, начиная с домашних устройств для автоматической регулировки температуры или автоматического пополнения запасов стирального порошка до контроля качества фармацевтической продукции в режиме реального времени. Очень распространенный сценарий — применение потоковой обработки к датчикам и устройствам для прогнозирования необходимости профилактического обслуживания. Это чем-то напоминает мониторинг приложений, только по отношению к аппаратному обеспечению, и встречается во множестве отраслей промышленности, включая фабричное производство, телекоммуникации (определение сбойных сотовых вышек), кабельное ТВ (выявление сбойных тюнеров до того, как клиент начнет жаловаться) и многое другое. У каждого сценария — свой паттерн, но цель одна — обработка в больших объемах поступающих от устройств событий, оповещающих о том, что устройства требуют техобслуживания. Примерами могут быть отброшенные пакеты в случае сетевого коммутатора, большие усилия, необходимые для закручивания винтов при производстве, и т. д.
- *Обнаружение мошенничества.* Известно также как *обнаружение аномалий*, представляет собой очень широкую область, связанную с поимкой мошенников/нечестных людей в системе. Примерами могут служить приложения для обнаружения мошенничества с кредитными картами, на фондовом рынке, жульничества в видеоиграх, а также система кибербезопасности. Во всех этих сферах чем раньше будет пойман мошенник, тем лучше, так что работающая в режиме реального времени система, способная быстро реагировать на события, например запретить выполнение подозрительной транзакции еще до ее одобрения, гораздо предпочтительнее пакетного задания, которое обнаружит мошенничество через три дня после события, когда все исправить будет гораздо сложнее. Опять же речь идет о задаче распознавания паттернов в крупномасштабном потоке событий.
- В сфере кибербезопасности существует метод, называющийся *сигнализацией* (beaconing). Вредоносное программное обеспечение, помещенное хакером в сеть организации, будет периодически обращаться наружу для получения команд. Обнаружить эти операции непросто, поскольку они могут производиться в любое время и с любой частотой. Обычно сети хорошо защищены от внешних атак, но более уязвимы к троянским коням внутри организации, отправляющим данные наружу. Благодаря обработке большого потока

событий сетевых подключений и распознавания аномальности паттерна обмена сообщениями (например, обнаружения того, что для конкретной машины является нетипичным обращение к конкретным IP-адресам) можно организовать раннее оповещение отдела безопасности — до того, как будет нанесен существенный ущерб.

Как выбрать фреймворк потоковой обработки

При выборе фреймворка потоковой обработки важно учесть тип будущего приложения. Для различных типов приложений подходят различные типы потоковой обработки.

- *Система ввода и обработки данных.* Подходит для случая, когда целью является ввод данных из одной системы в другую с внесением некоторых изменений в данные, чтобы они подходили для целевой системы.
- *Система, реагирующая в течение нескольких миллисекунд.* Любые приложения, требующие практически мгновенного ответа. В эту категорию попадают и некоторые сценарии обнаружения мошенничества.
- *Асинхронные микросервисы.* Большие бизнес-процессы делегируют подобным микросервисам выполнение простых действий, например обновление информации о наличии товара в магазине. Таким приложениям может потребоваться поддерживать локальное состояние, кэшируя события для повышения производительности.
- *Анализ данных в режиме псевдореального времени.* Подобные потоковые приложения выполняют сложные группировку и соединение, чтобы сформировать продольные и поперечные срезы данных, позволяющие почерпнуть из них полезную для бизнеса информацию.

Выбор системы потоковой обработки в значительной степени зависит от решаемой задачи.

- Если вы пытаетесь решить задачу ввода и обработки данных, стоит подумать еще раз, нужна ли вам система потоковой обработки или подойдет более примитивная система, ориентированная именно на ввод и обработку данных, например Kafka Connect. Если вы уверены, что нужна именно система потоковой обработки, убедитесь, что в избранной системе есть хороший выбор коннекторов, в том числе качественные коннекторы для нужных вам систем.
- Если вы пытаетесь решить задачу, требующую реакции в течение нескольких миллисекунд, также нужно еще раз подумать, прежде чем выбрать систему потоковой обработки. Для этой задачи лучше подходят паттерны типа «за-

прос — ответ». Если вы уверены, что хотите использовать систему потоковой обработки, выбирайте такую, которая поддерживает модель обработки по отдельным событиям с низким значением задержки, а не ориентированную на микропакетную обработку.

- При создании асинхронных микросервисов понадобится система потоковой обработки, хорошо интегрируемая с выбранной вами шиной сообщений (надеюсь, что ею будет Kafka), обладающая возможностями захвата изменений для удобной передачи произошедших далее по конвейеру изменений в локальное состояние микросервиса, а также обеспечивающая поддержку локального хранилища, которое могло бы выступать в роли кэша или материализованного представления данных микросервиса.
- При создании сложной системы для аналитики вам также понадобится система потоковой обработки с хорошей поддержкой локального хранилища — на этот раз не для хранения локальных кэшей и материализованных представлений, а для выполнения продвинутых операций агрегирования, оконных операций и соединений, реализовать которые без этого непросто. API такой системы должны включать поддержку пользовательских операций агрегирования, оконных операций и разнообразных типов соединений.

