

O'REILLY®



Apache Kafka

ПОТОКОВАЯ ОБРАБОТКА И АНАЛИЗ ДАННЫХ



Ния Нархид,
Гвен Шапира, Тодд Палино

Kafka: The Definitive Guide

Real-Time Data and Stream Processing at Scale

Neha Narkhede, Gwen Shapira, and Todd Palino

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Ния Нархид, Гвен Шапира, Тодд Палино

Apache Kafka

ПОТОКОВАЯ ОБРАБОТКА И АНАЛИЗ ДАННЫХ



Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону • Самара • Минск

2019

Ния Нархид, Гвен Шапира, Тодд Палино
Apache Kafka. Потокковая обработка и анализ данных

Серия «Бестселлеры O'Reilly»

Перевел с английского *И. Пальти*

Заведующая редакцией
Руководитель проекта
Ведущий редактор
Литературный редактор
Художественный редактор
Корректоры
Верстка

Ю. Сергиенко
О. Сивченко
Н. Гринчик
Н. Рощина
С. Заматевская
О. Андриевич, Е. Рафалюк-Бузовская
Г. Блинов

ББК 32.973.23 УДК 004.3

Нархид Ния, Шапира Гвен, Палино Тодд

H30 Apache Kafka. Потокковая обработка и анализ данных. — СПб.: Питер, 2019. — 320 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-0575-5

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 из этой замечательной книги вы 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.)

ISBN 978-1491936160 англ.

Authorized Russian translation of the English edition Kafka: The Definitive Guide
ISBN 9781491936160 © 2017 Neha Narkhede, Gwen Shapira, Todd Palino
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 978-5-4461-0575-5

© Перевод на русский язык ООО Издательство «Питер», 2019
© Издание на русском языке, оформление ООО Издательство «Питер», 2019
© Серия «Бестселлеры O'Reilly», 2019

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.

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: 08.2018. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —

Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 25.07.18. Формат 70×100/16. Бумага офсетная. Усл. п. л. 25,800. Тираж 700. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».

142300, Московская область, г. Чехов, ул. Полиграфистов, 1.

Сайт: www.chpk.ru. E-mail: marketing@chpk.ru Факс: 8(496) 726-54-10, телефон: (495) 988-63-87

Краткое содержание

Предисловие	15
Введение.....	18
Глава 1. Знакомьтесь: Kafka.....	22
Глава 2. Установка Kafka	42
Глава 3. Производители Kafka: запись сообщений в Kafka	65
Глава 4. Потребители Kafka: чтение данных из Kafka.....	88
Глава 5. Внутреннее устройство Kafka	120
Глава 6. Надежная доставка данных.....	140
Глава 7. Создание конвейеров данных.....	161
Глава 8. Зеркальное копирование между кластерами	184
Глава 9. Администрирование Kafka	213
Глава 10. Мониторинг Kafka	243
Глава 11. Поточковая обработка.....	278
Приложение. Установка Kafka на других операционных системах.....	315

Оглавление

Предисловие	15
Введение.....	18
Для кого предназначена эта книга.....	19
Условные обозначения	19
Использование примеров кода	20
Благодарности	21
Глава 1. Знакомьтесь: Kafka	22
Обмен сообщениями по типу «публикация/подписка»	22
С чего все начинается	23
Отдельные системы организации очередей.....	26
Открываем для себя систему Kafka	26
Сообщения и пакеты.....	26
Схемы.....	28
Темы и разделы.....	28
Производители и потребители.....	29
Брокеры и кластеры	31
Несколько кластеров	32
Почему Kafka?	34
Несколько производителей.....	34
Несколько потребителей.....	34
Сохранение информации на диске	34
Масштабируемость.....	35
Высокое быстродействие	35
Экосистема данных.....	35
Сценарии использования.....	36

История создания Kafka	38
Проблема LinkedIn	38
Рождение Kafka	40
Открытый исходный код.....	40
Название	41
Приступаем к работе с Kafka.....	41
Глава 2. Установка Kafka	42
Обо всем по порядку	42
Выбрать операционную систему	42
Установить Java.....	42
Установить ZooKeeper	43
Установка брокера Kafka	45
Конфигурация брокера.....	46
Основные настройки брокера.....	47
Настройки тем по умолчанию	49
Выбор аппаратного обеспечения	53
Пропускная способность дисков.....	54
Емкость диска	54
Память	55
Передача данных по сети.....	55
CPU.....	55
Кafka в облачной среде	56
Кластеры Kafka	56
Сколько должно быть брокеров?	57
Конфигурация брокеров.....	58
Тонкая настройка операционной системы.....	58
Промышленная эксплуатация	61
Параметры сборки мусора	61
Планировка ЦОД.....	62
Размещение приложений на ZooKeeper	63
Резюме.....	64
Глава 3. Производители Kafka: запись сообщений в Kafka	65
Обзор производителя.....	66
Создание производителя Kafka	68
Отправка сообщения в Kafka	70

Синхронная отправка сообщения	71
Асинхронная отправка сообщения	71
Настройка производителей	72
acks	73
buffer.memory	73
compression.type	74
retries	74
batch.size	75
linger.ms	75
client.id	75
max.in.flight.requests.per.connection	75
timeout.ms, request.timeout.ms и metadata.fetch.timeout.ms	76
max.block.ms	76
max.request.size	76
receive.buffer.bytes и send.buffer.bytes	76
Сериализаторы	77
Пользовательские сериализаторы	77
Сериализация с помощью Apache Avro	79
Использование записей Avro с Kafka	81
Разделы	84
Старые API производителей	86
Резюме	87
Глава 4. Потребители Kafka: чтение данных из Kafka	88
Принципы работы потребителей Kafka	88
Потребители и группы потребителей	88
Группы потребителей и переконфигурирование разделов	92
Создание потребителя Kafka	94
Подписка на темы	94
Цикл опроса	95
Настройка потребителей	97
fetch.min.bytes	97
fetch.max.wait.ms	97
max.partition.fetch.bytes	98
session.timeout.ms	98
auto.offset.reset	99
enable.auto.commit	99

partition.assignment.strategy	99
client.id	100
max.poll.records.....	100
receive.buffer.bytes и send.buffer.bytes	100
Фиксация и смещения	101
Автоматическая фиксация.....	102
Фиксация текущего смещения.....	103
Асинхронная фиксация	104
Сочетание асинхронной и синхронной фиксации	105
Фиксация заданного смещения	106
Прослушивание на предмет перебалансировки	107
Получение записей с заданными смещениями	109
Выход из цикла.....	112
Десериализаторы.....	113
Пользовательские сериализаторы	114
Использование десериализации Avro в потребителе Kafka	116
Автономный потребитель: зачем и как использовать потребитель без группы	117
Старые API потребителей	118
Резюме.....	119
Глава 5. Внутреннее устройство Kafka	120
Членство в кластере.....	120
Контроллер.....	121
Репликация.....	122
Обработка запросов.....	124
Запросы от производителей.....	127
Запросы на извлечение	127
Другие запросы.....	129
Физическое хранилище	131
Распределение разделов	131
Управление файлами.....	133
Формат файлов.....	134
Индексы	136
Сжатие	136
Как происходит сжатие.....	137
Удаленные события	138
Когда выполняется сжатие тем.....	139
Резюме.....	139

Глава 6. Надежная доставка данных	140
Гарантии надежности	141
Репликация.....	142
Настройка брокера.....	143
Коэффициент репликации	143
«Нечистый» выбор ведущей реплики	145
Минимальное число согласованных реплик.....	146
Использование производителей в надежной системе	147
Отправка подтверждений	148
Настройка повторов отправки производителями	149
Дополнительная обработка ошибок.....	150
Использование потребителей в надежной системе	151
Свойства конфигурации потребителей, важные для надежной обработки	152
Фиксация смещений в потребителях явным образом.....	153
Проверка надежности системы.....	156
Проверка конфигурации	157
Проверка приложений	158
Мониторинг надежности при промышленной эксплуатации.....	158
Резюме	160
Глава 7. Создание конвейеров данных	161
Соображения по поводу создания конвейеров данных	162
Своевременность.....	162
Надежность	163
Высокая/переменная нагрузка	164
Форматы данных.....	164
Преобразования.....	165
Безопасность.....	166
Обработка сбоев	166
Связывание и быстрота адаптации.....	167
Когда использовать Kafka Connect, а когда клиенты-производители и клиенты-потребители.....	168
Kafka Connect.....	168
Запуск Connect	169
Пример коннектора: файловый источник и файловый приемник	171
Пример коннектора: из MySQL в Elasticsearch.....	172
Взглянем на Connect поближе	178

Альтернативы Kafka Connect	181
Фреймворки ввода и обработки данных для других хранилищ.....	182
ETL-утилиты на основе GUI	182
Фреймворки потоковой обработки	182
Резюме	183
Глава 8. Зеркальное копирование между кластерами.....	184
Сценарии зеркального копирования данных между кластерами.....	185
Мультикластерные архитектуры.....	186
Реалии взаимодействия между различными ЦОД.....	186
Архитектура с топологией типа «звезда»	187
Архитектура типа «активный — активный».....	189
Архитектура типа «активный — резервный»	192
Потери данных и несогласованности при внеплановом восстановлении после сбоя.....	193
Начальное смещение для приложений после аварийного переключения.....	194
После аварийного переключения.....	198
Несколько слов об обнаружении кластеров.....	198
Эластичные кластеры	199
Утилита MirrorMaker (Apache Kafka)	200
Настройка MirrorMaker.....	201
Развертывание MirrorMaker для промышленной эксплуатации	202
Тонкая настройка MirrorMaker	206
Другие программные решения для зеркального копирования между кластерами	209
uReplicator компании Uber.....	209
Replicator компании Confluent.....	210
Резюме	211
Глава 9. Администрирование Kafka	213
Операции с темами	213
Создание новой темы.....	214
Добавление разделов	215
Удаление темы	216
Вывод списка всех тем кластера	216
Подробное описание тем	217

Группы потребителей.....	218
Вывод списка и описание групп	218
Удаление группы.....	220
Управление смещениями	220
Динамические изменения конфигурации	222
Переопределение значений настроек тем по умолчанию	222
Переопределение настроек клиентов по умолчанию	224
Описание переопределений настроек.....	225
Удаление переопределений настроек	225
Управление разделами	225
Выбор предпочтительной ведущей реплики.....	226
Смена реплик раздела	227
Изменение коэффициента репликации	230
Сброс на диск сегментов журнала.....	231
Проверка реплик	233
Потребление и генерация	234
Консольный потребитель.....	234
Консольный производитель.....	237
Списки управления доступом клиентов	239
Небезопасные операции.....	239
Перенос контроллера кластера	240
Отмена перемещения раздела	240
Отмена удаления тем.....	241
Удаление тем вручную.....	241
Резюме.....	242
Глава 10. Мониторинг Kafka	243
Основы показателей	243
Как получить доступ к показателям.....	243
Внешние и внутренние показатели	244
Контроль состояния приложения	245
Охват показателей	245
Показатели брокеров Kafka	246
Недореплицированные разделы.....	246
Показатели брокеров	252
Показатели тем и разделов	261
Мониторинг JVM.....	263

Мониторинг ОС	265
Журналирование.....	266
Мониторинг клиентов	267
Показатели производителя	267
Показатели потребителей	271
Квоты.....	274
Мониторинг отставания.....	275
Сквозной мониторинг.....	276
Резюме.....	277
Глава 11. Поточковая обработка.....	278
Что такое поточковая обработка	279
Основные понятия поточковой обработки.....	282
Время.....	282
Состояние	283
Таблично-поточковый дуализм	284
Временные окна.....	286
Паттерны проектирования поточковой обработки	287
Обработка событий по отдельности.....	288
Обработка с использованием локального состояния.....	288
Многоэтапная обработка/повторное разделение на разделы	290
Обработка с применением внешнего справочника: соединение потока данных с таблицей.....	292
Соединение потоков	294
Внеочередные события	295
Повторная обработка.....	296
Kafka Streams в примерах	297
Подсчет количества слов.....	298
Сводные показатели фондовой биржи.....	301
Обогащение потока событий перехода по ссылкам.....	303
Kafka Streams: обзор архитектуры.....	305
Построение топологии.....	306
Масштабирование топологии.....	306
Как пережить отказ.....	310
Сценарии использования поточковой обработки	310
Как выбрать фреймворк поточковой обработки	312
Резюме.....	314

Приложение. Установка Kafka на других операционных системах.....	315
Установка на Windows	315
Использование Windows Subsystem для Linux.....	315
Использование Java естественным образом.....	316
Установка на MacOS.....	318
Использование Homebrew.....	319
Установка вручную	319

Предисловие

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 уже 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 подошло.

В конце концов договорились 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 их поток и проектировать информационные системы — и, конечно, архитектуру данных — на этой основе.

Эта идея нашла 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, и также пересматривают свои фундаментальные процессы и системы с ориентацией на Kafka и банки.

Так что же это такое, Kafka? Чем она 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 оказаться 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 различий с традиционными системами 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 привязываемых к различным приложениям, Kafka предоставляет централизованную платформу, гибко масштабирующуюся под обработку всех потоков данных компании. Во-вторых, Kafka способна хранить данные столько времени, сколько нужно. Это дает колоссальные преимущества при ее использовании в качестве соединительного 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 вычислять производные потоки и наборы данных динамически, причем при гораздо меньшем количестве кода. Эти отличия ставят Kafka довольно обособленно.

Kafka можно также рассматривать 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 также можно в очень больших масштабах хранить и непрерывно обрабатывать потоки данных. 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 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 используемые в сфере складирования данных, присущее Kafka низкое значение задержки делает ее подходящей 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 возможность дать ответ на отзывы пользователей и т. д.

Еще одна категория, с которой сравнивают Kafka, — ETL и утилиты интеграции данных. 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, а платформа, основанная на концепции потоков событий в режиме реального времени. Это значит, что она может не только стыковать стандартные приложения с информационными системами, но и обеспечивать функционирование пользовательских приложений, создаваемых для порождения этих самых потоков данных. Нам 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 отражаемые в финансовых ведомостях потоки наличных денег.

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

Однако все несколько отличается от традиционного представления, и создание приложений, ориентированных на работу с непрерывными потоками данных, требует существенной смены парадигмы мышления 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 захотел, чтобы эта книга была у меня, как только познакомился с описанной в ней темой». Именно с такой установкой мы и начали писать. Мы вспомнили опыт, полученный при разработке Kafka, запуске ее в промышленную эксплуатацию и поддержке множества компаний при создании 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 сюда то, что считаем необходимым для успешной работы Apache Kafka при промышленной эксплуатации и создании на ее основе устойчивых к ошибкам высокопроизводительных приложений. Уделили особое внимание популярным сценариям использования: шине сообщений для событийно управляемых микросервисов, приложениям, обрабатывающим потоки, а также широкомасштабным 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, применение 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 выбрать необходимые при программировании компромиссы.

Одна из проблем инженерии разработки ПО — наличие нескольких 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 пояснять, какие

в конкретном случае существуют варианты, каково соотношение выгод и потерь и то, когда следует и не следует 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 забывали 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 11, ориентированы на Java-разработчиков. Для их усвоения важно, чтобы читатель был знаком с основами языка программирования Java, включая такие вопросы, как обработка исключений и конкурентность. В других главах, особенно 2, 8, 9 и 10, предполагается, что у читателя есть опыт работы с Linux и он знаком с настройкой сети и хранилищ данных 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, работающие не непосредственно с Kafka, а 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 сослуживцам 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.

Рубленый шрифт

Им набраны URL, адреса электронной почты.

`Моноширинный шрифт`

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

Полужирный 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

Эта книга создана для того, чтобы помочь вам в работе. В целом приведенные примеры кода вы можете использовать в своих программах и документации. Обращаться к нам за разрешением нет необходимости, если только вы не копируете значительную часть кода. Например, написание программы, использующей несколько фрагментов кода из этой книги, не требует отдельного разрешения. Для продажи или распространения компакт-диска с примерами из книг O'Reilly, конечно, разрешение нужно. Ответ на вопрос путем цитирования этой книги, в том числе примеров кода, не требует разрешения. Включение значительного количества кода примеров из этой книги в документацию к вашему продукту может потребовать разрешения.

Мы ценим, хотя и не требуем, ссылки на первоисточник. Ссылка на первоисточник включает название, автора, издательство и ISBN. Например: «Нархид Н., Шапира Г., Палино Т. Apache Kafka. Поточковая обработка и анализ данных. — СПб.: Питер, 2018. — ISBN 978-1-491-93616-0».

Если вам кажется, что заимствование примеров кода выходит за рамки правомерного использования или данного ранее разрешения, не стесняясь, связывайтесь с нами 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 его 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 (Apurva 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), Jeff Holoman (Джеф Холомен), Джоэль Коши (Joel Koshy), Джонатан Сейдмэн (Jonathan Seidman), 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 издательства O'Reilly Шеннон Катт (Shannon Cutt) за терпение, поддержку и гораздо лучший по сравнению с нашим контроль ситуации. Работа с издательством O'Reilly — замечательный опыт для любого автора: предоставляемая ими 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.

И мы хотели бы поблагодарить свое руководство и коллег за помощь и содействие, которые получили при написании этой книги.

Гвен также хотела бы поблагодарить своего супруга, Омера Шапира (Omer Shapira), за терпение и поддержку на протяжении многих месяцев, потраченных на написание еще одной книги, а также кошек Люка и Лею за то, что они такие милые, и отца, Лиора Шапира, 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, все время ободрявших его. Их поддержка, не ослабевавшая несмотря на то, что написание книги потребовало дополнительного времени, и долгие часы пробежек для освежения головы помогали ему работать.

1

Знакомьтесь: Kafka

Деятельность любого предприятия питается данными. Мы получаем информацию, анализируем ее, выполняем над ней какие-либо действия и создаем новые данные в качестве результатов. Все приложения создают данные — журнальные сообщения, 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 мышью на интересующих нас товарах превращаются в демонстрируемые нам же чуть позже рекомендации.

От скорости этого процесса зависят адаптивность и быстрота реакции нашего предприятия. Чем меньше усилий мы тратим на перемещение данных, тем больше можем уделить внимания основной деятельности. Именно поэтому конвейер — ключевой компонент в ориентированном на работу с данными предприятии. Способ перемещения данных оказывается практически столь 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 Деграсс Тайсон

Обмен сообщениями по типу «публикация/подписка»

Прежде чем перейти к обсуждению нюансов Apache Kafka, важно разобраться в обмене сообщениями по типу «публикация/подписка» и причине, по которой оно столь важно. *Обмен сообщениями по типу «публикация/подписка»* (publish/subscribe 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 классы сообщений. 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 создавать прямое соединение между вашим приложением и приложением, отображающим показатели в инструментальной панели, и «проталкивать» последние через это соединение (рис. 1.1).

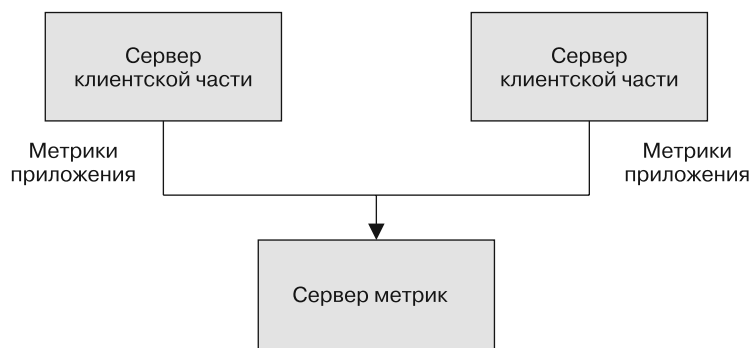


Рис. 1.1. Отдельный непосредственный издатель показателей

Это простое решение простой задачи, удобное для начала мониторинга. Но вскоре вам захочется анализировать показатели за больший период времени, а в инструментальной панели это не слишком удобно. Вы создадите новый сервис для получения показателей, их хранения и анализа. Для этого измените свое приложение так, чтобы оно могло записывать их в обе системы. К тому времени у вас появится 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.

Некоторая недоработка тут очевидна, так что вы решаете ее исправить. Создаете единое приложение, получающее показатели от всех имеющихся приложений и включающее сервер, предназначенный для запроса этих показателей для всех систем, которым они нужны. В результате сложность архитектуры снижается (рис. 1.3). Поздравляем, вы создали систему обмена сообщениями по типу «публикация/подписка»!

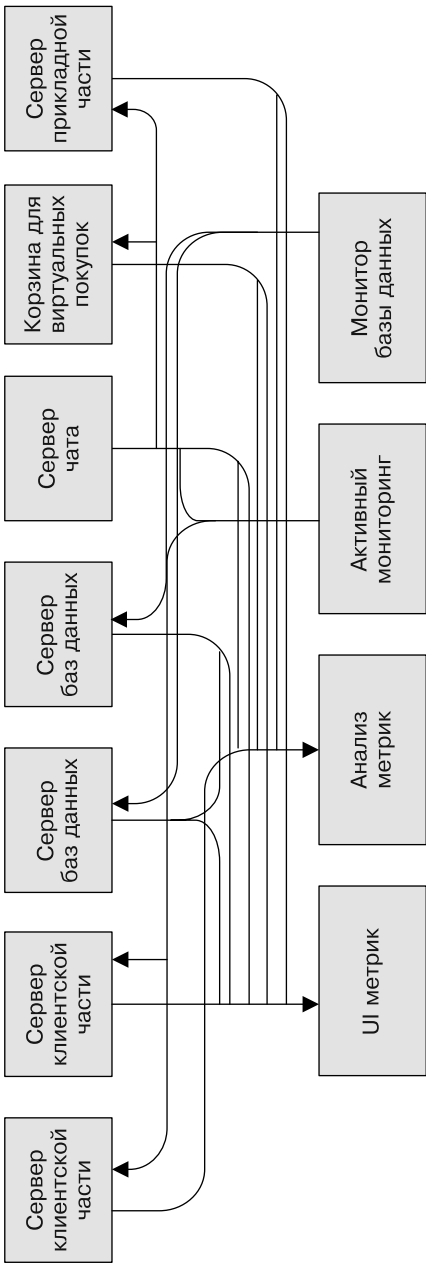


Рис. 1.2. Множество издателей показателей, использующих прямые соединения

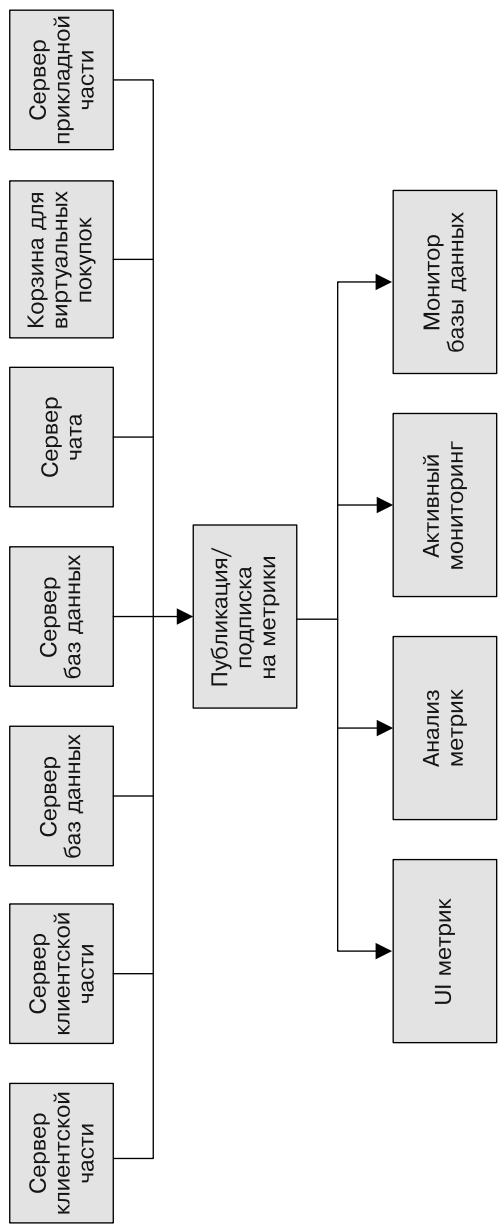


Рис. 1.3. Система публикации/подписки на показатели

Отдельные системы организации очередей

В то время как вы боролись с показателями, один из ваших коллег аналогичным образом трудился над журнальными сообщениями. 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 создания систем, расцепляющих издателей информации и подписчиков на нее. Подобная инфраструктура с тремя отдельными системами публикации/подписки показана на рис. 1.4.

Использовать ее намного удобнее, чем прямые соединения (как на 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 использования 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 — система публикации сообщений и подписки на них, предназначенная для решения поставленной задачи. Ее часто называют распределенным журналом фиксации транзакций или, в последнее время, распределенной платформой потоковой обработки. Файловая система или журнал фиксации транзакций 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, 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.

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 элемент метаданных, называемый *ключом* (key). Он также 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 по-

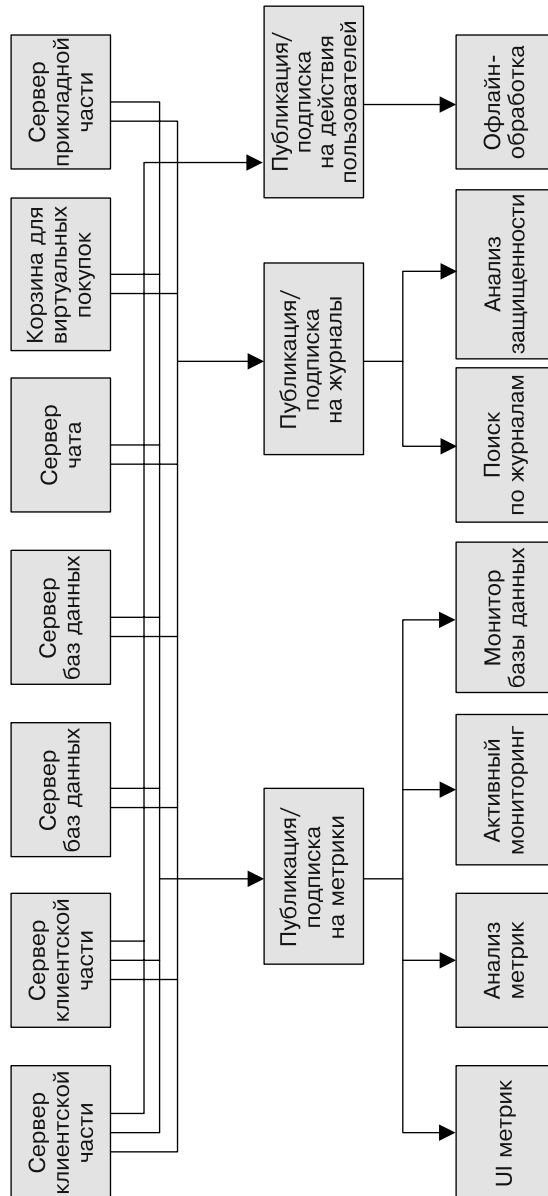


Рис. 1.4. Несколько систем публикации/подписки

следующим выбором номера раздела для сообщения путем деления указанного значения по модулю общего числа разделов в теме. Это гарантирует попадание сообщений с одним ключом в один раздел. Мы обсудим ключи подробнее в главе 3.

Для большей эффективности сообщения в Kafka записываются пакетами. *Пакет* (batch) представляет собой просто набор сообщений, относящихся к одной теме и одному 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 подвергаются сжатию, что позволяет передавать и хранить данные более эффективно за счет некоторого расхода вычислительных ресурсов.

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 *схем* 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 ошибкоустойчивая 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 расцеплять 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.

Темы и разделы

Сообщения в Kafka распределяются по *темам* (topics, иногда называют топиками). Ближайшая аналогия — таблица базы данных или каталог файловой системы. Темы 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 от начала к концу. Заметим: поскольку тема обычно состоит из нескольких разделов, нет никаких гарантий упорядоченности сообщений в пределах всей темы — лишь в пределах отдельного раздела. На рис. 1.5 показана тема 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 с целью достижения производительности, далеко выходящей за пределы возможностей одного сервера.



Рис. 1.5. Представление темы с несколькими разделами

При обсуждении данных, находящихся в таких системах, как Kafka, часто используется термин *поток данных* (stream). Чаще всего поток данных считается отдельной темой, независимо от количества разделов представляющей собой отдельный поток данных, перемещающихся от производителей к потребителям. Обычно 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. Их принцип работы подобен принципу работы офлайн-фреймворков, в частности Hadoop, предназначенных для работы с блоками данных. Обзор темы потоковой обработки приведен в главе 11.

Производители и потребители

Пользователи Kafka делятся на два основных типа: производители (генераторы) и потребители. Существуют также продвинутые клиентские API — API Kafka Connect для интеграции данных и Kafka Streams для потоковой обработки. Продвинутое клиенты используют 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). В целом производители сообщений создают их для конкретной темы. По умолчанию производителю не важно, в какой раздел записывается конкретное сообщение, он будет равномерно поставлять сообще-

ния во все разделы темы. В некоторых случаях производитель направляет сообщение в конкретный раздел, для чего 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 одну тему или более и читает сообщения в порядке их создания. 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 генерации. Смещения сообщений в конкретном разделе не повторяются. Благодаря сохранению смещения последнего полученного сообщения для каждого раздела в хранилище ZooKeeper или 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 темы. Организация в группы гарантирует чтение каждого раздела только одним членом группы. На рис. 1.6 представлены три потребителя, объединенные в одну группу для обработки темы. Два потребителя обрабатывают по одному разделу, а третий — два. Соответствие потребителя разделу иногда называют *принадлежностью* (ownership) раздела данному потребителю.

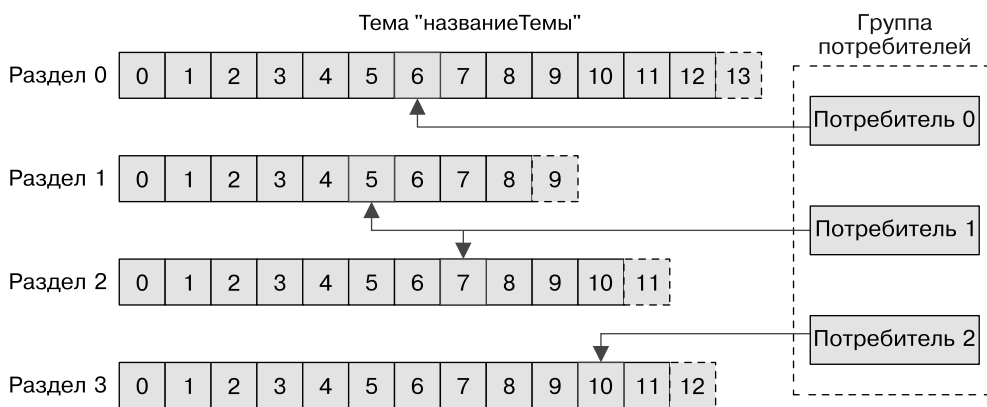


Рис. 1.6. Чтение темы группой потребителей

Таким образом, потребители получают возможность горизонтального масштабирования для чтения темы с большим количеством сообщений. Кроме того, в случае

сбоя отдельного потребителя оставшиеся члены группы перераспределят разделы так, чтобы взять на себя его задачу. Потребители и группы потребителей подробнее описываются в главе 4.

Брокеры и кластеры

Отдельный сервер Kafka называется *брокером* (broker). Брокер получает сообщения от производителей, присваивает им смещения и отправляет их в дисковое хранилище. Он также обслуживает потребителей и отвечает на запросы выборки из разделов, возвращая записанные на диск 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. Контроллер отвечает за административные операции, включая распределение разделов по брокерам и мониторинг отказов последних. Каждый раздел принадлежит одному из брокеров кластера, который называется его *ведущим* (leader). Раздел можно назначить нескольким брокерам, в результате чего произойдет ее репликация (рис. 1.7). Это обеспечивает избыточность сообщений в разделе, так что в случае сбоя ведущего другой брокер сможет занять его место. Однако все потребители и производители, работающие в этом разделе, должны соединяться с ведущим. Кластерные операции, включая репликацию разделов, подробно рассмотрены в главе 6.

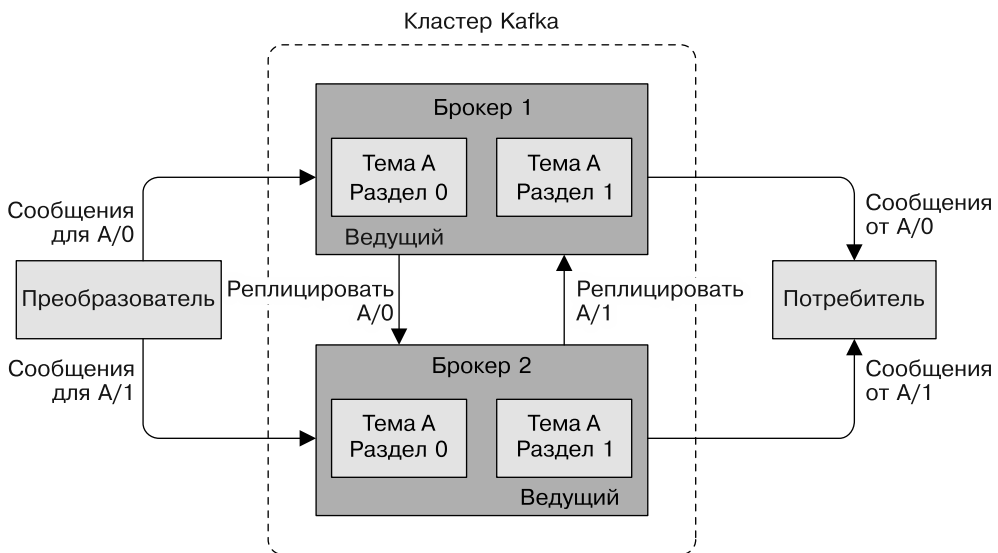


Рис. 1.7. Репликация разделов в кластере

Ключевая возможность Apache Kafka — *сохранение информации* (retention) в течение длительного времени. В настройки брокеров Kafka включается длительность хранения тем по умолчанию — или в течение определенного промежутка времени (например, 7 дней), или до достижения темой определенного размера в байтах (например, 1 Гбайт). Превысившие эти пределы сообщения становятся недействительными и удаляются, так что настройки сохранения соответствуют минимальному количеству доступной в каждый момент информации. Можно задавать настройки сохранения и для отдельных тем, чтобы сообщения хранились только до тех пор, пока они нужны. Например, тема для отслеживания действий пользователей можно хранить несколько дней, в то время как параметры приложений — лишь несколько часов. Можно также настроить для тем 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 катастроф).

При работе, 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 меняет 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 для этой цели утилиту *MirrorMaker*. 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 создает у вас ложного впечатления о его возможностях по созданию сложных конвейеров данных, которые мы подробнее рассмотрим в главе 7.

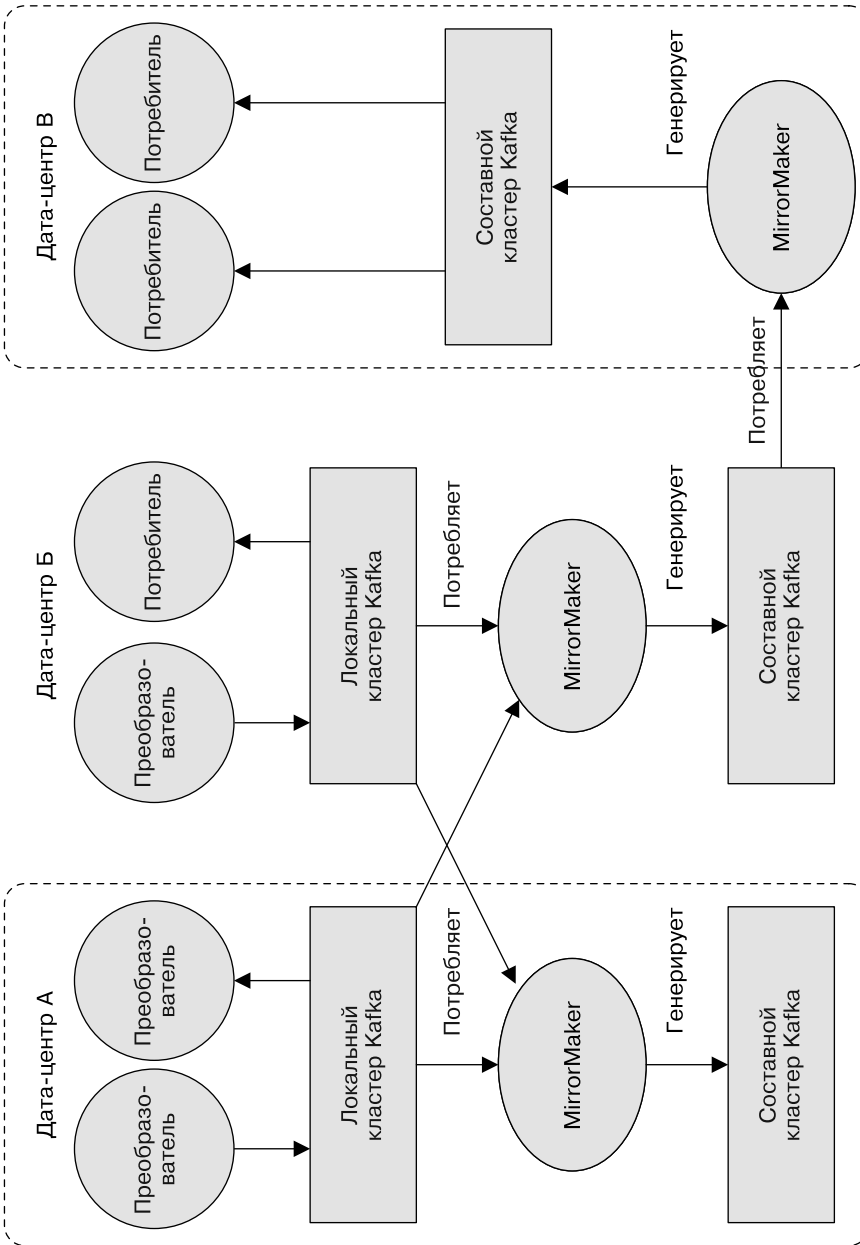


Рис. 1.8. Архитектура с несколькими ЦОД

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 одну тему 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 быть отдельная тема для просмотров страниц, в которую 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 тем, 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 каждой темы по отдельности. Благодаря этому различные потоки сообщений будут храниться в течение разного времени в зависимости от потребностей 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 потребителей 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 таким образом не прервется. Кластеры, которые должны выдерживать одновременные отказы нескольких брокеров, можно настроить, увеличив коэффициенты репликации. Подробнее репликация обсуждается в главе 6.

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 для легкой обработки очень больших потоков сообщений. Причем это можно делать параллельно с обеспечением менее чем секундной задержки сообщений по пути от производителя к потребителю.

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 использования.

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. В сочетании с системой, предоставляющей схемы сообщений, производители и потребители более

не требуют сильного сцепления или прямого соединения. По мере возникновения и исчезновения бизнес-моделей можно добавлять и удалять компоненты, причем производители не должно волновать то, какие приложения потребляют данные и сколько их.

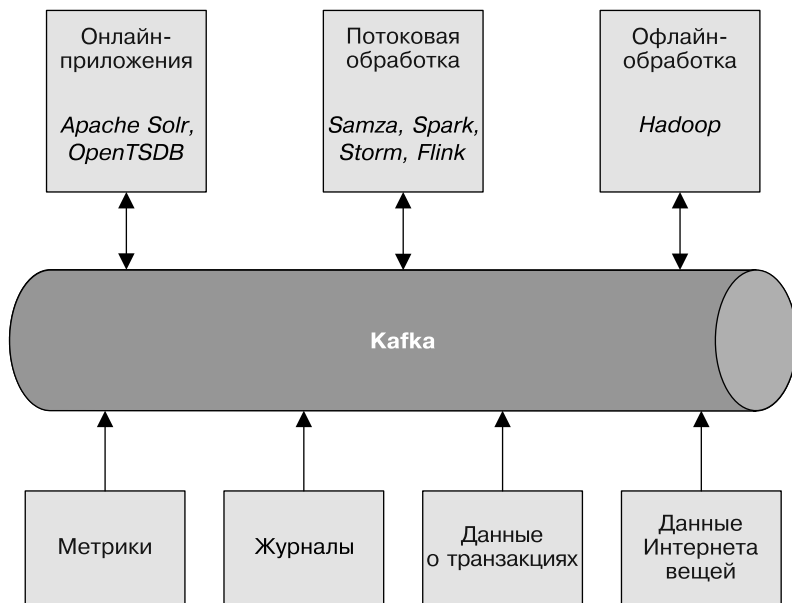


Рис. 1.9. Большая экосистема данных

Сценарии использования

Отслеживание действий пользователей

Первоначальный сценарий использования платформы Kafka, разработанный при ее создании в компании LinkedIn, состоял в отслеживании действий пользователей. Пользователи 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, добавлении пользователем данных в свой профиль.