Помимо мыслей по поводу сценариев применения, следует учитывать несколько общих соображений.

- *Удобство эксплуатации системы.* Легко ли разворачивать систему для промышленной эксплуатации? Легко ли выполнять мониторинг и искать причины проблем? Хорошо ли она масштабируется в обе стороны при необходимости? Хорошо ли она интегрируется с уже имеющейся у вас инфраструктурой? Что делать в случае ошибки и необходимости повторной обработки данных?
- *Простота использования API и легкость отладки.* Я сталкивался с ситуациями, когда написание хорошего приложения для другой версии того же фреймворка занимает на порядок больше времени. Время разработки и время вывода на рынок — важные факторы, так что выбирайте систему в расчете на максимальную эффективность своей работы.
- *Облегчение жизни.* Почти все подобные системы декларируют возможность выполнения продвинутых операций оконного агрегирования и поддержания локальных хранилищ, но вопрос вот в чем: упрощают ли они жизнь вам? Берут ли они на себя неприятные нюансы, связанные с масштабированием и восстановлением, или дают «дырявые» абстракции, предоставляя вам разгребать все проблемы? Чем больше чистых API и абстракций обеспечивает система и чем больше неприятных нюансов она берет на себя, тем выше производительность разработчиков.

- *Помощь сообщества разработчиков.* Многие из рассматриваемых вами потоковых приложений — приложения с открытым исходным кодом, так что активное сообщество разработчиков очень важно. Хорошее сообщество разработчиков означает регулярное получение новых прекрасных возможностей, довольно хорошее качество приложения (никто не захочет работать с плохим ПО), быстрое исправление программных ошибок и своевременные ответы на вопросы пользователей. Это значит также, что в случае какой-нибудь загадочной ошибки вы сможете найти информацию о ней, просто поискав в Интернете, так как пользователей системы много и все они сталкиваются с похожими проблемами.

Резюме

Мы начали эту главу с объяснения того, что такое потоковая обработка. Дали формальное ее определение и обсудили характерные черты парадигмы потоковой обработки. Мы также сравнили ее с другими парадигмами программирования.

Далее поговорили о важнейших понятиях потоковой обработки и проиллюстрировали их тремя примерами, написанными с применением библиотеки *Kafka Streams*.

После обсуждения всех нюансов примеров мы привели обзор архитектуры фреймворка *Kafka Streams* и рассказали о деталях его внутреннего устройства. В завершение главы и книги в целом привели примеры сценариев использования потоковой обработки и дали несколько советов по сравнению различных фреймворков потоковой обработки.

Установка Kafka в других операционных системах

Платформа Apache Kafka по большей части Java-приложение, а значит, может работать на любой операционной системе, где только можно установить JRE. Однако она была оптимизирована для работы в Linux-подобных операционных системах, так что наилучшую производительность демонстрирует именно там. Поэтому при использовании Kafka для разработки или тестирования в операционных системах для настольных компьютеров имеет смысл запускать ее на виртуальной машине, соответствующей среде будущей промышленной эксплуатации.

Установка в Windows

В Windows 10 можно запускать Kafka двумя способами. Традиционный способ — естественная Java-установка. Пользователи Windows 10 могут воспользоваться также Windows Subsystem для Linux. Настоятельно рекомендуется применять именно последний метод, поскольку в этом случае установка намного проще и среда ближе к типичной среде промышленной эксплуатации. Поэтому сначала мы рассмотрим именно его.

Использование Windows Subsystem для Linux

Работая под Windows 10, можно установить естественную поддержку операционной системы Ubuntu в Windows, воспользовавшись Windows Subsystem для Linux (WSL). На момент издания данной книги компания Microsoft все еще рассматривала WSL как экспериментальную возможность. Хотя WSL и работает аналогично виртуальной машине, ресурсов для нее необходимо меньше, а интеграция с операционной системой Windows у нее глубже.

Для установки WSL необходимо следовать указаниям, которые можно найти в размещенной в Сети документации для разработчиков Microsoft (MSDN) на странице «Что такое подсистема Windows для Linux?» (<https://oreil.ly/dULqm>). После этого необходимо установить JDK с помощью утилиты `apt` (при условии, что вы установили системный пакет Ubuntu для WSL):

```
$ sudo apt install openjdk-16-jre-headless
[sudo] password for username:
Reading package lists... Done
Building dependency tree
Reading state information... Done
[...]
done.
$
```

После установки JDK можно перейти к установке Kafka в соответствии с указаниями, приведенными в главе 2.

Использование Java естественным образом

В более старых версиях Windows или если вам не хочется использовать среду WSL, можно запустить Kafka естественным образом с помощью среды Java для Windows. Однако будьте осторожны, это может привести к возникновению ошибок, специфических для среды Windows. Подобные ошибки могут остаться незамеченными сообществом разработчиков Apache Kafka, в отличие от аналогичных проблем на Linux.