Сообщения публикуются в одной или нескольких темах, 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 использования.

Обмен сообщениями

Kafka используется также для обмена сообщениями, при котором 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 также идеально подходит для сбора показателей и журналов приложения и системы. В случае реализации этого сценария использования особенно ярко проявляется вероятность наличия нескольких приложений, генерирующих однотипные сообщения. Приложения регулярно публикуют в темах Kafka показатели, потребителями которых становятся системы мониторинга и оповещения. Их также можно использовать в таких офлайн-системах, как Hadoop, для долгосрочного анализа, например, 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: в случае необходимости изменения целевой системы (например, 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. Этот поток журнала изменений можно использовать и для репликации изменений базы данных в удаленной системе или для объединения изменений из нескольких систем в единое представление базы данных. Долгосрочное сохранение оказывается удобным для создания буфера для этого журнала изменений,

поскольку дает возможность повторного выполнения в случае сбоя приложений-потребителей. В качестве альтернативы можно воспользоваться темами 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 работает с данными в режиме реального времени со скоростью генерации сообщений. Потоковые 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 11.

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 со 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, 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 быстрой реакции приложения, в ней была продвинутая возможность отслеживания запросов, использовавшая систему мониторинга и позволявшая анализировать прохождение запроса пользователя по внутренним приложениям. У системы

мониторинга, однако, было 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 офлайн-обработку, 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 использовать 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 использования 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) — 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 справиться с таким объемом 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 чего 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, а позднее Чжан Рао. 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:

- ❑ расцепить 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 в масштабе миллиардов сообщений в день. Масштабируемость Kafka сыграла свою роль в том, что объем использования LinkedIn вырос до более чем триллиона сообщений и петабайта потребляемых данных ежедневно (по состоянию на август 2015 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 Kafka 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. Осенью 2014 года Джей Крепс, Ния Нархид и Чжан Рао покинули LinkedIn и основали Confluent — компанию, сосредоточившую усилия на обеспечении разработки, коммерческой поддержки и обучения Apache Kafka. Эти две компании продолжают разрабатывать и сопровождать Kafka, превращая ее в оптимальное средство создания больших конвейеров данных. В этом им помогает сообщество других разработчиков открытого ПО, чей вклад в работу постоянно растет.

Название

Часто можно услышать вопрос: почему Kafka получила такое название? 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»*.

Приступаем к работе с Kafka

Теперь, когда мы знаем все 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.

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 развертывания Kafka и критерии выбора 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.

Обо всем по порядку

Прежде чем использовать Kafka, необходимо проделать несколько вещей. В следующих разделах расскажем о них.

Выбрать операционную систему

Apache Kafka представляет собой Java-приложение, которое может работать на множестве операционных систем, в числе которых Windows, MacOS, Linux и др. В этой главе мы сосредоточимся на установке Kafka в среде Linux, поскольку именно на этой операционной системе платформу устанавливают чаще всего. Linux также является рекомендуемой операционной системой 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. Рекомендуется использовать Java 8, причем это может быть версия, как включенная в вашу операционную систему, так и непосредственно загруженная

с сайта java.com. Хотя ZooKeeper и Kafka будут работать с Java Runtime Edition, при разработке утилит и приложений удобнее использовать полный Java Development Kit (JDK). Приведенные шаги установки предполагают, что у вас в каталоге `/usr/java/jdk1.8.0_51` установлен JDK версии 8.0.51.

Установить ZooKeeper

Apache Kafka использует ZooKeeper для хранения метаданных о кластере Kafka, а также подробностей о клиентах-потребителях (рис. 2.1). Хотя ZooKeeper можно запустить и с помощью сценариев, включенных в дистрибутив Kafka, установка полной версии хранилища ZooKeeper из дистрибутива очень проста.

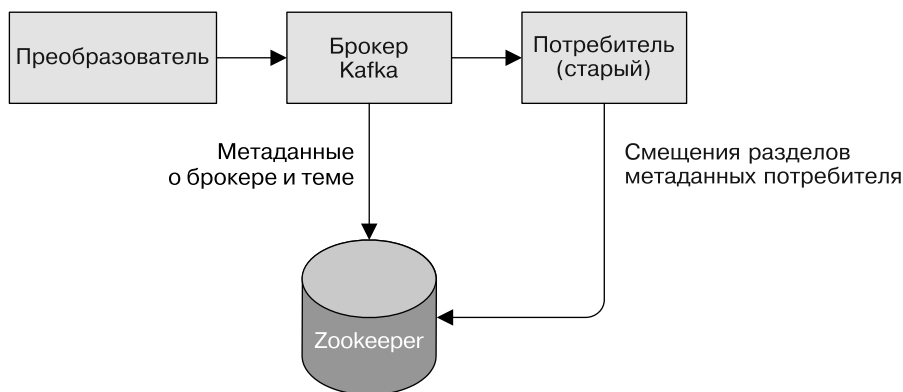


Рис. 2.1. Kafka и ZooKeeper

Kafka была тщательно протестирована со стабильной версией 3.4.6 хранилища ZooKeeper, которую можно скачать с сайта apache.org по адресу <http://bit.ly/2sDWSgJ>.

Автономный сервер

Следующий пример демонстрирует установку ZooKeeper с базовыми настройками в каталог `/usr/local/zookeeper` с сохранением данных в каталоге `/var/lib/zookeeper`:

```
# tar -zxf zookeeper-3.4.6.tar.gz
# mv zookeeper-3.4.6 /usr/local/zookeeper
# mkdir -p /var/lib/zookeeper
# cat > /usr/local/zookeeper/conf/zoo.cfg << EOF
> tickTime=2000
> dataDir=/var/lib/zookeeper
> clientPort=2181
> EOF
# export JAVA_HOME=/usr/java/jdk1.8.0_51
```

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

Теперь можете проверить, что ZooKeeper как полагается работает в автономном режиме, подключившись к порту клиента и отправив четырехбуквенную команду `srvr`:

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

Ансамбль ZooKeeper

Кластер ZooKeeper называется *ансамблем* (ensemble). Из-за особенностей самого алгоритма рекомендуется, чтобы ансамбль включал нечетное число серверов, например, 3, 5 и т. д., поскольку для того, чтобы ZooKeeper мог отвечать на запросы, должно функционировать большинство членов ансамбля (кворум). Это значит, что ансамбль из трех узлов может работать и при одном неработающем узле. Если в ансамбле три узла, 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, данные 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.

Для настройки работы 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 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=hostname:peerPort:LeaderPort` со следующими параметрами:

- ❑ `X` — идентификатор сервера. Обязан быть целым числом, но отсчет может вестись не от нуля и не быть последовательным;
- ❑ `hostname` — имя хоста или IP-адрес сервера;
- ❑ `peerPort` — TCP-порт, через который серверы ансамбля взаимодействуют друг с другом;
- ❑ `LeaderPort` — TCP-порт, через который осуществляется выбор ведущего узла.

Достаточно, чтобы клиенты могли подключаться к ансамблю через порт `clientPort`, 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 шагов можно запустить серверы, и они будут взаимодействовать друг с другом в ансамбле.

Установка брокера Kafka

После завершения настройки Java и ZooKeeper можно приступить к установке Apache Kafka. Актуальный выпуск Apache Kafka можно скачать по адресу <http://kafka.apache.org/downloads.html>. На момент публикации данной книги это версия 0.9.0.1, работающая под управлением Scala 2.11.0¹.

¹ На момент выхода из печати русского издания — 1.1.1 и Scala 2.12.6 соответственно. — *Примеч. пер.*

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 -zxvf kafka_2.11-0.9.0.1.tgz
# mv kafka_2.11-0.9.0.1 /usr/local/kafka
# mkdir /tmp/kafka-logs
# export JAVA_HOME=/usr/java/jdk1.8.0_51
# /usr/local/kafka/bin/kafka-server-start.sh -daemon
/usr/local/kafka/config/server.properties
#
```

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

Создание и проверка темы:

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

Генерация сообщений для темы `test`:

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

Потребление сообщений из темы `test`:

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

Конфигурация брокера

Пример конфигурации брокера, поставляемый вместе с дистрибутивом Kafka, вполне подойдет для пробного запуска автономного сервера, но для большинства установок его будет недостаточно. Существует множество параметров конфигурации Kafka, регулирующих все аспекты установки и настройки. Для многих из

них можно оставить значения по умолчанию, поскольку они относятся к нюансам настройки брокера Kafka, не применяемым до тех пор, пока вы не будете работать с требующим их использования конкретным сценарием.

Основные настройки брокера

Существует несколько настроек 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 обязательно поменять, чтобы брокер мог работать в кластере с другими брокерами.

broker.id

У каждого брокера Kafka должен быть целочисленный идентификатор, задаваемый посредством параметра `broker.id`. По умолчанию это значение равно 0, но может быть любым числом. Главное, чтобы оно не повторялось в пределах одного кластера Kafka. Выбор числа может быть произвольным, причем при необходимости ради удобства сопровождения его можно переносить с одного брокера на другой. Желательно, чтобы это число 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 (например, `host1.example.com`, `host2.example.com` и т. д.), эти числа будут удачным выбором для значений `broker.id`.

port

Типовой файл конфигурации запускает Kafka с прослушивателем на TCP-порту 9092. Этот порт можно изменить на любой другой доступный путем изменения параметра конфигурации `port`. Имейте в виду, что при выборе порта с номером менее 1024 Kafka должна запускаться от имени пользователя `root`. А запускать Kafka от имени пользователя `root` не рекомендуется.

zookeeper.connect

Путь, который ZooKeeper использует для хранения метаданных брокеров, задается с помощью параметра конфигурации `zookeeper.connect`. В образце конфигурации ZooKeeper работает на порту 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, используется корневой путь.

Если заданный путь `chroot` 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. Лучше 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.dirs`. Она представляет собой разделенный запятыми список путей в локальной системе. Если задано несколько путей, брокер будет сохранять разделы в них по принципу наименее используемых, с сохранением сегментов журналов одного раздела 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 пространства, так что 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 выгоды от применения такого подхода могут достичь нескольких часов в случае перезапуска брокера с большим числом разделов! 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 тему, когда:

- ☐ производитель начинает писать в тему сообщения;
- ☐ потребитель начинает читать из темы сообщения;
- ☐ любой клиент запрашивает метаданные темы.

Во многих случаях такое поведение может оказаться нежелательным, особенно из-за того, что не существует возможности проверить по протоколу Kafka существование темы, не вызвав ее создания. Если вы управляете созданием тем явным образом, вручную или посредством системы инициализации, то можете установить для параметра `auto.create.topics.enable` значение `false`.

Настройки тем по умолчанию

Конфигурация сервера Kafka задает множество настроек по умолчанию для создаваемых тем. Некоторые из этих параметров, включая число разделов и параметры сохранения сообщений, можно задавать для каждой темы отдельно с помощью инструментов администратора (рассматриваются в главе 9). Значения по умолчанию в конфигурации сервера следует устанавливать равными эталонным значениям, подходящим для большинства тем кластера.



Индивидуальное переопределение значений для каждой темы

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

num.partitions

Параметр `num.partitions` определяет, с каким количеством разделов создается новая тема, главным образом в том случае, когда включено автоматическое создание тем (что является поведением по умолчанию). Значение этого параметра по умолчанию — 1. Имейте в виду, что количество разделов для темы можно лишь увеличивать, но не уменьшать. Это значит, что если для нее требуется меньше разделов, чем указано в `num.partitions`, придется аккуратно создать ее вручную (это обсуждается в главе 9).

Как говорилось в главе 1, разделы представляют собой способ масштабирования тем 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 приведет к равномерному распределению нагрузки по сообщениям. Однако это не обязательное требование, ведь и наличие нескольких тем позволяет выравнивать нагрузку.



Как выбрать количество разделов

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

- Какой пропускной способности планируется достичь для темы? Например, планируете вы записывать 100 Кбайт/с или 1 Гбайт/с?
- Какая максимальная пропускная способность ожидается при потреблении сообщений из отдельного раздела? Из каждого раздела всегда будет читать не более чем один потребитель¹, так что если знать, что потребитель записывает данные в базу, которая не способна 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 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 из планируемого в будущем объема использования, а не текущего.
- 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 на выбор ведущего узла.

С учетом всего этого ясно, что разделов должно быть много, но не слишком много. Если у вас есть предварительные оценки целевой пропускной способности для темы и ожидаемой пропускной способности потребителей, можно получить требуемое число разделов путем деления целевой пропускной способности на ожидаемую пропускную способность потребителей. Так что если необходимо читать из темы 1 Гбайт/с и записывать столько же и мы знаем, что каждый потребитель способен обрабатывать лишь 50 Мбайт/с, то нам нужно как минимум 20 разделов. Таким образом, из темы 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.

¹ Имеется в виду, что раздел в целом должен быть прочитан одним потребителем или группой потребителей. — *Примеч. пер.*

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`. Все эти три параметра определяют одно и то же — промежуток времени, по истечении которого сообщения удаляются. Но рекомендуется использовать параметр `log.retention.ms`, ведь в случае указания нескольких параметров приоритет принадлежит наименьшей единице 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 осуществляется 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 9, 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 темы из восьми разделов и равного 1 Гбайт значения `log.retention.bytes` максимальный объем сохраняемых для этой темы данных будет 8 Гбайт. Отметим, что объем сохранения зависит от отдельных разделов, а не от темы. Это значит, что в случае увеличения числа разделов для темы максимальный объем сохраняемых при использовании `log.retention.bytes` данных также возрастет.



Настройка сохранения по размеру и времени

Если задать значения параметров `log.retention.bytes` и `log.retention.ms` (или любого 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.

log.segment.bytes

Упомянутые настройки сохранения журналов касаются сегментов журналов, а не отдельных сообщений. По мере генерации сообщений брокером Kafka они 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 темы отличаются низкой частотой генерации сообщений. Например, если в тему 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.segment.ms

Другой способ управления закрытием сегментов журнала — с помощью параметра `log.segment.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.segment.ms` не являются взаимоисключающими. Kafka закрывает сегмент журнала, когда 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.segment.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 время для этих разделов истечет тоже одновременно.

message.max.bytes

Брокер Kafka позволяет с помощью параметра `message.max.bytes` ограничивать максимальный размер генерируемых сообщений. Значение этого параметра по умолчанию равно 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. Также большие сообщения увеличивают объем записываемых на диск данных, что влияет на пропускную способность ввода/вывода.



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 каких-либо строгих требований к аппаратному обеспечению, она будет работать без проблем на любой системе. Но если говорить о производительности, то на нее влияют несколько факторов: емкость и пропускная способность дисков, оперативная память, сеть и CPU.

Сначала нужно определиться с тем, какие типы производительности важнее всего для вашей системы, после чего можно будет выбрать оптимальную конфигурацию аппаратного обеспечения, вписывающуюся в бюджет.

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 обладают более высокой относительной емкостью. 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, например, технология изготовления жесткого диска (к примеру, SAS или SATA), а также характеристики контроллера жесткого диска.

Емкость диска

Емкость — еще один аспект хранения. Необходимый объем дискового пространства определяется тем, сколько сообщений необходимо хранить одновременно. Если ожидается, что брокер будет получать 1 Тбайт трафика в день, то при 7-дневном 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 роста его с течением времени.

Емкость хранилища — один из факторов, которые необходимо учитывать при определении оптимального размера кластера Kafka и принятии решения о его расширении. Общий трафик кластера можно балансировать за счет нескольких разделов для каждой темы, 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 6).

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 выделения для JVM большого объема оперативной памяти в куче. Даже брокер, который обрабатывает X сообщений в секунду при скорости передачи данных X мегабит в секунду, может работать с кучей в 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 им 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. Зачастую это ключевой (наряду с объемом дискового хранилища) фактор выбора размера кластера. Затрудняет этот выбор присущий Kafka (вследствие поддержки нескольких потребителей) дисбаланс между входящим и исходящим сетевым трафиком. Производитель может генерировать 1 Мбайт сообщений в секунду для заданной темы, 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 6) и зеркальное копирование (обсуждается в главе 8). При интенсивном использовании сетевого интерфейса вполне возможно отставание репликации кластера, что вызовет неустойчивость его состояния.

CPU

Вычислительные мощности не так важны, как дисковое пространство и оперативная память, 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. Затем 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.

Kafka в облачной среде

Kafka часто устанавливают в облачной вычислительной среде, например, Amazon Web Services (AWS). AWS предоставляет множество виртуальных вычислительных узлов, все с различными комбинациями CPU, оперативной памяти и дискового пространства. Для выбора подходящей конфигурации виртуального узла нужно учесть в первую очередь факторы производительности Kafka. Начать можно с необходимого объема хранения данных, после чего учесть требуемую производительность генераторов. В случае потребности в очень низкой задержке могут понадобиться виртуальные узлы с оптимизацией по операциям ввода/вывода с локальными хранилищами на основе SSD. В противном случае может оказаться достаточно удаленного хранилища (например, AWS Elastic Block Store). После принятия этих решений можно будет выбрать из числа доступных вариантов CPU и оперативной памяти.

На практике это означает, что в случае задействования AWS можно выбрать виртуальные узлы типов 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.

Кластеры Kafka

Отдельный сервер 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 кластера Kafka. Более подробную информацию о репликации данных вы найдете в главе 6.

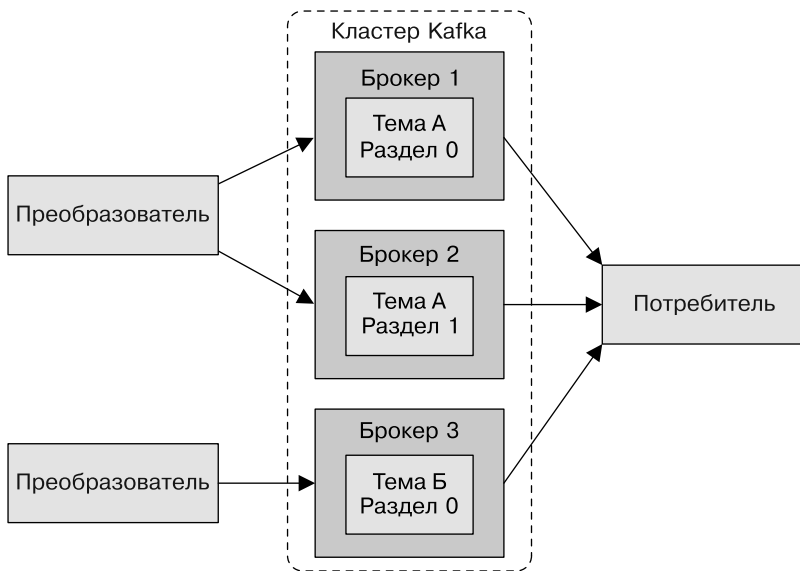


Рис. 2.2. Простой кластер Kafka

Сколько должно быть брокеров?

Размер кластера Kafka определяется несколькими факторами. Первый из них — 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, использование репликации может повысить требования к хранилищу минимум на 100 % (в зависимости от ее коэффициента) (см. главу 6). Это значит, что при использовании репликации тот же кластер должен будет содержать как минимум десять брокеров.

Еще один фактор, который нужно учесть, — возможности кластера по обработке запросов. Например, каковы возможности сетевых интерфейсов и способны ли они справиться с трафиком клиентов 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. Может быть полезно 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 при их работе 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 ее к специфике нагрузки Kafka.

Как и для большинства приложений, особенно тех, где важна пропускная способность, лучше 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 — 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 памяти воспользуется подкачкой вместо удаления страниц из страничного кэша. Лучше уменьшить размер страничного кэша, чем использовать подкачку.



Почему бы не сделать параметр `swappiness` равным 0?

Ранее рекомендовалось всегда задавать параметр `vm.swappiness` равным 0. Смысл этого значения был таков: никогда не использовать 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 использовать 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 и увеличением параметра `vm.dirty_ratio` до значения, превышающего 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 является несколько рискованным 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 3875
nr_writeback 29
nr_writeback_temp 0
#
```

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 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 также появилось отложенное выделение блоков, повышающее вероятность потери данных и повреждения файловой системы в случае системного отказа. В файловой системе XFS также используется алгоритм отложенного выделения, но более безопасный, чем в EXT4. Производительность XFS для типичной нагрузки Kafka тоже выше, причем нет необходимости производить 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`.

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 путем установки значения 1 параметра `net.ipv4.tcp_window_scaling`, что позволит клиентам передавать данные эффективнее и обеспечит возможность буферизации этих данных на стороне брокера. Значение параметра `net.ipv4.tcp_max_syn_backlog` большее, чем принятое по умолчанию 1024, позволяет повысить 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 количества пакетов, помещаемых в очередь для последующей обработки ядром.

Промышленная эксплуатация

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

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 (G1). G1 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 G1 требуются минимальные настройки. Для корректировки его производительности используются два параметра.

- ❑ `MaxGCPauseMillis`. Задаёт желаемую длительность паузы на каждый цикл сборки мусора. Это не фиксированный максимум — при необходимости G1 может превысить эту длительность. По умолчанию данное значение равно 200 мс. Это значит, что G1 будет стараться планировать частоту циклов сборки 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 сборка мусора не начинается. Значение по умолчанию равно 45. Это значит, что G1 не запустит цикл сборки мусора до того, как будет использоваться 45 % кучи, включая суммарное использование 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 размером 5 Гбайт. Этот брокер мог работать при значении 20 параметра `MaxGCPauseMillis`. А значение параметра `InitiatingHeapOccupancyPercent` установлено в 35, благодаря чему сборка мусора запускается несколько раньше, чем при значении по умолчанию.

Сценарий запуска Kafka по умолчанию использует не сборщик мусора G1, а новый параллельный сборщик мусора и конкурентный сборщик мусора маркировки и очистки. Это можно легко изменить посредством переменных среды. Изменим приведенную ранее команду запуска следующим образом:

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

Планировка ЦОД

При использовании ориентированных на разработку систем 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 выдаче данных означает потерю денег из-за невозможности или обслуживания поль-

зователей, или получить телеметрию их действий. При этом возрастает важность использования репликации в кластере Kafka (см. главу 6), а также физического расположения брокеров в стойках в ЦОД. Если не позаботиться об этом до разворачивания Kafka, могут потребоваться дорогостоящие работы по перемещению серверов.

Брокер Kafka ничего не знает о размещении по стойкам во время назначения новых разделов брокерам, а значит, он не способен учесть возможное расположение двух брокеров в одной физической стойке или в одной и той же зоне доступности (при работе в облачном сервисе, например, AWS), вследствие чего может случайно поставить все реплики раздела в соответствие брокерам, использующим в одной стойке одни и те же сетевые и силовые подключения. В случае отказа этой стойки разделы окажутся недоступными для клиентов. Кроме того, в результате «нечистых» выборов ведущего узла это может привести к дополнительной потере данных на восстановление (см. подробности в главе 6).

Рекомендуемая практика: 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 как минимум использование для брокеров серверов с резервным питанием (подключение к двум разным цепям электропитания) и двойных сетевых коммутаторов с объединенным интерфейсом к самим серверам для переключения на другой интерфейс без перебоев в работе. 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, темах и разделах. Запись в ZooKeeper выполняют только при изменении списков участников групп потребителей или изменениях в самом кластере Kafka. Объем трафика при этом минимален, так что использование выделенного ансамбля ZooKeeper для одного кластера Kafka не обоснованно. 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 и ZooKeeper

До Apache Kafka 0.9.0.0 потребители использовали не только брокеры, но и ZooKeeper для непосредственного сохранения информации о составе групп потребителей и соответствующих темах, 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 появился новый интерфейс потребителей, благодаря которому стало возможно делать это непосредственно с помощью брокеров Kafka. Он описан в главе 4.

Однако при работе потребителей и ZooKeeper с определенными настройками есть нюанс. Для фиксации смещений потребители могут использовать или ZooKeeper, или Kafka, причем интервал между фиксациями можно настраивать. Если потребители применяют ZooKeeper для смещений, то каждый потребитель будет производить операцию записи ZooKeeper через заданное время для каждого потребляемого им раздела. Обычный промежуток времени для фиксации смещений — 1 минута, 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 понадобиться увеличить интервал фиксации. Однако рекомендуется, чтобы работающие с актуальными библиотеками Kafka потребители использовали Kafka для фиксации смещений и не зависели от ZooKeeper.

Не считая 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 другие приложения, создающие нагрузку на диспетчер кластера в результате активного использования или неправильного функционирования, в отдельные ансамбли.

Резюме

В этой главе мы поговорили о том, как установить и запустить Apache Kafka. Рассмотрели, 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 дальнейшего потребления (глава 4).

3

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, служащих 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 Kafka.



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. Рассмотрение протокола передачи данных и внешних клиентов выходит за рамки текущей 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 использования означают также разнообразие требований: каждое ли сообщение критично 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 критично 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 допустима 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 по ссылке. Пропускная способность зависит от ожидаемого уровня пользовательской активности на сайте.

Различие в требованиях влияет на использование 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.

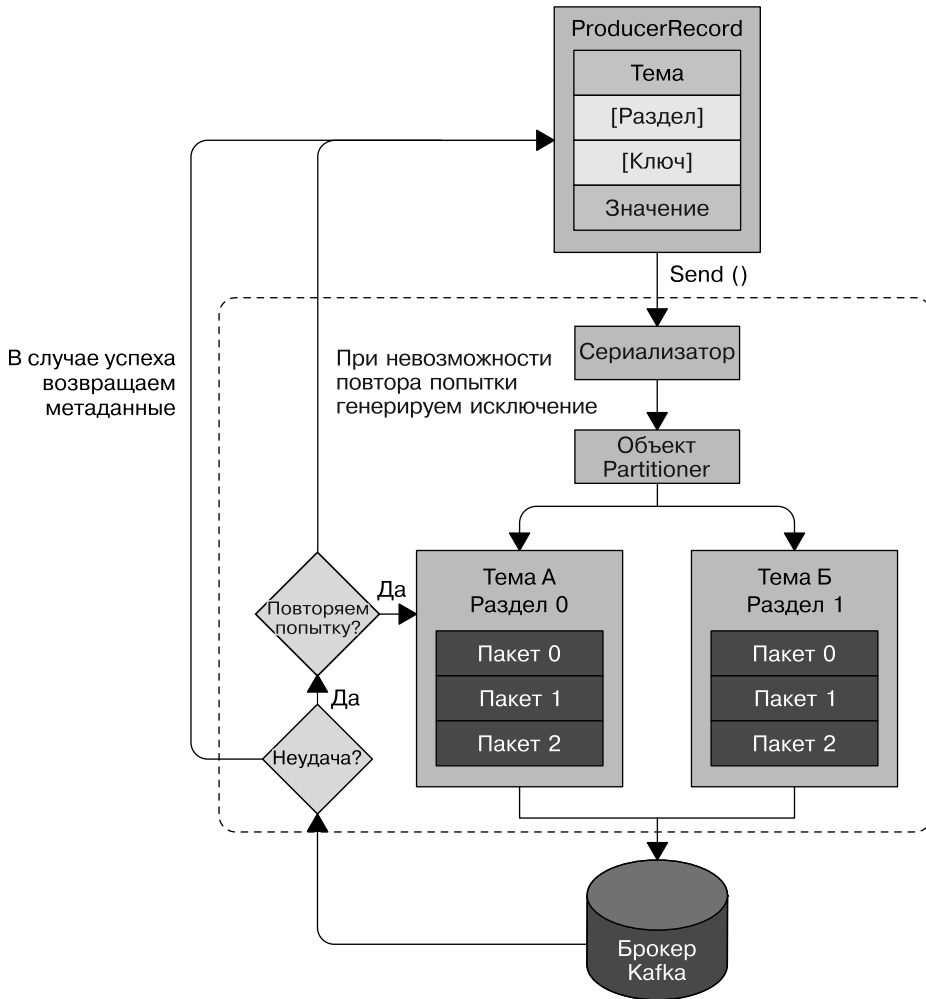


Рис. 3.1. Высокоуровневый обзор компонентов производителей Kafka

Для генерации сообщений для Kafka нам понадобится сначала создать объект `ProducerRecord`, включающий тему, в которую мы собираемся отправить запись, и значение. При желании можно задать также ключ и раздел. После отправки объекта `ProducerRecord` он прежде всего сериализует объекты ключа и значения в байтовые массивы для отправки по сети.

Далее данные попадают в объект `Partitioner`. Если в `ProducerRecord` был указан раздел, объект `Partitioner` ничего не делает и просто возвращает указанный раздел. Если же нет, он выбирает раздел, обычно в соответствии с ключом из `ProducerRecord`. Если раздел выбран, производитель будет знать, в какую тему и раздел должна попасть запись. После этого он помещает эту запись в пакет записей, предназначенных для отправки в соответствующие тему и раздел. За отправку этих 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 тему, 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 используются 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` и `IntegerSerializer`, 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;

- ❑ `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:

```
private 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 заданием соответствующих параметров конфигурации. В документации Apache Kafka описаны все параметры конфигурации. 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 часть сообщений при использовании этого метода теряется.
- ❑ *Синхронная отправка.* При отправке сообщения метод `send()` возвращает объект-фьючерс (объект класса `Future`), после чего можно воспользоваться методом `get()` для организации ожидания и выяснения того, успешно ли прошла отправка.
- ❑ *Асинхронная отправка.* Методу `send()` передается функция обратного вызова, которая вызывается при получении ответа от брокера Kafka.

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

Хотя все примеры в этой главе однопоточны, объект производителя может использоваться для отправки сообщений несколькими потоками. Лучше всего начать с одного производителя и одного потока. Чтобы повысить пропускную способность, можно будет добавить дополнительные потоки, использующие тот же производитель. А когда этот метод перестанет приносить плоды — добавить в приложение дополнительные производители для еще большего повышения пропускной способности.

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 темы, в которую отправляются данные, и отсылаемые в Kafka ключ и значение (тоже строки). Типы ключа и значения должны соответствовать объектам `serializer` и `producer`.
- 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://www.bit.ly/2rG7Cg6>), включающий `RecordMetadata`, но поскольку мы просто игнорируем возвращаемое 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. Это может быть исключение `SerializationException` при неудач-

ной сериализации сообщения, `BufferExhaustedException` или `TimeoutException` при переполнении буфера, `InterruptedException` при сбое отправляющего потока.

Синхронная отправка сообщения

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.
- ❷ Если перед отправкой данных в Kafka или во время нее возникли ошибки, если брокер Kafka вернул сообщение об ошибке, исключающие повтор отправки или лимит на повторы исчерпан, 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» может быть исправлена выбором нового ведущего узла для раздела. 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 вернет исключение сразу же.

Асинхронная отправка сообщения

Пусть время прохождения сообщения между нашим приложением и кластером Kafka и обратно равно 10 мс. Если мы будем ждать ответа после отправки каждого сообщения, отправка 100 сообщений займет около 1 секунды. В то же время, если не ожидать ответов, отправка всех сообщений не займет практически никакого времени. В большинстве случаев ответ и не требуется — Kafka возвращает тему,

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.

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, большинство из которых описаны в документации Apache Kafka (<http://kafka.apache.org/documentation.html#producerconfigs>). Значения по умолчанию многих из них вполне разумны, так что нет смысла возиться с каждым параметром. Однако некоторые из этих параметров существенно влияют на использование памяти, производительность и надежность производителей. Рассмотрим их.

acks

Параметр `acks` задает число получивших сообщение реплик разделов, требующееся для признания производителем операции записи успешной. Этот параметр существенно влияет на вероятность потери сообщений. Он может принимать три значения.

- ❑ При `acks=0` производитель не ждет 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.
- ❑ При `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, а в качестве нового ведущего узла была выбрана реплика, не содержащая этого сообщения (из-за «нечистых» выборов ведущего узла). В этом случае пропускная способность зависит от того, отправляем мы сообщения синхронно или асинхронно. Ожидание ответа от сервера клиентским кодом (посредством вызова метода `get()`, возвращаемого при отправке сообщения объекта `Future`), очевидно, существенно повысит длительность задержки (по крайней мере на время перемещения данных по сети в обе стороны). При использовании клиентом функций обратного вызова задержка будет незаметна, но пропускная способность окажется ограничена числом передаваемых в данный момент сообщений, то есть числом сообщений, которое производитель может отправить до получения ответов от сервера.
- ❑ При `acks=all` производителю приходит ответ от брокера об успешном получении сообщения после того, как оно дойдет до всех синхронизируемых реплик. Это самый безопасный режим, поскольку есть уверенность, что сообщение получено более чем одним брокером и оно не пропадет даже в случае аварийного сбоя (больше информации по этому вопросу вы найдете в главе 5). Однако в случае `acks=1` задержка будет еще выше, ведь придется ждать получения сообщения более чем одним брокером.

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

исчерпаться свободное место и дополнительные вызовы метода `send()` приведут или к блокировке, или генерации исключения в зависимости от значения параметра `block.on.buffer.full`. (В версии 0.9.0.0 он заменен параметром `max.block.ms`, предоставляющим возможность блокировки на заданное время с последующей генерацией исключения.)

compression.type

По умолчанию сообщения отправляются в несжатом виде. Возможные значения этого параметра — `snappy`, `gzip` и `lz4`, 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 использует больше ресурсов CPU и занимает больше времени, но обеспечивает лучшую степень сжатия. Поэтому его рекомендуется использовать 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.

retries

Производитель получил от сервера сообщение об ошибке. Возможно, она окажется временной, например, отсутствует ведущий узел для раздела. В таком случае значение параметра `retries` будет определять число предпринятых производителем попыток отправить сообщение, по достижении которого он прекратит это делать и известит клиента о проблеме. По умолчанию производитель ждет 100 мс перед следующей попыткой, но это поведение можно изменить с помощью параметра `retry.backoff.ms`. Мы рекомендуем проверить, сколько времени занимает восстановление после аварийного сбоя брокера (то есть сколько времени займет выбор новых ведущих узлов для всех разделов), и задать такие количество повторов и задержку между ними, чтобы общее время, потраченное на повторы, превышало время восстановления кластера Kafka после сбоя, иначе производитель прекратит попытки повтора слишком рано. Не при всех ошибках производителю имеет смысл пытаться повторять отправку. Некоторые ошибки носят не временный характер, и при их возникновении производитель не будет пытаться повторить отправку (например, такова ошибка «сообщение слишком велико»). В целом, поскольку повторами отправки занимается производитель, нет смысла делать это в коде приложения. Лучше сосредоточить усилия на обработке ошибок, которые нельзя исправить путем повтора отправки, или случаев, когда число попыток повтора было исчерпано без результата.

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 наполовину полные пакеты 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. Задание слишком маленького размера пакета обусловит дополнительный расход памяти, поскольку производителю придется отправлять сообщения чаще.

linger.ms

Параметр `linger.ms` управляет длительностью ожидания дополнительных сообщений перед отправкой текущего пакета. `KafkaProducer` отправляет пакет сообщений или при наполнении текущего пакета, или по достижении лимита `linger.ms`. По умолчанию производитель будет отправлять сообщения, как только появится свободный поток для этого, даже если в пакете содержится лишь одно сообщение. Устанавливая параметр `linger.ms` в значение больше 0, мы указываем производителю на необходимость подождать несколько миллисекунд, чтобы перед отправкой пакета брокерам в него были добавлены дополнительные сообщения. Это повышает длительность задержки, но повышает и пропускную способность, поскольку чем больше сообщений отправляются одновременно, тем меньше накладные расходы из расчета на одно сообщение.

client.id

Значение этого параметра может быть любой строкой. Его впоследствии будут использовать брокеры для идентификации отправленных клиентом сообщений. Применяется для журналирования и показателей, а также задания квот.

max.in.flight.requests.per.connection

Управляет количеством сообщений, которые производитель может отправить серверу, не получая ответов. Высокое значение этого параметра приведет к повышенному использованию памяти, но одновременно к увеличению пропускной способности. Однако слишком высокое значение уменьшит пропускную способность вследствие снижения эффективности пакетной обработки. Равное 1 значение этого параметра гарантирует запись сообщений в брокер в порядке их отправки даже в случае повторов отправки.

`timeout.ms`, `request.timeout.ms` и `metadata.fetch.timeout.ms`

Эти параметры управляют длительностью ожидания производителем ответа от сервера при отправке данных (`request.timeout.ms`) и запросе метаданных, например, текущих ведущих узлов разделов, в которые производится запись (`metadata.fetch.timeout.ms`). Если время ожидания истекло, а ответ получен не был, производитель или повторит попытку отправки, или вернет ошибку посредством генерации исключения или функции обратного вызова. Параметр `timeout.ms` управляет длительностью ожидания брокером от синхронизируемых реплик подтверждения того, что сообщение получено, в соответствии с параметром `acks`. Брокер вернет ошибку, если время ожидания истечет, а подтверждения получены не будут.

`max.block.ms`

Управляет длительностью блокировки при вызове производителем метода `send()` и запросе явным образом метаданных посредством вызова `partitionsFor()`. Эти методы выполняют блокировку при переполнении буфера отправки производителя или недоступности метаданных. По истечении времени ожидания `max.block.ms` генерирует соответствующее исключение.

`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 как максимум отправить 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.



Гарантии упорядоченности

Apache Kafka сохраняет порядок сообщений внутри раздела. Это значит, что брокер запишет в раздел сообщения, отправленные из производителя в определенном порядке, в том же порядке и все потребители будут читать их. Для некоторых сценариев использования упорядоченность играет важную роль. Помещение на счет 100 долларов и их снятие в дальнейшем сильно отличается от снятия с последующим помещением! Однако для некоторых сценариев использования порядок не имеет значения.

Установка для параметра `retries` ненулевого значения, а для `max.in.flights.requests.per.connection` — значения, превышающего 1, означает, что брокер, которому не удалось записать первый пакет сообщений, сможет успешно записать второй (уже отправленный), после чего вернуться к первому и успешно повторить попытку его записи, в результате чего порядок будет изменен на противоположный.

Обычно в надежной системе нельзя взять и установить нулевое число повторов, так что если обеспечение упорядоченности критично важно, рекомендуем задать `max.in.flight.requests.per.connection=1`, чтобы гарантировать, что во время повтора попытки не будут отправляться другие сообщения, поскольку это может привести к изменению порядка на противоположный и существенному ограничению пропускной способности производителя, поэтому так следует поступать лишь тогда, когда порядок важен.

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 и байтовых массивов, 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 либо Protobuf, или создать пользовательский сериализатор для уже используемых объектов. Мы настоятельно рекомендуем вам воспользоваться универсальной библиотекой сериализации. Но чтобы разобраться, как работают сериализаторы и почему лучше использовать 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 мы столкнемся 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 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 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 номер факса — необязателен, по умолчанию это неопределенное значение.

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 будем использовать 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: хотя 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://www.bit.ly/2t9FmEb>).
- ❑ У десериализатора должен быть доступ к схеме, использованной 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 воспользуемся Confluent Schema Registry. Код Confluent Schema Registry можно найти на GitHub (<https://github.com/confluentinc/schema-registry>), его также можно установить в виде части платформы Confluent (<http://docs.confluent.io/current/installation.html>). Если вы решите использовать этот реестр схем, рекомендуем заглянуть в его документацию (<http://docs.confluent.io/current/schema-registry/docs/index.html>).

Идея 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. 3.2.

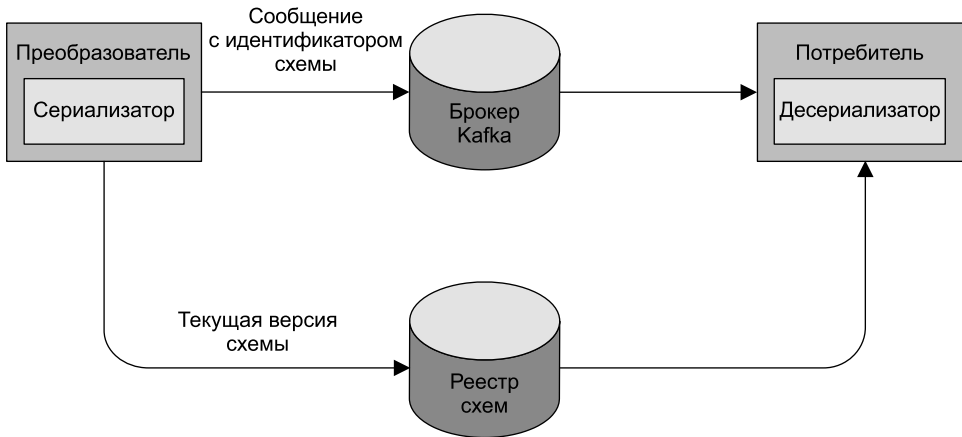


Рис. 3.2. Блок-схема сериализации и десериализации записей Avro

Вот пример отправки сгенерированных Avro объектов в Kafka (см. документацию Avro по адресу <http://avro.apache.org/docs/current/>, чтобы получить информацию о генерации кода с его помощью):

```

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";
int wait = 500;

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

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

- ❶ Для сериализации объектов с помощью Avro мы используем класс `KafkaAvroSerializer`. Обратите внимание на то, что `AvroSerializer` может работать с простыми типами данных, именно поэтому в дальнейшем объект `String` используется в качестве ключа записи и объект `Customer` — в качестве значения.
- ❷ `schema.registry.url` — новый параметр, указывающий на место хранения схем.
- ❸ `Customer` — сгенерированный объект. Мы сообщаем производителю, что значение наших записей будет представлять собой объект `Customer`.
- ❹ Мы также создаем экземпляр класса `ProducerRecord` с объектом `Customer` в качестве типа значения и передаем объект `Customer` при создании новой записи.
- ❺ Вот и все. Мы отправили запись с объектом `Customer`, а `KafkaAvroSerializer` позаботится обо всем остальном.