Перед установкой ZooKeeper и Kafka необходимо настроить среду Java. Установите последнюю версию Oracle Java 16, которую можно найти на странице загрузки Oracle Java SE (<https://jdk.java.net>). Скачайте полный установочный пакет JDK, чтобы у вас были все утилиты Java, и следуйте указаниям по установке.



Осторожнее с путями

Мы настоятельно рекомендуем не использовать при установке Java и Kafka пути, содержащие пробелы. Windows разрешает применять в путях пробелы, однако предназначенные для Unix приложения настраиваются иначе, так что задание путей может вызвать проблемы. Учитывайте это при задании пути установки Java. Например, разумно будет при установке JDK 16.0.1 выбрать путь `C:\Java\jdk-16.0.1`.

После установки Java необходимо настроить переменные среды. Это можно сделать на панели управления Windows, хотя точное место зависит от используемой вами версии Windows. В Windows 10 нужно сделать следующее.

1. Выбрать Система и безопасность (System and Security).
2. Выбрать Система (System).
3. Выбрать Дополнительные параметры системы (Advanced system settings), в результате чего откроется окно Свойства системы (System Properties).
4. На вкладке Дополнительно (Advanced) нажать кнопку Переменные среды (Environment Variables).

В этом разделе вы сможете добавить новую пользовательскую переменную `JAVA_HOME` (рис. А.1) и задать ее значение в соответствии с путем, по которому установили Java. Затем поменяйте значение системной переменной `Path`, добавив туда новый элемент `%JAVA_HOME%\bin`. Сохраните эти настройки и выйдите из панели управления.

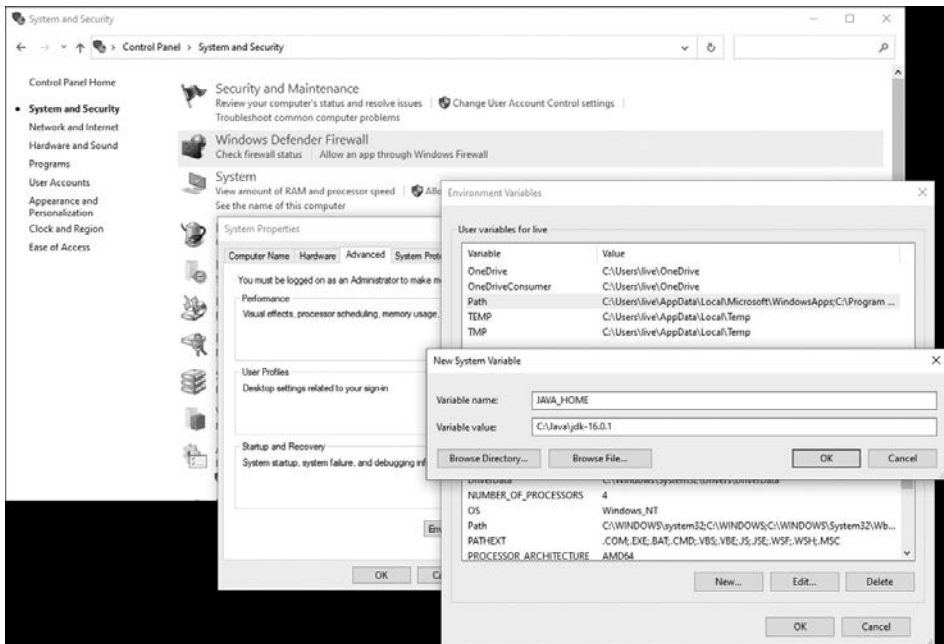


Рис. А.1. Добавление переменной `JAVA_HOME`

Теперь можно приступить к установке Apache Kafka. Установочный пакет включает ZooKeeper, так что устанавливать его отдельно не нужно. Текущую версию Kafka (<https://oreil.ly/xpwy1>) можно скачать в Интернете. На момент выхода данной книги актуальной является версия 2.8.0, работающая со Scala 2.13.0. Файл, который вы скачаете, представляет собой gzip-архив, заархивированный и упакованный с помощью утилиты `tar`, так что для его распаковки вам понадобится

Windows-приложение, например 8 Zip. Как и при установке в операционной системе Linux, у вас будет возможность выбрать, куда выполнять разархивирование. В данном примере мы будем считать, что установочный пакет Kafka разархивирован в каталог `C:\kafka_2.13-2.8.0`.