Но что если нам понадобится использовать обобщенные объекты Avro вместо сгенерированных объектов Avro? Никаких проблем. В этом случае нужно всего лишь указать 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<String,
            GenericRecord>("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 Avro, поскольку ее уже не предоставляет сгенерированный Avro 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 темы, ключ и значение. Сообщения Kafka представляют собой пары «ключ/значение», и хотя можно создавать объекты `ProducerRecord` только с темой и значением или с неопределенным значением по умолчанию для ключа, большинство приложений отправляют записи с ключами. Ключи служат для двух целей: они представляют собой дополнительную информацию, сохраняемую вместе с сообщением, и на их основе определяется, в какой раздел темы записывать сообщение. Все сообщения с одинаковым ключом попадут в один раздел. Это значит, что, если каждый процесс читает лишь часть разделов темы (подробнее об этом в главе 4), все записи для конкретного ключа будет читать один и тот же процесс. Для создания записи типа «ключ/значение» нужно просто создать объект `ProducerRecord`, вот так:

```

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

```

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

```

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

```

- ❶ В этом примере ключ будет равен `null`, что может указывать, например, на то, что имя покупателя в форме опущено.

Если ключ равен `null` и используется метод секционирования по умолчанию, то запись будет отправлена в один из доступных разделов темы случайным образом.

Для балансировки сообщений по разделам при этом будет использоваться циклический алгоритм.

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 темы, 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 6, когда мы будем обсуждать репликацию и доступность Kafka.

Соответствие ключей разделам остается согласованным лишь до тех пор, пока число разделов в теме 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 тему 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 темы с достаточным количеством разделов (в главе 2 мы приводили соображения по поводу выбора оптимального количества разделов) и никогда не добавлять новые.

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 покупатель — компания Banana, производящая карманные устройства. Допустим, что более 10 % ваших ежедневных транзакций приходится на этого покупателя. При использовании хеш-секционирования, принятого по умолчанию, записи Banana 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 Banana отдельный раздел, после чего применить хеш-секционирование для распределения остальных заказчиков по 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.

Старые API производителей

В этой главе мы обсудили Java клиент-производитель, включенный в пакет `org.apache.kafka.clients`. Однако в Apache Kafka все еще содержатся¹ два более старых клиента, написанных на языке Scala и включенных в пакет `kafka.producer` (часть базового модуля Kafka). Эти производители называются `SyncProducer` (в зависимости от значения параметра `acks` он может ждать подтверждения сервером получения каждого сообщения или пакета сообщений, прежде чем отправлять новые сообщения) и `AsyncProducer` (формирует пакеты сообщений в фоновом режиме, отправляет их в отдельном потоке и не предоставляет клиенту никакой информации об успехе или неудаче отправки).

Поскольку текущий производитель поддерживает оба типа поведения, намного более надежен и предоставляет разработчику гораздо бóльшие возможности кон-

¹ Начиная с версии 0.11.0, они считаются устаревшими и оставлены только из соображений совместимости. — *Примеч. пер.*

троля, мы не будем обсуждать эти старые API. Если вас интересует возможность их использования, подумайте еще раз, после чего обратитесь к документации Kafka за дополнительной информацией.

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 используемый 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 Kafka.

4 Потребители Kafka: чтение данных из Kafka

Приложения, читающие данные из Kafka, используют объект `KafkaConsumer` для подписки на темы Kafka и получения из них сообщений. Чтение данных из Kafka несколько отличается от чтения данных из других систем обмена сообщениями: некоторые принципы его работы и идеи 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 использования 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 темы Kafka, проверяющее их и записывающее результаты в другое хранилище данных. Это приложение должно будет создать объект-потребитель, подписаться на соответствующую тему 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 тему 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 тем. Необходимо, чтобы несколько потребителей могли делить между собой данные и читать из одной темы подобно тому, как несколько производителей могут писать в одну тему.

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

Рассмотрим тему T1 с четырьмя разделами. Предположим, что мы создали новый потребитель C1 — единственный потребитель в группе G1 и подписали его на тему T1. C1 будет получать все сообщения из всех четырех разделов темы (рис. 4.1).

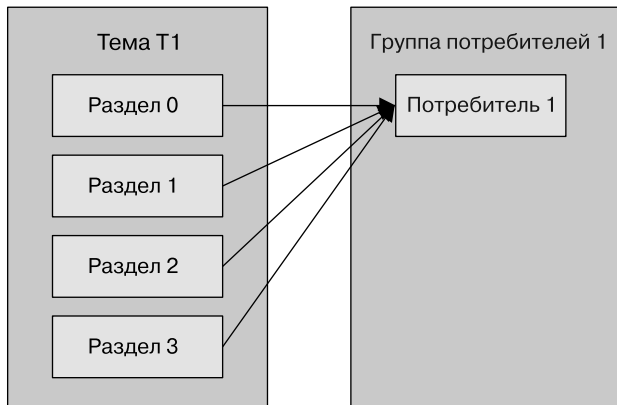


Рис. 4.1. Одна группа потребителей с четырьмя разделами

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).

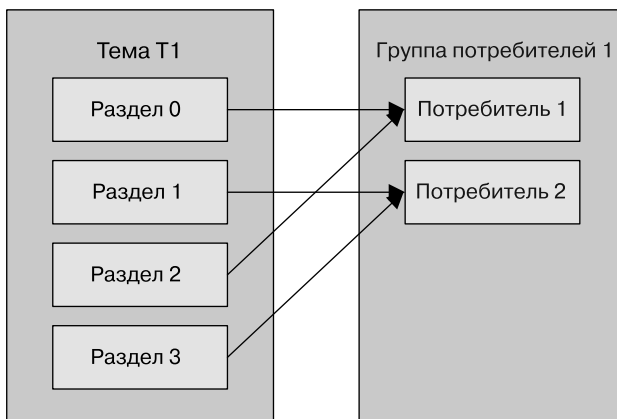


Рис. 4.2. Четыре раздела разбиты по двум потребителям

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

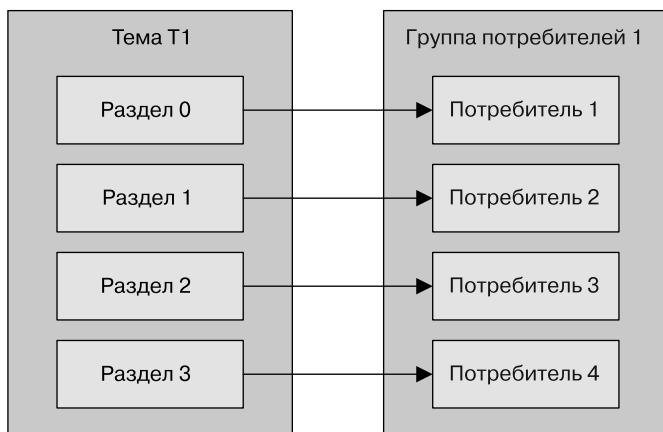


Рис. 4.3. Четыре потребителя, по одному разделу каждый

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

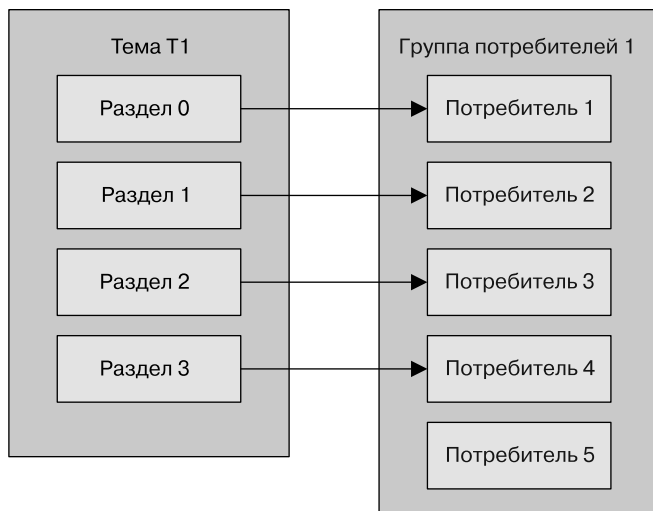


Рис. 4.4. Потребителей больше, чем разделов, поэтому часть из них простаивает

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 тему, и разделение нагрузки путем добавления новых потребителей, каждый из которых отвечает лишь за часть разделов и сообщений, — основной метод масштабирования. Поэтому имеет смысл

создавать темы с большим количеством разделов, ведь это дает возможность добавлять новых потребителей при возрастании нагрузки. Помните, что нет смысла добавлять столько потребителей, сколько нужно, чтобы их стало больше, чем разделов в теме, — часть из них будет просто простаивать. В главе 2 мы приводили соображения по поводу выбора количества разделов в теме.

Помимо добавления потребителей для масштабирования отдельного приложения широко распространено чтение несколькими приложениями данных из одной темы. На самом деле одной из главных задач создания Kafka было обеспечение возможности использования записанных в темы 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 темы, у нее 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 добавим в предыдущем примере новую группу G2 с одним потребителем, он прочитает все сообщения из темы 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).

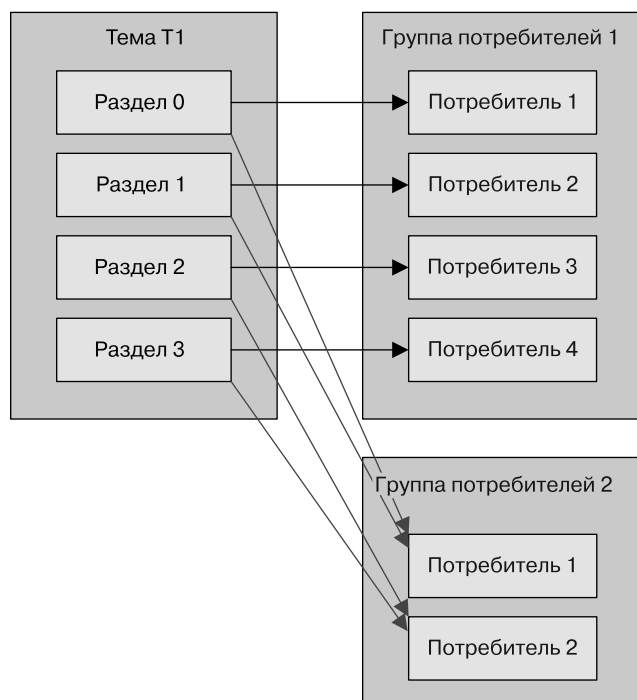


Рис. 4.5. Вторая группа потребителей тоже читает все сообщения

Резюмируем: для каждого приложения, которому нужны все сообщения из одной темы или нескольких, создается новая группа потребителей. В существующую группу их добавляют при необходимости масштабирования чтения и обработки сообщений из тем, 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 тем, 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 тем, которые читает группа (например, при добавлении администратором новых разделов).

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. Во время перебалансировки потребители не могут получать сообщения, так что она фактически представляет собой краткий интервал недоступности всей группы потребителей. Кроме того, при передаче разделов от одного потребителя другому потребитель утрачивает свое текущее состояние: если он кэшировал какие-либо данные, ему придется обновить кэши, что замедлит приложение на время восстановления состояния. В этой главе мы обсудим, как сделать так, чтобы перебалансировка не представляла опасности, и как избежать нежелательной перебалансировки.

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). До тех пор, пока потребитель регулярно отправляет контрольные сигналы, он считается активным, нормально работающим и обрабатывающим сообщения с относящихся к нему разделов. Контрольные сигналы отправляются во время опросов (то есть при извлечении потребителем записей) и при фиксации полученных им записей.

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

этом, и координатор группы 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, и их настройку подходящим для вас образом.

ИЗМЕНЕНИЯ ЛОГИКИ РАБОТЫ КОНТРОЛЬНЫХ СИГНАЛОВ В ПОСЛЕДНИХ ВЕРСИЯХ КАФКА

В версии 0.10.1 Kafka появился отдельный поток для контрольных сигналов, отправляющий их и между опросами. Он дает возможность сделать частоту контрольных сигналов (а значит, и время, которое нужно группе потребителей для обнаружения аварийного сбоя потребителя и отсутствия от него контрольных сигналов) независимой от частоты опросов (определяемой временем, необходимым для обработки возвращаемых брокерами данных). В новых версиях Kafka можно задать максимальную длительность работы потребителя без опросов, по истечении которой его признают отказавшим и будет инициирована перебалансировка. Этот параметр используется для предотвращения динамической взаимоблокировки (livelock), при которой не происходит аварийного сбоя приложения, но и задача не выполняется. Он не зависит от параметра `session.timeout.ms`, соответствующего промежутку времени, необходимому для обнаружения отказа потребителя и прекращения отправки контрольных сигналов.

Оставшаяся часть данной главы посвящена обсуждению некоторых проблем более старых версий Kafka и тому, как разработчик может с ними справиться. Мы обсудим также, что делать с приложениями, которые обрабатывают записи слишком долго. Если вы работаете с версией Kafka 0.10.1 или более новой, эта информация окажется для вас не слишком актуальной. Если при использовании одной из новых версий приходится иметь дело с записями, обработка которых занимает больше времени, просто подберите значение параметра `max.poll.interval.ms`, подходящее для более длительных промежутков между опросами.



Как происходит распределение разделов по брокерам

Когда потребитель хочет присоединиться к группе, он отправляет координатору группы запрос `JoinGroup`. Первый присоединившийся к группе потребитель становится ведущим потребителем группы. Ведущий 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 две стратегии назначения разделов, которые мы подробнее обсудим в посвященном настройке разделе. После распределения разделов ведущий потребитель группы 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.

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 мы пока сделаем вид, что является. Это свойство задает группу потребителей, к которой относится экземпляр `KafkaConsumer`. Хотя можно создавать и потребителей, не принадлежащих 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 темы

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

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

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

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

Для подписки на все темы `test` можно выполнить следующий вызов:

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

Цикл опроса

В самой основе API потребителей лежит простой цикл опроса сервера. После подписки потребителя на тему цикл опроса осуществляет координацию и перебалансировку разделов, отправляет контрольные сигналы и извлекает данные. Разработчику предоставляется чистый API, просто возвращающий доступные данные из соответствующих разделов. Основной код потребителя выглядит следующим образом:

```
try {
    while (true) {
        ❶ ConsumerRecords<String, String> records = consumer.poll(100);
        ❷ for (ConsumerRecord<String, String> record : records)
        ❸ {
            log.debug("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(4));
        }
    }
} finally {
    consumer.close();
}
  ❹
  ❺
```

- 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, возврат из метода `poll()` произойдет немедленно, в противном случае он будет ожидать поступления данных от брокера указанное число миллисекунд. Значение обычно определяется потребностью приложения в скорости откликов, то есть тем, насколько быстро нужно вернуть управление выполняющему опрос потоку.
- ❸ Метод `poll()` возвращает список записей, каждая из которых содержит тему и раздел, из которого она поступила, смещение записи в разделы 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.
- ❺ Всегда закрывайте (выполняйте операцию `close()`) потребитель перед завершением его выполнения. Эта операция приводит к закрытию сетевых подключений и сокетов. Она также инициирует немедленную перебалансировку, не дожидаясь обнаружения координатором группы того, что потребитель перестал отправлять контрольные сигналы и, вероятно, более не функционирует. Это заняло бы намного больше времени и, следовательно, увеличило период, в течение которого потребители не могут получать сообщения из некоторого подмножества разделов.

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. И конечно, в цикле опроса отправляются контрольные сигналы, подтверждающие функционирование потребителей. Поэтому необходимо гарантировать быстроту и эффективность всего кода, выполняемого между итерациями.



Потокобезопасность

Несколько потребителей, относящихся к одной группе, не могут работать в одном потоке, и несколько потоков не могут безопасно использовать один потребитель. Железное правило: один потребитель на один поток. Для работы нескольких потребителей в одной группе в одном приложении необходимо выполнять каждый из них в отдельном потоке. Не помешает обернуть логику потребителя в отдельный объект и воспользоваться объектом типа `ExecutorService` языка Java для запуска нескольких потоков, каждый — со своим потребителем. В блоге Confluent вы можете найти руководство (bit.ly/2tfVu6O) по выполнению этих действий.

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 (<http://kafka.apache.org/documentation.html#new-consumerconfigs>). 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. Если брокеру поступает запрос на записи от потребителя, но новые записи оказываются на несколько байт 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 темах 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.

`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 тему Kafka

данных. Если нужно ограничить потенциальную задержку (обычно из-за соглашения 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.

`max.partition.fetch.bytes`

Это свойство определяет максимальное число байт, возвращаемых сервером из расчета на один раздел. Значение по умолчанию равно 1 Мбайт. Это значит, что при возврате методом `KafkaConsumer.poll()` объекта `ConsumerRecords` объект записи будет занимать не более `max.partition.fetch.bytes` на каждый назначенный потребителю раздел. Так что если в теме 20 разделов, а потребителей — 5, каждому из них понадобится 4 Мбайт доступной для объекта `ConsumerRecords` памяти. На практике обычно приходится выделять больше памяти, поскольку в случае отказа потребителя каждому из оставшихся придется взять на себя работу с большим числом разделов. Значение `max.partition.fetch.bytes` должно быть больше максимально приемлемого для брокера размера сообщения (который задается параметром `max.message.size` конфигурации брокера), иначе брокер может выдать сообщение, которое потребитель не сможет прочитать, в результате чего зависнет. Еще один важный нюанс — количество времени, которое нужно потребителю для обработки данных. Как вы помните, потребитель должен вызывать метод `poll()` достаточно часто, чтобы не истекло время ожидания и не произошла перебалансировка. Если возвращаемый при отдельном вызове метода `poll()` объем данных очень велик, потребителю может потребоваться слишком много времени на обработку, из-за чего он не успеет начать следующую итерацию цикла опроса вовремя и время ожидания истечет. В этом случае можно или уменьшить значение `max.partition.fetch.bytes`, или увеличить время ожидания сеанса.

`session.timeout.ms`

По умолчанию потребитель может находиться вне связи с брокерами и продолжать считаться работающим не более 3 с. Если потребитель не отправляет контрольный сигнал координатору группы в течение промежутка времени, большего, 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 методом `poll()` объекта `KafkaConsumer` 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 обычно 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` меньше, 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 из-за более длительной итерации цикла опроса или сборки мусора. Задание более высокого значения `session.timeout.ms` снизит вероятность случайной перебалансировки, но и для обнаружения настоящего сбоя в этом случае потребуются больше времени.

auto.offset.reset

Этот параметр определяет поведение 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.

enable.auto.commit

Далее в этой главе мы обсудим различные варианты фиксации смещений. 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`.

partition.assignment.strategy

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

- ❑ *Диапазонная (Range)*. Каждому потребителю присваиваются последовательные подмножества разделов из тем, на которые он подписан. Так что если потребители C1 и C2 подписаны на темы T1 и T2, оба по три раздела, то потребителю C1

будут назначены разделы 0 и 1 из тем T1 и T2, а C2 — раздел 2 из тем T1 и T2. Поскольку в каждой из тем нечетное количество разделов, а распределение разделов по потребителям выполняется для каждой темы отдельно, у первого потребителя окажется больше разделов, чем у второго. Подобное происходит всегда, когда используется диапазонная стратегия распределения, а количество потребителей не делится нацело на количество разделов в каждой теме.

- ❑ *Циклическая (RoundRobin)*. Все разделы от всех подписанных тем 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 темы T1 и раздел 1 из темы T2, а C2 были бы назначены раздел 1 из темы T1 и разделы 0 и 2 из темы T2. Когда все потребители подписаны на одни и те же темы (очень распространенный сценарий), циклическое распределение дает одинаковое количество разделов у всех потребителей (или, в крайнем случае, различие в 1 раздел).

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`. 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 сообщений. Используется при журналировании и для показателей, а также при задании квот.

max.poll.records

Этот параметр задает максимальное число записей, возвращаемое при одном вызове метода `poll()`. Он удобен для управления объемом данных, обрабатываемым приложением в цикле опроса.

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.

Фиксация и смещения

Метод `poll()` при вызове возвращает записанные в Kafka данные, еще не прочитанные потребителями из нашей группы. Это означает возможность отследить, 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 разделе *фиксацией* (`commit`).

Каким образом потребители фиксируют смещение? Путем отправки в специальную тему `__consumer_offsets` сообщения, содержащего 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 переконфигурирования 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. 4.6).



Рис. 4.6. Повторно обрабатываемые события

Если зафиксированное смещение превышает смещение последнего фактически обработанного клиентом сообщения, расположенные в этом промежутке сообщения будут пропущены группой потребителей (рис. 4.7).

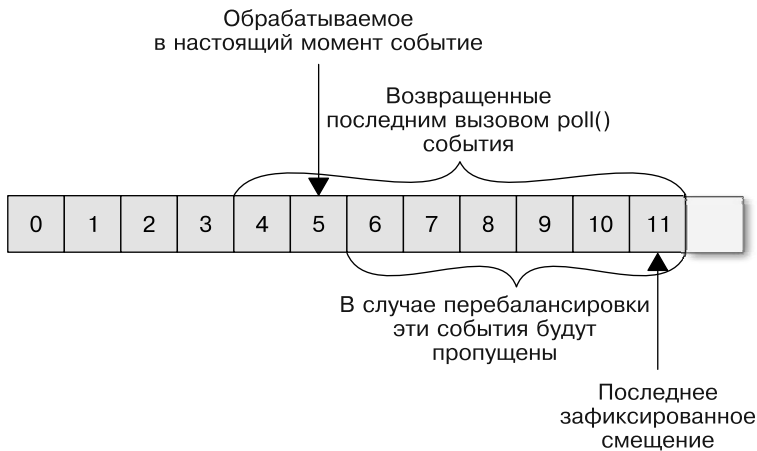


Рис. 4.7. Пропущенные события между смещениями

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

Автоматическая фиксация

Простейший способ фиксации смещений — делегировать эту задачу потребителю. При значении `true` параметра `enable.auto.commit` потребитель каждые 5 с будет автоматически фиксировать максимальное 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 время ли выполнить фиксацию, и, если да, фиксирует возвращенные при последнем опросе смещения.

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

Учтите, что по умолчанию автоматическая фиксация происходит каждые 5 с. Представьте, что после последней фиксации прошло 3 с и запустилась переконфигурация. После переконфигурации все потребители начнут 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 вызов метода `poll()` всегда 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()` также 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 дублирования сообщений при перебалансировке. В API потребителей есть возможность фиксировать текущее смещение в нужный разработчику приложения момент вместо фиксации по таймеру.

При задании параметра `auto.commit.offset=false` смещения будут фиксироваться только тогда, когда приложение потребует этого явным образом. 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()` фиксирует последнее возвращенное методом `poll()` смещение, так что не забудьте вызвать `commitSync()` после завершения обработки всех записей в наборе, или вы рискуете пропустить сообщения, как описывалось ранее. При запуске перебалансировки все сообщения с начала самого недавнего пакета и до момента начала перебалансировки окажутся обработанными дважды.

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

```
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    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, что 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 использования.
- 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 непоравимая 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, возникающих при перебалансировке.

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

```
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    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 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 смеще-

ние 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:

```
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    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 носит временный характер, 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):

```
try {
    while (true) {
        ConsumerRecords<String, String> records = consumer.poll(100);
        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
    }
} catch (Exception e) {
    log.error("Unexpected error", e);
} finally {
    try {
        consumer.commitSync(); ❷
    } 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 обрабатывали.

К счастью, API потребителей предоставляет возможность вызывать методы `commitAsync()` или `commitSync()`, передавая им ассоциативный словарь разделов и смещений, которые нужно зафиксировать. Если идет процесс обработки пакета записей и смещение последнего полученного вами из раздела 3 в теме «покупатели» сообщения равно 5000, то можете вызвать метод `commitSync()` для фиксации смещения 5000 для раздела 3 в теме «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;

....

while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    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. 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 Я решил вызвать здесь метод `commitAsync()`, 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 два доступных для реализации метода:

- ❑ `public void onPartitionsRevoked(Collection<TopicPartition> partitions)` — вызывается до начала перебалансировки и после завершения получения сообщений потребителем. Именно в этом методе необходимо фиксировать смещения, чтобы следующий потребитель, которому будет назначен этот раздел, знал, с какого места начинать;
- ❑ `public void onPartitionsAssigned(Collection<TopicPartition> partitions)` — вызывается после переназначения разделов потребителю, но до того, как он начнет получать сообщения.

В следующем примере вы увидите, как использовать метод `onPartitionsRevoked()` для фиксации смещений перед сменой принадлежности раздела (в следующем разделе приведем развернутый пример, демонстрирующий использование метода `onPartitionsAssigned()`):

```
private Map<TopicPartition, OffsetAndMetadata> currentOffsets =
    new HashMap<>();

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(100);
        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, "no metadata"));
    }
    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. Обратите внимание на то, что мы фиксируем последние обработанные смещения, а не последние смещения во все еще обрабатываемом пакете. Делаем это из-за возможной смены принадлежности раздела в ходе обработки пакета. 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 дальнейшей обработки всех сообщений по очереди. Однако иногда необходимо начать чтение с другого смещения.

Если вы хотели бы начать чтение всех сообщений с начала раздела или хотя бы пропустить все до конца разделы и получать только новые сообщения, то можно применить специализированные API `seekToBeginning(TopicPartition tp)` и `seekToEnd(TopicPartition tp)`.

Однако API Kafka дает возможность переходить и к конкретному смещению. Ее можно использовать для множества различных целей, например, чтобы вернуться на

несколько сообщений назад или пропустить несколько сообщений (если чувствительному к задержкам приложению, отстающему от графика, нужно перескочить к более актуальным сообщениям). Самый интересный сценарий использования этой возможности — когда смещения хранятся не в Kafka.

Рассмотрим распространенный сценарий использования: приложение читает события из Kafka (например, поток данных о маршрутах перемещения пользователей по веб-сайту), обрабатывает данные (к примеру, удаляет записи, соответствующие посещению сайта автоматизированными программами, а не живыми пользователями), после чего сохраняет результаты в базе данных, NoSQL-хранилище или Hadoop. Допустим, что нам не хотелось бы ни терять какие-либо данные, ни сохранять одни и те же результаты в базе данных дважды. В этом случае цикл потребителя выглядел бы примерно так:

```
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records)
    {
        currentOffsets.put(new TopicPartition(record.topic(),
            record.partition()),
            record.offset());
        processRecord(record);
        storeRecordInDB(record);
        consumer.commitAsync(currentOffsets);
    }
}
```

В этом примере мы ведем себя совершенно параноидально, сохраняя смещения после обработки каждой записи. Однако все равно остается вероятность аварийного сбоя приложения после сохранения записи в базе данных, но до фиксации смещений, вследствие чего запись будет обработана снова и в базе данных окажутся дубликаты.

Этого можно было бы избежать, одномоментно сохраняя записи и смещения. При этом или запись и смещение фиксировались бы вместе, или не фиксировались бы ни то ни другое. Но при сохранении записей в базе данных, а смещений — в Kafka это невозможно.

А если заносить и запись, и смещение в базу данных в одной транзакции? Тогда можно быть уверенным, что или и запись обработана, и смещение зафиксировано, или не выполнено ни то ни другое и запись будет обработана повторно.

Остается единственная проблема: если смещение хранится в базе данных, а не в Kafka, то как наш потребитель узнает, откуда начинать чтение при назначении ему нового раздела? В этом нам поможет метод `seek()`. При начале работы потребителя или назначении нового раздела им можно воспользоваться для поиска смещения в базе данных и перехода к нужному месту.

Вот набросок примера использования этой методики. Чтобы начинать обработку с хранимых в базе данных смещений, используем класс `ConsumerRebalanceListener` и метод `seek()`:

```
public class SaveOffsetsOnRebalance implements
    ConsumerRebalanceListener {

    public void onPartitionsRevoked(Collection<TopicPartition>
        partitions) {
        commitDBTransaction(); ❶
    }

    public void onPartitionsAssigned(Collection<TopicPartition>
        partitions) {
        for(TopicPartition partition: partitions)
            consumer.seek(partition, getOffsetFromDB(partition)); ❷
        }
    }

consumer.subscribe(topics, new SaveOffsetOnRebalance(consumer));
consumer.poll(0);

for (TopicPartition partition: consumer.assignment())
    consumer.seek(partition, getOffsetFromDB(partition)); ❸

while (true) {
    ConsumerRecords<String, String> records =
        consumer.poll(100);
    for (ConsumerRecord<String, String> record : records)
    {
        processRecord(record);
        storeRecordInDB(record);
        storeOffsetInDB(record.topic(), record.partition(),
            record.offset()); ❹
    }
    commitDBTransaction();
}
```

- ❶ Используем тут фиктивный метод, предназначенный для фиксации транзакций в базе данных. Идея состоит во вставке записей и смещений в базу данных в процессе обработки записей, так что для обеспечения сохранности данных нужно только зафиксировать транзакции перед «потерей» раздела.
- ❷ У нас также есть фиктивный метод для извлечения смещений из базы данных, после чего путем вызова `seek()` мы переходим к этим записям при назначении новых разделов.
- ❸ В самом начале работы потребителя после подписки на темы мы один раз вызываем метод `poll()` для присоединения к группе потребителей и назначения разделов, после чего сразу же переходим с помощью `seek()` к нужному смещению в назначенных разделах. Не забывайте, что `seek()` лишь меняет место,

откуда мы получаем данные, так что нужные сообщения будут извлечены при следующем вызове метода `poll()`. Если в вызове `seek()` содержалась ошибка (например, такого смещения не существует), `poll()` сгенерирует исключение.

- ❹ Еще один фиктивный метод: на этот раз мы обновляем таблицу в базе данных, в которой хранятся смещения. Мы предполагаем, что обновление записей происходит быстро, так что обновляем все записи, а фиксация — медленно, поэтому фиксируем транзакцию только в конце обработки пакета. Однако есть несколько вариантов оптимизации этих действий.

Существует множество способов реализации строго однократной доставки путем хранения смещений и данных во внешнем хранилище, но все они требуют использования класса `ConsumerRebalanceListener` и метода `seek()`, чтобы гарантировать своевременное сохранение смещений и чтение сообщений потребителем с правильного места.

Выход из цикла

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

Если вы решите выйти из цикла опроса, 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. Обрабатывать 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 перераспределение, и вам не придется ждать истечения времени ожидания 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 по адресу: <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();
    }
    });
...
try {
    // Выполняем цикл вплоть до нажатия ctrl+c, об очистке
    // при завершении выполнения позаботится ShutdownHook
    while (true) {
        ConsumerRecords<String, String> records =
            movingAvg.consumer.poll(1000);
        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 ❷
} finally {
    consumer.close(); ❸
    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 В результате вызова `wakeup` из другого потока `poll` сгенерирует исключение `WakeupException`. Его лучше перехватить, чтобы не произошло непредвиденного завершения выполнения приложения, но ничего делать с ним не требуется.
- ❸ Перед завершением выполнения потребителя аккуратно закрываем его.

Десериализаторы

Как уже обсуждалось в предыдущей главе, для производителей Kafka требуются *сериализаторы* для преобразования объектов в отправляемые в Kafka байтовые массивы. А для потребителей Kafka необходимы *десериализаторы* для преобразования полученных из Kafka 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, а также 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 каждую из тем, и гарантировать, что темы содержат только такие данные, которые понятны используемым вами десериализаторам. Как раз в этом состоит одно из преимуществ сериализации и десериализации с помощью Avro и репозитория схем — `AvroSerializer` гарантирует, что все записываемые в конкретную тему данные совместимы со схемой темы, 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
                    IntegerDeserializer is shorter than expected");

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

            byte[] nameBytes = new Array[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 serializing
                Customer
                to byte[] " + 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 простую задачу.

- ② Мы просто меняем логику сериализатора на противоположную ей — извлекаем идентификатор и имя покупателя из байтового массива и используем их для формирования нужного объекта.

Использующий этот сериализатор код потребителя будет выглядеть примерно так:

```
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.CustomerDeserializer");

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

consumer.subscribe("customerCountries")

while (true) {
    ConsumerRecords<String, Customer> records =
        consumer.poll(100);
    for (ConsumerRecord<String, Customer> record : records)
    {
        System.out.println("current customer Id: " +
            record.value().getID() + " and
            current customer name: " + record.value().getName());
    }
}
```

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 воспользоваться стандартным форматом 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 совместимости схем 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:

```
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("schema.registry.url", schemaUrl); ❷
String topic = "customerContacts"

KafkaConsumer consumer = new
    KafkaConsumer(createConsumerConfig(brokers, groupId, url));
consumer.subscribe(Collections.singletonList(topic));

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

while (true) {
    ConsumerRecords<String, Customer> records =
        consumer.poll(1000); 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 потребителям 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 темы или из конкретного раздела темы. В этом случае оснований для организации группы потребителей или перебалансировки нет, достаточно просто назначить потребителю соответствующие тему и/или разделы, получать сообщения и периодически фиксировать смещения.