Запуск ZooKeeper и Kafka под операционной системой Windows несколько отличается от запуска под Linux, поскольку приходится использовать предназначенные для Windows пакетные файлы, а не сценарии командной оболочки, как в случае других платформ. Эти пакетные файлы также не поддерживают работу приложения в фоновом режиме, так что вам понадобится отдельная командная оболочка для каждого приложения. Сначала запустим ZooKeeper:

```
PS C:\> cd kafka_2.13-2.8.0
PS C:\kafka_2.13-2.8.0> bin\windows\zookeeper-server-start.bat C:\
kafka_2.13-2.8.0\config\zookeeper.properties
[2021-07-18 17:37:12,917] INFO Reading configuration from: C:
kafka_2.13-2.8.0\config\zookeeper.properties (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
[...]
[2021-07-18 17:37:13,135] INFO PrepRequestProcessor (sid:0) started, reconfigEnabled=false (org.apache.zookeeper.server.PrepareRequestProcessor)
[2021-07-18 17:37:13,144] INFO Using checkIntervalMs=60000 maxPerMinute=10000 (org.apache.zookeeper.server.ContainerManager)
```

После успешного запуска ZooKeeper можете открыть еще одно окно для запуска Kafka:

```
PS C:\> cd kafka_2.13-2.8.0
PS C:\kafka_2.13-2.8.0> .\bin\windows\kafka-server-start.bat C:
kafka_2.13-2.8.0\config\server.properties
[2021-07-18 17:39:46,098] INFO Registered kafka:type=kafka.Log4jController
MBean (kafka.utils.Log4jControllerRegistration$)
[...]
[2021-07-18 17:39:47,918] INFO [KafkaServer id=0] started (kafka.server.KafkaServer)
[2021-07-18 17:39:48,009] INFO [broker-0-to-controller-send-thread]: Recorded new controller, from now on will use broker 192.168.0.2:9092 (id: 0 rack: null) (kafka.server.BrokerToControllerRequestThread)
```

Установка в macOS

macOS основан на Darwin — Unix-подобной операционной системе, ведущей свое происхождение в том числе и от FreeBSD. Это значит, что многого из того, что мы можем ожидать от нее, можно ожидать и от любой Unix-подобной операционной системы, так что установка в нее разработанных для Unix приложений, например Apache Kafka, не составляет труда. Можно или выбрать

простейший вариант — воспользоваться системой управления пакетами, например Homebrew, или установить Java и Kafka вручную ради расширенного контроля версий.

Использование Homebrew

Если у вас уже установлена Homebrew (<https://brew.sh/>) для macOS, можете воспользоваться ею для установки Kafka за один шаг. При этом сначала будет установлена Java, а потом — Apache Kafka 02.8.0 (версия по состоянию на момент написания данной книги).

Если вы еще не установили Homebrew, сделайте это, следуя указаниям, приведенным на странице описания установки (<https://oreil.ly/ZVEvc>). Затем можно будет установить саму Kafka. Система управления пакетами Homebrew гарантирует, что сначала будут установлены все зависимости, включая Java:

```
$ brew install kafka
==> Installing dependencies for kafka: openjdk, openssl@1.1 and zookeeper
==> Installing kafka dependency: openjdk
==> Pouring openjdk--16.0.1.big_sur.bottle.tar.gz
[...]
==> Summary
/usr/local/Cellar/kafka/2.8.0: 200 files, 68.2MB
$
```

Homebrew затем установит Kafka в каталог `/usr/local/Cellar`, но файлы будут привязаны к другим каталогам:

- исполняемые файлы и сценарии будут находиться в каталоге `/usr/local/bin`;
- настройки Kafka — в `/usr/local/etc/kafka`;
- настройки ZooKeeper — в `/usr/local/etc/zookeeper`;
- параметр конфигурации `log.dirs` (расположение данных Kafka) будет установлен в значение `/usr/local/var/lib/kafka-logs`.

После завершения установки можно запустить ZooKeeper и Kafka (в данном примере Kafka запускается в фоновом режиме):

```
$ /usr/local/bin/zkServer start
ZooKeeper JMX enabled by default
Using config: /usr/local/etc/zookeeper/zoo.cfg
Starting zookeeper ... STARTED
$ /usr/local/bin/kafka-server-start /usr/local/etc/kafka/server.properties
[2021-07-18 17:52:15,688] INFO Registered kafka:type=kafka.Log4jController
MBean (kafka.utils.Log4jControllerRegistration$)
[...]
```

```
[2021-07-18 17:52:18,187] INFO [KafkaServer id=0] started (kafka.server.Kafka
Server)
[2021-07-18 17:52:18,232] INFO [broker-0-to-controller-send-thread]: Recorded
new controller, from now on will use broker 192.168.0.2:9092 (id: 0 rack: null)
(kafka.server.BrokerToControllerRequestThread)
```

Установка вручную

Аналогично установке вручную в операционной системе Windows при установке Kafka в macOS необходимо сначала установить JDK. Для получения нужной версии для macOS можно воспользоваться той же страницей загрузки Oracle Java SE (<https://jdk.java.net>), а затем установить Apache Kafka, опять же аналогично Windows. В данном примере будем считать, что скачанный установочный пакет Kafka разархивируется в каталог `/usr/local/kafka_2.13-2.8.0`.