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

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

```
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(1000);

        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 данной теме разделов. Если вы собираетесь получать только данные из конкретного раздела, то можете эту часть пропустить.
- ❷ Выяснив, какие разделы нам нужны, вызываем метод `assign()`, передавая ему их список.

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

Старые API потребителей

В этой главе мы обсудили Java-клиент `KafkaConsumer`, включенный в пакет `org.apache.kafka.clients`. Однако на момент написания данной книги в Apache Kafka все еще есть два более старых клиента, написанных на языке Scala и включенных в пакет `kafka.consumer`, являющийся частью базового модуля Kafka. Один из этих потребителей носит название `SimpleConsumer` (и он не так уж прост). `SimpleConsumer` представляет собой тонкий адаптер для API Kafka, позволяющий получать данные из конкретных разделов и с конкретных смещений. Еще один старый API называется высокоуровневым потребителем или `ZookeeperConsumerConnector`. Высоко-

уровневый потребитель чем-то похож на текущий класс потребителя наличием групп потребителей и перебалансировкой разделов, но использует Zookeeper для управления группами потребителей и не предоставляет таких широких возможностей контроля фиксации и перебалансировки, как имеющийся в новой версии.

Как уже говорилось, поскольку текущий потребитель поддерживает оба типа поведения, гораздо более надежен и предоставляет разработчику гораздо бóльшие возможности контроля, мы не будем обсуждать эти старые API. Если вас интересует их использование, подумайте еще раз, после чего обратитесь к документации Kafka за дополнительной информацией.

Резюме

Мы начали главу с подробного описания групп потребителей Kafka и обеспечиваемых ими возможностей разделения работы по чтению событий из тем между несколькими потребителями. После теории привели практический пример потребителя, подписывающегося на тему 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.

Теперь, когда вы научились генерировать и получать события с помощью Kafka, в следующей главе мы объясним некоторые внутренние особенности реализации платформы.

5

Внутреннее устройство Kafka

Для промышленной эксплуатации Kafka или написания использующих ее приложений знать внутреннее устройство платформы необязательно. Однако понимание особенностей работы Kafka 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 трех вопросах, особенно актуальных для тех, кому приходится иметь дело с этой платформой.

- ❑ 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://www.bit.ly/2s3MYNh>). Различные компоненты Kafka 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 зарегистриро-

ваться, но ему это не удастся, 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 тем (см. далее раздел «Репликация») содержит идентификаторы брокеров для реплик. Таким образом, в случае невозможности сбоя брокера можно запустить новый брокер с тем же ID, и он немедленно присоединится к кластеру вместо старого и получит те же разделы и темы.

Контроллер

Контроллер — это брокер Kafka, который помимо выполнения обычных своих функций отвечает за выбор ведущих реплик для разделов (обсудим ведущие реплики разделов и их функциональность в следующем разделе). 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. Другие брокеры из кластера будут оповещены об этом посредством таймеров ZooKeeper и попытаются сами создать узел-контроллер. Первый узел, создавший контроллер, и станет новым узлом-контроллером, а остальные получат исключение «узел уже существует» и пересоздадут свои таймеры на новом узле-контроллере. Каждый новый выбранный контроллер получает новое большее значение *начала отсчета контроллера* (`controller epoch`) с помощью операции условного инкремента ZooKeeper. Текущее начало отсчета контроллера известно брокерам, так что они будут игнорировать полученные от контроллера сообщения с более старым значением.

Если контроллер посредством отслеживания соответствующего пути ZooKeeper обнаруживает, что брокер вышел из состава кластера, 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 в списке реплик этого раздела) и отправляет запрос всем брокерам, содержащим или новые ведущие реплики, или существующие ведомые реплики для данных разделов. Запрос содержит информацию о новой ведущей и ведомых репликах для этих разделов. 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.

Контроллер, получив информацию о присоединении брокера к кластеру, задействует 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.

Репликация

Репликация — основа основ архитектуры Kafka. Первая же фраза документации¹ платформы описывает его 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 темам. 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 темам и разделам.

Существует два типа реплик.

- ❑ *Ведущие*. Одна реплика из каждого раздела назначается ведущей (leader). Через нее выполняются все запросы на генерацию и потребление, чтобы обеспечить согласованность.
- ❑ *Ведомые*. Все реплики раздела, не являющиеся ведущими, называются ведомыми (followers). Они не обслуживают клиентские запросы, их единственная зада-

¹ Речь идет о документации к Kafka версии 0.9.0. — *Примеч. пер.*

ча — 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.

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 потребления 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.

Реплика запросит сообщение 1, затем 2, потом 3 и не запросит сообщения 4 до тех пор, пока не получит все предыдущие. Это значит, что ведущая реплика знает, что ведомая получила все сообщения вплоть до 3, когда последняя запрашивает сообщение 4. Ведущая реплика на основе последних запрошенных репликами смещений может определить, насколько отстает каждая из них. Если реплика не запрашивала сообщений более 10 секунд или запрашивала, но не отстает более чем на 10 секунд, 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 6, 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 темы. Предпочтительная она потому, что при первоначальном создании разделов производится распределение ведущих реплик между брокерами (его алгоритм обсудим далее в этой главе). В результате можно ожидать, что, когда ведущие реплики всех разделов кластера будут одновременно и предпочтительными, распределяться нагрузка по брокерам будет 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, иницилируя 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 10.) 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 перераспределении 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. 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 будет 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. Наглядно этот процесс показан на рис. 5.1.

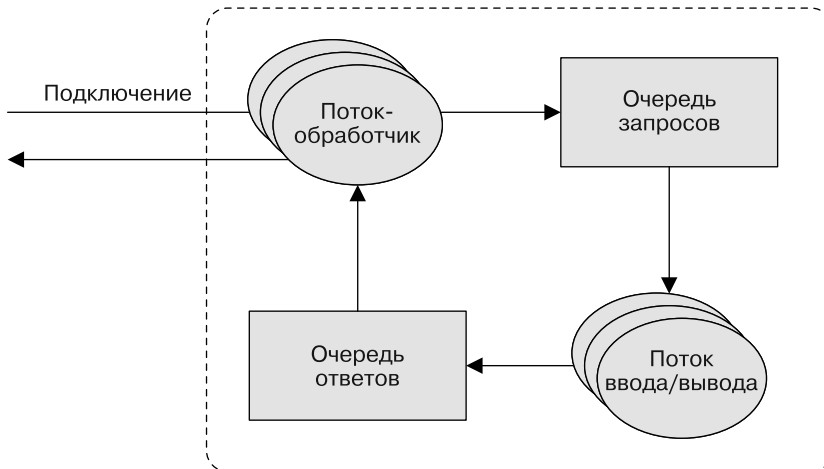


Рис. 5.1. Обработка запросов внутри Apache Kafka

После помещения запросов в очередь ответственность за их обработку передается *потокам ввода/вывода* (IO threads). Наиболее распространенные типы 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.

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 тем, интересующих клиента. Ответ сервера содержит информацию о существующих в этих темах 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 темы, например, не был ли добавлен еще один брокер и не была ли перенесена на него часть реплик (рис. 5.2). 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.

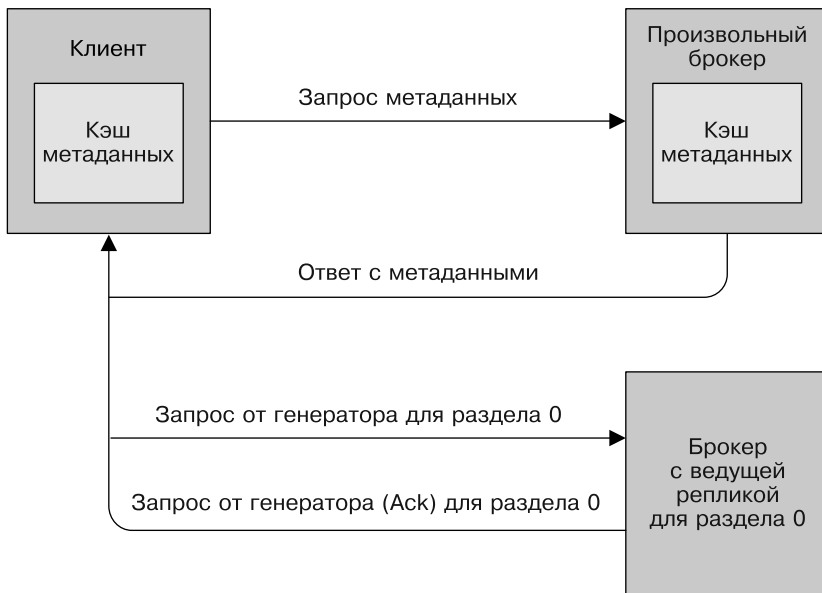


Рис. 5.2. Маршрутизация запросов клиентов

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`), всеми согласованными репликами (`acks=all`) или 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 эту тему?
- ☐ 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 6, когда будем обсуждать гарантии сохраняемости и надежности Kafka.)

Затем он 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 тем, разделов и смещений, — что-то вроде «Пожалуйста, отправьте мне сообщения, начинающиеся со смещения 53 раздела 0 темы Test, и сообщения, начинающиеся со смещения 64 раздела 3 темы 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 тем 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. 5.3). При этом читается в целом тот же самый объем данных при намного меньшем объеме взаимодействий, а следовательно, меньших накладных расходах.

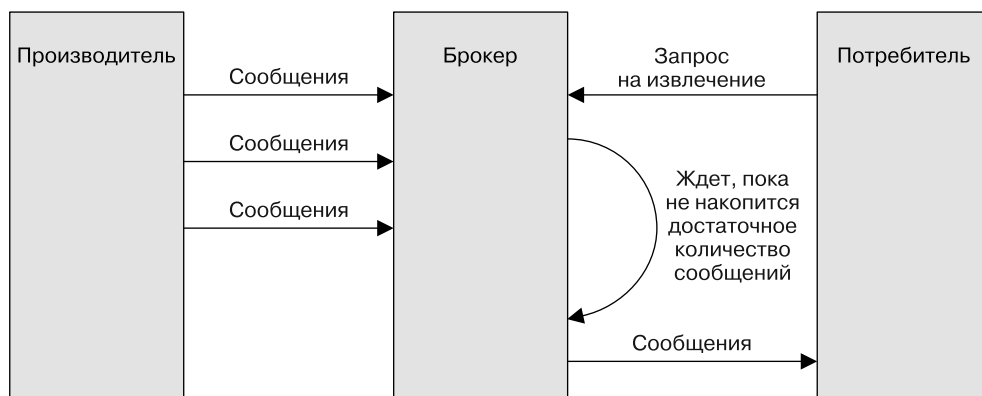


Рис. 5.3. Брокер откладывает ответ до тех пор, пока не накопит достаточно данных

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 прочитать 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 дождаться получения сообщения всеми согласованными репликами и лишь затем разрешать потребителям его читать (рис. 5.4). 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` — промежутком времени, по истечении которого реплика, отстающая в репликации новых сообщений, будет сочтена рассогласованной.

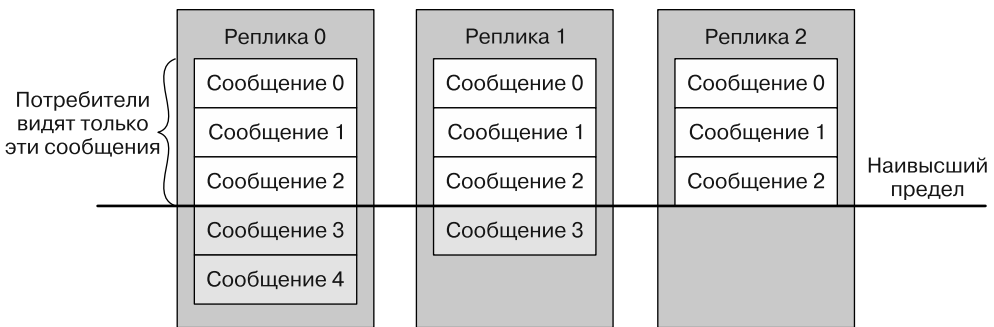


Рис. 5.4. Потребители видят только те сообщения, которые реплицированы на согласованные реплики

Другие запросы

Мы только что обсудили самые распространенные типы запросов, применяемые клиентами Kafka: `Metadata`, `Produce` и `Fetch`. Важно помнить, что мы говорим про обобщенный двоичный протокол, используемый клиентами для взаимодействия по сети. Хотя Kafka включает Java-клиенты, реализованные и поддерживаемые

участниками проекта Apache Kafka, существуют также клиенты на иных языках программирования, например, C, Python, Go и многих других. Полный список можно найти на сайте Apache Kafka (<http://www.bit.ly/2skvTjx>), причем все они взаимодействуют с брокерами Kafka по этому протоколу.

Кроме того, этот же протокол применяется для взаимодействия между самими брокерами Kafka. Эти запросы носят внутренний характер и не должны использоваться клиентами. 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.

Протокол Kafka в настоящий момент включает более 20 типов запросов, в дальнейшем будут добавлены и другие. Этот протокол постоянно совершенствуется — он должен развиваться по мере добавления функциональных возможностей 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 мы решили перестать использовать для этого ZooKeeper, а хранить смещения в отдельной теме Kafka. Для этого пришлось добавить в протокол несколько типов запросов: `OffsetCommitRequest`, `OffsetFetchRequest` и `ListOffsetsRequest`. Теперь, когда приложение обращается к API клиента `commitOffset()`, клиент ничего не записывает в ZooKeeper, а отправляет запрос `OffsetCommitRequest` в Kafka.

Темы по-прежнему создаются с помощью утилит командной строки, непосредственно обновляющих список тем в ZooKeeper, а брокеры по этому списку отслеживают добавление новых тем. Мы находимся в процессе усовершенствования Kafka и добавления типа запроса `CreateTopicRequest`, с помощью которого все клиенты (даже в языках программирования, где нет библиотеки ZooKeeper) могли бы создавать темы, непосредственно обращаясь к брокерам Kafka.

Помимо усовершенствования протокола путем добавления новых типов запросов мы иногда меняем существующие запросы, добавляя некоторые новые возможности. Например, при переходе от Kafka 0.9.0 к 0.10.0 мы решили, что клиентам не помешает информация о текущем контроллере, и добавили ее в ответ `Metadata`. В результате появилась 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, а только потом обновлять клиенты — новые брокеры смогут обрабатывать старые запросы, но не наоборот.

В версии Kafka 0.10.0 мы добавили 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.

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 состоять или из отдельного диска при использовании дискового массива JBOD, или из нескольких дисков, когда задействуется 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 включает по 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 рассмотрим, 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 Kafka превращается в долговременное хранилище данных, и опишем, как она функционирует.

Распределение разделов

При создании темы Kafka сначала принимает решение о распределении разделов по брокерам. Допустим, у нас есть 6 брокеров и мы хотим создать тему на 10 разделов с коэффициентом репликации 3. Kafka нужно распределить 30 реплик разделов по 6 брокерам. Основные задачи этого распределения следующие.

- ❑ Равномерно распределить реплики по брокерам — в нашем случае выделить на каждый брокер 5 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 желательно 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 6 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, 1 и 2 находятся в одной стойке, а брокеры 3, 4 и 5 — в другой. Вместо того чтобы подбирать брокеры по порядку от 0 до 5, мы упорядочиваем их в следующем порядке: 0, 3, 1, 4, 2, 5 — за каждым брокером следует брокер из другой стойки (рис. 5.5). В таком случае, если ведущая реплика раздела 0 находится на брокере 4, то первая реплика окажется на брокере 2, находящемся в другой стойке. Это замечательно, потому что, если первая стойка выйдет из строя, у нас все равно окажется работающая реплика и раздел по-прежнему будет доступен. Это справедливо для всех реплик, так что мы гарантировали доступность в случае отказа одной из стоек.

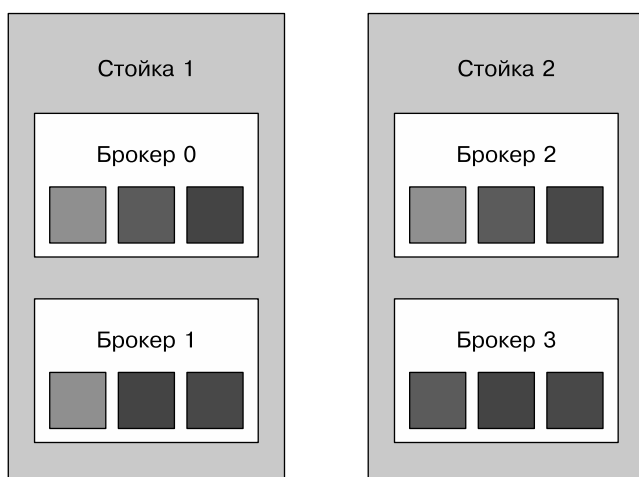


Рис. 5.5. Разделы и реплики распределяются по брокерам, находящимся в разных стойках

Выбрав нужные брокеры для всех разделов и реплик, мы должны определиться с каталогом для новых разделов. Сделаем это отдельно для каждого раздела. Принцип очень 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 каждой темы 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). Активный сегмент 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.

Каждое сообщение помимо ключа, значения и смещения содержит такие вещи, как размер сообщения, контрольную сумму для обнаружения порчи данных, волшебный байт с версией формата сообщения, кодек сжатия (Snappy, GZip или LZ4) и метку даты/времени (добавлена в версии 0.10.0). В зависимости от настроек метка даты/времени присваивается или производителем при отправке сообщения, или брокером при его получении.

Если производитель отправляет сообщения в сжатом виде, то все сообщения из одного пакета сжимаются вместе и отправляются в виде значения сообщения-адаптера (рис. 5.6). Так что брокер получает одно сообщение, которое затем переправляет потребителю. Но потребитель, распаковав значение этого сообщения-адаптера, увидит все содержащиеся в пакете сообщения с их метками даты/времени и смещениями.

Это значит, что при задействовании сжатия на производителе (что рекомендуется!) отправка больших пакетов приводит к более высокой степени сжатия как при передаче по сети, так и при размещении на дисках брокера. Это также значит, что при изменении используемого потребителями формата сообщений (например, добавлении в сообщение метки даты/времени) придется поменять как формат данных на диске, так и протокол передачи данных, а брокеры Kafka должны будут знать, что делать в случаях, когда из-за появления новых версий файлы содержат сообщения в двух форматах.

Брокеры Kafka идут в комплекте с утилитой `DumpLogSegment`, позволяющей просматривать сегменты разделов в файловой системе и исследовать их содержимое. Она выводит смещение, контрольную сумму, волшебный байт, размер и кодек сжатия для каждого сообщения. Запустить ее можно с помощью следующей команды:

```
bin/kafka-run-class.sh kafka.tools.DumpLogSegments
```

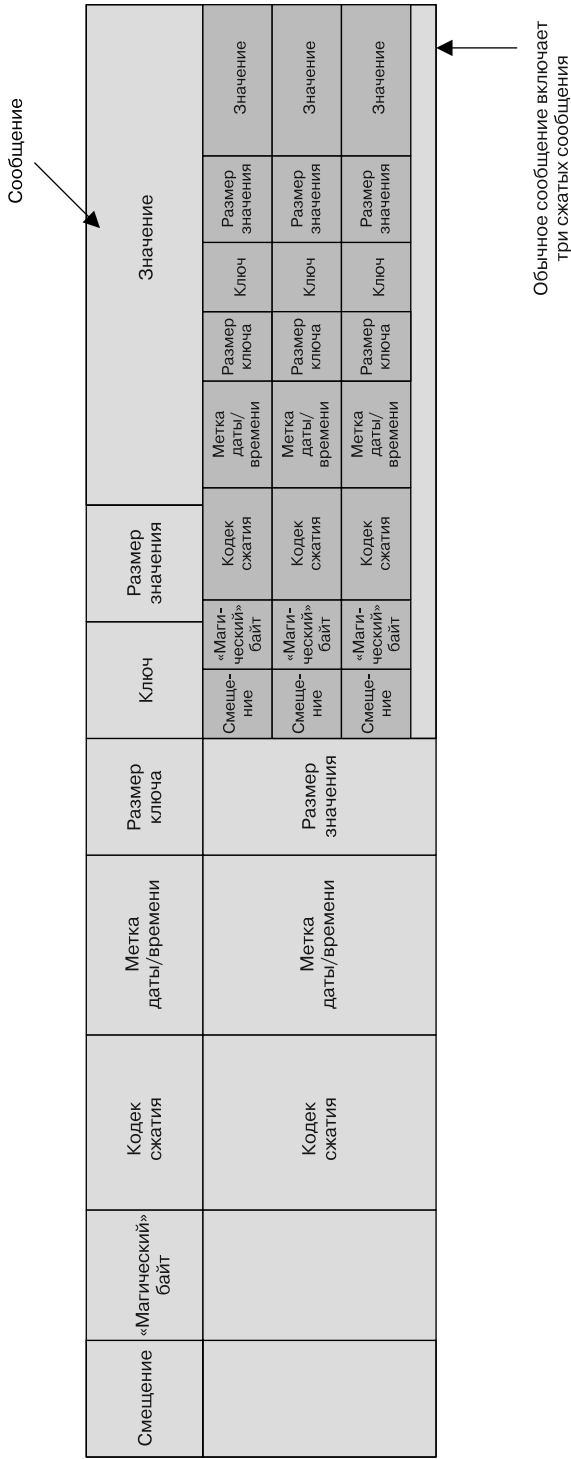


Рис. 5.6. Обычное сообщение и сообщение-адаптер

Если задать параметр `--deep-iteration`, утилита отобразит информацию о сжатых сообщениях, содержащихся внутри сообщений-адаптеров.

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.

Индексы также 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 — 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 адреса тех покупателей, которые давно никуда не переезжали, станут храниться столько, сколько нужно. Другой сценарий использования — приложение, использующее 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 таком 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 темы: *delete* (удалять), при которой события, чей возраст превышает интервал хранения, удаляются, и *compact* (сжимать), при которой сохраняется только последнее значение для каждого из ключей темы. Конечно, вторая стратегия имеет смысл только для тех тем, для которых приложения генерируют события, содержащие как ключ, так и значение. В случае использования неопределенных ключей (`null`) попытка сжатия приведет к сбою.

Как происходит сжатие

Каждый из журналов условно делится на две части (рис. 5.7):

- ❑ «чистая» — сжатые ранее сообщения. Она содержит только по одному значению для каждого ключа — последнему на момент предыдущего сжатия;
- ❑ «грязная» — сообщения, записанные после последнего сжатия.

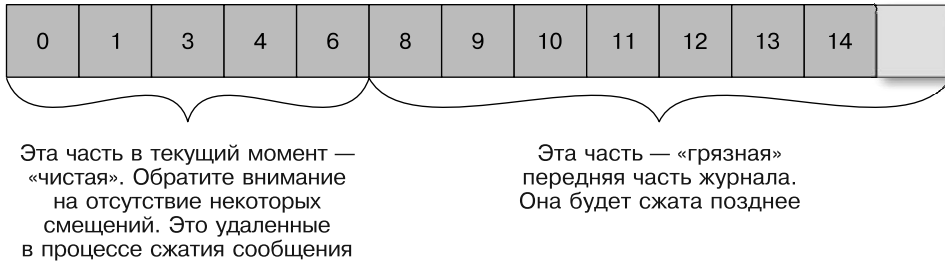


Рис. 5.7. Журнал с «чистым» и «грязным» разделами

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 этих потоков выбирает раздел с максимальным отношением числа «грязных» сообщений к полному размеру раздела и очищает его.

Для сжатия раздела поток очистки читает его «грязную» часть 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 в 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 задействовать 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 необязательно, 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 начнет 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, оно 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 переходим к следующему сегменту. В конце этого процесса для каждого ключа остается одно значение — наиболее новое (рис. 5.8).

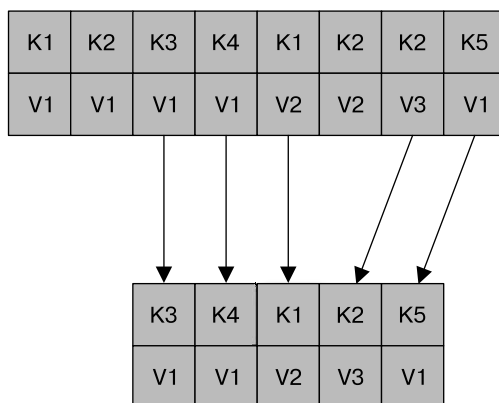


Рис. 5.8. Сегмент раздела до и после сжатия

Удаленные события

Допустим, мы всегда сохраняем последнее сообщение для каждого ключа. Тогда 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.

Когда выполняется сжатие тем

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.

В версии 0.10.0 и более старых Kafka начинается сжатие, когда 50 % темы содержит «грязные» сообщения. Задача заключается в том, чтобы не сжимать слишком часто — это может негативно повлиять на производительность чтения/записи для данной темы, 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 темой дискового пространства, после чего сжать их за один раз. Этот параметр настраивает администратор.

В будущих версиях мы планируем добавить период отсрочки, на протяжении которого гарантируется, что сообщения не будут сжаты. Благодаря этому приложениям, которым необходимо просмотреть каждое записанное в тему сообщение, хватит времени, даже если они несколько отстают.

Резюме

Тема внутреннего устройства Kafka намного обширнее, чем можно было охватить в этой главе. Но мы надеемся, что вы смогли прочувствовать реализованные во время работы над Kafka проектные решения и оптимизации и, вероятно, разобрались в некоторых малопонятных видах ее поведения и настроек.

Если внутреннее устройство Kafka вас действительно интересует, никакого другого выхода, кроме чтения кода, нет. Среди разработчиков Kafka (почтовая ссылка 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.

6

Надежная доставка данных

Надежная доставка данных относится к числу тех неотъемлемых свойств системы, проектирование которых нельзя оставить на потом. Как и производительность, она должна учитываться в системе, начиная с самой первой схемы работы. Надежность не получится «прикрутить» к уже готовому продукту. Более того, это свойство всей системы, а не отдельного компонента, так что хотя мы будем говорить о гарантиях надежности Apache Kafka, но 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 надежного продукта.

Apache Kafka очень гибка в том, что касается надежной доставки данных. У Kafka 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 быстродействие и простота. Kafka спроектирована в расчете на довольно широкие возможности настройки, а ее клиентский API достаточно гибок для любых компромиссов. 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. Потом обсудим брокеры и темы 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

При 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 критичных 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 Б было записано после сообщения А с помощью одного производителя в одном разделе, то Kafka гарантирует, что смещение сообщения Б будет превышать смещение сообщения А и потребители прочитают сообщение Б после сообщения А.
- ❑ 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 соображениями, 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.

Темы 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 производителями в ведущую реплику и потребляются из нее. Другие реплики просто должны быть согласованы с ведущей и своевременно реплицировать все недавние события. В случае недоступности ведущей реплики одна из согласованных становится новой ведущей.

Реплика считается согласованной, если она является ведущей репликой раздела или ведомой, которая:

- ❑ отправляла в ZooKeeper контрольный сигнал в последние 6 секунд (настраивается), что означает наличие текущего сеанса связи с ZooKeeper;
- ❑ извлекала сообщения из ведущей реплики в последние 10 секунд (настраивается);
- ❑ извлекала наиболее свежие сообщения из ведущей реплики в последние 10 секунд (настраивается). То есть недостаточно, чтобы ведомая реплика продолжала получать сообщения от ведущей, требуется еще и отсутствие задержки.

Если реплика теряет соединение с ZooKeeper, прекращает извлекать новые сообщения или отстает более чем на 10 секунд, 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 после временных неполадок, но может занять и много времени, если брокер, на котором находится реплика, долго не работал.



Рассогласованные реплики

Картина, при которой одна или несколько реплик быстро перепрыгивают из согласованного состояния в рассогласованное и наоборот, — верный признак проблем с кластером. Причина часто заключается в неправильной настройке сборки мусора Java на брокере. Неправильная настройка сборки мусора может вызвать приостановку работы брокера на несколько секунд, за время которой он теряет подключение к ZooKeeper. А когда брокер Kafka теряет подключение к ZooKeeper, то становится рассогласованным с кластером, что и обуславливает описанное поведение.

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 тем системы, а также на уровне отдельных тем.

Возможность управлять связанными с надежностью компромиссами на уровне отдельных тем означает, что один и тот же кластер Kafka можно задействовать как для надежных, так и для ненадежных тем. Например, в банке администратор, вероятно, захочет установить очень высокий уровень надежности по умолчанию для всего кластера, за исключением тем, 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 темы называется `replication.factor`. А на уровне брокера для автоматически создаваемых тем используется параметр `default.replication.factor`.

До сих пор мы предполагали, что коэффициент репликации тем равен 3, то есть каждый раздел реплицируется три раза на трех различных брокерах. Это было вполне разумное допущение, соответствующее умолчанию Kafka, но пользователи могут менять это поведение. Даже после создания темы можно добавлять или удалять реплики, меняя таким образом коэффициент репликации.

Коэффициент репликации N означает возможность потери $N - 1$ брокеров при сохранении надежности чтения из темы и записи в нее. 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 вам 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.

Так как же определить правильное число реплик для темы? Ответ зависит от степени ее важности и от ресурсов, которые вы готовы потратить для повышения доступности. А также от уровня вашей паранойи.

Если вас устраивает, что, возможно, конкретная тема будет недоступна при перезапуске отдельного брокера (это нормально при обычном функционировании кластера), то вполне будет достаточно коэффициента репликации 1. Не забудьте убедиться, что ваше начальство и пользователей также устраивает этот компромисс, — вы экономите на жестких дисках или серверах, но теряете высокую доступность. Коэффициент репликации 2 означает: потеря одного брокера не страшна, что представляется вполне достаточным. Однако не забывайте: потеря одного брокера (особенно на старых версиях Kafka) может сделать кластер нестабильным, вследствие чего придется перезапустить еще один брокер — контроллер Kafka. Это значит, что при коэффициенте репликации 2 для восстановления после возникшей проблемы может потребоваться перевести систему в состояние недоступности. Непростой выбор.

Поэтому мы рекомендуем использовать коэффициент репликации 3 для всех тем, для которых важна доступность. В редких случаях это считается недостаточно безопасным вариантом: мы встречали банки, в которых для критично важных тем создавались пять реплик, просто на всякий случай.

Размещение реплик также играет важную роль. По умолчанию Kafka размещает каждую реплику раздела на отдельном брокере. Однако в некоторых случаях этот вариант недостаточно безопасен. Если все реплики раздела размещены на брокерах в одной стойке, в случае сбоя коммутатора *top-of-rack* 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 название стойки для каждого брокера. При заданных названиях стоек Kafka обеспечит

распределение раздела по нескольким стойкам, что гарантирует еще более высокую доступность. Если вас интересуют детали, перечитайте главу 5, где 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 равно `true`.

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, то две доступные реплики не будут согласованы.

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 несколько часов.
- ❑ 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 последующих отчетов. Кроме того, реплика 2 вернется в работу и станет ведомой у новой ведущей реплики. При этом она удалит все сообщения, опережающие текущую ведущую реплику. 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.

Установка значения параметра `unclean.leader.election.enable` в `true` означает, что мы разрешаем рассогласованным репликам становиться ведущими (этот процесс называется *«нечистым» выбором* (`unclean election`)), хотя и знаем, что в результате этого потеряем данные. Установка же его в `false` означает ожидание восстановления функционирования исходной ведущей реплики, что приводит к снижению доступности. Обычно возможность «нечистого» выбора ведущей реплики отключают (параметр равен `false`) в системах, в которых критически важны качество и согласованность данных, например, банковских (большинство банков предпочитают невозможность обрабатывать платежи по кредитным картам в течение минут или даже часов риску неправильной обработки платежа). В системах же, где более значима доступность, например, системах анализа маршрутов перемещения пользователей по веб-сайту, возможность «нечистого» выбора ведущей реплики часто включена.

Минимальное число согласованных реплик

Соответствующий параметр уровня как темы, так и брокера называется `min.insync.replicas`.

Как мы уже видели, в некоторых случаях даже при трех репликах в теме 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 утрачены.

Если нужно гарантировать, что зафиксированные данные будут записаны более чем в одну реплику, можно задать более высокое минимальное число согласованных реплик. Если в теме три реплики и для параметра `min.insync.replicas` установ-

лено значение 2, то записывать в раздел темы можно будет 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.

Использование производителей в надежной системе

Даже если конфигурация брокеров самая надежная из всех возможных, система в целом может иногда терять данные, если не настроить производители достаточно надежным образом.

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 записано 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 не было на них записано, оно будет утрачено. Но приложение-производитель считает, что оно было 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 производитель 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.

Так как же избежать подобных неприятных результатов? 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`.
- ❑ Правильная обработка ошибок как в настройках, так и исходном коде.

Режимы работы 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 недоступен или если весь кластер Kafka решит часок-другой отдохнуть от работы, никакие ошибки возвращены не будут. Это значит, что даже в случае ожидаемого «чистого» выбора ведущей реплики производитель будет терять сообщения просто из-за отсутствия информации о недоступности ведущей реплики на протяжении выбора новой ведущей реплики. Быстродействие при `acks=0` очень высокое, именно поэтому существует столько тестов производительности при подобной конфигурации. Если вы решите идти этим путем, то сможете добиться потрясающей пропускной способности и эффективно использовать большую часть полосы пропускания, но заведомо потеряете часть сообщений.
- ❑ `acks=1` означает, что ведущая реплика в момент получения сообщения и записи его в файл данных раздела (но необязательно на диск) отправит подтверждение или сообщение об ошибке. Это значит, что при обычных условиях во время выбора ведущей реплики производитель получит исключение `LeaderNotAvailableException`. И если он обработает это исключение правильно (см. следующий раздел), то повторит попытку отправки сообщения, так что

оно успешно попадет в новую ведущую реплику. Возможна потеря данных в случае аварийного сбоя ведущей реплики, если часть успешно записанных на нее и подтвержденных сообщений не были реплицированы на ведомые реплики до сбоя.

- ❑ `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 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 самым низким быстродействием — производитель ждет получения сообщения всеми репликами, прежде чем отметить пакет сообщений как обработанный и продолжить работу. Эти эффекты можно смягчить за счет использования асинхронного режима производителя и отправки пакетов большего размера, но данный вариант обычно снижает пропускную способность.

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 теми возвращаемыми брокером ошибками, которые можно разрешить путем повтора отправки. 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 целом, если ваша цель — не терять ни одного сообщения, то лучше всего настроить производитель на повтор отправки сообщений в тех случаях, когда это имеет смысл. Почему? Потому что отсутствие ведущей реплики или проблемы с сетевым подключением обычно разрешаются за несколько секунд, и если просто предоставить производителю возможность продолжать попытки вплоть до достижения успеха, то можно избавиться от необходимости заниматься этим. Меня часто спрашивают: «На какое количество повторов необходимо настроить производитель?». Ответ зависит от того, что вы собираетесь делать после генерации производителем

исключения о прекращении попыток после N проделанных. Если хотите перехватить исключение и выполнить еще несколько попыток, то вам определенно необходимо установить большее число повторов — и пусть производитель продолжает предпринимать попытки. Если же вы собираетесь просто отказаться от сообщения, считая, что смысла в дальнейших повторах нет, или записать его где-нибудь еще и заняться им позже, то пора прекратить повторы. Обратите внимание на то, что имеющаяся в Kafka утилита репликации между ЦОД (MirrorMaker, которую мы обсудим в главе 8) настроена по умолчанию на бесконечное число повторов (то есть число повторов равно `MAX_INT`), поскольку высоконадежная утилита репликации никогда не должна просто выбрасывать сообщения.

Отметим, что попытка повтора отправки сообщения в случае неудачи часто означает небольшой риск того, что оба сообщения будут успешно записаны на брокер, а это приведет к дублированию. Например, если подтверждение от брокера не достигло производителя из-за проблем с сетью, но сообщение было успешно записано и реплицировано, то производитель сочтет отсутствие подтверждения временными проблемами с сетью и попытается повторить отправку сообщения, поскольку не знает, что оно было получено. В этом случае на брокере окажется две копии сообщения. Повторы отправки и тщательная обработка ошибок позволяют гарантировать сохранение сообщения *по крайней мере один раз*, но в текущей версии Kafka (0.10.0) нельзя гарантировать, что оно будет сохранено *ровно один раз*. Многие находящиеся в промышленной эксплуатации приложения добавляют в каждое сообщение уникальный идентификатор, чтобы можно было обнаруживать дубликаты и убирать их при чтении сообщений. Другие приложения делают сообщения *идемпотентными* в том смысле, что отправка сообщения дважды не повлияет на правильность. Например, сообщение «Сумма на счету равна \$110» идемпотентно, ведь многократная его отправка не меняет результата. А сообщение «Добавить на счет \$10» не идемпотентно, поскольку меняет результат при каждой отправке.

Дополнительная обработка ошибок

С помощью встроенного в производителя механизма повторов можно легко должным образом обработать множество разнообразных ошибок без потери сообщений, но, как разработчику, вам 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.

В главе 3 мы обсуждали написание обработчиков ошибок как для синхронного, так и для асинхронного методов отправки сообщений. Логика этих обработчиков ошибок меняется в зависимости от вашего приложения и его задач: выбрасываете ли вы «плохие» сообщения? Заносите ли ошибки в журнал? Храните ли эти сообщения в каталоге на локальном диске? Иницилируете ли обратный вызов к другому приложению?

Эти решения зависят от вашей архитектуры. Просто отметим, что если все обработчики ошибок пытаются только повторить отправку сообщения, то в этом лучше положиться на функциональность 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 событий, находит в нем последнее смещение и запрашивает следующий пакет событий, начиная с последнего полученного смещения. Благодаря этому потребители Kafka всегда получают новые данные в правильной последовательности и не пропускают событий.

В случае останова 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 самым же 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 события, 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 одну тему, каждый получает в обработку подмножество разделов темы и будет читать только часть сообщений, но группа в целом прочитает все сообщения. Если необходимо, чтобы отдельный потребитель увидел каждое из сообщений темы, 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 когда потребитель 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 вас по расписанию или самостоятельно фиксировать смещения в своем коде? Основное преимущество автоматической фиксации смещений в том, что при реализации потребителей на одну заботу окажется меньше. Если выполнять всю обработку прочитанных записей внутри цикла опроса потребителя, то автоматическая фиксация смещений гарантирует невозможность фиксации необработанного смещения (если вы не помните,

что такое цикл опроса потребителя, — обратитесь к главе 4). Основной недостаток автоматической фиксации смещений — отсутствие контроля числа дубликатов, которые придется обработать, поскольку потребитель останавливается после обработки части записей, но до запуска автоматической фиксации. Если вы занимаетесь чем-то замысловатым вроде передачи записей другому потоку для обработки в фоновом режиме, то могут оказаться автоматически зафиксированы смещения для уже прочитанных, но, возможно, еще не обработанных потребителем записей.

Четвертый из этих параметров связан с третьим и называется `auto.commit.interval.ms`. Если вы выберете автоматическую фиксацию смещений, то этот параметр даст вам возможность настроить ее частоту. Значение по умолчанию — каждые 5 секунд. В целом более высокая частота фиксации приводит к дополнительным вычислительным расходам, но снижает число дубликатов, которые могут возникнуть при останове потребителя.

Фиксация смещений в потребителях явным образом

Если вы используете автоматическую фиксацию смещений, то фиксация смещений явным образом вас не интересует. Но если вам не помешал бы больший контроль за временем фиксации смещений, чтобы уменьшить число дубликатов или, допустим, потому что обработка событий происходит вне основного цикла опроса потребителя, то стоит обдумать, как фиксировать смещения.

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 событий

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 события в конце цикла опроса.

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 (возможно, после каждого обработанного события), или фиксировать только один раз в несколько итераций. Фиксация подобно отправке сообщений при `acks=all` обуславливает некоторые накладные расходы, так что вам нужно найти компромисс.

Убедитесь, что фиксируете правильные смещения

Часто допускаемая при фиксации во время цикла опроса ошибка состоит в случайной фиксации последнего прочитанного при опросе смещения, а не последнего обработанного. Помните, что чрезвычайно важно всегда фиксировать смещения сообщений после их обработки — фиксация смещений для прочитанных, но еще не обработанных сообщений приводит к потере сообщений потребителями. В главе 4 приведены соответствующие примеры.

Перераспределение

При проектировании приложения не забывайте, что время от времени будут происходить перераспределения потребителей и вам придется обрабатывать их должным образом. В главе 4 приведено несколько примеров, но в более глобальном смысле это значит, что перед сменой принадлежности разделов придется зафиксировать смещения и при назначении новых разделов очистить сохраненные состояния.

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, вы 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», вы фиксируете смещения, но не подтверждаете получение отдельных сообщений. Это значит, что если вы не смогли обработать запись № 30, но успешно обработали запись № 31, то фиксировать 31-ю запись не следует — это приведет к фиксации всех записей до 31-й, включая 30-ю, что было бы нежелательно. Вместо этого попробуйте один из следующих двух 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, и продолжайте обработку (см. пояснения в главе 4). Вам может потребоваться продолжить выполнение цикла опроса вместе с обработкой всех записей. Можете воспользоваться методом `pause()` потребителя для упрощения повторов, чтобы гарантировать, что дополнительные опросы не вернут дополнительные данные.
2. Столкнувшись с ошибкой, которую можно разрешить путем повтора, запишите ее в отдельную тему и продолжайте выполнение. Для обработки записей из

этой темы для повторов можно воспользоваться отдельной группой потребителей. Или один и тот же потребитель может подписаться как на основную тему, так и на тему для повторов с приостановкой между повторами потребления данных из темы 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 событий. 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 тему для результатов одновременно с фиксацией смещения. Это значит, что поток может начать работу с того места, где остановился, подхватив последнее накопленное значение. Однако это не решает проблемы, ведь Kafka пока не предоставляет функциональности транзакций. Может произойти аварийный сбой после записи последнего результата, но до фиксации смещений, или наоборот. В целом, это довольно сложная проблема, и мы рекомендуем, вместо того чтобы пытаться решить ее своими силами, обратиться к таким библиотекам, как Kafka Streams, предоставляющим высокоуровневые DSL-подобные API для агрегирования, реализации соединений, оконных функций и другой сложной аналитики.

Длительная обработка записей

Иногда обработка записей занимает много времени. Например, в случае взаимодействия с блокирующим обработку или выполняющим очень длительные вычисления сервисом. Как вы помните, в некоторых версиях Kafka невозможно остановить выполнение опросов более чем на несколько секунд (см. подробности в главе 4). Даже если вы не хотите обрабатывать дальнейшие записи, все равно должны продолжать опросы, чтобы клиент мог отправлять контрольные сигналы брокеру. Распространенная схема действий в подобных случаях такова: передать по возможности данные пулу из множества потоков для ускорения работы за счет параллельной обработки. После передачи данных потокам-исполнителям можно приостановить потребитель и продолжать выполнение опросов без фактического извлечения дополнительных данных вплоть до завершения выполнения потоков-исполнителей. После этого можно возобновить работу потребителя. А поскольку потребитель не прекращает выполнение опросов, то контрольные сигналы будут отправляться по плану и перераспределение запущено не будет.

Строго однократная доставка

Для некоторых приложений требуется доставить данные не просто как минимум один раз (то есть без потерь данных), а ровно один раз. Хотя в настоящий момент Kafka не обеспечивает поддержки строго однократной доставки, у потребителей есть в запасе несколько трюков, позволяющих гарантировать, что каждое сообщение в Kafka будет записано во внешнюю систему ровно один раз (обратите внимание на то, что при этом не отсеиваются дубликаты, которые могли возникнуть при отправке данных в Kafka).

Простейший и, наверное, самый распространенный способ строго однократной доставки — запись результатов в систему, поддерживающую уникальные ключи. В числе таких систем все хранилища данных типа «ключ/значение», все реляционные базы данных, Elasticsearch и, вероятно, множество других хранилищ данных. При записи данных в такую систему, как реляционная база данных или Elasticsearch, или сама запись содержит уникальный ключ (довольно распространенная ситуация), или можно создать такой ключ на основе сочетания темы, раздела и смещения, которое однозначно идентифицирует запись Kafka. Если вы уже занесли запись в виде значения с уникальным ключом, а позднее случайно прочитали ее снова, то вы просто запишете те же ключ и значение. Хранилище данных при этом перезапишет существующую запись, и результат будет точно таким же, как и без случайного дубликата. Эта очень распространенная и удобная схема называется *идемпотентной операцией записи* (idempotent write).

Другой вариант возможен при записи в систему с поддержкой транзакций. Простейший пример — реляционные базы данных, но в HDFS есть атомарные переименования, часто используемые для тех же целей. Суть процедуры состоит в объединении записей и их смещении в одну транзакцию, чтобы добиться согласованности. В начале работы извлекаются смещения последних записей, занесенных во внешнее хранилище, после этого применяется метод `consumer.seek()`, чтобы начать потребление данных с этих смещений. Пример реализации такого варианта приведен в главе 4.

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?

Можете так и поступить, но лучше сначала выполнить хотя бы небольшую проверку. Мы рекомендуем 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 и рекомендуется поступить по двум причинам.

- ❑ Благодаря этому можно проверить, соответствует ли выбранная конфигурация вашим требованиям.
- ❑ Это хорошее упражнение на прослеживание ожидаемого поведения системы. Данная глава носит скорее теоретический характер, так что важно проверить, насколько эта теория применима на практике.

Kafka содержит две утилиты, предназначенные для такой проверки. Пакет `org.apache.kafka.tools` включает классы `VerifiableProducer` и `VerifiableConsumer`. 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 производитель можно настраивать точно так же, как и ваш собственный, задавая нужное значение параметра `acks`, количество попыток повторов 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 перераспределении.

Стоит также задуматься о том, какие тесты имеет смысл выполнить.

- ❑ Выбор ведущей реплики: что произойдет, если остановить 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, вы запускаете контрольный производитель и контрольный потребитель и выполняете выбранный сценарий — например, останавливаете ведущую реплику раздела, для которой генерирует данные производитель. Если вы ожидаете лишь небольшой паузы, после которой функционирование возобновится без потери каких-либо данных, то проверьте, совпадает ли число

сообщений, сгенерированных производителем, и число сообщений, потребленных потребителем.

Репозиторий исходного кода Apache Kafka включает обширный набор тестов. Многие из них основаны на одном и том же принципе — например, проверке функционирования плавающих обновлений с помощью контрольного производителя и контрольного потребителя.

Проверка приложений

Убедившись, что настройки брокера и клиента соответствуют требованиям, можете приступить к проверке того, обеспечивает ли ваше приложение необходимые гарантии. Это включает проверку таких вещей, как пользовательский код обработки ошибок, фиксация смещений, перераспределение прослушивателей и других мест, в которых логика приложения взаимодействует с клиентскими библиотеками Kafka.

Конечно, поскольку приложение ваше, вам виднее, как его тестировать, мы можем лишь подсказать некоторые моменты. Надеемся, что в процессе разработки вы используете комплексные тесты. Но как бы вы ни проверяли приложение, мы рекомендуем запускать тесты при различных сбойных состояниях:

- ☐ потере клиентами соединения с сервером (помочь в моделировании сетевых сбоев может системный администратор);
- ☐ выборе ведущей реплики;
- ☐ плавающем перезапуске брокеров;
- ☐ плавающем перезапуске потребителей;
- ☐ плавающем перезапуске производителей.

В каждом из этих сценариев существует *ожидаемое поведение* — то, что вы хотели получить, когда создавали приложение. И вы можете выполнить тест, чтобы увидеть, что произойдет на самом деле. Например, когда вы планировали плавающий перезапуск потребителей, то ожидали небольшой паузы вследствие перераспределения потребителей, после которой потребление продолжилось бы при не более чем 1000 дублирующихся значений. Тест покажет, действительно ли приложение фиксирует смещения и выполняет перераспределение 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 9 приведены подробные советы по мониторингу состояния кластера Kafka, но помимо этого важно контролировать также клиенты и потоки данных в системе.

Во-первых, 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 теме/разделе [тема 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...). Как говорилось в разделе «Использование производителей в надежной системе» ранее в данной главе, вы можете или увеличить число повторов, или первым делом устранить вызывающую ошибки проблему.

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://github.com/linkedin/Burrow>).

Мониторинг потока данных означает также проверку того, чтобы все сформированные производителями данные были потреблены своевременно (смысл слова «своевременно» определяется вашими требованиями). Для обеспечения этого нужно знать, когда данные поступили от производителей. Для облегчения этого Kafka, начиная с версии 0.10.0, включает во все сообщения метку даты/времени. Если ваши клиенты работают с более ранней версией Kafka, рекомендуем для каждого события регистрировать метку даты/времени, имя приложения-производителя и имя хоста, на котором было создано сообщение. Это поможет в дальнейшем при выяснении причин различных проблем.

Чтобы убедиться в том, что все сформированные производителями сообщения были потреблены за приемлемое время, производители должны регистрировать число сгенерированных сообщений (обычно в виде количества событий в секунду). Потребители должны будут регистрировать как число потребленных сообщений (также в виде количества событий в секунду), так и информацию о задержках между генерацией сообщений и их потреблением на основе меток даты/времени событий. Далее вам 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 не слишком велики. Для более продвинутого мониторинга можете добавить специальный потребитель, подписав его на наибо-

лее важные темы, который бы подсчитывал события и сравнивал их количество с информацией от производителей с целью точного мониторинга производителей даже в случае, когда в конкретный момент времени никто не получает исходящие от них события. Реализация подобных систем сквозного мониторинга непроста 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 (<http://www.confluent.io/product/control-center>).

Резюме

Как мы говорили в начале главы, надежность — вопрос не только конкретных возможностей Kafka. Необходимо сделать надежной систему в целом, включая архитектуру вашего приложения, методы применения приложением API производителей и потребителей, настройки производителей и потребителей, настройки тем и брокеров. Обеспечение надежности приложения всегда означает определенный компромисс между сложностью приложения, его производительностью, доступностью и использованием дискового пространства. Понимая все доступные варианты, распространенные схемы действий и требования для конкретных сценариев, можно принимать разумные решения по поводу нужной степени надежности вашего приложения и развернутого Kafka, а также того, на какие компромиссы имеет смысл пойти в конкретном случае.

7

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 задействуется 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, разделяя внутри конвейера производителей данных и их потребителей. 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 свою задачу 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 нечто специфичное для Kafka, а скорее 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 у нас нет возможности 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, как 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 6. 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, то 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 производитель. 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 в состоянии 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 нацелены не просто на масштабирование, а на распараллеливание работы. В следующем разделе мы опишем, как платформа Kafka дает возможность источникам и приемникам данных распределять работы по нескольким потокам выполнения и задействовать 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 при записи в S3.

Самой Kafka и API Kafka Connect форматы данных совершенно не важны. 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 Hive. Хороший конвейер данных при добавлении столбца в MySQL обеспечивает добавление его и в Hive, 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 HDFS 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. Существуют 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. Такие конвейеры называют также высокоточными конвейерами (*high-fidelity pipeline*) или архитектурой озер данных (*data-lake architecture*). 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 ресурсы обходятся недорого, и желательно по возможности вынести обработку из этих систем.

Безопасность

Безопасность важна всегда. В терминологии конвейеров данных основные проблемы безопасности состоят в следующем.

- ❑ 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?

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 тема 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 не-санкционированного. Написав немного дополнительного кода, можно отследить, откуда поступили события в каждой теме и кто их менял, и таким образом получить полную историю каждой записи.

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 долго хранит все события, можно при необходимости вернуться назад во времени и ликвидировать последствия ошибок.

Связывание и быстрота адаптации

Одна из важнейших задач конвейеров данных — 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, 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 появляющейся в компании системы 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 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 задействовать традиционные клиент-производитель и клиент-потребитель, как описано в главах 3 и 4, или воспользоваться API Kafka Connect и коннекторами, как мы покажем далее. Прежде чем углубиться в нюансы Kafka Connect, задумаемся о том, 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 задействовать для подключения Kafka к хранилищам данных, созданным не вами, код 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. Если для хранилищ уже существует коннектор, Kafka Connect могут использовать и не программисты, а простые пользователи, которым необходимо только настроить коннекторы.

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 задействовать 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. Kafka Connect берет большую часть этих забот на себя, благодаря чему вы можете сосредоточиться на перемещении данных во внешние хранилища и обратно.

Kafka Connect

Фреймворк Kafka Connect — часть Apache Kafka, обеспечивающая масштабируемый и гибкий способ перемещения 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 воспользоваться 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 использует *преобразователи формата* (convertors) — поддержка формата JSON встроена в Apache Kafka, а реестр схем Confluent предоставляет преобразователи форматов Avro. Благодаря этому пользователи могут выбирать формат хранения данных в Kafka независимо от задействованных коннекторов.

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 Connect

Kafka Connect поставляется вместе с Apache Kafka, так что установить 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. 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 этом кластере коннектор может оказаться запущенным на любом из исполнителей кластера, как и его задачи.
- ❑ **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` — это составная часть реестра схем Confluent.

У некоторых преобразователей формата есть особые параметры конфигурации. Например, сообщения в формате JSON могут включать или не включать схему. Чтобы конкретизировать эту возможность, нужно установить параметр `key.converter.schema.enable=true` или `false` соответственно. Аналогичную настройку можно выполнить для преобразователя значений, установив параметр `value.converter.schema.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`.

Настройки `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:

```
gwen$ curl http://localhost:8083/
{"version":"0.10.1.0-SNAPSHOT","commit":"561f45d747cd2a8c"}
```

В результате обращения к корневому URI REST должна быть возвращена текущая версия. Мы работаем с предварительным выпуском Kafka 0.10.1.0. Можно также посмотреть доступные плагины коннекторов:

```
gwen$ curl http://localhost:8083/connector-plugins
[{"class":"org.apache.kafka.connect.file.FileStreamSourceConnector"},
{"class":"org.apache.kafka.connect.file.FileStreamSinkConnector"}]
```

У нас запущена чистая Apache Kafka, так что доступны только плагины коннекторов для файлового источника и файлового приемника.

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. Connect в автономном режиме обычно удобнее использовать для разработки и отладки, а также 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 тему 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 --header
"contentType:application/json"

{"name":"load-kafka-config", "config":{"connector.class":"FileStreamSource",
"file":"config/server.properties", "topic":"kafka-config-topic", "name":"load-kafka-
config"}, "tasks":[]}
```

Для создания коннектора мы написали JSON-текст, включающий название коннектора — `load-kafka-config` — и ассоциативный массив его настроек, включающий класс коннектора, загружаемый файл и тему, в которую мы хотим его загрузить.

Воспользуемся консольным потребителем Kafka для проверки загрузки настроек в тему:

```
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 этой темы 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://localhost: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 тем в один файл, в то время как источник позволяет записывать только в одну тему.

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
```

Если вы заглянете в журнал исполнителя `Connect` после удаления коннектора, то обнаружите, что все остальные коннекторы перезапускают задания. Это происходит для перераспределения оставшихся задач между исполнителями и обеспечения равномерной нагрузки после удаления коннектора.

Пример коннектора: из MySQL в Elasticsearch

Теперь, когда заработал простой пример, пора заняться чем-то более полезным. Возьмем таблицу MySQL, отправим ее в тему 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. Если вы работаете с Confluent OpenSource, то они уже должны быть установлены как часть платформы. В противном случае можно выполнить сборку коннекторов из кода в GitHub.

1. Перейдите по адресу <https://github.com/confluentinc/kafka-connect-elasticsearch>.
2. Клонировать этот репозиторий.
3. Выполните команду `mvn install` для сборки проекта.
4. Повторите эти действия для коннектора JDBC (<https://github.com/confluentinc/kafka-connect-jdbc>).

Теперь скопируйте JAR-файлы, появившиеся в результате сборки в подкаталогах `target` каталогов, в которых выполнялась сборка коннекторов, в место, соответствующее пути к классам Kafka Connect:

```
gwen$ mkdir libs
gwen$ cp ../kafka-connect-jdbc/target/kafka-connect-jdbc-3.1.0-SNAPSHOT.jar
libs/
gwen$ cp ../kafka-connect-elasticsearch/target/kafka-connect-
elasticsearch-3.2.0-SNAPSHOT-package/share/java/kafka-connect-elasticsearch/*
libs/
```

Если исполнители Kafka Connect еще не запущены, сделайте это и проверьте, что новые плагины коннекторов перечислены в списке:

```
gwen$ bin/connect-distributed.sh config/connect-distributed.properties &

gwen$ curl http://localhost:8083/connector-plugins
[{"class":"org.apache.kafka.connect.file.FileStreamSourceConnector"},
{"class":"io.confluent.connect.elasticsearch.ElasticsearchSinkConnector"},
{"class":"org.apache.kafka.connect.file.FileStreamSinkConnector"},
{"class":"io.confluent.connect.jdbc.JdbcSourceConnector"}]
```

Как видите, в кластере теперь доступны новые плагины коннекторов. JDBC-источнику требуется драйвер MySQL для работы с MySQL. Мы скачали JDBC-драйвер для MySQL с сайта Oracle, разархивировали его и скопировали файл `mysql-connector-java-5.1.40-bin.jar` в каталог `libs/` при копировании коннекторов.

Следующий шаг — создание таблицы в базе данных MySQL, которую потом можно будет передать в Kafka с помощью JDBC-коннектора:

```
gwen$ mysql.server restart

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)
```

```
mysql> commit;
Query OK, 0 rows affected (0.01 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, а можно воспользоваться API REST для их выяснения:

```
gwen$ curl -X PUT -d "{}" localhost:8083/connector-plugins/JdbcSourceConnector/
config/validate --header "content-Type:application/json" | python -m json.tool
```

```
{
  "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>
    }
  ]
}
```

По сути, мы запросили у API REST проверку настроек коннектора, передав их пустой перечень. В качестве ответа получили JSON-описание всех доступных настроек (для повышения удобочитаемости результат пропустили через Python).

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","validate.non.null":false,"timestamp.column.name":"login_time","topic.prefix":"mysql."}}' | curl -X POST -d @- http://localhost:8083/connectors --header "content-Type:application/json"
```

```
{
  "name": "mysql-login-connector",
  "config": {
    "connector.class": "JdbcSourceConnector",
    "connection.url": "jdbc:mysql://127.0.0.1:3306/test?user=root",
    "mode": "timestamp",
    "table.whitelist": "login",
    "validate.non.null": "false",
    "timestamp.column.name": "login_time",
    "topic.prefix": "mysql.",
    "name": "mysql-login-connector"
  },
  "tasks": []
}
```

Прочитаем данные из темы `mysql.login`, чтобы убедиться, что она работает:

```
gwen$ bin/kafka-console-consumer.sh --new --bootstrap-server=localhost:9092 --topic mysql.login --from-beginning
```

```
<more stuff>
```

```
{
  "schema": {
    "type": "struct",
    "fields": [
      {
        "type": "string",
        "optional": true,
        "field": "username"
      },
      {
        "type": "int64",
        "optional": true,
        "name": "org.apache.kafka.connect.data.Timestamp",
        "version": 1,
        "field": "login_time"
      }
    ],
    "optional": false,
    "name": "login"
  },
  "payload": {
    "username": "gwenshap",
    "login_time": 1476423962000
  }
},
{
  "schema": {
    "type": "struct",
    "fields": [
      {
        "type": "string",
        "optional": true,
        "field": "username"
      },
      {
        "type": "int64",
        "optional": true,
        "name": "org.apache.kafka.connect.data.Timestamp",
        "version": 1,
        "field": "login_time"
      }
    ],
    "optional": false,
    "name": "login"
  },
  "payload": {
    "username": "tpalino",
    "login_time": 1476423981000
  }
}
```

Если вы не увидите никаких данных или получите сообщение, гласящее, что темы 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)
```

С первого раза обычно не получается задать правильную строку соединения. Среди возможных проблем также отсутствие драйвера по пути к классам или прав на чтение таблицы.

Обратите внимание на то, что вставленные во время работы коннектора в таблицу `login` дополнительные строки должны сразу же отразиться в теме `mysql.login`.

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" : "Hammerhead",
"cluster_name" : "elasticsearch_gwen",
"cluster_uuid" : "42D5GrxOQFebf83DYgNl-g",
"version" : {
  "number" : "2.4.1",
  "build_hash" : "c67dc32e24162035d18d6fe1e952c4cbcbe79d16",
  "build_timestamp" : "2016-09-27T18:57:55Z",
  "build_snapshot" : false,
  "lucene_version" : "5.5.2"
},
>tagline" : "You Know, for Search"
}

```

Теперь 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
"contentType:application/json"

```

```

{"name":"elastic-login-connector", "config":{"connector.class":
"ElasticsearchSinkConnector", "connection.url":"http://localhost:9200", "type.name":
"mysqldata", "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. Каждая тема в Kafka по умолчанию становится отдельным индексом Elasticsearch (остается название темы). Внутри темы необходимо описать тип записываемых данных. Мы считаем, что тип у всех событий в теме одинаков, так что просто «зашиваем» его: `type.name=mysql-data`. Записываем в Elasticsearch только одну тему — `mysql.login`. При описании таблицы в MySQL мы не задали для нее первичного ключа. В результате ключи событий в Kafka не определены. А поскольку у событий в Kafka нет ключей, необходимо сообщить коннектору Elasticsearch, чтобы в качестве ключей событий он использовал название темы, идентификаторы разделов и смещения.

Проверим, что индекс с данными таблицы `mysql.login` создан:

```

gwen$ curl 'localhost:9200/_cat/indices?v'
health status index          pri rep docs.count docs.deleted store.size
pri.store.size
yellow open      mysql.login    5    1          3              0      10.7kb
10.7kb

```

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" : 29,
"timed_out" : false,
"_shards" : {
  "total" : 5,
  "successful" : 5,
  "failed" : 0
},
"hits" : {
  "total" : 3,
  "max_score" : 1.0,
  "hits" : [ {
    "_index" : "mysql.login",
    "_type" : "mysql-data",
    "_id" : "mysql.login+0+1",
    "_score" : 1.0,
    "_source" : {
      "username" : "tpalino",
      "login_time" : 1476423981000
    }
  }, {
    "_index" : "mysql.login",
    "_type" : "mysql-data",
    "_id" : "mysql.login+0+2",
    "_score" : 1.0,
    "_source" : {
      "username" : "nnarkede",
      "login_time" : 1476672246000
    }
  }, {
    "_index" : "mysql.login",
    "_type" : "mysql-data",
    "_id" : "mysql.login+0+0",
    "_score" : 1.0,
    "_source" : {
      "username" : "gwenshap",
      "login_time" : 1476423962000
    }
  } ]
}
}

```

Если добавить новые записи в таблицу в MySQL, они автоматически появятся в теме `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 список всех известных коннекторов (<http://www.confluent.io/product/connectors/>), включая поддерживаемые как различными компаниями, так и сообществом разработчиков. Можете выбрать из списка любой коннектор, какой вам только хочется попробовать, собрать его из кода в GitHub репозитории, настроить — прочитав документацию или получив настройки из API REST — и запустить его на своем кластере исполнителей Connect.



Создание своих собственных коннекторов

API коннекторов общедоступен, так что каждый может создать новый коннектор. На самом деле именно так большинство их в Connector Hub и появилось — люди создавали коннекторы и сообщали нам о них. Так что мы призываем вас написать собственный коннектор, если хранилище данных, к которому вам нужно подключиться, в Connector Hub отсутствует. Можете даже представить его сообществу разработчиков, чтобы другие люди его увидели и смогли использовать. Обсуждение всех нюансов создания коннектора выходит за рамки данной главы, но вы можете узнать о них из официальной документации (<http://docs.confluent.io/3.0.1/connect/devguide.html>). 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 вашим вопросам и просьбам о помощи, а также демонстрации новых коннекторов в почтовой рассылке сообщества разработчиков Apache Kafka (users@kafka.apache.org).

Взглянем на Connect поближе

Чтобы понять, как работает Connect, необходимо разобраться с тремя основными его понятиями и их взаимодействием друг с другом. Как мы уже объясняли и демонстрировали на примерах, для использования Connect вам нужен работающий кластер исполнителей и потребуется запускать/останавливать 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, состоящее 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 `max.tasks` 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 значениями первичного ключа таблицы). 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, 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 он 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 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 их перемещению, 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 реализация всего этого может занять несколько месяцев. 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 преобразователь формата, который будет применяться для сохранения данных в Kafka. В настоящий момент в этом качестве можно использовать Avro, JSON и строки. 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 (Avro, JSON или строковое значение) 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 коннекторов (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 этом файле. В JDBC-источнике раздел может быть таблицей базы данных, а смещение — идентификатором записи 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. Механизм хранения является подключаемым, обычно это тема Kafka. Благодаря этому коннекторы могут начинать обработку событий с последнего сохраненного после перезапуска или аварийного сбоя смещения.

Последовательность выполняемых коннекторами приемников действий аналогична с точностью до наоборот: они читают записи Kafka, в которых уже есть идентификаторы темы, 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. Часть разработчиков 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 и Elasticsearch, а Kafka — лишь одно из многих средств ввода данных в эту систему, 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. Задействовать их имеет смысл лишь для того, чтобы не переходить на что-то новое: если вы везде применяете, например, Pentaho, то зачем добавлять еще одну систему интеграции данных специально для Kafka? Это имеет смысл также, если вы создаете конвейеры ETL на основе GUI. Основной недостаток 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. Как упоминалось в разделе «Преобразования» ранее в этой главе, 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 задействовать его же для интеграции данных. Это часто позволяет исключить из технологического процесса потоковой обработки необходимость хранить обрабо-

танные 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 Connect работает, 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 это делаем мы. Убедитесь, что ваша система интеграции данных способна без потерь сообщений выдержать остановку процессов, аварийные сбои машин, сетевые задержки и высокие нагрузки. В конце концов, 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 уверенным 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.

8

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 нескольких кластеров.

В некоторых случаях кластеры совершенно независимы — действуют в различных 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 различаются 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 для работы достаточно 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 такой компании может 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 на центральный кластер, чтобы бизнес-аналитики могли сформировать отчеты о прибыли в масштабе всей компании.
- ❑ *Избыточность (переключение при аварийных сбоях).* Приложения выполняются на одном кластере Kafka, для них не требуются данные из других мест, но вас беспокоит вероятность недоступности кластера по каким-либо причинам. Вам нужен еще один кластер Kafka со всеми данными с первого кластера, чтобы в случае аварии можно было перенаправить приложения на второй кластер и продолжать работу как ни в чем не бывало.
- ❑ *Миграция в облако.* Сейчас многие компании пользуются как локальным ЦОД, так и услугами облачных провайдеров. Часто ради избыточности приложения выполняются в различных регионах провайдеров облачных услуг, а иногда используются и различные провайдеры. В подобных случаях нередко задействуется как минимум по одному кластеру Kafka в каждом локальном ЦОД и каждом регионе облака. Эти кластеры Kafka задействуются 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.

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 данными 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 следующие вещи:

- ❑ *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 с помощью увеличения числа попыток повтора производителей, а с длительными задержками — за счет увеличения размеров 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 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. 8.1).

Существует также упрощенный вариант этой архитектуры с двумя кластерами — ведущим и ведомым (рис. 8.2).

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.

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, а каждый потребитель всегда читает данные из одного кластера, то такую архитектуру легко развертывать, настраивать и контролировать.

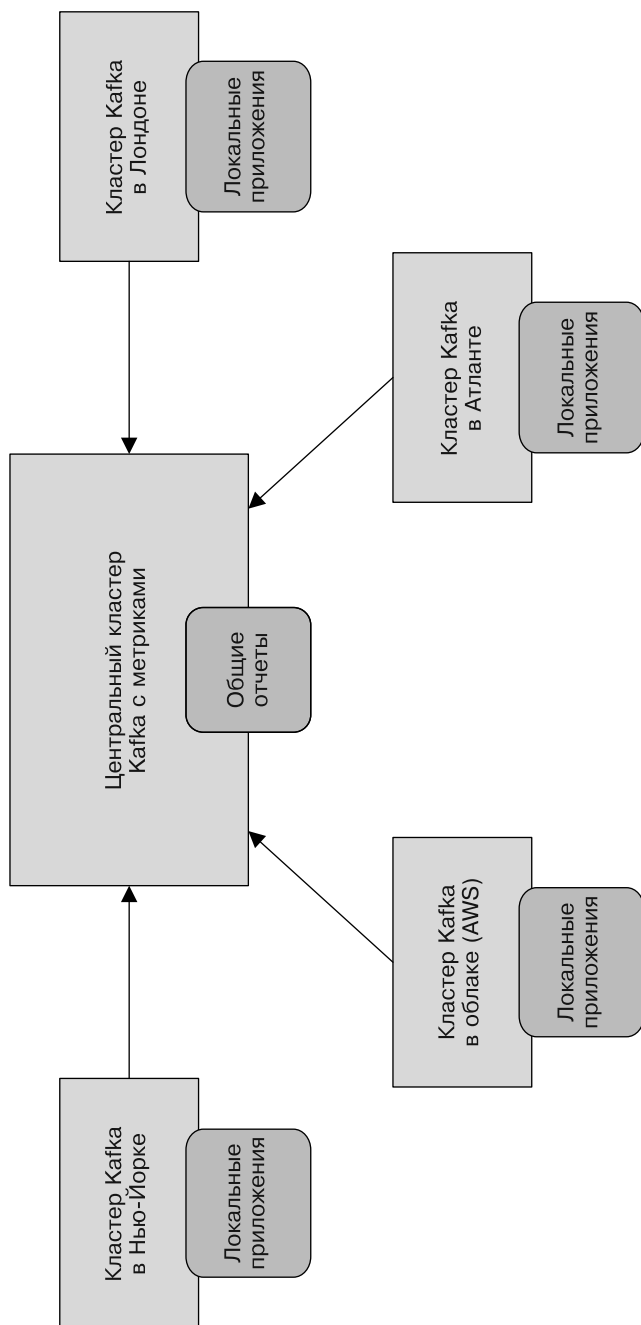


Рис. 8.1. Архитектура с топологией типа «звезда»

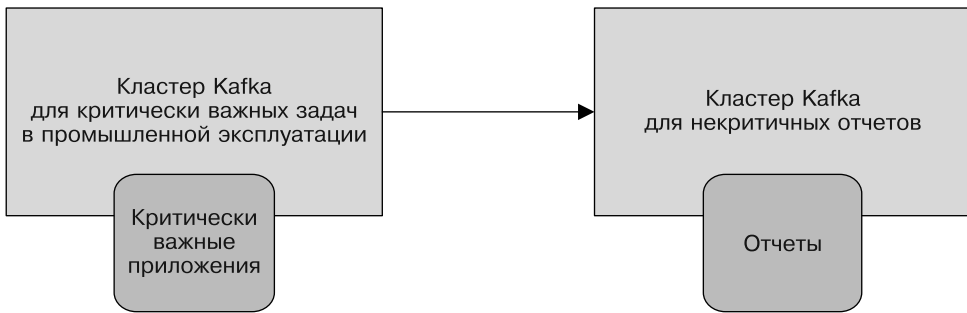


Рис. 8.2. Упрощенная версия архитектуры с топологией типа «звезда»

Основные недостатки этой архитектуры вытекают 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, используемый 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 их в центральный кластер. Если в нескольких ЦОД существует одна и та же тема, можно записать все события из нее в одну тему с тем же названием на центральном кластере или заносить события из каждого ЦОД в отдельную тему.

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. 8.3).

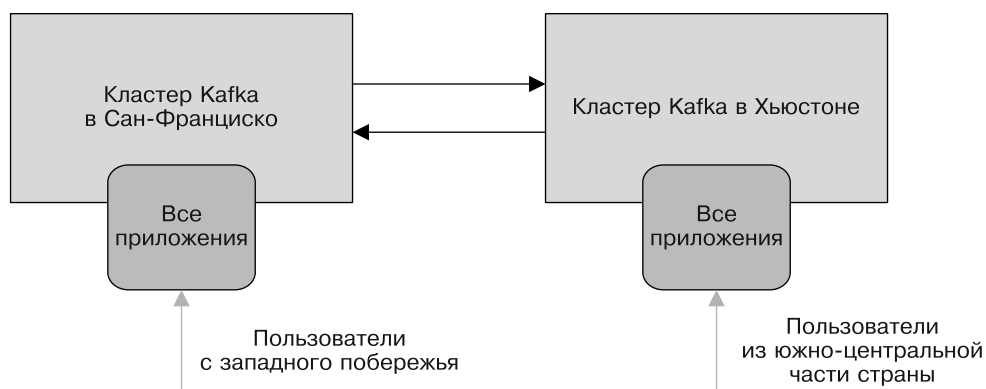


Рис. 8.3. Модель архитектуры типа «активный — активный»

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 не просто 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 в 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 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, кроме 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 будет 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, 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 использования 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 затратоэффективная 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 понадобится процесс зеркального копирования для каждой пары ЦОД и каждого направления. При пяти ЦОД придется поддерживать работу как минимум 20 процессов зеркального копирования, а вероятнее всего 40, поскольку для высокой доступности необходима избыточность.

Кроме того, необходимо избегать заикливания — бесконечного зеркального копирования одного и того же события туда и обратно. Для этого можно выделить для каждой логической темы отдельную тему в каждом ЦОД и не реплицировать темы, производителем которых является удаленный ЦОД. Например, логическая тема *users* будет называться *SF.users* в одном ЦОД и *NYC.users* — в другом. При зеркальном копировании *SF.users* будет скопирована 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 обе темы, *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 темы. В нашем примере речь идет о пространствах имен NYC и SF.

Отметим, что в ближайшем будущем (возможно, еще до того, как к вам в руки попадет эта книга) в Apache Kafka появятся заголовки записей¹. Это позволит пометать записи ЦОД, в котором они впервые появились, предотвращая с помощью

¹ Были добавлены в версии 0.11.0.0. См. KAFKA-4208. — *Примеч. пер.*

информации из этого заголовка бесконечные циклы зеркального копирования, а также организовывать обработку событий из разных ЦОД по отдельности. Эту функциональность можно реализовать и с помощью структурированного формата данных для значений записей (мой 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 тексасском ЦОД, вероятно, будут бездействующие («холодные») копии всех приложений, которые администраторы могут запустить в случае чрезвычайной ситуации и которые будут применять второй кластер (рис. 8.4). Зачастую это требование закона, а не реальный план действий, но все равно лучше быть готовым к подобной ситуации.

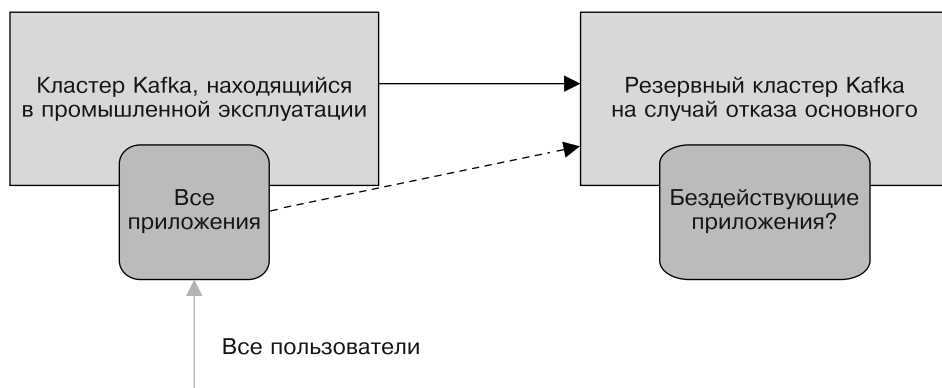


Рис. 8.4. Архитектура типа «активный — резервный»

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 особенностями.

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 — 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.

Потери данных и несогласованности 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 стоит ожидать, что DR-кластер

будет отставать от основного на несколько сотен или даже тысяч сообщений. Если ваш кластер Kafka обрабатывает миллион 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 в Kafka сейчас нет понятия транзакций, так что если часть событий из нескольких тем 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

Вероятно, сложнее всего 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. В случае использования старых потребителей, фиксирующих смещения в ZooKeeper, если не производится зеркальное копирование этих смещений в рамках плана DR, необходимо выбрать один из этих параметров: либо читать с начала имеющихся данных и обрабатывать большое количество дубликатов, либо перескочить в конец раздела, пропустив при этом некое (надемся, небольшое) число событий.
- ❑ *Репликация темы для смещений.* Потребители новых (0.9 и выше) версий Kafka будут фиксировать смещения в специальную тему `__consumer_offsets`. Если зеркально копировать ее 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 тему через неделю после ее создания. В этом случае первым смещением в основном кластере будет, например, смещение 57000000 (старые события за первые четыре дня уже удалены), а первое смещение 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 темы, так что и темы основного кластера, и DR-темы будут начинаться со смещения 0, а повторы попыток отправки производителем могут привести к расхождению смещений. Проще говоря, не существует варианта зеркального копирования Kafka, которое сохраняло бы 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 отсутствия в настоящий момент в Kafka транзакций 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. 8.5).

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 смещения в основном 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 темы или перескочить к ее 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.* При использовании действительно новых (0.10.0 и выше) версий потребителей Kafka каждое сообщение включает метку даты/времени, соответствующую времени отправки сообщения в Kafka. В еще более новых версиях (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?

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.

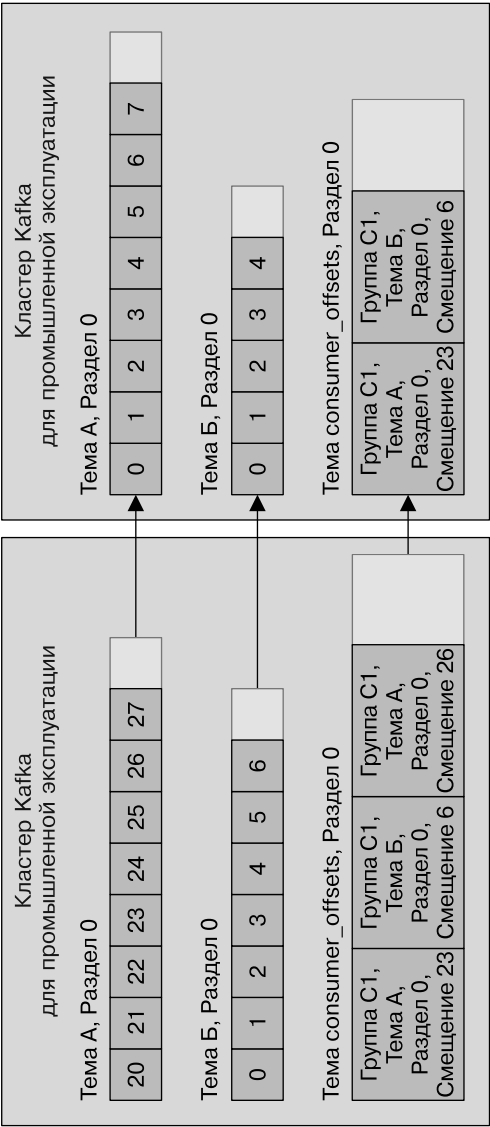


Рис. 8.5. Переключение при сбое приводит к появлению зафиксированных смещений без соответствующих записей

Этот параметр — замечательная вещь, если сразу писать все приложения таким образом. Но что если вы уже создали приложение без него? Можно без особых проблем написать небольшую утилиту, которая будет принимать на входе метку даты/времени, извлекать для нее смещения с помощью новых API, после чего фиксировать смещения для списка разделов и тем как конкретной группы потребителей. Мы надеемся добавить такую утилиту в Kafka в ближайшем будущем, но вы можете написать ее и самостоятельно. При запуске подобной утилиты необходимо приостановить работу группы потребителей и возобновить ее непосредственно после завершения работы утилиты.

Это вариант можно рекомендовать для тех, кто использует новые версии Kafka, хочет быть уверенным в том, что при аварийном переключении система будет вести себя правильно, и не против написать немного собственного кода в процессе.

- ❑ *Внешнее хранение соответствий смещений.* Одна из главных проблем при зеркальном копировании темы для смещений состоит в возможных расхождениях смещений в основном и DR-кластере. Из-за этого некоторые компании предпочитают использовать внешнее хранилище данных, например Apache Cassandra, для хранения соответствий смещений одного кластера смещениям другого. Они настраивают свои утилиты зеркального копирования Kafka так, что при генерации события для отправки в DR-кластер оба смещения отправляются во внешнее хранилище. Или же сохраняют оба смещения только в случае изменения различия между ними. Например, если смещение 495 на основном кластере соответствует смещению 500 в DR-кластере, мы записываем во внешнее хранилище пару (495, 500). Если в дальнейшем разница меняется и смещение 596 соответствует уже смещению 600, то записываем новое соответствие (596, 600). Необходимости хранить все промежуточные соответствия смещений между 495 и 596 нет, 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. При этом реализуется один из двух описанных ранее методов, чтобы потребители начинали чтение с новых смещений из карты соответствий. Однако 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.

Это решение довольно сложно и, по моему мнению, практически никогда не оправдывает дополнительных усилий. Оно появилось еще в те времена, когда не существовало индексов, которые можно использовать для аварийного переключения. Сейчас я предпочел бы обновить кластер и воспользоваться переключением по времени, а не тратить усилия на хранение соответствий смещений, все равно не охватывающее все сценарии аварийного переключения.

После аварийного переключения

Предположим, что аварийное переключение было успешным. Все прекрасно работает на DR-кластере. Теперь нам нужно что-то сделать с основным кластером. Например, сформировать 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 возникают 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 — зачастую и то и другое.
- ❑ Кроме того, по обсуждавшимся выше 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 не согласованными.

Поэтому 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 — клиентам 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.

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 первоначального. *Эластичные кластеры* (stretch clusters) предназначены для предотвращения отказа кластера Kafka в случае сбоя всего ЦОД. Это достигается за счет установки одного кластера Kafka в нескольких ЦОД.

Эластичные кластеры принципиально отличаются от других мультикластерных архитектур. Начать нужно с того, 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.

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 указанное 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, а значит, останется доступен кластер ZooKeeper. А следовательно, и кластер Kafka.

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 простая утилита для зеркального копирования данных между двумя ЦОД. Она называется MirrorMaker и представляет собой набор потребителей (по историческим причинам именуемых в документации MirrorMaker *потоками данных* (streams)), состоящих в одной группе и читающих данные из выбранного вами для репликации набора тем. У каждого из процессов MirrorMaker имеется один производитель. Последовательность выполняемых действий очень проста: MirrorMaker запускает для каждого потребителя поток выполнения. Потребители читают события из назначенных им тем и разделов исходного кластера и используют разделяемый производитель для отправки этих событий на целевой кластер. Каждые 60 секунд (по умолчанию) потребители сообщают производителю, чтобы он отправил все имеющиеся события в Kafka и подождет от нее подтверждения их успешного получения. Затем потребители обращаются к исходному кластеру Kafka для фиксации смещений этих событий. Такая схема гарантирует отсутствие потерь данных (Kafka подтверждает получение сообщений до фиксации смещений в исходном кластере), а в случае аварийного сбоя процесса MirrorMaker образуются дубликаты максимум за 60 секунд (рис. 8.6).

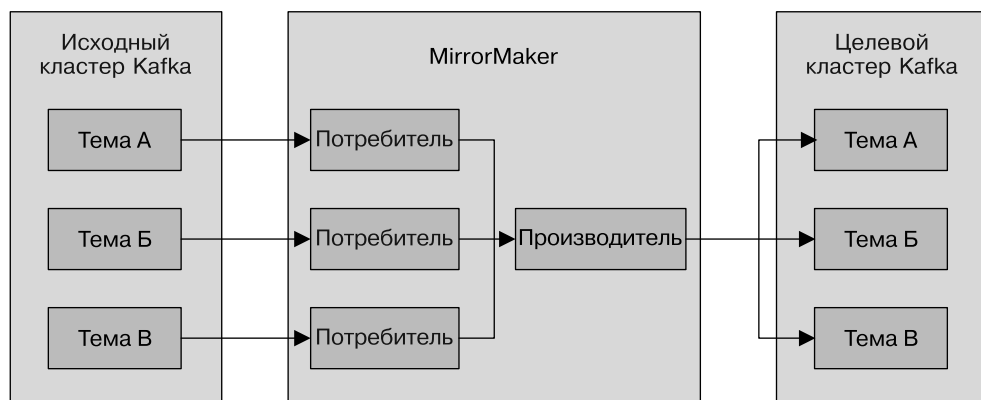


Рис. 8.6. Процесс MirrorMaker в Kafka



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. К моменту выхода версии Kafka 0.10.0.0 MirrorMaker переписывалась уже четырежды и, возможно, не в последний раз. Приведенные в следующих разделах описание и различные подробности относятся к MirrorMaker такой, какой она была версиях Kafka с 0.9.0.0 по 0.10.2.0.

Настройка MirrorMaker

У MirrorMaker очень широкие возможности настройки. Во-первых, утилита использует один производитель и несколько потребителей, так что при ее настройке можно задействовать все конфигурационные параметры производителей и потребителей. Кроме того, у самой MirrorMaker есть обширный список параметров конфигурации, зачастую взаимосвязанных довольно запутанным образом. 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:

```
bin/kafka-mirror-maker --consumer.config etc/kafka/consumer.properties --producer.config etc/kafka/producer.properties --new.consumer --num.streams=2 --whitelist ".*"
```

Рассмотрим основные аргументы командной строки MirrorMaker:

- ❑ **consumer.config.** Это настройки для всех потребителей, извлекающих данные из исходного кластера. Конфигурационный файл у них один на всех, а значит, можно использовать только один исходный кластер и один `group.id`. Поэтому все потребители окажутся в составе одной группы потребителей — это именно то, что нам и требуется. Обязательные настройки в этом файле — `bootstrap.servers` (для исходного кластера) и `group.id`. Но можете задать любые необходимые вам дополнительные настройки. А вот параметр `auto.commit.enable=false` вам вряд ли стоит менять. Для MirrorMaker необходима возможность фиксировать свои смещения после их успешной доставки в целевой кластер Kafka. Изменение этого параметра может привести к потере данных. Имеет смысл также поменять значение параметра `auto.offset.reset`. По умолчанию оно равно `latest`, то есть MirrorMaker будет зеркально копировать только события, поступившие в исходный кластер после запуска MirrorMaker. Если вы предпочли бы скопировать и существующие данные, поменяйте его значение на `earliest`. Мы обсудим другие настройки конфигурации в разделе «Тонкая настройка MirrorMaker» далее в этой главе.
- ❑ **producer.config.** Эти настройки производителей используются MirrorMaker для записи в целевой кластер. Единственная обязательная настройка среди

них — `bootstrap.servers` (для целевого кластера). Обсудим дополнительные настройки конфигурации в разделе «Тонкая настройка MirrorMaker».

- ❑ `new.consumer`. MirrorMaker может задействовать потребитель версии 0.8 или 0.9. Рекомендуем использовать потребители версии 0.9, поскольку они в настоящий момент стабильнее.
- ❑ `num.streams`. Как мы уже объясняли, каждый поток означает еще один потребитель, читающий данные из исходного кластера. Помните, что все потребители из одного процесса MirrorMaker делят между собой одни и те же производители. Для потребления всех данных от производителя потребуется несколько потоков. Для большего повышения производительности понадобится еще один процесс MirrorMaker.
- ❑ `whitelist`. Регулярное выражение для названий зеркально копируемых тем. Все темы, чьи названия соответствуют данному регулярному выражению, будут копироваться. В данном примере мы реплицируем их все, но часто имеет смысл использовать регулярное выражение вроде `prod.*`, чтобы не реплицировать тестовые темы. Или можно при архитектуре типа «активный — активный» задать для MirrorMaker, копирующего данные из нью-йоркского ЦОД в ЦОД Сан-Франциско, значение этого параметра равным `whitelist="NYC.*"`, чтобы избежать обратного копирования тем из Сан-Франциско.

Развертывание MirrorMaker для промышленной эксплуатации

В предыдущем примере MirrorMaker запускалась как утилита командной строки. 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. Формально у MirrorMaker есть параметр командной строки `-daemon`, которая делает все перечисленное, но в последних выпусках Kafka она не работает как полагается.

У большинства использующих MirrorMaker компаний написаны собственные сценарии запуска, включающие применяемые параметры. Для автоматизации развертывания и управления параметрами и файлами конфигураций задействуются также такие системы развертывания для промышленной эксплуатации, как Ansible, Puppet, Chef и Salt.

Более продвинутый вариант развертывания, который становится все более популярным, — запуск MirrorMaker в контейнере Docker. Утилита MirrorMaker совершенно не сохраняет состояние, для нее не нужно никакого дискового хранилища — все данные и состояние хранятся в самой Kafka. Благодаря использованию Docker в качестве адаптера для MirrorMaker становится возможной также

работа нескольких экземпляров утилиты на одной машине. Поскольку отдельный экземпляр MirrorMaker ограничивается пропускной способностью одного производителя, часто бывает важно запустить несколько экземпляров MirrorMaker, а Docker эту задачу облегчает. Он также упрощает вертикальное масштабирование в обе стороны — можно добавить контейнеры, если возникает потребность в дополнительной пропускной способности в период пиковой нагрузки, а затем убрать их. При работе MirrorMaker в облачной среде можно даже подключать дополнительные серверы, чтобы запускать на них контейнеры в зависимости от нагрузки и потребностей в обслуживании.

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 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 эти события 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, то 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. Воздействует оно и на сами брокеры Kafka. Если трафик между ЦОД требует шифрования, 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 схему локального потребления и удаленной генерации, позаботьтесь, чтобы MirrorMaker никогда не теряла событий, задав параметр `acks=all` и достаточное число повторов попыток. Настройте MirrorMaker так, чтобы она завершала работу, если отправить события невозможно, — обычно это безопаснее, чем продолжать работу и рисковать потерей данных.

Если важно понизить до минимума отставание целевого кластера от исходного, лучше запустить не менее двух экземпляров MirrorMaker на двух различных серверах с использованием обоими одной и той же группы потребителей. В случае

останова одного из серверов второй экземпляра MirrorMaker продолжит зеркальное копирование данных.

При развертывании MirrorMaker для промышленной эксплуатации важно не забывать контролировать ее работу.

- *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. 8.7).

На рис. 8.7 последнее смещение исходного кластера равно 7, а последнее смещение целевого — 5, то есть величина отставания составляет два сообщения.

Существует два способа отслеживания этого отставания, но 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 она делает это раз в минуту, так что вы обнаружите, что отставание растет в течение минуты, после чего неожиданно резко уменьшается. На схеме фактическое отставание равно 2, но утилита `kafka-consumer-groups` покажет, что оно равно 4, поскольку MirrorMaker пока еще не зафиксировала смещения для более новых сообщений. Утилита Burrow 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 заполняет эту брешь в мониторинге, контролируя количество сообщений и контрольные суммы.

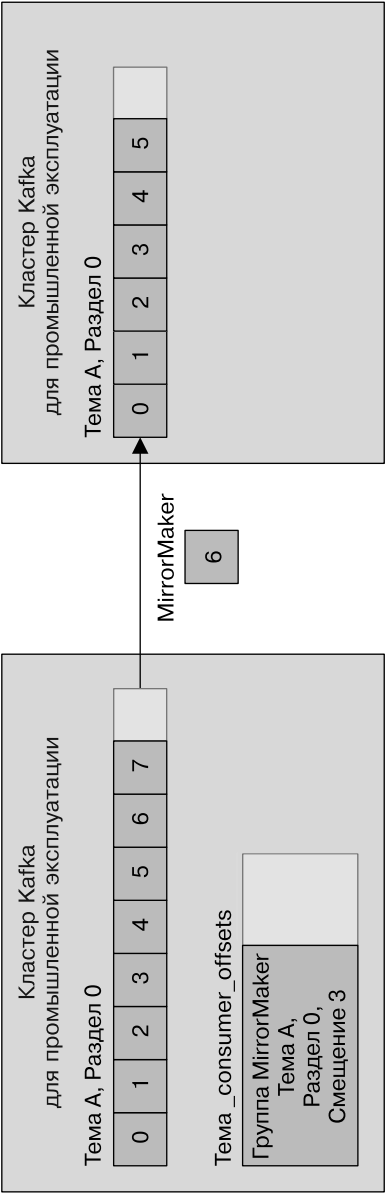


Рис. 8.7. Мониторинг величины отставания в смысле смещений

- ❑ *Мониторинг показателей.* 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. 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»¹. Если вы контролируете все остальное, «канарейка» не нужна, но иметь ее в качестве дополнительного слоя мониторинга не помешает. Она представляет собой процесс, который ежеминутно отправляет событие в специальную тему в исходном кластере, после чего пытается прочесть это событие из целевого кластера. И уведомляет вас в случае, если передача этого события занимает слишком много времени (то есть MirrorMaker отстает или вообще не работает).

Тонкая настройка MirrorMaker

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 использовалась 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 она его наверстает.

Далее неплохо бы оценить пропускную способность MirrorMaker при различном количестве потоков потребителей — оно настраивается с помощью параметра `num.streams`. Мы можем дать некоторые приближенные оценки (для LinkedIn это 6 Мбайт/с при 8 потоках потребителей и 12 Мбайт/с при 16), но 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) на исходном кластере, а затем подключить MirrorMaker и начать эти данные зеркально копировать. Протестируйте MirrorMaker при 1, 2, 4, 8, 16 и 32 потоках потребителей. Найдите число потоков, при котором производительность начинает падать, и установите значение параметра

¹ Название напоминает о том, что долгое время шахтеры использовали канареек в качестве высокочувствительных живых детекторов рудничного газа. — *Примеч. пер.*

тра `num.streams` чуть меньше него. 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 потоков выполнения. Таким образом вы сможете определить максимальную пропускную способность, достижимую с помощью одного экземпляра MirrorMaker. Если ее недостаточно, попробуйте добавить дополнительные экземпляры, а затем и дополнительные серверы.

Кроме того, полезно будет выделить важные темы — те, для которых обязательно низкое значение задержки, так что кластер-зеркало должен находиться как можно ближе к исходному кластеру — в отдельном кластере MirrorMaker со своей группой потребителей. Это позволит предотвратить замедление работы самого важного из ваших конвейеров из-за слишком раздутой темы или вышедшего из-под контроля производителя.

По большому счету это все, что можно настроить в самой MirrorMaker. Однако у вас есть еще возможности повысить производительность потоков потребителей и экземпляров MirrorMaker.

Для оптимизации настроек сети в операционной системе Linux можно сделать следующее:

- ☐ увеличить размер буфера TCP (`net.core.rmem_default`, `net.core.rmem_max`, `net.core.wmem_default`, `net.core.wmem_max`, `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 работающие в MirrorMaker производители и потребители. Во-первых, нужно разобраться, производитель или потребитель представляет собой узкое место — производитель ожидает, пока потребитель прочитает данные, или наоборот? Один из способов выяснения этого — 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 большая 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:

- ❑ `max.in.flight.requests.per.connection`. По умолчанию MirrorMaker допускает наличие только одного активного запроса. Это значит, что прежде чем можно будет отправить следующее сообщение, целевой кластер должен подтвердить получение каждого отправленного производителем запроса. Это может ограничить пропускную способность, особенно если брокеры подтверждают получение со значительной задержкой. MirrorMaker ограничивает число активных запросов, потому что это единственный способ гарантировать сохранение Kafka порядка сообщений в том случае, если отправку некоторых из них придется повторить несколько раз, прежде чем их успешное получение будет подтверждено. Если для сценария использования не важен порядок сообщений, то можно повысить значение параметра `max.in.flight.requests.per.connection` и значительно увеличить пропускную способность;
- ❑ `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.

Следующие параметры конфигурации потребителя могут помочь увеличить его пропускную способность.

- ❑ Стратегия распределения разделов по потребителям в MirrorMaker (то есть алгоритм выбора того, какому потребителю какие разделы назначаются) со значением по умолчанию `range`. В стратегии `range` есть свои преимущества, именно поэтому она используется по умолчанию, но она же может обусловить неравномерное распределение разделов по потребителям. Применительно к MirrorMaker обычно лучше изменить стратегию на циклическую, особенно в случае зеркального копирования большого количества тем и разделов. Для этого в файл конфигурации потребителя необходимо добавить параметр `partition.assignment.strategy=org.apache.kafka.clients.consumer.RoundRobinAssignor`.
- ❑ `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 с брокера столько данных, сколько ему разрешается. Если у вас есть резервы памяти, можете попробовать увеличить значение па-

параметра `fetch.max.bytes`, позволив тем самым потребителю читать больший объем данных при каждом запросе.

- ❑ `fetch.min.bytes` и `fetch.max.wait` — 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`, 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 на то, как они обходят его ограничения и проблемы.

uReplicator компании Uber

Компания Uber использовала MirrorMaker очень широко и по мере роста числа тем и разделов и повышения производительности кластера столкнулась со следующими проблемами.

- ❑ *Задержки из-за перераспределения.* MirrorMaker применяют обычные потребители. Добавление потоков и экземпляров MirrorMaker, перезапуск экземпляров MirrorMaker или даже просто добавление новых тем, соответствующих регулярному выражению из параметра `whitelist`, — все это приводит к перераспределению потребителей. Как мы видели в главе 4, это вызывает останов всех потребителей до тех пор, пока всем им не будут назначены новые разделы. При очень большом числе тем 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 огромного объема событий, требующих зеркального копирования. Восстановление после подобной ситуации могло быть длительным. В результате возникала очень большая задержка чтения событий потребителями из целевого кластера.
- ❑ *Сложности при добавлении тем.* Использование регулярного выражения в качестве `whitelist` темы означает, что MirrorMaker придется выполнять перераспределение при каждом добавлении соответствующей темы в исходный кластер. Ранее мы уже видели, что в Uber перераспределение приводило к особенно неприятным последствиям. Чтобы избежать подобных внезапных

перераспределений, в компании решили просто перечислять все темы, которые нужно зеркально копировать. Но это значит, что приходилось вручную добавлять все подлежащие копированию новые темы в `whitelist` на всех экземплярах `MirrorMaker` и перезапускать эти экземпляры, что все равно приводило к перераспределению. Конечно, оно происходило при запланированном техобслуживании, а не всякий раз, когда кто-нибудь добавлял тему, но объем трудозатрат все равно был большим. Это также означало, что в случае ошибки при техобслуживании, вследствие которой у разных экземпляров окажутся разные списки тем, `MirrorMaker` после запуска станет бесконечно производить перераспределение, поскольку потребители не смогут договориться, на какую тему кому подписываться.

Из-за этих проблем Uber пришлось написать собственный клон утилиты `MirrorMaker`, получивший название `uReplicator`. В компании решили воспользоваться `Apache Helix` в качестве центрального высокодоступного контроллера, который отвечал бы за список тем и разделов, назначаемых различным экземплярам `uReplicator`. В нем для добавления новых тем 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 потребителя Kafka, который они написали сами 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 можете найти в главе 4). В результате потребитель `Helix` избегает перераспределений, прослушивая на предмет поступающих от `Helix` изменений в назначениях разделов.

В блоге Uber написали сообщение, где описали архитектуру более подробно и показали, каких положительных результатов достигли. На момент написания данной книги нам неизвестно о каких-либо компаниях, кроме Uber, которые использовали бы `uReplicator`. Вероятно, это следствие того, что большинство компаний не работают в таких масштабах, как Uber, и не сталкиваются с подобными проблемами, или того, что зависимость от `Apache Helix` означает необходимость изучения совершенно нового компонента и работы с ним, что усложняет проект в целом.

Replicator компании Confluent

Параллельно с разработкой `uReplicator` компанией Uber в компании Confluent независимо создали свой `Replicator`. Несмотря на схожесть названий, проекты не имеют практически ничего общего — это два различных решения двух различных наборов проблем `MirrorMaker`. `Replicator` компании Confluent был разработан для решения проблем, с которыми сталкиваются корпоративные заказчики, когда используют `MirrorMaker` при развертывании своих мультикластерных архитектур.

- ❑ *Расхождение конфигураций кластеров.* `MirrorMaker` обеспечивает согласованность данных в исходном и целевом кластерах, но согласованность только их. В результате у тем могут появиться различные число разделов, разные коэффи-

циенты репликации и различные настройки уровня темы. Если вы увеличите длительность хранения информации с одной до трех недель на исходном кластере и забудете проделать это на DR-кластере, то при аварийном переключении вас ждет довольно неприятный сюрприз в виде потери информации за несколько недель. Попытка поддержания согласованности всех этих настроек вручную может вызвать ошибки и сбой расположенных далее по конвейеру приложений или даже самой репликации в случае рассогласованности подсистем.

- ❑ *Проблемы управления кластером.* Мы уже видели, что MirrorMaker обычно развертывается в виде кластера из нескольких экземпляров. А это еще один кластер, нужно как-то развернуть его, управлять им и выполнять мониторинг. Управление конфигурацией MirrorMaker из-за двух файлов конфигурации и большого числа параметров и так представляет собой непростую задачу. Ее сложность только возрастает, если кластеров больше двух, а репликация односторонняя. При трех кластерах и архитектуре типа «активный — активный» вам придется развертывать шесть кластеров MirrorMaker, вероятно, с тремя экземплярами в каждом как минимум, выполнять их мониторинг и настройку. При пяти кластерах типа «активный — активный» количество кластеров MirrorMaker возрастает до 20.

Компания Confluent ради минимизации издержек администрирования и без того загруженных корпоративных IT-отделов приняла решение реализовать Replicator в виде коннектора производителя для фреймворка Kafka Connect, который бы читал данные из другого кластера Kafka, а не из базы данных. Вспомните архитектуру Kafka Connect из главы 7: каждый коннектор делит нагрузку между задачами, число которых можно настроить. В Replicator каждая задача представляет собой пару «потребитель/производитель». Фреймворк Connect распределяет эти задачи по различным рабочим узлам по мере необходимости, так что все задачи могут быть на одном сервере или распределяться по нескольким. Таким образом, отпадает необходимость вручную подсчитывать, сколько потоков MirrorMaker должно приходиться на один экземпляр и сколько экземпляров — на машину. В Kafka Connect есть также API REST для централизованного управления настройками коннекторов и задач. Если считать, что по другим причинам в большинство схем развертывания Kafka включен Kafka Connect (очень популярный сценарий использования — отправка событий изменения базы данных в Kafka), то можно уменьшить число кластеров, которыми нужно будет управлять, за счет запуска Replicator внутри Kafka Connect. Другое важное преимущество состоит в том, что коннектор Replicator реплицирует не только данные из списка тем Kafka, но и настройки тем ZooKeeper.

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 утилитами, начав с MirrorMaker Apache Kafka и обсудив немало нюансов ее промышленной эксплуатации. Завершили главу обзором двух 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.

9

Администрирование Kafka

Kafka предоставляет пользователям несколько утилит командной строки (CLI), удобных для администрирования кластеров. Они реализованы в виде классов Java, для правильного вызова которых имеются наборы сценариев. Эти утилиты позволяют выполнять простейшие действия, но более сложные операции реализовать с их помощью нельзя. В этой главе мы опишем доступные в проекте Apache Kafka 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 темами, но для большинства операций над кластером они пока не поддерживаются. Это значит, что вышеупомянутые утилиты командной строки можно использовать без всякой аутентификации, то есть такие операции, как изменение темы, можно выполнять без какой-либо проверки на безопасность или аудита. Соответствующая функциональность находится в процессе разработки и скоро будет добавлена в Kafka.

Операции с темами

Утилита `kafka-topics.sh` позволяет легко выполнить большинство операций с темами (изменять настройки тем с ее помощью не рекомендуется, соответствующая функциональность была перенесена в утилиту `kafka-configs.sh`). Она позволяет создавать, менять, удалять и выводить информацию об имеющихся в кластере темах. Для ее использования необходимо задать строку подключения ZooKeeper для кластера с помощью аргумента `--zookeeper`. В приводимых далее примерах предполагается, что строка подключения ZooKeeper имеет вид `zoo1.example.com:2181/kafka-cluster`.



Проверьте версию

Многие утилиты командной строки Kafka работают непосредственно с хранимыми в ZooKeeper метаданными, а не подключаются 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 новой темы

Для создания новой темы в кластере необходимо задать три аргумента (их следует указывать, несмотря на то что для некоторых из них могут быть заданы значения по умолчанию на уровне брокера):

- ❑ *название темы* — название создаваемой темы;
- ❑ *коэффициент репликации* — количество реплик темы в кластере;
- ❑ *разделы* — число создаваемых для данной темы разделов.



Задание настроек темы

Во время создания темы можно также явным образом указывать для нее реплики или переопределять настройки. Ни одну из этих операций мы не будем рассматривать. Сведения о переопределении настроек вы можете найти далее в этой главе. Передать их сценарию `kafka-topics.sh` можно с помощью параметра командной строки `--config`. Переназначение разделов также описывается далее в этой главе.

Названия тем могут содержать алфавитно-цифровые символы, а также символы подчеркивания, тире и точки.



Именованние тем

Разрешается, хотя и не рекомендуется начинать названия тем с двух символов подчеркивания. Подобные темы считаются предназначенными для внутренних целей кластера (как, например, `__consumer_offsets`, предназначенная для хранения смещений для группы потребителей). Не рекомендуется применять в названиях в одном кластере и точки, и подчеркивания, поскольку при использовании названий тем в названиях показателей внутри Kafka точки заменяются подчеркиваниями (например, `topic.1` в показателях превращается в `topic_1`).

Выполните сценарий `kafka-topics.sh` со следующими параметрами:

```
kafka-topics.sh --zookeeper <zookeeper connect> --create --topic <string>
--replication-factor <integer> --partitions <integer>
```

В результате кластер создаст тему с заданными названием и количеством 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 тему `my-topic` с 8 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".
#
```



Игнорирование ошибки в том случае, когда тема уже существует

При использовании этого сценария для автоматизации может оказаться полезным аргумент `--if-not-exists`, при котором не возвращается ошибка в случае, когда тема уже существует.

Добавление разделов

Иногда оказывается нужно увеличить количество разделов темы. Темы масштабируются и реплицируются в кластере посредством разделов, так что чаще всего число разделов увеличивают при необходимости расширения темы или снижения нагрузки на отдельный раздел. Можно также увеличивать темы, если требуется несколько экземпляров одного и того же потребителя в рамках одной группы, поскольку только один участник группы потребителей может читать один раздел.



Тонкая настройка тем с ключами

С точки зрения потребителей добавление разделов в темы, 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 тем, содержащих сообщения с ключами, однократно при их создании и стараться не менять их размер.



Игнорирование ошибки в случае, когда темы не существует

Хотя для команды `--alter` существует аргумент `--if-exists`, использовать его не рекомендуется, поскольку в этом случае команда не будет возвращать ошибки, если модифицируемой темы не существует. Это может замаскировать проблему с отсутствием нужной темы.

Например, увеличим количество разделов для темы my-topic до 16:

```
# kafka-topics.sh --zookeeper zoo1.example.com:2181/kafka-cluster
--alter --topic my-topic --partitions 16
WARNING: If partitions are increased for a topic that has a key,
the partition logic or ordering of the messages will be affected
Adding partitions succeeded!
#
```



Уменьшение количества разделов

Уменьшить количество разделов темы невозможно. Причина, по которой эта возможность не поддерживается, такова: удаление раздела из темы 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 следует удалить тему и создать ее заново.

Удаление темы

Даже не содержащая сообщений тема расходует ресурсы кластера, включая дисковое пространство, открытые дескрипторы файлов и оперативную память. Если тема больше не нужна, следует ее удалить, чтобы освободить эти ресурсы. Для этого параметр конфигурации брокеров кластера `delete.topic.enable` должен быть равен `true`. Если же этот параметр установлен в `false`, запросы на удаление тем будут проигнорированы.



Данные будут утеряны

Удаление темы приводит к удалению всех ее сообщений. Эта операция необратима, так что выполняйте ее осторожно.

Например, удалим тему с названием my-topic:

```
# kafka-topics.sh --zookeeper zoo1.example.com:2181/kafka-cluster
--delete --topic my-topic
Topic my-topic is marked for deletion.
Note: This will have no impact if delete.topic.enable is not set to true.
#
```

Вывод списка всех тем кластера

Утилита для работы с темами может также вывести список всех тем в кластере. Формат списка — по одной теме в строке, упорядоченность отсутствует.

Например, выведем список всех тем кластера:

```
# kafka-topics.sh --zookeeper zoo1.example.com:2181/kafka-cluster
--list
my-topic - marked for deletion
other-topic
#
```

Подробное описание тем

В числе прочего можно получить подробную информацию по одной или нескольким темам кластера. Выводимая информация включает количество разделов, переопределения настроек тем и список разделов с распределением реплик. Можно ограничиться информацией по одной теме, указав для команды аргумент `--topic`.

Например, выведем описание всех тем кластера:

```
# kafka-topics.sh --zookeeper zoo1.example.com:2181/kafka-cluster --describe
Topic:other-topic    PartitionCount:8      ReplicationFactor:2  Configs:
Topic:other-topic    Partition: 0          ... Replicas: 1,0       Isr: 1,0
Topic:other-topic    Partition: 1          ... Replicas: 0,1       Isr: 0,1
Topic:other-topic    Partition: 2          ... Replicas: 1,0       Isr: 1,0
Topic:other-topic    Partition: 3          ... Replicas: 0,1       Isr: 0,1
Topic:other-topic    Partition: 4          ... Replicas: 1,0       Isr: 1,0
Topic:other-topic    Partition: 5          ... Replicas: 0,1       Isr: 0,1
Topic:other-topic    Partition: 6          ... Replicas: 1,0       Isr: 1,0
Topic:other-topic    Partition: 7          ... 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 темы или разделы кластера, соответствующие заданному критерию. Эти параметры не работают с командой `list`, описанной в предыдущем разделе.

Для поиска всех тем, у которых были переопределены настройки, воспользуйтесь аргументом `--topics-with-overrides`. При этом будут выведены только те темы, чьи настройки отличаются от настроек кластера по умолчанию.

Существует два фильтра, которые можно использовать для поиска проблемных разделов. Аргумент `--under-replicated-partitions` выводит все разделы, одна или более реплик которых не согласованы с ведущей репликой. Аргумент `--unavailable-partitions` выводит все разделы, у которых нет ведущей реплики. Это более серьезная проблема, означающая, что раздел в настоящий момент отключен и недоступен для клиентов-потребителей и клиентов-производителей.

Например, выведем список недореплицированных разделов:

```
# kafka-topics.sh --zookeeper zoo1.example.com:2181/kafka-cluster
--describe --under-replicated-partitions
    Topic: other-topic    Partition: 2    Leader: 0    Replicas: 1,0
    Isr: 0
    Topic: other-topic    Partition: 4    Leader: 0    Replicas: 1,0
    Isr: 0
#
```

Группы потребителей

Управление группами потребителей в Kafka осуществляется в двух местах: информация для потребителей старых версий хранится в ZooKeeper, новых — в брокерах Kafka. Утилиту `kafka-consumer-groups.sh` можно использовать для вывода и описания обоих типов групп, а также для удаления информации о группах потребителей и смещениях, но только если это группы потребителей старых версий, хранимых в ZooKeeper. Работая с группами потребителей старых версий, можно обращаться к кластеру Kafka посредством задания для утилиты параметра командной строки `--zookeeper`. А для групп потребителей новых версий необходимо использовать параметр `--bootstrap-server` с указанием имени хоста и порта брокера Kafka, к которому должно осуществляться подключение.

Вывод списка и описание групп

Для вывода списка групп потребителей, если это старые клиенты-потребители, необходимо использовать параметры `--zookeeper` и `--list`. Для новых воспользуйтесь параметрами `--bootstrap-server`, `--list` и `--new-consumer`.

Например, выведем список групп старой версии:

```
# kafka-consumer-groups.sh --zookeeper
zoo1.example.com:2181/kafka-cluster --list
console-consumer-79697
myconsumer
#
```

И список групп новой версии:

```
# kafka-consumer-groups.sh --new-consumer --bootstrap-server
kafka1.example.com:9092 --list kafka-python-test
my-new-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 темы, которые читает группа, а также смещения для всех разделов тем.

Например, выведем подробную информацию о группе потребителей старой версии с названием `testgroup`:

```
# kafka-consumer-groups.sh --zookeeper zoo1.example.com:2181/kafka-cluster
--describe --group testgroup
GROUP                                TOPIC                                PARTITION
CURRENT-OFFSET  LOG-END-OFFSET  LAG                                OWNER
myconsumer      1688            0                                my-topic      0
myconsumer_host1.example.com-1478188622741-7dab5ca7-0
myconsumer      1418            0                                my-topic      1
myconsumer_host1.example.com-1478188622741-7dab5ca7-0
myconsumer      1314            1                                my-topic      2
myconsumer_host1.example.com-1478188622741-7dab5ca7-0
myconsumer      2012            0                                my-topic      3
myconsumer_host1.example.com-1478188622741-7dab5ca7-0
myconsumer      1089            0                                my-topic      4
myconsumer_host1.example.com-1478188622741-7dab5ca7-0
myconsumer      1429            3                                my-topic      5
myconsumer_host1.example.com-1478188622741-7dab5ca7-0
myconsumer      1634            0                                my-topic      6
myconsumer_host1.example.com-1478188622741-7dab5ca7-0
myconsumer      2261            0                                my-topic      7
myconsumer_host1.example.com-1478188622741-7dab5ca7-0
#
```

В табл. 9.1 описаны выводимые здесь поля.

Таблица 9.1. Поля вывода информации о группе потребителей старой версии с названием `testgroup`

Поле	Описание
GROUP	Название группы потребителей
TOPIC	Название читаемой темы
PARTITION	Идентификатор читаемого раздела
CURRENT-OFFSET	Последнее смещение, зафиксированное группой потребителей для данного раздела темы. Представляет собой позицию потребителя в разделе
LOG-END-OFFSET	Текущее максимальное смещение для данного раздела темы из имеющихся в брокере
LAG	Разница между Current-Offset потребителя и Log-End-Offset брокера для данного раздела темы
OWNER	Член группы потребителей, который в настоящий момент выполняет потребление данного раздела темы. Представляет собой произвольный идентификатор, задаваемый самим членом группы. Не обязан содержать имя хоста потребителя

Удаление группы

Удаление группы потребителей поддерживается только для клиентов — потребителей старых версий. Это действие приводит к удалению из ZooKeeper всей группы, включая все сохраненные смещения для всех потребляемых группой тем. Чтобы выполнить удаление, необходимо прекратить работу всех потребителей группы. Если не сделать этого заранее, поведение потребителей может оказаться непредсказуемым из-за удаления метаданных ZooKeeper для группы во время их использования.

Например, удалим группу потребителей testgroup:

```
# kafka-consumer-groups.sh --zookeeper
zoo1.example.com:2181/kafka-cluster --delete --group testgroup
Deleted all consumer group information for group testgroup in zookeeper.
#
```

Ту же самую команду можно использовать и для удаления смещений для отдельной читаемой группой темы, не удаляя всю группу. Опять же рекомендуется предварительно остановить работу группы потребителей или исключить с помощью настроек потребление ею удаляемой темы.

Например, удалим смещения для темы my-topic из группы потребителей testgroup:

```
# kafka-consumer-groups.sh --zookeeper
zoo1.example.com:2181/kafka-cluster --delete --group testgroup
--topic my-topic
Deleted consumer group information for group testgroup topic my-topic in zookeeper.
#
```

Управление смещениями

Помимо отображения и удаления смещений групп потребителей с помощью клиентов старых версий 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 сообщение, обработать которое у потребителя не получается).



Управление зафиксированными в Kafka смещениями

Сейчас не существует утилиты для управления зафиксированными в Kafka смещениями клиента-потребителя. Эта возможность доступна только для потребителей, фиксирующих смещения в ZooKeeper. Чтобы управлять смещениями группы потребителей, фиксирующих смещения в Kafka, для фиксации смещений придется воспользоваться имеющимися API клиента.

Экспорт смещений

Именованного сценария для экспорта смещений не существует, но можно воспользоваться сценарием `kafka-run-class.sh`, чтобы выполнить соответствующий Java-класс в подходящей среде. В результате экспорта смещений будет создан файл, содержащий разделы всех тем группы и их смещения в подходящем для утилиты импорта формате. В этом файле в каждой строке будет по одному разделу темы в следующем формате:

```
/consumers/GROUPNAME/offsets/topic/TOPICNAME/PARTITIONID:OFFSET.
```

Например, экспортируем смещения группы потребителей `testgroup` в файл `offsets`:

```
# kafka-run-class.sh kafka.tools.ExportZkOffsets
--zkconnect zoo1.example.com:2181/kafka-cluster --group testgroup
--output-file offsets
# cat offsets
/consumers/testgroup/offsets/my-topic/0:8905
/consumers/testgroup/offsets/my-topic/1:8915
/consumers/testgroup/offsets/my-topic/2:9845
/consumers/testgroup/offsets/my-topic/3:8072
/consumers/testgroup/offsets/my-topic/4:8008
/consumers/testgroup/offsets/my-topic/5:8319
/consumers/testgroup/offsets/my-topic/6:8102
/consumers/testgroup/offsets/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 прошлого раздела, и устанавливает на его основе текущие смещения группы потребителей. Общепринятая практика — экспортировать текущие смещения группы потребителей, делать резервную копию файла и вносить в файл изменения, задавая для смещений нужные значения. Обратите внимание на то, что для команды импорта параметр `--group` не используется, поскольку название группы потребителей включено в импортируемый файл.



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 перекроют импортированные.

Например, импортируем смещения группы потребителей `testgroup` из файла `offsets`:

```
# kafka-run-class.sh kafka.tools.ImportZkOffsets --zkconnect
zoo1.example.com:2181/kafka-cluster --input-file offsets
#
```

Динамические изменения конфигурации

Существует возможность перекрытия настроек тем и квот клиентов во время работы кластера. В будущем планируется добавить еще больше динамических настроек, поэтому мы поместили их в отдельную утилиту командной строки, `kafka-configs.sh`. Это позволяет задавать настройки для отдельных тем и ID клиентов. Кластер использует заданные настройки на постоянной основе. Они хранятся в ZooKeeper, и каждый брокер читает их при запуске. В утилитах и документации о подобных динамических настройках говорят как о *настройках для отдельных тем* (per-topic) или *настройках для отдельных клиентов* (per-client), а также используют термин «*переопределение настроек*» (override).

Как и для предыдущих утилит, необходимо указать строку подключения ZooKeeper для кластера с помощью аргумента `--zookeeper`. В последующих примерах предполагается, что строка подключения ZooKeeper имеет вид `zoo1.example.com:2181/kafka-cluster`.

Переопределение значений настроек тем по умолчанию

Сочетать различные сценарии использования в одном кластере можно, меняя множество настроек отдельных тем. У большинства этих настроек есть задаваемые в конфигурации брокера значения по умолчанию, которые и будут применяться, если их не переопределить.

Формат команды изменения настроек темы:

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

В табл. 9.2 приведены возможные настройки (ключи конфигурации) тем.

Таблица 9.2. Допустимые ключи [конфигурации] тем

Ключ конфигурации	Описание
cleanup.policy	При значении compact сообщения темы будут отбрасываться, а из сообщений с заданным ключом будет сохраняться только самое последнее (сжатые журналы)
compression.type	Тип сжатия, используемый брокером при записи на диск пакетов сообщений для данной темы. Допустимые значения gzip, snappy и lz4
delete.retention.ms	Длительность (в миллисекундах) хранения отметок об удалении для данной темы. Имеет смысл только для тем со сжатием журналов

Ключ конфигурации	Описание
file.delete.delay.ms	Длительность (в миллисекундах) ожидания перед удалением сегментов журнала и индексов для данной темы с диска
flush.messages	Количество сообщений, которое может быть получено, прежде чем будет выполнен принудительный сброс сообщений данной темы на диск
flush.ms	Промежуток времени (в миллисекундах) перед принудительным сбросом сообщений данной темы на диск
index.interval.bytes	Допустимое количество байтов сообщений, генерируемых между записями индекса сегмента журнала
max.message.bytes	Максимальный размер отдельного сообщения данной темы (в байтах)
message.format.version	Используемая брокером при записи сообщений на диск версия формата сообщений. Должна представлять собой допустимую версию API (например, 0.10.0)
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 данной темы утилитой сжатия журналов (в виде отношения числа несжатых сегментов журнала к общему числу сегментов). Имеет смысл только для тем со сжатием журналов
min.insync.replicas	Минимальное количество согласованных реплик, необходимое для того, чтобы раздел темы считался доступным
preallocate	При установке в true место под сегменты журналов для этой темы будет выделяться заранее, при создании нового сегмента
retention.bytes	Объем хранимых сообщений этой темы (в байтах)
retention.ms	Длительность (в миллисекундах) хранения сообщений данной темы
segment.bytes	Объем сообщений (в байтах), записываемый в отдельный сегмент журнала в разделе
segment.index.bytes	Максимальный размер (в байтах) отдельного индекса сегмента журнала
segment.jitter.ms	Максимальное число миллисекунд, задаваемых случайным образом и добавляемых к ключу segment.ms при создании сегментов журналов
segment.ms	Частота (в миллисекундах) чередования сегментов журналов для каждого из разделов
unclean.leader.election.enable	При установке в false для данной темы будет запрещен «нечистый» выбор ведущей реплики

Например, установим для темы my-topic длительность сохранения, равную 1 ч (3 600 000 мс):

```
# kafka-configs.sh --zookeeper zoo1.example.com:2181/kafka-cluster
--alter --entity-type topics --entity-name my-topic --add-config
retention.ms=3600000
Updated config for topic: "my-topic".
#
```

Переопределение настроек клиентов по умолчанию

Единственные настройки, которые можно переопределить для клиентов Kafka, — квоты производителей и потребителей. И то и другое представляет собой объем данных (в байтах в секунду), который разрешается генерировать/потреблять всем клиентам с заданным идентификатором клиента в расчете на одного брокера. То есть если вы зададите квоту в 10 Мбайт/с на клиент для кластера в пять брокеров, то этот клиент сможет генерировать 10 Мбайт/с данных для каждого из брокеров, что в итоге составит 50 Мбайт/с.



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 уникальный идентификатор клиента для каждой группы потребителей, причем каким-то образом идентифицирующий ее. Благодаря этому группа потребителей сможет использовать квоту совместно, а поиск по журналам, какая группа отвечает за запрос, значительно упростится.

Формат команды изменения настроек клиента:

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

В табл. 9.3 приведены настройки (ключи конфигурации) клиентов.

Таблица 9.3. Настройки (ключи конфигурации) клиентов

Ключ конфигурации	Описание
producer_bytes_rate	Допустимый объем генерируемых для одного брокера в секунду по отдельному идентификатору клиента сообщений (в байтах)
consumer_bytes_rate	Допустимый объем потребляемых из одного брокера в секунду по отдельному идентификатору клиента сообщений (в байтах)

Описание переопределений настроек

С помощью утилиты командной строки можно вывести список всех переопределений настроек и просмотреть конкретные настройки темы или клиента. Как и в остальных утилитах, здесь для этого используется команда `--describe`.

Например, выведем список всех переопределений настроек для темы `my-topic`:

```
# kafka-configs.sh --zookeeper zoo1.example.com:2181/kafka-cluster
--describe --entity-type topics --entity-name my-topic
Configs for topics:my-topic are
retention.ms=3600000,segment.ms=3600000
#
```



Только переопределения настроек темы

Описание конфигурации выводит только переопределения и не включает настройки кластера по умолчанию. Сейчас не существует возможности ни через ZooKeeper, ни по протоколу Kafka динамически выяснить настройки самих брокеров. Это значит, что в случае использования в ходе автоматизации вышеупомянутой утилиты для выяснения настроек темы или клиента необходимо отдельно предоставить ей 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 темы `my-topic`:

```
# kafka-configs.sh --zookeeper zoo1.example.com:2181/kafka-cluster
--alter --entity-type topics --entity-name my-topic
--delete-config retention.ms
Updated config for topic: "my-topic".
#
```

Управление разделами

Среди утилит Kafka есть два сценария для управления разделами: один предназначен для повторного выбора ведущих реплик, а второй представляет собой низкоуровневую утилиту распределения разделов по брокерам. С их помощью можно должным образом сбалансировать трафик сообщений в кластере брокеров Kafka.

Выбор предпочтительной ведущей реплики

Как обсуждалось в главе 6, у разделов может быть по несколько реплик, для большей надежности. Однако только одна из них может быть ведущей репликой раздела, и вся генерация и потребление выполняются на соответствующем брокере. Эта реплика определяется внутри Kafka как первая согласованная в списке реплик, но в случае останова и перезапуска брокера не становится автоматически ведущей для каких-либо разделов.



Автоматическое перераспределение ведущих реплик

Существует настройка брокера для автоматического перераспределения ведущих реплик, но использовать ее при промышленной эксплуатации не рекомендуется. Модуль автоматической балансировки серьезно влияет на производительность и может привести к длительной приостановке трафика клиента, если кластеры большие.

Один из способов вернуть брокерам статус ведущих — инициировать выбор предпочтительной ведущей реплики. При этом контроллер кластера выбирает оптимальные ведущие реплики для разделов. Обычно эта операция не влияет на производительность, так как у клиентов есть возможность следить за сменой ведущей реплики автоматически. Инициировать эту операцию вручную можно с помощью утилиты `kafka-preferred-replica-election.sh`.

Например, запустим выбор предпочтительной ведущей реплики для всех тем кластера с одной темой из восьми разделов:

```
# kafka-preferred-replica-election.sh --zookeeper
zoo1.example.com:2181/kafka-cluster
Successfully started preferred replica election for partitions
Set([my-topic,5], [my-topic,0], [my-topic,7], [my-topic,4],
[my-topic,6], [my-topic,2], [my-topic,3], [my-topic,1])
#
```

Если число разделов у кластеров велико, запуск отдельной процедуры выбора предпочтительной ведущей реплики может завершиться неудачей. Запрос должен быть записан в z-узел (znode) ZooKeeper в метаданных кластера, и если его размер превышает размер z-узла (по умолчанию 1 Мбайт), то произойдет ошибка. В этом случае необходимо создать файл с JSON-объектом, содержащим список разделов, и разбить запрос на несколько шагов. Формат для этого JSON-файла следующий:

```
{
  "partitions": [
    {
      "partition": 1,
      "topic": "foo"
    },
  ],
}
```

```

    {
      "partition": 2,
      "topic": "foobar"
    }
  ]
}

```

Например, инициируем выбор предпочтительной ведущей реплики с заданным списком разделов, находящимся в файле `partitions.json`:

```

# kafka-preferred-replica-election.sh --zookeeper
zoo1.example.com:2181/kafka-cluster --path-to-json-file
partitions.json
Successfully started preferred replica election for partitions
Set([my-topic,1], [my-topic,2], [my-topic,3])
#

```

Смена реплик раздела

Время от времени возникает необходимость изменить распределение реплик для раздела. Вот несколько примеров таких случаев.

- ❑ Разделы темы не сбалансированы в масштабах кластера, что приводит к неравномерной нагрузке брокеров.
- ❑ Один из брокеров отключился, и раздел недореплицирован.
- ❑ Был добавлен новый брокер, и ему необходимо выделить часть нагрузки кластера.

Для выполнения этой операции можно воспользоваться утилитой `kafka-reassign-partitions.sh`. Ее необходимо использовать как минимум в два этапа. На первом этапе на основе списка брокеров и тем генерируется набор перемещений, выполняемых на втором этапе. Существует также необязательный третий этап, на котором сгенерированный список используется для проверки степени завершенности перераспределений разделов.

Для генерации набора перемещений разделов необходимо создать файл с JSON-объектом, содержащим список тем. Формат JSON-объекта следующий (номер версии в настоящий момент всегда равен 1):

```

{
  "topics": [
    {
      "topic": "foo"
    },
    {
      "topic": "foo1"
    }
  ],
  "version": 1
}

```

Например, сгенерируем перемещения разделов, чтобы перенести перечисленные в файле `topics.json` разделы на брокеры с идентификаторами 0 и 1:

```
# kafka-reassign-partitions.sh --zookeeper
zoo1.example.com:2181/kafka-cluster --generate
--topics-to-move-json-file topics.json --broker-list 0,1
Current partition replica assignment
```

```
{ "version": 1, "partitions": [ { "topic": "my-topic", "partition": 5, "replicas": [ 0, 1 ] },
{ "topic": "my-topic", "partition": 10, "replicas": [ 1, 0 ] }, { "topic": "my-topic",
"partition": 1, "replicas": [ 0, 1 ] }, { "topic": "my-topic", "partition": 4, "replicas": [ 1, 0 ] },
{ "topic": "my-topic", "partition": 7, "replicas": [ 0, 1 ] }, { "topic": "my-topic",
"partition": 6, "replicas": [ 1, 0 ] }, { "topic": "my-topic", "partition": 3, "replicas": [ 0, 1 ] },
{ "topic": "my-topic", "partition": 15, "replicas": [ 0, 1 ] }, { "topic": "my-topic",
"partition": 0, "replicas": [ 1, 0 ] }, { "topic": "my-topic", "partition": 11, "replicas":
[ 0, 1 ] }, { "topic": "my-topic", "partition": 8, "replicas": [ 1, 0 ] }, { "topic": "my-topic",
"partition": 12, "replicas": [ 1, 0 ] }, { "topic": "my-topic", "partition": 2, "replicas":
[ 1, 0 ] }, { "topic": "my-topic", "partition": 13, "replicas": [ 0, 1 ] }, { "topic": "my-topic",
"partition": 14, "replicas": [ 1, 0 ] }, { "topic": "my-topic", "partition": 9, "replicas":
[ 0, 1 ] } ] }
```

Proposed partition reassignment configuration