Запуск ZooKeeper и Kafka не отличается от их запуска в Linux, хотя сначала нужно убедиться, что задано значение переменной `JAVA_HOME`:

```
$ export JAVA_HOME=`/usr/libexec/java_home -v 16.0.1`
$ echo $JAVA_HOME
/Library/Java/JavaVirtualMachines/jdk-16.0.1.jdk/Contents/Home

$ /usr/local/kafka_2.13-2.8.0/bin/zookeeper-server-start.sh -daemon /usr/local/
kafka_2.13-2.8.0/config/zookeeper.properties
$ /usr/local/kafka_2.13-2.8.0/bin/kafka-server-start.sh /usr/local/
kafka_2.13-2.8.0/config/server.properties
[2021-07-18 18:02:34,724] INFO Registered kafka:type=kafka.Log4jController
MBean (kafka.utils.Log4jControllerRegistration$)
[...]
[2021-07-18 18:02:36,873] INFO [KafkaServer id=0] started (kafka.server.Kafka
Server)
[2021-07-18 18:02:36,915] INFO [broker-0-to-controller-send-thread]: Recorded
new controller, from now on will use broker 192.168.0.2:9092 (id: 0 rack: null)
(kafka.server.BrokerToControllerRequestThread)((("macOS, installing Kafka on",
startref="ix_macOS")))((("operating systems", "other than Linux, installing
Kafka on", startref="ix_OSinstall")))
```


Дополнительные инструменты Kafka

Сообщество Apache Kafka создало надежную экосистему инструментов и платформ, которые значительно облегчают задачи запуска и использования Kafka. Хотя этот список ни в коем случае не является исчерпывающим, здесь представлены несколько наиболее популярных инструментов, чтобы помочь вам начать работу.



Предостережение

Хотя мы связаны с некоторыми из компаний и проектов, включенных в этот список, ни мы, ни издательство O'Reilly не отдадут предпочтение тем или иным инструментам в ущерб другим. Пожалуйста, обязательно проведите собственное исследование на предмет пригодности этих платформ и инструментов для работы, которую вам необходимо выполнить.

Комплексные платформы

Несколько компаний предлагают полностью интегрированные платформы для работы с Apache Kafka. Они включают в себя управляемые развертывания всех компонентов, что позволяет вам сосредоточиться на использовании Kafka, а не на том, как ее запустить. Это может стать идеальным решением для тех случаев, когда недоступны ресурсы (или вы не хотите их выделять) для обучения правильной работе с Kafka и необходимой для этого инфраструктурой. Некоторые из них также предоставляют инструменты, такие как управление схемами, интерфейсы REST, и в некоторых случаях поддержку клиентских библиотек, чтобы вы могли быть уверены в правильном взаимодействии компонентов.

Название: **Confluent Cloud**

URL-адрес: <https://www.confluent.io/confluent-cloud>

Описание: вполне уместно, что компания, созданная одними из первоначальных разработчиков для развития и поддержки Kafka, предлагает управляемое решение. Confluent Cloud объединяет ряд необходимых инструментов, включая управление схемами, клиентов, RESTful-интерфейс и мониторинг, в одном предложении. Оно доступно на всех трех основных облачных платформах (AWS, Microsoft Azure и Google Cloud Platform) и поддерживается значительной частью основных разработчиков Apache Kafka, работающих в компании Confluent. Многие компоненты, входящие в состав платформы, такие как реестр схем и прокси-сервер REST, доступны в качестве отдельных инструментов под лицензией сообщества Confluent (<https://oreil.ly/IAFga>), которая ограничивает некоторые сценарии использования.

Название: **Aiven**

URL-адрес: <https://aiven.io>

Описание: компания Aiven предоставляет управляемые решения для многих платформ данных, включая Kafka. Для поддержки этого она разработала Karapace (<https://karapace.io>), который представляет собой реестр схем и прокси-сервер REST, оба API-совместимы с компонентами Confluent, но поддерживаются по лицензии Apache 2.0 (<https://oreil.ly/a96F0>), которая не ограничивает сценарии использования. Помимо трех основных облачных провайдеров Aiven поддерживает также DigitalOcean (<https://www.digitalocean.com>) и UpCloud (<https://upcloud.com>).

Название: **CloudKafka**

URL-адрес: <https://www.cloudkafka.com>

Описание: компания CloudKafka фокусируется на предоставлении управляемого решения Kafka с интеграциями для популярных инфраструктурных сервисов, таких как DataDog или Splunk. Она поддерживает использование реестра схем Confluent и прокси-сервера REST со своей платформой, но только в версии 5.0 до изменения лицензии Confluent. CloudKafka предоставляет свои услуги как на платформе AWS, так и на Google Cloud.

Название: **Amazon Managed Streaming for Apache Kafka (Amazon MSK)**

URL-адрес: <https://aws.amazon.com/msk>

Описание: компания Amazon также предоставляет также собственную управляемую платформу Kafka, поддерживаемую только на AWS. Поддержка схем

обеспечивается за счет интеграции с AWS Glue (<https://oreil.ly/hvjoV>), в то время как прокси-сервер REST напрямую не поддерживается. Amazon поощряет использование инструментов сообщества, таких как Cruise Control, Burrow и прокси-сервер REST от Confluent, но не поддерживает их напрямую. Таким образом, MSK несколько менее интегрирован, чем другие предложения, но все же может обеспечить основной кластер Kafka.