```
{ "version": 1, "partitions": [ { "topic": "my-topic", "partition": 5, "replicas": [ 0, 1 ] },
{ "topic": "my-topic", "partition": 10, "replicas": [ 1, 0 ] }, { "topic": "my-topic",
"partition": 1, "replicas": [ 0, 1 ] }, { "topic": "my-topic", "partition": 4, "replicas": [ 1, 0 ] },
{ "topic": "my-topic", "partition": 7, "replicas": [ 0, 1 ] }, { "topic": "my-topic",
"partition": 6, "replicas": [ 1, 0 ] }, { "topic": "my-topic", "partition": 15, "replicas": [ 0, 1 ] },
{ "topic": "my-topic", "partition": 0, "replicas": [ 1, 0 ] }, { "topic": "my-topic",
"partition": 3, "replicas": [ 0, 1 ] }, { "topic": "my-topic", "partition": 11, "replicas": [ 0, 1 ] },
{ "topic": "my-topic", "partition": 8, "replicas": [ 1, 0 ] }, { "topic": "my-topic",
"partition": 12, "replicas": [ 1, 0 ] }, { "topic": "my-topic", "partition": 13, "replicas":
[ 0, 1 ] }, { "topic": "my-topic", "partition": 2, "replicas": [ 1, 0 ] }, { "topic": "my-topic",
"partition": 14, "replicas": [ 1, 0 ] }, { "topic": "my-topic", "partition": 9,
"replicas": [ 0, 1 ] } ] }
```

#

Список брокеров задается через командную строку утилиты в виде разделенного запятыми списка идентификаторов брокеров. В результате утилита выведет в стандартный поток вывода два JSON-объекта с описанием текущего распределения разделов для тем и предлагаемого распределения. Формат этих JSON-объектов: `{ "partitions": [{ "topic": "my-topic", "partition": 0, "replicas": [1, 2] }], "version": 1 }`.

Первый JSON-объект имеет смысл сохранить на случай необходимости возврата к прежнему распределению. Второй JSON-объект — с предлагаемым распределением — необходимо сохранить в новый файл. Этот файл затем передается обратно утилите `kafka-reassign-partitions.sh` для второго этапа.

Например, выполните предлагаемое перераспределение разделов из файла `reassign.json`:

```
# kafka-reassign-partitions.sh --zookeeper
zoo1.example.com:2181/kafka-cluster --execute
--reassignment-json-file reassign.json
```


Current partition replica assignment

```
{ "version": 1, "partitions": [ { "topic": "my-topic", "partition": 5, "replicas": [ 0, 1 ] },
{ "topic": "my-topic", "partition": 10, "replicas": [ 1, 0 ] }, { "topic": "my-topic",
"partition": 1, "replicas": [ 0, 1 ] }, { "topic": "my-topic", "partition": 4, "replicas":
[ 1, 0 ] }, { "topic": "my-topic", "partition": 7, "replicas": [ 0, 1 ] }, { "topic":
"my-topic", "partition": 6, "replicas": [ 1, 0 ] }, { "topic": "my-topic", "partition": 3,
"replicas": [ 0, 1 ] }, { "topic": "my-topic", "partition": 15, "replicas": [ 0, 1 ] }, { "topic":
"my-topic", "partition": 0, "replicas": [ 1, 0 ] }, { "topic": "my-topic", "partition": 11,
"replicas": [ 0, 1 ] }, { "topic": "my-topic", "partition": 8, "replicas": [ 1, 0 ] }, { "topic":
"my-topic", "partition": 12, "replicas": [ 1, 0 ] }, { "topic": "my-topic", "partition": 2,
"replicas": [ 1, 0 ] }, { "topic": "my-topic", "partition": 13, "replicas": [ 0, 1 ] },
{ "topic": "my-topic", "partition": 14, "replicas": [ 1, 0 ] }, { "topic": "my-topic",
"partition": 9, "replicas": [ 0, 1 ] } ] }
```

Save this to use as the --reassignment-json-file option during rollback
 Successfully started reassignment of partitions { "version": 1, "partitions":
 [{ "topic": "my-topic", "partition": 5, "replicas": [0, 1] }, { "topic": "my-topic",
 "partition": 0, "replicas": [1, 0] }, { "topic": "my-topic", "partition": 7, "replicas":
 [0, 1] }, { "topic": "my-topic", "partition": 13, "replicas": [0, 1] }, { "topic":
 "my-topic", "partition": 4, "replicas": [1, 0] }, { "topic": "my-topic", "partition": 12,
 "replicas": [1, 0] }, { "topic": "my-topic", "partition": 6, "replicas": [1, 0] }, { "topic":
 "my-topic", "partition": 11, "replicas": [0, 1] }, { "topic": "my-topic", "partition": 10,
 "replicas": [1, 0] }, { "topic": "my-topic", "partition": 9, "replicas": [0, 1] }, { "topic":
 "my-topic", "partition": 2, "replicas": [1, 0] }, { "topic": "my-topic", "partition": 14,
 "replicas": [1, 0] }, { "topic": "my-topic", "partition": 3, "replicas": [0, 1] }, { "topic":
 "my-topic", "partition": 1, "replicas": [0, 1] }, { "topic": "my-topic", "partition": 15,
 "replicas": [0, 1] }, { "topic": "my-topic", "partition": 8, "replicas": [1, 0] }] }

#

Эта команда запустит перераспределение заданных реплик разделов по новым брокерам. Контроллер кластера выполняет это действие посредством добавления новых реплик в список реплик каждого из разделов, увеличивая 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.



Повышение эффективности использования сети при переназначении реплик

При удалении множества разделов с отдельного брокера, например, при удалении этого брокера из кластера, рекомендуется останавливать и перезапускать брокер перед началом переназначения. Это приведет к тому, что ведущими репликами для находящихся на этом брокере разделов станут другие брокеры кластера (если не включена возможность автоматического выбора ведущей реплики). Это значительно повышает производительность выполнения переназначений и снижает влияние на кластер, поскольку связанный с репликацией трафик будет распределяться по нескольким брокерам.

Во время выполнения переназначения и после его завершения можно использовать утилиту `kafka-reassign-partitions.sh` для проверки состояния переназначения. Это позволит выяснить, сколько переназначений выполняется в настоящий момент, какие из них были завершены и какие завершились неудачей в случае возникновения ошибок. Для этого у вас должен быть файл с JSON-объектом, заданным на этапе выполнения.

Например, проверим состояние выполняющегося переназначения разделов из файла `reassign.json`:

```
# kafka-reassign-partitions.sh --zookeeper
zoo1.example.com:2181/kafka-cluster --verify
--reassignment-json-file reassign.json
Status of partition reassignment:
Reassignment of partition [my-topic,5] completed successfully
Reassignment of partition [my-topic,0] completed successfully
Reassignment of partition [my-topic,7] completed successfully
Reassignment of partition [my-topic,13] completed successfully
Reassignment of partition [my-topic,4] completed successfully
Reassignment of partition [my-topic,12] completed successfully
Reassignment of partition [my-topic,6] completed successfully
Reassignment of partition [my-topic,11] completed successfully
Reassignment of partition [my-topic,10] completed successfully
Reassignment of partition [my-topic,9] completed successfully
Reassignment of partition [my-topic,2] completed successfully
Reassignment of partition [my-topic,14] completed successfully
Reassignment of partition [my-topic,3] completed successfully
Reassignment of partition [my-topic,1] completed successfully
Reassignment of partition [my-topic,15] completed successfully
Reassignment of partition [my-topic,8] completed successfully
#
```



Пакетные переназначения

Переназначения разделов серьезно влияют на производительность кластера, поскольку вызывают изменения согласованности страничного кэша памяти и используют сеть и дисковые операции ввода/вывода. Будет отличной идеей минимизировать эти эффекты за счет разбиения переназначений на множество мелких шагов.

Изменение коэффициента репликации

Утилита переназначения разделов предоставляет недокументированную возможность, позволяющую увеличивать/уменьшать коэффициент репликации раздела. Это может пригодиться при создании раздела с неправильным коэффициентом репликации (например, если на момент создания темы не было доступно достаточного числа брокеров). Сделать это можно посредством создания JSON-объекта

в том же формате, который применялся на этапе переназначения разделов для добавления/удаления реплик с целью задания нужного коэффициента репликации. Кластер завершит операцию переназначения, сохранив новое значение коэффициента.

Например, рассмотрим текущее распределение реплик для темы `my-topic` с одним разделом и коэффициентом репликации, равным 1:

```
{
  "partitions": [
    {
      "topic": "my-topic",
      "partition": 0,
      "replicas": [
        1
      ]
    }
  ],
  "version": 1
}
```

Если передать следующий JSON-объект на этапе выполнения переназначений разделов, то коэффициент репликации будет увеличен до 2:

```
{
  "partitions": [
    {
      "partition": 0,
      "replicas": [
        1,
        2
      ],
      "topic": "my-topic"
    }
  ],
  "version": 1
}
```

Аналогично уменьшить коэффициент репликации раздела можно, передав JSON-объект с сокращенным списком реплик.

Сброс на диск сегментов журнала

В случае, когда необходимо найти сообщение с конкретным содержимым, например, «отравленную таблетку» — сообщение, которое потребитель не может обработать, можно воспользоваться вспомогательной утилитой 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.

Например, декодируем файл `0000000000052368601.log` сегмента журнала и выведем сводную информацию по сообщениям:

```
# kafka-run-class.sh kafka.tools.DumpLogSegments --files
0000000000052368601.log
Dumping 0000000000052368601.log
Starting offset: 52368601
offset: 52368601 position: 0 NoTimestampType: -1 invalid: true
payloadsize: 661 magic: 0 compresscodec: GZIPCompressionCodec crc:
1194341321
offset: 52368603 position: 687 NoTimestampType: -1 invalid: true
payloadsize: 895 magic: 0 compresscodec: GZIPCompressionCodec crc:
278946641
offset: 52368604 position: 1608 NoTimestampType: -1 invalid: true
payloadsize: 665 magic: 0 compresscodec: GZIPCompressionCodec crc:
3767466431
offset: 52368606 position: 2299 NoTimestampType: -1 invalid: true
payloadsize: 932 magic: 0 compresscodec: GZIPCompressionCodec crc:
2444301359
...
```

Или декодируем файл `0000000000052368601.log` сегмента журнала с выводом раз-
вернутой информации по сообщениям:

```
# kafka-run-class.sh kafka.tools.DumpLogSegments --files
0000000000052368601.log --print-data-log
offset: 52368601 position: 0 NoTimestampType: -1 invalid: true
payloadsize: 661 magic: 0 compresscodec: GZIPCompressionCodec crc:
1194341321 payload: test message 1
offset: 52368603 position: 687 NoTimestampType: -1 invalid: true
payloadsize: 895 magic: 0 compresscodec: GZIPCompressionCodec crc:
278946641 payload: test message 2
offset: 52368604 position: 1608 NoTimestampType: -1 invalid: true
payloadsize: 665 magic: 0 compresscodec: GZIPCompressionCodec crc:
3767466431 payload: test message 3
offset: 52368606 position: 2299 NoTimestampType: -1 invalid: true
payloadsize: 932 magic: 0 compresscodec: GZIPCompressionCodec crc:
2444301359 payload: test message 4
...
```

Можно также использовать эту утилиту для проверки 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, используемых 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.

Например, проверим файл индекса для файла `0000000000052368601.log` сегмента журнала на отсутствие повреждений:

```
# kafka-run-class.sh kafka.tools.DumpLogSegments --files
0000000000052368601.index,0000000000052368601.log
--index-sanity-check
Dumping 0000000000052368601.index
0000000000052368601.index passed sanity check.
Dumping 0000000000052368601.log
Starting offset: 52368601
offset: 52368601 position: 0 NoTimestampType: -1 invalid: true
payloadsize: 661 magic: 0 compresscodec: GZIPCompressionCodec crc:
1194341321
offset: 52368603 position: 687 NoTimestampType: -1 invalid: true
payloadsize: 895 magic: 0 compresscodec: GZIPCompressionCodec crc:
278946641
offset: 52368604 position: 1608 NoTimestampType: -1 invalid: true
payloadsize: 665 magic: 0 compresscodec: GZIPCompressionCodec crc:
3767466431
...
```

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 с брокера, в этом случае ведомый брокер не будет заполнять промежутки.

Чтобы проверить согласованность всех реплик разделов темы в кластере, можно воспользоваться утилитой `kafka-replica-verification.sh`. Она извлекает сообщения из всех реплик заданного набора разделов темы и проверяет наличие всех сообщений во всех репликах. Утилите необходимо передать регулярное выражение, соответствующее всем темам, которые нужно проверить. Если его не указать, то будут проверяться все темы. Необходимо также указать явным образом список брокеров для подключения.



Осторожно: влияет на производительность кластера

Утилита проверки реплик влияет на производительность кластера так же, как и перераспределение 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 тем, название которых начинается с «my-», на брокерах 1 и 2:

```
# kafka-replica-verification.sh --broker-list
kafka1.example.com:9092,kafka2.example.com:9092 --topic-white-list 'my-.*'
2016-11-23 18:42:08,838: verification process is started.
2016-11-23 18:42:38,789: max lag is 0 for partition [my-topic,7]
at offset 53827844 among 10 partitions
2016-11-23 18:43:08,790: max lag is 0 for partition [my-topic,7]
at offset 53827878 among 10 partitions
```

Потребление и генерация

В ходе работы с Apache Kafka вам часто придется вручную потреблять сообщения или генерировать образцы сообщений для проверки работы ваших приложений. Для этого существуют две вспомогательные утилиты: `kafka-console-consumer.sh` и `kafka-console-producer.sh`. Они представляют собой адаптеры для клиентских библиотек Java, которые обеспечивают возможность взаимодействия с темами Kafka, так что писать для этой цели отдельное приложение не нужно.



Конвейерная передача вывода в отдельное приложение

Хотя существует возможность написания приложений-адаптеров для консольного производителя и потребителя (например, для чтения сообщений и конвейерной передачи их другому приложению на обработку), эти приложения часто ненадежны, так что лучше их избегать. Очень непросто наладить взаимодействие с консольным потребителем так, чтобы при этом сообщения не терялись. Аналогично, доступны не все возможности консольного производителя, так что отправлять данные должным образом непросто. Лучше воспользоваться или непосредственно клиентскими библиотеками Java, или сторонними клиентскими библиотеками для других языков программирования, которые применяют протокол Kafka напрямую.

Консольный потребитель

Утилита `kafka-console-consumer.sh` позволяет потреблять сообщения из одной или нескольких тем кластера Kafka. Сообщения выводятся в стандартный поток вывода и разделяются символом новой строки. По умолчанию выводятся неформатированные сообщения (с помощью `DefaultFormatter`). Обязательные параметры описаны далее.



Проверьте версию утилиты

Очень важно использовать потребитель той же версии, что и кластер Kafka. Консольные потребители более старых версий могут повредить кластер из-за того, что взаимодействуют с ZooKeeper недопустимым образом.

Первый из обязательных параметров указывает, применять ли потребитель новой версии, включающий ссылку на сам кластер Kafka. При использовании потребителя старой версии для этого требуется только один аргумент — параметр `--zookeeper` с последующей строкой подключения для кластера. В приведенных ранее примерах он может выглядеть вот так: `--zookeeper zoo1.example.com:2181/kafka-cluster`. В случае нового потребителя необходимо задать как флаг `--new-consumer`, так и параметр `--bootstrap-server` с разделенным запятыми списком брокеров после него, например: `--bootstrap-server kafka1.example.com:9092,kafka2.example.com:9092`.

Далее необходимо указать потребляемые темы. Для этого существует три параметра: `--topic`, `--whitelist` и `--blacklist`, из которых можно указать только один. Параметр `--topic` задает для потребления одну конкретную тему. После каждого из параметров `--whitelist` и `--blacklist` должно следовать регулярное выражение (не забудьте экранировать его должным образом, чтобы командная строка оболочки не изменила его). При использовании параметра `--whitelist` будут потребляться все темы, соответствующие регулярному выражению, а в случае `--blacklist` — все темы, *кроме* соответствующих регулярному выражению.

Например, для потребления отдельной темы `my-topic` с помощью потребителя старой версии нужно сделать следующее:

```
# kafka-console-consumer.sh --zookeeper
zoo1.example.com:2181/kafka-cluster --topic my-topic
sample message 1
sample message 2
^CProcessed a total of 2 messages
#
```

Помимо простейших команд командной строки можно передавать консольному потребителю и все обычные параметры конфигурации. Сделать это можно двумя способами в зависимости от количества передаваемых параметров. Первый способ: через файл конфигурации потребителя, задав ключ `--consumer.config CONFIGFILE`, где `CONFIGFILE` — полный путь к файлу с параметрами конфигурации. Другой способ: указать параметры в командной строке с одним или большим количеством аргументов в виде `--consumer-property KEY=VALUE`, где `KEY` — название параметра, а `VALUE` — его задаваемое значение. Такой способ удобен при задании таких параметров конфигурации, как идентификатор группы потребителей.



Параметры командной строки, которые легко перепутать

Как для консольного потребителя, так и для консольного производителя существует параметр командной строки `--property`, который не следует путать с параметрами `--consumer-property` и `--producer-property`. Параметр `--property` используется для передачи конфигурации только подпрограмме форматирования сообщений, а не самому клиенту.

Существует несколько часто используемых параметров консольного потребителя, которые не мешает знать:

- ❑ `--formatter CLASSNAME` — задает используемый при декодировании сообщений класс подпрограммы форматирования сообщений. По умолчанию `kafka.tools.DefaultMessageFormatter`;
- ❑ `--from-beginning` — потреблять сообщения из заданной (-ых) темы (тем), начиная с первого смещения. В противном случае потребление начинается с последнего смещения;
- ❑ `--max-messages NUM` — читать не более `NUM` сообщений, прежде чем завершить работу;
- ❑ `--partition NUM` — читать только из раздела с идентификатором `NUM` (для этого требуется новый потребитель).

Параметры подпрограммы форматирования сообщений

Помимо используемой по умолчанию существуют три подпрограммы форматирования сообщений:

- ❑ `kafka.tools.LoggingMessageFormatter` — выводит сообщения посредством механизма журналирования вместо стандартного потока вывода. Сообщения выводятся на уровне INFO и включают метку даты/времени, ключ и значение;
- ❑ `kafka.tools.ChecksumMessageFormatter` — выводит только контрольные суммы сообщений;
- ❑ `kafka.tools.NoOpMessageFormatter` — потребляет сообщения, но не выводит их вообще.

У `kafka.tools.DefaultMessageFormatter` есть также несколько удобных параметров, которые можно задавать с помощью параметра командной строки `--property`:

- ❑ `print.timestamp` — установите в `true` для отображения метки даты/времени каждого из сообщений (при наличии);
- ❑ `print.key` — установите в `true` для отображения наряду со значением ключа сообщения;
- ❑ `key.separator` — задает символ-разделитель, выводимый между ключом и значением сообщения;
- ❑ `line.separator` — задает символ-разделитель, выводимый между сообщениями;
- ❑ `key.deserializer` — позволяет задать имя класса, используемого для десериализации ключа сообщения перед выводом;
- ❑ `value.deserializer` — позволяет задать имя класса, используемого для десериализации значения сообщения перед выводом.

Классы десериализации должны реализовывать интерфейс `org.apache.kafka.common.serialization.Deserializer`. Для отображения вывода консольный потребитель вызывает их метод `toString`. Обычно эти десериализаторы реализуют в виде Java-класса, размещаемого на пути классов для консольного потребителя посредством задания значения переменной среды `CLASSPATH`, перед выполнением сценария `kafka_console_consumer.sh`.

Чтение тем смещений

Иногда бывает удобно посмотреть, какие смещения для групп потребителей кластера были зафиксированы. Возможно, вам захочется узнать, фиксировала ли смещения конкретная группа потребителей или насколько часто это происходило. Сделать это можно посредством чтения специальной внутренней темы `__consumer_offsets` через консольный потребитель. Смещения для всех потребителей записываются в эту тему в виде сообщений. Для декодирования этих сообщений нужно воспользоваться классом `kafka.coordinator.GroupMetadataManager$OffsetsMessageFormatter`¹.

Например, прочитаем отдельное сообщение из темы смещений:

```
# kafka-console-consumer.sh --zookeeper
zoo1.example.com:2181/kafka-cluster --topic __consumer_offsets
--formatter 'kafka.coordinator.GroupMetadataManager$OffsetsMessage
Formatter' --max-messages 1
[my-group-name,my-topic,0]::[OffsetMetadata[481690879,NO_METADATA]
,CommitTime 1479708539051,ExpirationTime 1480313339051]
Processed a total of 1 messages
#
```

Консольный производитель

Аналогично консольному потребителю утилита `kafka-console-producer.sh` может использоваться для записи сообщений в тему Kafka вашего кластера. По умолчанию сообщения читаются по одному в строке, в качестве разделителя ключа и значения применяется символ табуляции (если он отсутствует, считается, что ключ пустой).



Изменение способа чтения строк

Можно написать собственный класс построчного чтения для нестандартных операций. Этот класс, отвечающий за создание объекта `ProducerRecord`, должен расширять интерфейс `kafka.common.MessageReader`. Указать свой класс можно через командную строку с помощью параметра `--line-reader`. Убедитесь, что содержащий ваш класс JAR-файл включен в путь классов.

¹ Начиная с версии 0.11, этот класс называется `kafka.coordinator.group.GroupMetadataManager$OffsetsMessageFormatter`. — *Примеч. пер.*

Консольный производитель требует задания как минимум двух обязательных аргументов. Один из них — параметр `--broker-list`, в котором указывается один или несколько брокеров в виде разделенного запятыми списка элементов вида `hostname:port`. Второй обязательный параметр — `--topic`, определяющий тему, для которой генерируются сообщения. По окончании генерации отправьте символ конца файла (EOF) для закрытия клиента.

Например, вот так можно генерировать два сообщения для темы `my-topic`:

```
# kafka-console-producer.sh --broker-list
kafka1.example.com:9092,kafka2.example.com:9092 --topic my-topic
sample message 1
sample message 2
^D
#
```

Как и в случае консольного потребителя, консольному производителю можно передавать любые обычные параметры настройки. Сделать это можно двумя способами в зависимости от количества передаваемых параметров. Первый способ: через файл конфигурации производителя путем задания ключа `--producer.config CONFIGFILE`, где `CONFIGFILE` — полный путь к файлу с параметрами конфигурации. Другой способ: указать параметры в командной строке с одним или несколькими аргументами в виде `--producer-property KEY=VALUE`, где `KEY` — название параметра, а `VALUE` — его задаваемое значение. Это способ удобен при задании таких параметров производителей, как настройки пакетной передачи сообщений (например, `linger.ms` или `batch.size`).

У консольного производителя имеется множество аргументов командной строки, предназначенных для тонкой настройки его поведения. Среди наиболее удобных:

- ❑ `--key-serializer CLASSNAME` — задает используемый при сериализации ключей сообщений класс подпрограммы кодирования. По умолчанию `kafka.serializer.DefaultEncoder`;
- ❑ `--value-serializer CLASSNAME` — задает используемый при сериализации значений сообщений класс подпрограммы кодирования. По умолчанию `kafka.serializer.DefaultEncoder`;
- ❑ `--compression-codec STRING` — задает используемый при генерации сообщений тип сжатия. Возможные значения — `none`, `gzip`, `snappy` и `lz4`. Значение по умолчанию `gzip`;
- ❑ `--sync` — генерирует сообщения синхронно, ожидая подтверждения получения каждого из них перед отправкой следующего.



Создание пользовательского сериализатора

Пользовательские сериализаторы должны расширять класс `kafka.serializer.Encoder`. Их можно использовать для преобразования полученных из стандартного потока ввода строк в подходящую для темы кодировку, например Avro или Protobuf.

Параметры построчного чтения

У класса `kafka.tools.LineMessageReader`, отвечающего за чтение из стандартного потока ввода и создание записей для производителей, также есть несколько удобных параметров, которые можно передать консольному производителю с помощью параметра командной строки `--property`:

- ❑ `ignore.error` — установите в `false`, чтобы при отсутствии разделителя ключей при `parse.key=true` генерировалось исключение. По умолчанию `true`;
- ❑ `parse.key` — установите в `false`, чтобы ключ всегда оставался пустым. По умолчанию `true`;
- ❑ `key.separator` — определяет, какой разделитель между ключом и значением сообщения будет использоваться при чтении.

При генерации сообщений `LineMessageReader` будет разбивать поступившие входные данные в точке первого вхождения `key.separator`. Если далее не остается никаких символов, значение сообщения будет пустым. Если в строке отсутствует символ-разделитель для ключа или `parse.key=false`, то ключ будет пустым.

Списки управления доступом клиентов

Для взаимодействия с механизмом управления доступом (ACL) клиентов Kafka существует утилита командной строки, `kafka-acls.sh`. Дополнительную информацию об ACL и безопасности можно найти на сайте Apache Kafka (<https://kafka.apache.org/>).

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 решения конкретной ошибки. Эти операции обычно не документированы, не поддерживаются, и их выполнение несет некоторые риски для приложения.

Несколько наиболее распространенных из них описаны далее, чтобы в непредвиденной ситуации был лишний вариант восстановления. При нормальном функционировании кластера использовать их не рекомендуется, так что, прежде чем их запустить, стоит дважды подумать.



Опасность: Неизвестная территория

Операции из этого раздела 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 контроллер, представляющий собой запущенный на одном из брокеров поток выполнения. Контроллер отвечает за управление операциями кластера, и время от времени рекомендуется переносить его на другой брокер. 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 не представляет особого риска, но это нестандартная операция, и выполнять ее регулярно не стоит.

Брокер, который является текущим контроллером, регистрируется с помощью узла ZooKeeper `/controller` на верхнем уровне пути кластера. Если удалить этот узел вручную, то текущий контроллер сложит полномочия и кластер выберет новый.

Отмена перемещения раздела

Обычная последовательность выполняемых действий при перераспределении разделов.

1. Выполняется запрос перераспределения (создание узла ZooKeeper).
2. Контроллер добавляет кластер разделов в новые брокеры.
3. Новые брокеры приступают к репликации разделов и делают это вплоть до достижения согласованности.
4. Контроллер кластера удаляет старые брокеры из списка реплик разделов.

Поскольку запрошенные перераспределения запускаются параллельно, при нормальных обстоятельствах причин для отмены происходящего перераспределения нет. Одно из исключений — сбой брокера посередине процесса перераспределения и невозможность его (брокера) немедленного перезапуска. В результате перераспределение никогда не завершится, тем самым препятствуя запуску дополнительных перераспределений (например, для удаления разделов со сбойного брокера и распределения их по другим брокерам). В подобных случаях можно заставить кластер забыть об этом перераспределении.

Для удаления выполняемого в данный момент перераспределения сделайте следующее.

1. Удалите узел ZooKeeper `/admin/reassign_partitions` из пути кластера Kafka.
2. Принудительно иницилируйте перемещение контроллера (см. подробности в разделе «Перенос контроллера кластера» ранее в главе).



Проверка коэффициентов репликации

При отмене текущего перемещения раздела, действия над которым еще не были завершены, старые брокеры не будут удалены из списка реплик. Это значит, что коэффициент репликации части разделов может оказаться больше, чем планировалось. Брокер не позволит выполнять некоторые административные операции с темами, в которых есть разделы с несогласованными коэффициентами репликации. Рекомендуется просмотреть обрабатываемые разделы и убедиться, что их коэффициенты репликации верны.

Отмена удаления тем

При удалении темы с помощью утилит командной строки узел ZooKeeper запрашивает удаление. При обычных обстоятельствах кластер выполняет это немедленно. Однако утилита командной строки никак не может знать, включена ли возможность удаления тем в кластере. В результате удаление темы запрашивается в любом случае, так что если эта настройка отключена, вас будет ждать неприятный сюрприз. Избежать этого помогает возможность отмены ожидающих выполнения запросов на удаление тем.

Удаление темы запрашивается созданием в `/admin/delete_topic` дочернего узла ZooKeeper, название которого соответствует названию темы. Удаление этих узлов ZooKeeper (но не родительского узла `/admin/delete_topic`) приводит к удалению ожидающих выполнения запросов.

Удаление тем вручную

Если удаление тем в вашем кластере отключено или вам понадобилось убрать какие-либо темы 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 темы в ZooKeeper во время работы кластера.

Для удаления темы из кластера сделайте следующее.

1. Остановите все брокеры кластера.
2. Удалите каталог ZooKeeper `/brokers/topics/TOPICNAME` 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 *TOPICNAME-NUM*, где *NUM* — 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 тем 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.

Конечно, управлять кластером Kafka невозможно без должного мониторинга. В главе 10 мы обсудим способы мониторинга состояния 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 клиентов — как производителей, так и потребителей.

10 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 темам и разделам. Благодаря им у вас будет подробная информация обо всех производимых на брокере операциях, но они же могут стать настоящим проклятием ответственных за мониторинг системы.

В этом разделе мы подробно опишем важнейшие показатели, которые следует контролировать постоянно, и расскажем, как реагировать на их изменения. Мы также опишем 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 рекомендуемых к использованию при мониторинге и уведомлении пользователей практиках. 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, лучшим 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`.

Внешние и внутренние показатели

Предоставляемые по интерфейсу JMX показатели — внутренние, они формируются и выдаются самим контролируемым приложением. Для большинства внутренних показателей, например, хронометража отдельных этапов запроса, это оптимальный вариант. Лишь у самого приложения имеется настолько подробная информация. Существуют и другие показатели, например, общее время выполнения запроса или доступность конкретного типа запроса, которые можно оценить извне приложения. То есть сервер (в нашем случае брокер) будет получать показатели от клиента Kafka или другого стороннего приложения. В их числе зачастую оказываются такие, как доступность (доступен ли брокер?) и время задержки (сколько времени занимает выполнение запроса?). Они позволяют оценить приложение со стороны, что зачастую более информативно.

Один из традиционных примеров ценности внешних показателей — мониторинг состояния сайта. Веб-сервер работает нормально и все его показатели говорят, что сайт функционирует. Однако существует проблема с сетью между веб-сервером и внешними пользователями, из-за чего веб-сервер недоступен пользователям. Внешний мониторинг, выполняемый за пределами сети, с помощью которого проверяют доступность сайта, может обнаружить подобную проблему и уведомить вас о ситуации.

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.

Охват показателей

С учетом числа доступных в Kafka показателей важно тщательно выбирать, на какие из них обращать внимание. Особую важность это приобретает при настройке уведомлений на их основе. Слишком легко поддаться усталости от уведомлений, когда их всплывает так много, что невозможно понять, какая проблема серьезна, а какая нет. Непросто также задать должным образом пороговые значения для всех показателей и поддерживать их актуальность. Если уведомлений слишком много или они часто оказываются некорректными, перестаешь верить, что они правильно описывают состояние приложения.

Гораздо лучше иметь несколько высокоуровневых уведомлений. Например, вы можете получать одно уведомление о масштабной проблеме, с тем чтобы сразу начать собирать дополнительные данные для выяснения конкретики. Аналогом можно считать индикатор «Проверить двигатель» в автомобиле. Сотня индикаторов на панели, каждый из которых сигнализировал бы об отдельных проблемах с воздушным фильтром, маслом, выхлопной трубой и т. д., только запутала бы водителя. Вместо этого существует один индикатор, сообщающий о проблеме, и есть возможность получить более подробную информацию о том, в чем именно заключается проблема. На протяжении данной главы мы будем отмечать показатели с максимальным охватом, позволяющие упростить систему оповещения.

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 также 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 Б и наоборот. Какой бы способ вы ни выбрали, главное, чтобы мониторинг и оповещение о проблемах Kafka не зависели от ее функционирования.

Начнем с обсуждения показателей недореплицированных разделов как характеристики общей производительности, а также с того, как реагировать на их изменения. Остальные рассматриваемые показатели дополняют высокоуровневую картину брокера. 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.

Недореплицированные разделы

Если у вас есть возможность контролировать лишь один показатель брокера Kafka, выберите число недореплицированных разделов. Этот показатель, взятый для каждого брокера кластера, представляет собой количество разделов, для которых данный 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. 10.1 приведена более подробная информация по поводу недореплицированных разделов.

Таблица 10.1. Показатели и соответствующие им недореплицированные разделы

Показатель	Недореплицированные разделы
Управляемый компонент (MBean) JMX	kafka.server:type=ReplicaManager,name=UnderReplicatedPartitions
Диапазон значений	Целое число, равное нулю или больше него

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 показателей. В этом случае необходимо выяснить, что случилось с брокером, и решить проблему. Зачастую дело оказывается в отказе аппаратного обеспечения, но проблему может вызывать и проблема с операционной системой или Java.



Выбор предпочтительной реплики

Прежде чем пытаться выяснить причины проблемы, следует убедиться, что недавно была выбрана предпочтительная реплика (см. главу 9). Брокеры Kafka не становятся автоматически вновь ведущими для разделов (разве что включена настройка для автоматического перераспределения ведущих реплик, но это не рекомендуется) после потери статуса ведущей реплики, например, в случае сбоя или останова брокера. А значит, ведущие реплики запросто могут стать несбалансированными. Выбор предпочтительной реплики — безопасная и простая операция, так что стоит в первую очередь выполнить его и посмотреть, не исчезнет ли проблема.

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 на одном брокере, то обычно причина проблемы именно в нем и ошибка указывает на то, что у других брокеров возникают проблемы при репликации сообщений с 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 недоразумения 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 9.

Например, выведем список недореплицированных разделов кластера:

```
# kafka-topics.sh --zookeeper zoo1.example.com:2181/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 категорий:

- ☐ дисбаланс нагрузки;
- ☐ истощение ресурсов.

Источник первой из них — дисбаланса разделов или ведущих реплик — найти нетрудно, но решить ее может оказаться сложно. Для диагностики от брокеров кластера понадобятся данные по следующим показателям:

- ☐ количество разделов;
- ☐ количество ведущих разделов;
- ☐ суммарная по всем темам входящая скорость передачи данных [байт/с];
- ☐ суммарная по всем темам исходящая скорость передачи данных [байт/с];
- ☐ суммарная частота входящих сообщений по всем темам.

Изучите эти показатели. В идеально сбалансированном кластере они будут примерно одинаковыми для всех брокеров кластера, как в табл. 10.2.

Таблица 10.2. Показатели использования кластера

Брокер	Раздел	Ведущая реплика	Входящих байтов, Мбайт/с	Исходящих байтов, Мбайт/с
1	100	50	3,56	9,45
2	101	49	3,66	9,25
3	100	50	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 с более нагруженных брокеров на менее нагруженные. Сделать это можно с помощью утилиты `kafka-reassign-partitions.sh`, описанной в главе 9.



Вспомогательные утилиты для балансировки кластера

Сами брокеры Kafka не позволяют автоматически перераспределять 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://github.com/linkedin/kafka-tools>). 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 вход;
- ☐ пропускная способность сети на выход;
- ☐ среднее время ожидания диска;
- ☐ процент использования диска.

Исчерпание любого из этих ресурсов будет проявляться одинаково — в виде недо-реплицированных 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 ваших 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 темам 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 на 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;
- ❑ 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 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.

Сбои 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 на одном-единственном брокере может свести на нет производительность всего кластера. Дело в том, что клиенты-производители подключаются ко всем брокерам, на которых располагаются ведущие разделы для темы, 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 вызваны 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 также могут выразиться в проблемах операционной системы, например, недостаточном размере сетевых буферов или 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 растёт, 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 испытывает какие-то проблемы. Воспользуйтесь системными утилитами, например `top`, для поиска процессов, которые используют 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 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 контролируемом останове брокеров зачастую возникают проблемы (подробности относительно признака текущего контроллера смотрите в табл. 10.3).

Таблица 10.3. Показатель «признак текущего контроллера»

Показатель	Признак текущего контроллера
Управляемый компонент (MBean) JMX	kafka.controller:type=KafkaController, name=ActiveControllerCount
Диапазон значений	0 или 1

Если ни один из брокеров не претендует на звание контроллера кластера, последний не сможет должным образом реагировать на изменения состояния, включая создание тем/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.

Коэффициент простоя обработчиков запросов

Kafka использует два пула потоков выполнения для обработки всех запросов клиентов: сетевые потоки и обработчики запросов. Сетевые потоки отвечают за чтение данных и их запись в клиенты по сети. Это не требует больших вычислительных затрат, то есть шансы исчерпать ресурсы сетевых потоков невелики. 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 загрузки брокеров влияние на этот пул потоков существенно возрастает (подробности о коэффициенте простоя обработчиков запросов смотрите в табл. 10.4).

Таблица 10.4. Коэффициент простоя обработчиков запросов

Показатель	Признак текущего контроллера
Управляемый компонент (MBean) JMX	kafka.server:type=KafkaRequestHandlerPool, name=RequestHandlerAvgIdlePercent
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 на 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 слабого 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 на брокерах тоже на 0.10. Это приведет к колоссальному снижению использования потоков обработчиков запросов.

Суммарная входящая скорость передачи данных

Суммарная по всем темам 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 перераспределения разделов кластера (подробности в табл. 10.5).

Таблица 10.5. Подробная информация о показателе «суммарная по всем темам входящая скорость передачи данных»

Показатель	Входящая скорость, байт/с
Управляемый компонент (MBean) JMX	kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec
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. 10.6).

Таблица 10.6. Подробная информация о показателе «суммарная по всем темам 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



Учет потоков извлечения данных из реплик

Исходящая скорость включает и трафик реплик. Это значит, что если коэффициент репликации всех тем 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.

Суммарное по всем темам 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. 10.7).

Таблица 10.7. Подробная информация о показателе «суммарное по всем темам 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?

Меня 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 информацию о том, сколько сообщений в нем содержится. Следовательно, брокер фактически не знает, сколько было отправлено сообщений. Единственный возможный в такой ситуации показатель — количество извлечений в секунду — представляет собой частоту выполнения запросов, а не количество сообщений.

Количество разделов

Показатель количества разделов для брокера обычно незначительно меняется с течением времени, ведь он представляет собой общее количество разделов, назначенных данному брокеру. Он включает все реплики на брокере, неважно, ведущие или ведомые. Мониторинг этого показателя более интересен в случае кластера, на котором включено автоматическое создание тем, поскольку при этом владелец кластера не всегда контролирует создание тем (подробности в табл. 10.8).

Таблица 10.8. Подробная информация о показателе «количество разделов»

Показатель	Количество разделов
Управляемый компонент (MBean) JMX	kafka.server:type=ReplicaManager,name=PartitionCount
Диапазон значений	Целое число, равное нулю или больше него

Количество ведущих реплик

Количество 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 количество 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 перераспределение ведущих реплик). В этом случае данный показатель покажет меньшее число (или даже 0) ведущих реплик, указывая на необходимость запуска выбора предпочтительной реплики для перераспределения ведущих реплик кластера (подробности в табл. 10.9).

Таблица 10.9. Подробная информация о показателе «количество ведущих реплик»

Показатель	Количество ведущих реплик
Управляемый компонент (MBean) JMX	kafka.server:type=ReplicaManager,name=LeaderCount
Диапазон значений	Целое число, равное нулю или больше него

Удобно использовать этот показатель совместно с количеством 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 количества недореплицированных разделов, мониторинг количества отключенных разделов критически важен (табл. 10.10). Этот показатель предоставляет только брокер — контроллер кластера (все остальные брокеры будут возвращать 0), причем он показывает количество разделов кластера, у которых сейчас нет ведущей реплики. Без ведущей реплики раздел 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 10.10. Показатель «число отключенных разделов»

Показатель	Число ведущих реплик
Управляемый компонент (MBean) JMX	kafka.controller:type=KafkaController,name=OfflinePartitionsCount
Диапазон значений	Целое число, равное нулю или больше него

При промышленной эксплуатации кластера Kafka отключенные разделы могут влиять на клиенты-производители, приводя к потере сообщений или отрицательному обратному влиянию в приложении. Чаще всего это заканчивается тем, что сайт становится неработоспособным и требует немедленного вмешательства.

Показатели запросов

В протоколе Kafka, описанном в главе 5, есть множество различных типов запросов. С помощью определенных показателей контролируют функционирование следующих типов запросов:

- ❑ ApiVersions;
- ❑ ControlledShutdown;
- ❑ CreateTopics;
- ❑ DeleteTopics;
- ❑ DescribeGroups;

- ☐ Fetch;
- ☐ FetchConsumer;
- ☐ FetchFollower;
- ☐ GroupCoordinator;
- ☐ Heartbeat;
- ☐ JoinGroup;
- ☐ LeaderAndIsr;
- ☐ LeaveGroup;
- ☐ ListGroups;
- ☐ Metadata;
- ☐ OffsetCommit;
- ☐ OffsetFetch;
- ☐ Offsets;
- ☐ Produce;
- ☐ SaslHandshake;
- ☐ StopReplica;
- ☐ SyncGroup;
- ☐ UpdateMetadata.

Для каждого из них существует восемь показателей, позволяющих отслеживать обработку каждой из фаз запроса. Например, показатели для запроса `Fetch` перечислены в табл. 10.11.

Таблица 10.11. 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>
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 этапы обработки запроса.

- ❑ *Общее время* — оценка общего 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:

- ❑ процентиля — `50thPercentile`, `75thPercentile`, `95thPercentile`, `98thPercentile`, `99thPercentile`, `999thPercentile`;
- ❑ `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.