Название: **Azure HDInsight**

URL-адрес: <https://azure.microsoft.com/en-us/services/hdinsight>

Описание: корпорация Microsoft предоставляет управляемую платформу для Kafka в HDInsight, которая поддерживает также Hadoop, Spark и другие компоненты больших данных. Как и MSK, HDInsight фокусируется на основном кластере Kafka, оставляя многие другие компоненты, включая реестр схем и прокси-сервер REST, на усмотрение пользователя. Некоторые сторонние разработчики предоставили шаблоны для выполнения таких развертываний, но они не поддерживаются компанией Microsoft.

Название: **Cloudera**

URL-адрес: <https://www.cloudera.com/products/open-source/apache-hadoop/apache-kafka.html>

Описание: компания Cloudera с первых дней своего существования является неотъемлемым участником сообщества Kafka и предоставляет управляемую Kafka в качестве компонента потоковых данных в своем комплексном продукте Customer Data Platform (CDP). Однако CDP фокусируется не только на Kafka и работает как в публичных облачных средах, так и в частных.

Развертывание и управление кластером

При запуске Kafka за пределами управляемой платформы вам понадобится несколько вещей, которые помогут вам правильно управлять кластером. Сюда входит помощь в инициализации и развертывании, балансировке данных и визуализации кластеров.

Название: **Strimzi**

URL-адрес: <https://strimzi.io>

Описание: Strimzi предоставляет операторы Kubernetes для развертывания кластеров Kafka, чтобы упростить ее настройку в среде Kubernetes. Компания

не предоставляет управляемые сервисы, однако упрощает процесс запуска и работы в облаке, как в публичном, так и в частном. Он также предоставляет Strimzi Kafka Bridge, который представляет собой реализацию прокси-сервера REST, поддерживаемую лицензией Apache 2.0 (<https://oreil.ly/a96F0>). На данный момент Strimzi не поддерживает реестр схем из-за проблем с лицензиями.

Название: **AKHQ**

URL-адрес: <https://akhq.io>

Описание: AKHQ — это графический интерфейс для управления кластерами Kafka и взаимодействия с ними. Он поддерживает управление конфигурацией, включая пользователей и списки управления доступом, а также обеспечивает некоторую поддержку таких компонентов, как реестр схем и Kafka Connect. Он также предоставляет инструменты для работы с данными в кластере в качестве альтернативы консольным инструментам.

Название: **JulieOps**

URL-адрес: <https://github.com/kafka-ops/julie>

Описание: JulieOps (ранее Kafka Topology Builder) обеспечивает автоматизированное управление топиками и списками управления доступом, используя модель GitOps. Помимо просмотра состояния текущей конфигурации, JulieOps предоставляет средства для декларативной конфигурации и управления изменениями топиков, схем, списков управления доступом и многого другого в долгосрочной перспективе.

Название: **Cruise Control** («Круиз-контроль»)

URL-адрес: <https://github.com/linkedin/cruise-control>

Описание: Cruise Control — это ответ LinkedIn на вопрос, как управлять сотнями кластеров с тысячами брокеров. Этот инструмент начинался как решение для автоматической перебалансировки данных в кластерах, но эволюционировал, включив в себя обнаружение аномалий и административные операции, такие как добавление и удаление брокеров.

Название: **Conduktor**

URL-адрес: <https://www.conduktor.io>

Описание: хотя Conduktor не имеет открытого исходного кода, это популярный настольный инструмент для управления кластерами Kafka и взаимодействия с ними. Он поддерживает многие управляемые платформы, включая Confluent, Aiven и MSK, и множество различных компонентов, таких как Connect, kSQL

и Streams. Он также позволяет взаимодействовать с данными в кластерах, в отличие от использования консольных инструментов. Предоставляется бесплатная лицензия для применения в целях разработки и работы с одним кластером.

Мониторинг и исследование данных

Важнейшая часть работы Kafka — обеспечение работоспособности кластера и клиентов. Как и многие приложения, Kafka предоставляет множество показателей и других телеметрических данных, но разобраться в них может быть непросто. Многие крупные платформы мониторинга, например Prometheus (<https://prometheus.io>), могут легко получать показатели от брокеров и клиентов Kafka. Существует также ряд инструментов, помогающих разобраться во всех этих данных.

Название: **Xinfra Monitor**

URL-адрес: <https://github.com/linkedin/kafka-monitor>

Описание: Xinfra Monitor (ранее Kafka Monitor) был разработан компанией LinkedIn для мониторинга доступности кластеров и брокеров Kafka. Для этого он использует набор топиков для создания искусственных данных через кластер и измерения их задержки, доступности и полноты. Это ценный инструмент для измерения состояния работоспособности развертывания Kafka, не требующий прямого взаимодействия с вашими клиентами.

Название: **Burrow**

URL-адрес: <https://github.com/linkedin/burrow>

Описание: Burrow — это еще один инструмент, изначально созданный компанией LinkedIn, который обеспечивает комплексный мониторинг отставания потребителей в кластерах Kafka. Он позволяет получить представление о здоровье потребителей без необходимости непосредственного взаимодействия с ними. Burrow активно поддерживается сообществом и имеет собственную экосистему инструментов (<https://oreil.ly/yNPRQ>) для связи с другими компонентами.

Название: **Kafka Dashboard**

URL-адрес: <https://www.datadoghq.com/dashboards/kafka-dashboard>

Описание: для тех, кто использует DataDog для мониторинга, он предоставляет отличную панель инструментов Kafka Dashboard, которая поможет вам начать

интегрировать кластеры Kafka в стек мониторинга. Она разработана для обеспечения однопанельного представления вашего кластера Kafka и упрощает просмотр многих показателей.

Название: **Streams Explorer** («Исследователь потоков»)

URL-адрес: <https://github.com/bakdata/streams-explorer>

Описание: Streams Explorer — это инструмент для визуализации потока данных через приложения и соединители в развертывании Kubernetes. Хотя он в значительной степени зависит от структурирования ваших развертываний с использованием Kafka Streams или Faust с помощью инструментов bakdata, но может затем предоставить хорошо понятное представление этих приложений и их показателей.

Название: **kcat**

URL-адрес: <https://github.com/edenhill/kafkacat>

Описание: kcat (ранее kafkacat) — это популярная альтернатива консольным производителю и потребителю, которые являются частью основного проекта Apache Kafka. Он небольшой, быстрый и написан на языке C, поэтому у него нет накладных расходов JVM. Он поддерживает также ограниченное представление о состоянии кластера, показывая метаданные, выводимые для него.

Клиентские библиотеки

Проект Apache Kafka предоставляет клиентские библиотеки для Java-приложений, но одного языка никогда не бывает достаточно. Существует множество реализаций клиента Kafka, причем такие популярные языки, как Python, Go и Ruby, имеют несколько вариантов. Кроме того, прокси-серверы REST, например, от Confluent, Strimzi или Kgarase, могут охватывать множество сценариев использования. Вот несколько клиентских реализаций, которые выдержали испытание временем.

Название: **librdkafka**

URL-адрес: <https://github.com/edenhill/librdkafka>

Описание: librdkafka — это реализация клиента Kafka на языке C, которая считается одной из самых производительных доступных библиотек, настолько хорошей, что Confluent поддерживает клиенты для Go, Python и .NET, которые были созданы как обертки вокруг librdkafka. Она лицензируется по простой

лицензии BSD, состоящей из двух пунктов (<https://oreil.ly/dLoe8>), что позволяет легко использовать ее в любых приложениях.

Название: **Sarama**

URL-адрес: <https://github.com/Shopify/sarama>

Описание: компания Shopify создала клиент Sarama как собственную реализацию Golang. Он выпускается под лицензией MIT (<https://oreil.ly/sajdS>).

Название: **kafka-python**

URL-адрес: <https://github.com/dpkp/kafka-python>

Описание: kafka-python — это еще одна нативная реализация клиента, на этот раз на языке Python. Она выпускается под лицензией Apache 2.0 (<https://oreil.ly/a96F0>).

Потоковая обработка

Хотя проект Apache Kafka включает Kafka Streams для создания приложений, это не единственный вариант для потоковой обработки данных из Kafka.

Название: **Samza**

URL-адрес: <https://samza.apache.org>

Описание: Apache Samza — это фреймворк для потоковой обработки, который был специально разработан для Kafka. Хотя он появился раньше Kafka Streams, их разрабатывали во многом одни и те же люди, в результате у них много общих концепций. Однако, в отличие от Kafka Streams, Samza работает на Yarn и предоставляет полный фреймворк для приложений.

Название: **Spark**

URL-адрес: <https://spark.apache.org>

Описание: Spark — еще один проект Apache, ориентированный на пакетную обработку данных. Он обрабатывает потоки, рассматривая их как быстрые микро-пакеты. Это означает, что задержка немного выше, но отказоустойчивость просто достигается путем повторной обработки пакетов, а архитектура Lambda проста. Преимуществом этой технологии является широкая поддержка сообщества.

Название: **Flink**

URL-адрес: <https://flink.apache.org>

Описание: Apache Flink специально ориентирован на потоковую обработку и работает с очень низкой задержкой. Как и Samza, он поддерживает Yarn, но также работает с Mesos, Kubernetes или автономными кластерами. Он поддерживает Python и R с помощью высокоуровневых API.

Название: **Beam**

URL-адрес: <https://beam.apache.org>

Описание: Apache Beam не предоставляет потоковую обработку напрямую, а вместо этого позиционирует себя как унифицированную модель программирования для пакетной и потоковой обработки. Она использует такие платформы, как Samza, Spark и Flink, в качестве исполнителей для компонентов общего конвейера обработки.

Об авторах

Гвен Шапира (Gwen Shapira) — главный инженер в компании Confluent. Возглавляет команду облачно-ориентированной Kafka, специализирующуюся на повышении эластичности, масштабируемости и мультиарендности возможностей Kafka для Confluent Cloud. У Гвен 15-летний опыт создания масштабируемых архитектур данных. Она часто выступает с докладами на отраслевых конференциях и является членом комитета по управлению проектом (PMC) Apache Kafka.

Тодд Палино (Todd Palino) — главный штатный инженер по надежности сайта в LinkedIn, решающий задачи управления пропускной способностью и эффективностью всей платформы. Ранее он отвечал за архитектуру, повседневную работу и разработку инструментов для Kafka и ZooKeeper в LinkedIn, включая создание расширенной системы мониторинга и уведомлений. Тодд является разработчиком проекта с открытым исходным кодом Burrow, инструмента мониторинга потребителей Kafka. Его можно встретить на отраслевых конференциях, где он делится своим опытом в области SRE. Тодд более 20 лет проработал в технологической отрасли, управляя инфраструктурными сервисами, в том числе в качестве системного инженера в компании Verisign.

Раджини Сиварам (Rajini Sivaram) — главный инженер в компании Confluent, проектирует и разрабатывает функции межкластерной репликации для Kafka и функции безопасности для Confluent Platform и Confluent Cloud. Она является разработчиком программного обеспечения Apache Kafka и членом комитета по управлению программами Apache Kafka. До прихода в Confluent работала в компании Pivotal, создавая высокопроизводительный реактивный API для Kafka на основе Project Reactor. Ранее Раджини работала в IBM, занимаясь разработкой Kafka-as-a-Service для платформы IBM Bluemix. Имеет опыт работы как с параллельными и распределенными системами, так и с виртуальными машинами Java и системами обмена сообщениями.

Крит Петти (Krit Petty) — менеджер SRE Kafka в компании LinkedIn. До того как стать менеджером, работал старшим инженером по обеспечению надежности, расширяя и усиливая Kafka, преодолевая трудности, связанные с масштабированием Kafka до невиданных ранее высот, в том числе делая первые шаги по переносу крупномасштабных развертываний Kafka в LinkedIn в облако Microsoft Azure. Крит имеет степень магистра в области компьютерных наук и ранее работал управляющим менеджером систем Linux, а также инженером-программистом — он разрабатывал программное обеспечение для проектов высокопроизводительных вычислений в нефтегазовой промышленности.

Иллюстрация на обложке

Птица, изображенная на обложке книги, — это синекрылый зимородок — кукабара (*Dacelo leachii*). Она относится к семейству Alcedinidae и встречается на юге Новой Гвинеи и в менее засушливых районах северной Австралии. Ее причисляют к речным зимородкам.

Самец кукабары очень красив. Нижние перья его крыльев и хвоста голубые, отсюда и название, у самок же хвосты красновато-коричневые с черными полосами. У обоих полов нижняя часть тела кремового цвета с коричневыми прожилками, а радужная оболочка глаз — белая. Взрослые кукабары меньше других зимородков — 38–43 см в длину и в среднем весят около 260–330 г.

Кукабары плотоядные, причем их добыча немного меняется в зависимости от времени года. Например, в летние месяцы птица питается ящерицами, лягушками и насекомыми, которых много в это время, а в более засушливую пору в рационе появляется больше раков, рыбы, грызунов и даже более мелких птиц. Однако они не одиноки в том, что едят других птиц, поскольку саму кукабару в сезон включают в свое меню красные ястребы-тетеревятники и пурпурные совы.

Размножается синекрылый зимородок в период с сентября по декабрь. Гнезда он строит в дуплах на высоких деревьях. Выкармливание птенцов — это совместная работа: у мамы и папы есть как минимум одна птица-помощник. Кукабары откладывают 3–4 яйца и высиживают их около 26 дней. Птенцы оперяются примерно через 36 дней после вылупления — если выживают: известны случаи, когда старшие братья и сестры убивали младших на первой неделе их агрессивной и конкурентной жизни. Тех, кто не стал жертвой братоубийства или не умер по другим причинам, родители научат охотиться в течение 6–10 недель, прежде чем молодые птицы отправятся в самостоятельную жизнь.

Многие из животных, изображенных на обложках книг издательства O'Reilly, находятся под угрозой исчезновения, а ведь все они важны для мира. Чтобы узнать больше о том, как можно им помочь, посетите сайт animals.oreilly.com.

Цветная иллюстрация принадлежит Карен Монтгомери, она создана по черно-белой гравюре из *English Cyclopaedia*.

Гвен Шапира, Тодд Палино, Раджини Сиварам, Крит Петти

Apache Kafka. Потокковая обработка и анализ данных

2-е издание

Перевел с английского С. Черников

Руководитель дивизиона	<i>Ю. Сергиенко</i>
Руководитель проекта	<i>А. Питиримов</i>
Ведущий редактор	<i>Н. Гринчик</i>
Научный редактор	<i>В. Дмитриуценок</i>
Литературный редактор	<i>Н. Рощина</i>
Художественный редактор	<i>В. Мостипан</i>
Корректоры	<i>Е. Павлович. Е. Рафалюк-Бузовская</i>
Верстка	<i>Г. Блинов</i>

Изготовлено в России. Изготовитель: ООО «Прогресс книга».

Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,

Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 02.2023. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортёр в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 10.01.23. Формат 70×100/16. Бумага офсетная. Усл. п. л. 41,280. Тираж 700. Заказ 0000.