Что такое процентиль

Процентили — распространенный способ представления показателей хронометража. В частности, 99-й 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 процентили. При этом становится понятно, как происходит обработка среднестатистического запроса и 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 конкретной темы, 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 тем и разделов

Помимо множества доступных на брокерах показателей, описывающих функционирование брокеров Kafka в целом, существуют и показатели уровня тем и разделов. В больших кластерах их может быть очень много, так что не получится систематизировать их все в процессе обычного функционирования. Однако они очень удобны при отладке конкретных проблем с клиентами. Например, показатели уровня тем можно использовать для поиска конкретной темы, которая 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. 10.12 мы используем имя темы `TOPICNAME` и раздел 0. При обращении к описанным показателям не забудьте поменять имя темы и номер раздела на соответствующие значения для вашего кластера.

Таблица 10.12. Показатели уровня темы

Название	Управляемый компонент (MBean) JMX
Скорость входящей передачи данных, байт/с	<code>kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec,topic=TOPICNAME</code>
Скорость исходящей передачи данных, байт/с	<code>kafka.server:type=BrokerTopicMetrics,name=BytesOutPerSec,topic=TOPICNAME</code>
Частота неудачного извлечения данных	<code>kafka.server:type=BrokerTopicMetrics,name=FailedFetchRequestsPerSec,topic=TOPICNAME</code>
Частота неудачной генерации сообщений	<code>kafka.server:type=BrokerTopicMetrics,name=FailedProduceRequestsPerSec,topic=TOPICNAME</code>
Частота входящих сообщений	<code>kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec,topic=TOPICNAME</code>
Частота запросов на извлечение данных	<code>kafka.server:type=BrokerTopicMetrics,name=TotalFetchRequestsPerSec,topic=TOPICNAME</code>
Частота запросов от производителей	<code>kafka.server:type=BrokerTopicMetrics,name=TotalProduceRequestsPerSec,topic=TOPICNAME</code>

Показатели уровня темы

Показатели уровня темы очень схожи с описанными ранее показателями брокеров. Фактически единственное отличие состоит в указании названия темы, к которой будут относиться показатели. С учетом числа доступных показателей и в зависимости от количества тем 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 темы. Кроме того, они слишком многочисленны, ведь в сотнях тем легко могут насчитываться 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. 10.13). Если их суммировать, можно узнать объем хранимых данных отдельной темы, что удобно при выделении ресурсов для отдельных клиентов Kafka. Расхождение между размерами двух разделов одной и той же темы может указывать на проблему с неравномерным распределением сообщений по использовавшемуся при генерации ключу. Показатель количества сегментов журнала отражает

количество файлов сегментов журнала на диске для данного раздела. Он удобен для отслеживания ресурсов, как и показатель размера разделов.

Таблица 10.13. Показатели запроса

Название	Управляемый компонент (MBean) JMX
Размер раздела	<code>kafka.log:type=Log,name=Size,topic=TOPICNAME,partition=0</code>
Количество сегментов журнала	<code>kafka.log:type=Log,name=NumLogSegments,topic=TOPICNAME,partition=0</code>
Начальное смещение журнала	<code>kafka.log:type=Log,name=LogEndOffset,topic=TOPICNAME,partition=0</code>
Конечное смещение журнала	<code>kafka.log:type=Log,name=LogStartOffset,topic=TOPICNAME,partition=0</code>

Показатели конечного и начального смещения 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 потеряло 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 очень большого числа требующих отслеживания параметров. Намного легче контролировать количество недореплицированных разделов в масштабах брокера, используя утилиты командной строки (описаны в главе 9) 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, 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. 10.14 показано, какие компоненты использовать в случае JRE Oracle Java 1.8 со сборкой мусора G1.

Таблица 10.14. 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 (полный или молодой) 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, на которой работает брокер. В ее составе есть два полезных атрибута, данные по которым можно собрать тут, но непросто получить в операционной системе: `MaxFileDescriptorCount` и `OpenFileDescriptorCount`. `MaxFileDescriptorCount` представляет собой максимально 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, быстро исчерпается.

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 соотношений 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 непрерывающегося 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 загрузке значение будет равно числу CPU в системе. Например, если в системе 24 процессора, то средняя нагрузка, равная 24, соответствует 100%-ной загрузке.

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 темы число битов исходящих данных при отсутствии потребителей. В зависимости от количества потребителей входящий сетевой трафик может оказаться на порядки больше исходящего. 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 забудет 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, разделить 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 темы, изменении состояния брокера, а также такие операции кластера, как выбор предпочтительной реплики и перемещения разделов. Вторым механизмом 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 мешать.

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`. Он заносит в журнал информацию обо всех отправленных брокеру запросах. На уровне `DEBUG` данный журнал включает конечные точки соединений, хронометраж запросов и сводную информацию. На уровне `TRACE` — также информацию о теме 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

Новый клиент-производитель Kafka существенно повысил компактность имеющихся показателей, сделав их доступными в виде атрибутов небольшого числа управляемых компонентов (`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 *CLIENTID*. В случаях, когда название компонента содержит идентификатор брокера, этот идентификатор заменен на *BROKERID*. Названия тем заменены на *TOPICNAME*. Пример приведен в табл. 10.15.

Таблица 10.15. Управляемые компоненты показателей производителя Kafka

Название	Управляемый компонент (MBean) JMX
В целом по производителю	<code>kafka.producer:type=producer-metrics,client-id=CLIENTID</code>
Для отдельного брокера	<code>kafka.producer:type=producer-node-metrics,client-id=CLIENTID,node-id=node-BROKERID</code>
Для отдельной темы	<code>kafka.producer:type=producer-topic-metrics,client-id=CLIENTID,topic=TOPICNAME</code>

У каждого из компонентов показателей в табл. 10.15 есть несколько атрибутов, предназначенных для описания состояния производителя. Конкретные наиболее полезные атрибуты описаны в следующем разделе. Прежде чем двигаться дальше, убедитесь, что хорошо понимаете семантику работы производителя, описанную 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 используются 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, и если он больше нуля, 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. Повышение времени задержки запроса означает замедление запросов от производителей. Причина может быть в проблемах с сетью или 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 от производителей брокерам запросов в секунду. И, конечно, каждое сообщение состоит из определенного числа байтов. Эти показатели отнюдь не станут лишними на инструментальной панели вашего приложения.



Почему бы не воспользоваться `ProducerRequestMetrics`?

Существует компонент для показателя `ProducerRequestMetrics`, с помощью которого можно получить как проценты задержки запроса, так и несколько скользящих средних частоты запросов. Почему же его не рекомендуется использовать? Проблема в том, что этот показатель выдается отдельно для каждого потока производителя. В приложениях, в которых по соображениям производительности используется несколько потоков, согласование этих показателей может оказаться непростой задачей. Обычно достаточно атрибутов, предоставляемых единым общим компонентом для производителя.

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 темы, в байтах. `record-size-avg` показывает средний размер отдельной записи в байтах. В случае применения производителя с одной темой эти показатели предоставляют полезную информацию о сгенерированных сообщениях. Если используются производители с несколькими темами, 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 наполнения пакета, в соответствии с параметром конфигурации `batch.size`;
- ❑ с момента отправки прошлого пакета пройдет достаточный промежуток времени, соответствующий 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 тем будет применяться первое условие, а для медленных тем — второе. Показатель `record-queue-time-avg` показывает, сколько времени занимает генерация сообщений, а следовательно, будет полезен при настройке этих двух параметров конфигурации с целью удовлетворения требований вашего приложения к времени задержки.

Показатели уровня брокера и темы

Помимо общих показателей производителей существуют компоненты показателей с ограниченным набором атрибутов для подключения к отдельным брокерам Kafka, а также для каждой темы, для которой 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 теме.

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 настоящих показателей в каждый момент времени может быть различным в зависимости от состояния кластера Kafka.

Показатели тем интереснее, чем показатели брокеров, но они могут принести пользу только при использовании производителей, работающих с более чем одной темой. Кроме того, применять их на постоянной основе можно, только если производитель не работает с большим числом тем. Например, `MirrorMaker` может генерировать сотни, если не тысячи тем. 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 тем лучше всего задействовать при поиске причин конкретной проблемы. Атрибуты `record-send-rate` и `record-error-rate`, например, можно использовать для выяснения того, к какой теме относятся отмененные сообщения (или подтверждения того, что они имеются во всех темах). Кроме того, существует показатель `byte-rate` — общая частота сообщений темы (в байтах в секунду).

Показатели потребителей

Аналогично новому клиенту-производителю новый 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. В потребителе в силу того, что логика потребления сообщений сложнее простой отправки сообщений брокерам Kafka, есть несколько дополнительных показателей (табл. 10.16).

Таблица 10.16. Управляемые компоненты показателей потребителя Kafka

Название	Управляемый компонент (MBean) JMX
В целом по потребителю	<code>kafka.consumer:type=consumer-metrics,client-id=CLIENTID</code>
Диспетчер извлечения	<code>kafka.consumer:type=consumer-fetch-manager-metrics,client-id=CLIENTID</code>
Для отдельной темы	<code>kafka.consumer:type=consumer-fetch-manager-metrics,client-id=CLIENTID,topic=TOPICNAME</code>
Для отдельного брокера	<code>kafka.consumer:type=consumer-node-metrics,client-id=CLIENTID,node-id=node-BROKERID</code>
Координатор	<code>kafka.consumer:type=consumer-coordinator-metrics,client-id=CLIENTID</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 показателей 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 на извлечение к брокерам. 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 медленной темы 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 тем 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 отражает текущее отставание в виде числа сообщений 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 использовать 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 состояния потребителя. 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 помо-

щью 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 темы

Показатели, предоставляемые клиентом-потребителем по каждому из соединений брокера и каждому из потребляемых тем, 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 тем. 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 темы оказываются полезны при чтении более чем одной темы. 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 тем (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 конкретной темы сообщений в байтах в секунду, а `records-consumed-rate` — ту же информацию в виде количества сообщений. Показатель `fetch-size-avg` представляет собой средний размер запроса на извлечение для данной темы в байтах.

Показатели координатора потребителя

Как описывалось в главе 4, клиенты-потребители обычно работают в составе группы потребителей. Эта группа 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, инициируя 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 специальную тему. 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. Существуют 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. 10.17.

Таблица 10.17. Показатели, требующие отслеживания

Клиент	Название компонента
Потребитель	bean kafka.consumer:type=consumer-fetch-manager-metrics,client-id=CLIENTID,attribute fetch-throttle-time-avg
Производитель	bean kafka.producer:type=producer-metrics,client-id=CLIENTID, attribute produce-throttle-time-avg

По умолчанию квоты на 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 с их мониторинга, а не добавлять показатели потом.

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 или неточен, или вообще недоступен.

Предпочтительный метод мониторинга отставания потребителя — использование внешнего процесса, который может отслеживать состояние как раздела на брокере (путем наблюдения за последним сгенерированным сообщением), так и потребителя (путем наблюдения 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 9 приводилась информация по использованию 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 темы, получающей 100 сообщений в час, необходимо иное пороговое значение, чем для темы, получающей 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 тем, задача будет не из легких.

Один из способов снижения сложности мониторинга групп потребителей — использование Burrow. 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 использования 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 каждой темы отдельно. Однако в большинстве случаев неразумно раздувать трафик тем за счет искусственной добавки. Можно, однако, по крайней мере получить ответы на данные вопросы для каждого из брокеров кластера, и именно это делает Kafka Monitor

(<https://github.com/linkedin/kafka-monitor>). Эта утилита с открытым исходным кодом, созданная командой разработчиков Kafka компании LinkedIn, непрерывно генерирует и потребляет данные из темы, распределенной 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 использования 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, обсудили использование внешних систем мониторинга с целью отслеживания отставания потребителей и сквозной доступности кластера. Эта глава хоть и не претендует на звание исчерпывающего списка доступных показателей, но охватывает наиболее важные из них, требующие постоянного отслеживания.

11

Потоковая обработка

Kafka 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.

Специалисты-аналитики в данной сфере иногда утверждают, что все эти системы потоковой обработки по сути аналогичны сложным системам обработки событий (СЕР), существующим уже около 20 лет. Нам представляется, что популярность потоковой обработки растет потому, что она появилась после Kafka, а значит, есть возможность использовать Kafka в качестве надежного источника потоков событий. По мере роста популярности Kafka сначала в качестве простой шины сообщений, а потом системы интеграции данных во многих компаниях начали появляться системы, содержащие множество потоков интересных данных, хранимых 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. 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 обра-

ботки Apache Kafka — ее задачами и архитектурой. Мы покажем небольшой пример использования библиотеки Kafka Streams для подсчета скользящего среднего цен акций. Затем обсудим другие примеры удачных сценариев применения потоковой обработки и завершим главу небольшим перечнем критериев выбора фреймворка потоковой обработки для использования совместно с Apache Kafka. Эта глава задумана как краткое введение в потоковую обработку и не охватывает всех возможностей Kafka Streams, равно как в ней не делается попытка обсудить и сравнить все существующие фреймворки потоковой обработки — эти темы заслуживают отдельной книги, а может, и нескольких.

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. Я уже сталкивался с подобным 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 их 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 я сначала кладу деньги на счет в банке, а затем их трачу, сильно отличается от последовательности, при которой я сначала трачу деньги, а затем гашу 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 содержаться 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 со множеством заголовков), они могут быть полностью 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. Потоковая обработка означает непрерывную обработку одного или более потоков событий. Потоковая 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 в 02:00 каждый день или каждый час, 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. Он заполняет пробел между вселенной «запрос/ответ», где приходится ждать событий, а обработка занимает всего 2 мс, и вселенной пакетной обработки, где обработка данных производится раз в день и 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 вы непрерывно читаете данные из неограниченного набора данных, что-то с ними делаете и выводите результаты, значит, вы выполняете потоковую обработку. Но обработка 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: пишется 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.

Время

Время — вероятно, важнейшее из понятий потоковой обработки, а заодно и наиболее запутанное. Если вы хотите получить представление о том, насколько сложным может быть время в сфере распределенных систем, рекомендуем заглянуть в превосходную статью Джастина Шийи (Justin Sheehy) «Настоящего не существует» (There is No Now) (<http://www.bit.ly/2rXXdLr>). В контексте потоковой 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 потоковой 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 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 вашего приложения о *времени события*, например, при создании записей Kafka на основе записи

базы данных через какое-либо время после фактического события, желательно добавить время события в виде поля самой записи. 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. В версиях 0.10.0 и более поздних брокеры Kafka автоматически добавляют это время в получаемые записи, если Kafka настроена соответствующим образом или если записи поступили от производителей более старых версий и не содержат меток даты/времени. В потоковой 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.
- ❑ *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.



Не забывайте о часовых поясах

Работая с временем, важно не забывать 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 требуется обрабатывать события по отдельности, потоковая обработка — вещь очень простая. Например, для простого чтения потока транзакций о покупках в интернет-магазине из Kafka, поиска среди них транзакции на сумму более 10 000 долларов и отправки сообщения электронной почты о них соответствующему торговцу достаточно нескольких строк кода с использованием потребителя Kafka и SMTP-библиотеки.

Наиболее интересной потоковая 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, хранимую дополнительно к событиям, *состоянием* (state).

Заманчиво было бы хранить состояние в локальных переменных приложения потоковой обработки, например, хранить скользящие средние в простой хэш-таблице. Однако такой подход к хранению состояния при потоковой обработке ненадежен, поскольку при останове приложения потоковой 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 потоковой обработке используются несколько типов состояния.

- ❑ *Локальное (внутреннее) состояние.* Состояние, доступное только конкретному экземпляру приложения потоковой 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 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 — повышение времени задержки и привносимая еще одной системой дополнительная сложность. 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 кон-

критный 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. Потоки представляют 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 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 захвата изменений данных (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 магазин, продающий обувь. Потоковое 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 (рис. 11.1). Для того чтобы увидеть, насколько оживленно идет торговля, можно посмотреть на поток данных в целом и узнать, что было выполнено пять транзакций. Возможно, нам захочется также выяснить, почему вернули синие туфли.

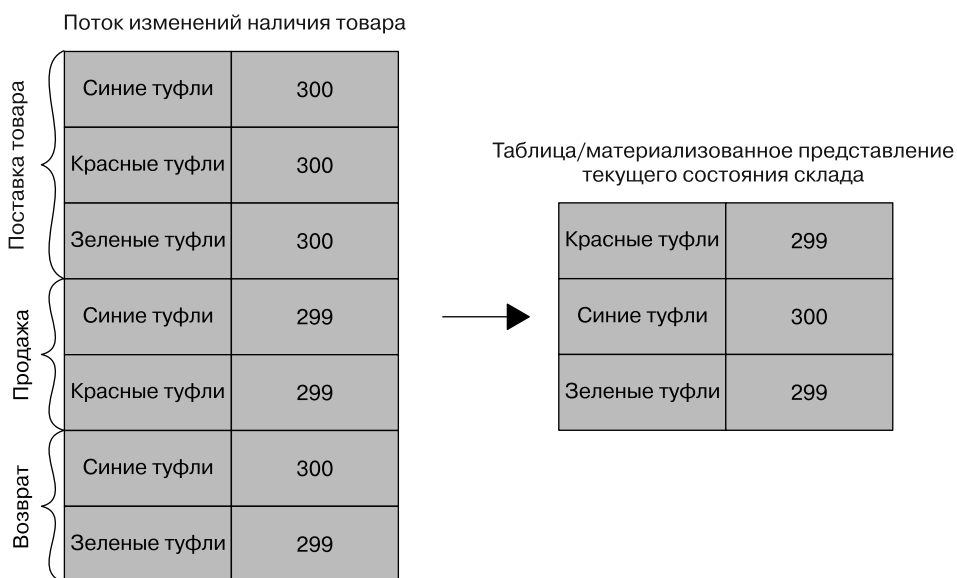


Рис. 11.1. Материализация изменений товарных остатков

Временные окна

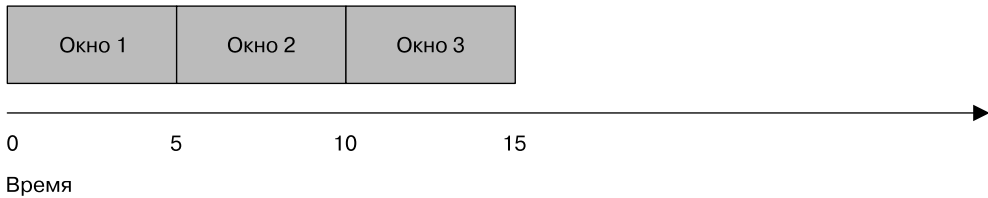
Большинство операций над потоками данных — оконные, то есть оперирующие над временными интервалами: скользящие средние, самые продаваемые товары за неделю, 99-й процентиль нагрузки на систему и т. д. Операции объединения двух потоков данных также носят оконный характер — при этом объединяются события, 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.
- ❑ Насколько часто окно сдвигается (*интервал опережения*, *advance interval*): обновлять ли пятиминутные средние значения каждую минуту, секунду или при каждом поступлении нового события? Окно, размер которого равен его *интервалу опережения*, иногда называют «кувыркающимся» (*tumbling window*). Окно, которое перемещается при каждой новой записи, иногда называют *скользящим* (*sliding window*).
- ❑ В течение какого времени сохраняется возможность обновления окна? Допустим, что пятиминутное скользящее среднее подсчитывается для окна 00:00–00:05.

А через час мы получаем еще несколько результатов, полученных в 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, 03:17–03:22. Скользящие окна никогда не выравниваются, потому что перемещаются при каждой новой записи. Различия между этими типами окон показаны на рис. 11.2.

«Кувыркающееся» окно: пятиминутное окно, перемещается каждые пять минут



«Прыгающее» окно: пятиминутное окно, перемещается каждую минуту. Окна перекрываются, так что одно и то же событие может относиться к нескольким окнам

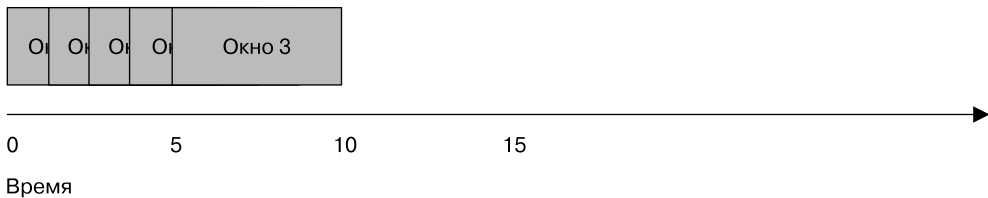


Рис. 11.2. «Кувыркающиеся» и «прыгающие» окна

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 на этапе отображения события преобразуются, 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. Подобные приложения могут не 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, можно просто делегировать обработку событий другому экземпляру приложения.

Для этого паттерна вполне достаточно простого производителя и потребителя (рис. 11.3).

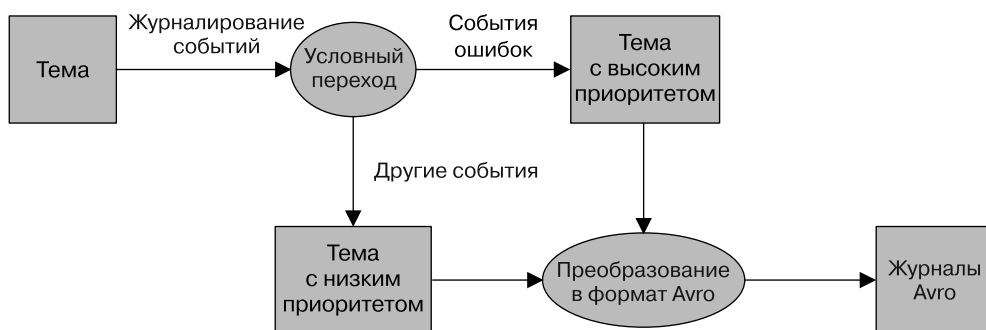


Рис. 11.3. Топология обработки событий по отдельности

Обработка с использованием локального состояния

Для большинства приложений потоковой обработки важную роль играет агрегирование информации, особенно по временным окнам. Примером этого может служить поиск минимальной и максимальной цены акций для каждого дня торгов и вычисление скользящего среднего.

Подобное агрегирование требует сохранения *состояния* потока данных. В нашем примере для вычисления минимальной и средней цены акций за день необходимо

хранить встречавшиеся до сих пор минимальное и максимальное значения и сравнивать с ними каждое новое значение из потока данных.

Для этого можно использовать *локальное* (не разделяемое) 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. 11.4).

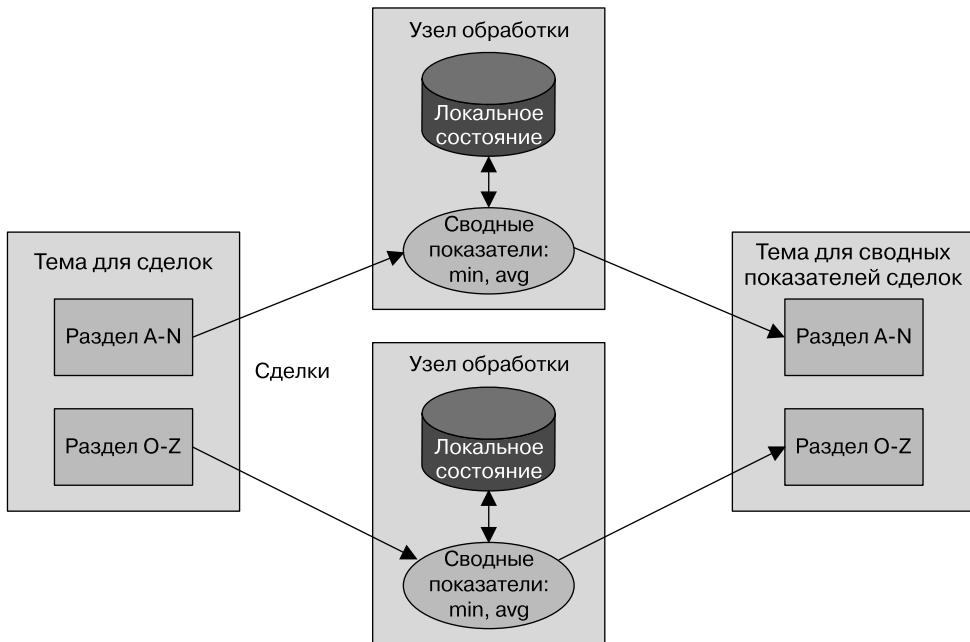


Рис. 11.4. Топология обработки событий с применением локального состояния

Приложения потоковой обработки существенно усложняются при наличии у приложения локального состояния, так как возникает несколько проблем, которые должно решить такое приложение.

- ❑ *Использование памяти.* Локальное состояние должно помещаться в доступной экземпляру приложения оперативной памяти.
- ❑ *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 тему Kafka. В случае останова узла потока данных локальное состояние не утрачивается — его можно легко восстановить путем повторного чтения событий из темы 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 тем сжатие журналов, чтобы гарантировать, что они не будут расти до бесконечности, и иметь возможность восстановить состояние.

- *Перераспределение.* 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, проверьте, 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 задействовать всю доступную информацию? Например, допустим, что нужно каждый день публиковать 10 самых быстро растущих ценных бумаг — 10 ценных бумаг, стоимость которых сильнее всего выросла за день торгов (с открытия торгов до закрытия биржи). Конечно, никаких локальных действий на отдельном экземпляре приложения не будет для этого достаточно, поскольку все 10 нужных ценных бумаг могут находиться на разделах, относящихся к другим экземплярам. Нам понадобится двухэтапный подход. Сначала нужно вычислить ежедневный рост/падение цены для каждого из символов акций. Это можно сделать на каждом из отдельных экземпляров с помощью локального состояния. Затем следует записать результаты в новую тему из одного раздела. Далее отдельный экземпляр приложения читает этот раздел и находит там 10 наиболее быстро растущих в цене акций. Вторая тема, содержащая лишь сводные показатели по символам акций, очевидно, будет намного меньше (как и объем ее трафика), чем темы, содержащие саму информацию о сделках, а значит, ее сможет обработать один экземпляр приложения. Иногда для получения результата необходимо больше шагов (рис. 11.5).

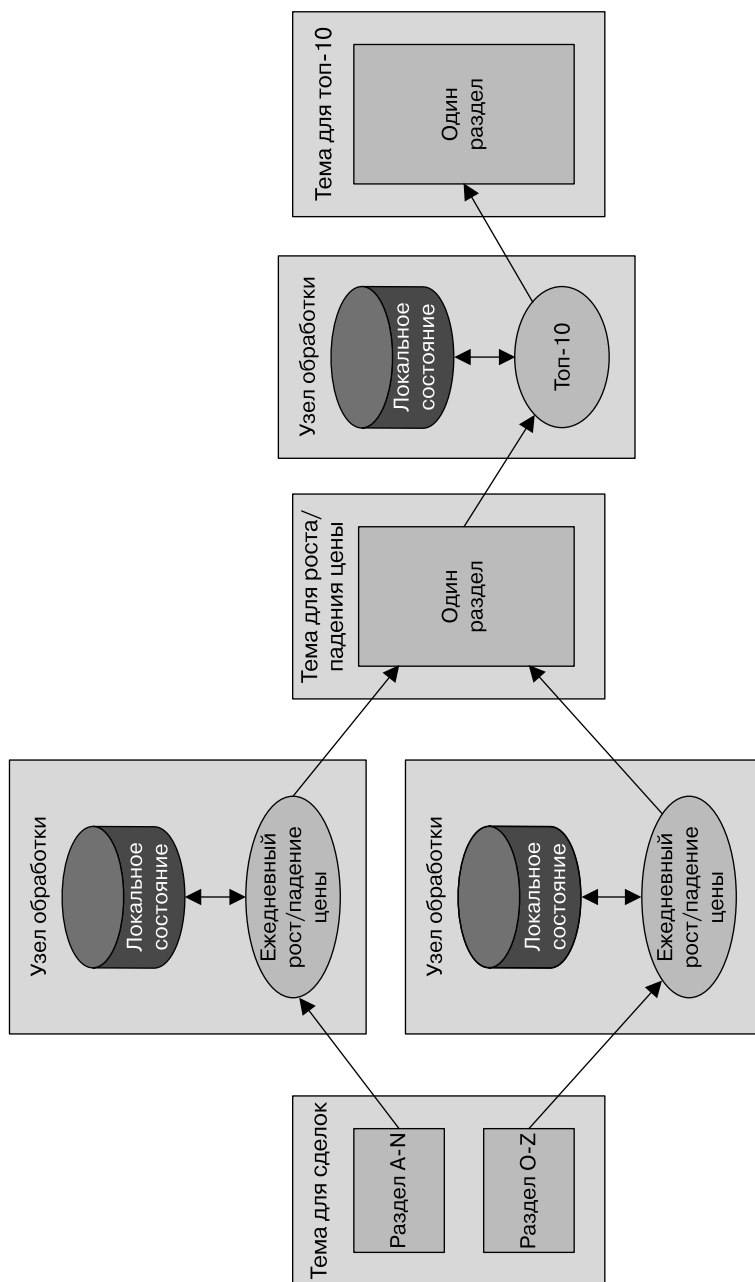


Рис. 11.5. Топология, включающая как использование локального состояния, так и повторное секционирование

Такая разновидность многоэтапной обработки хорошо знакома тем, кому случалось писать код отображения/свертки, 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 использования 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 другую тему событие, включающее первоначальный щелчок на ссылке плюс возраст и пол пользователя (рис. 11.6).

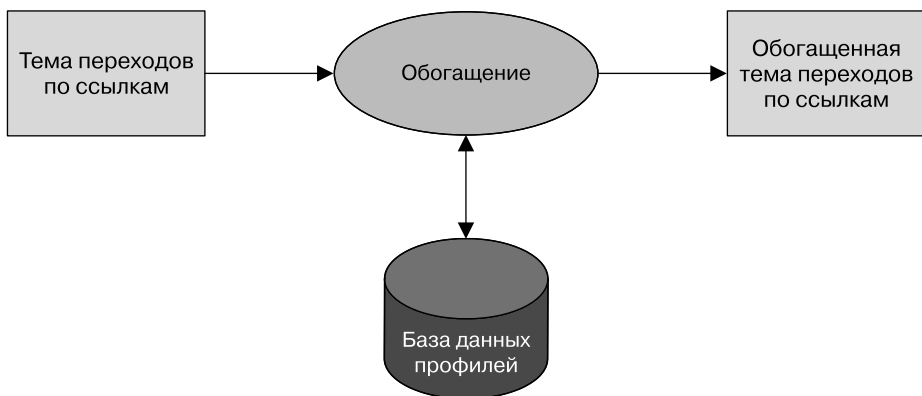


Рис. 11.6. Поточковая обработка с использованием внешнего источника данных

Проблема с этим напрашивающимся вариантом состоит в том, что внешний справочник существенно увеличивает время обработки каждой записи — обычно на 5–15 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. Хотелось бы найти решение, которое бы масштабировалось лучше.

Чтобы обеспечить хорошую производительность и масштабирование, 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 CDC. Разработчики, использующие Kafka Connect, обнаружат там множество коннекторов, предназначенных для выполнения CDC и преобразования таблиц базы данных в поток событий изменения. Благодаря этому вы сможете хранить свою собственную копию таблицы, обновляя ее соответствующим образом при получении уведомления о каждом событии изменения базы данных (рис. 11.7).

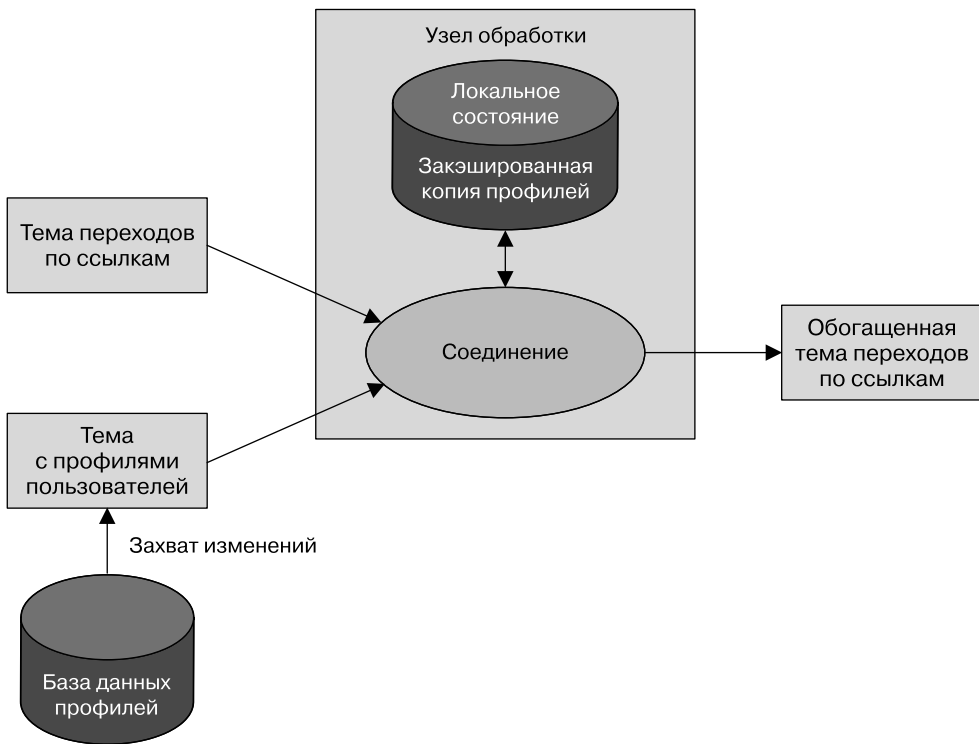


Рис. 11.7. Топология соединения таблицы и потока событий, благодаря которой нет необходимости использовать внешний источник данных при потоковой обработке

Далее при получении событий переходов пользователей по ссылкам вы сможете найти `user_id` в локальном кэше и выполнить обогащение события. Благодаря применению локального кэша 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 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 и 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 мышью 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 в несколько секунд, и искать соответствие результатов для каждого из них (рис. 11.8).

В библиотеке Kafka Streams это работает следующим образом: оба потока данных, запросы и щелчки на ссылках секционируются по одним и тем же ключам, которые представляют собой и ключи соединения. При этом все события щелчков от пользователя `user_id:42` попадают в раздел 5 темы событий щелчков, а все события поиска для `user_id:42` — в раздел 5 темы событий поиска. После этого Kafka Streams обеспечивает назначение раздела 5 обеих тем одной задаче Kafka, так что этой задаче оказываются доступны все соответствующие события для пользователя `user_id:42`. Она хранит во встроенном кэше RocksDB временное окно соединения для обеих тем и благодаря этому может выполнить соединение.

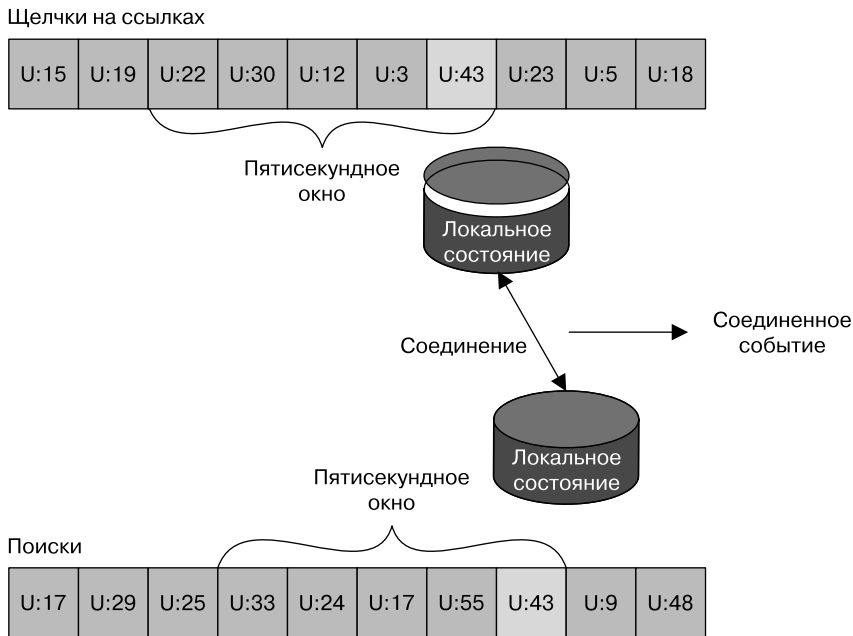


Рис. 11.8. Соединение двух потоков событий. В таких случаях всегда применяется временное окно

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. 11.9). 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).

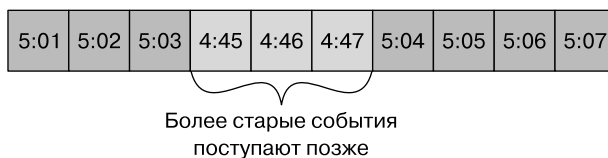


Рис. 11.9. Внеочередные события

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 темы результатов. Обычно они представляют собой сжатые темы, 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, перезаписывая предыдущий.

Повторная обработка

Последний из важных паттернов — 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 существующее 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 тем (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 навестает 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 без риска потерять критически важные данные или внести ошибки в процессе очистки.

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 по причине простоты его применения, а также потому что он поставляется вместе с уже имеющимся у вас 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, но, как вы увидите, это редко используется на практике.

Создание приложения, задействующего API DSL, всегда начинается с формирования с помощью *StreamBuilder* *топологии* обработки — ориентированного ациклического графа (DAG) преобразований, применяемых ко всем событиям потоков. Затем на основе топологии создается исполняемый объект *KafkaStreams*. При запуске объекта *KafkaStreams* создается несколько потоков выполнения, каждый из которых использует топологию обработки к событиям из потоков. Обработка завершается по закрытии объекта *KafkaStreams*.

Мы рассмотрим несколько примеров использования *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 потокковой обработки необходимо настроить *Kafka Streams*. У него есть множество параметров, которые мы не станем 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.KEY_SERDE_CLASS_CONFIG,
            Serdes.String().getClass().getName()); ❸
        props.put(StreamsConfig.VALUE_SERDE_CLASS_CONFIG,
            Serdes.String().getClass().getName());
```

- ❶ У каждого приложения *Кафка* должен быть свой идентификатор приложения. Он используется для координации действий экземпляров приложения, а также

именования внутренних локальных хранилищ и относящихся к ним тем. Среди приложений Kafka Streams, работающих в пределах одного кластера, идентификаторы не должны повторяться.

- ❷ Приложения Kafka Streams всегда читают данные из тем Kafka и записывают результаты в темы 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:

```
KStreamBuilder builder = new KStreamBuilder(); ❶

KStream<String, String> source =
    builder.stream("wordcount-input");

final Pattern pattern = Pattern.compile("\\W+");

KStream counts = source.flatMapValues(value->
    Arrays.asList(pattern.split(value.toLowerCase())) ❷
    .map((key, value) -> new KeyValue<Object,
        Object>(value, value))
    .filter((key, value) -> (!value.equals("the"))) ❸
    .groupByKey() ❹
    .count("CountStore").flatMapValues(value->
        Long.toString(value)).toStream(); ❺
counts.to("wordcount-output"); ❻
```

- ❶ Создаем объект класса `KStreamBuilder` и приступаем к описанию потока, передавая название входной темы.
- ❷ Все читаемые из темы-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` просто 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 лишь запустить его:

```
KafkaStreams streams = new KafkaStreams(builder, 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 (мы 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 мы рекомендуем запустить полный вариант примера. Инструкции по выполнению полного примера можно найти в файле `README` репозитория на GitHub (<http://www.bit.ly/2sOXzUN>).

Отметим, что для запуска всего примера не требуется ничего устанавливать, кроме самой Apache Kafka. Вы могли наблюдать подобное при использовании Spark, например, в *локальном режиме* (Local Mode). Основное различие: если во входной теме 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 Spark — то, что использовать его в локальном режиме очень просто, но для эксплуатации кластера необходимо установить YARN или Mesos, после чего установить Spark на всех машинах, а затем разобраться с запуском приложения на кластере. В случае же API Kafka Streams, можно 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 оконных сводных 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 на странице:

```
Properties props = new Properties();
props.put(StreamsConfig.APPLICATION_ID_CONFIG, "stockstat");
props.put(StreamsConfig.BootstrapServersConfig,
Constants.BROKER);
props.put(StreamsConfig.KeySerdeClassConfig,
Serdes.String().getClass().getName());
props.put(StreamsConfig.ValueSerdeClassConfig,
TradeSerde.class.getName());
```

Основное отличие — в использовании классов `Serde`. В примере подсчета количества слов на странице 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 проектов, как GSON, Avro, Protobufs и т. п.

Теперь, когда настройка завершена, можно заняться топологией:

```

KStream<TickerWindow, TradeStats> stats = source.groupByKey() ❶
    .aggregate(TradeStats::new, ❷
        (k, v, tradestats) -> tradestats.add(v), ❸
        TimeWindows.of(5000).advanceBy(1000), ❹
        new TradeStatsSerde(), ❺
        "trade-stats-store") ❻
    .toStream(((key, value) -> new TickerWindow(key.key(),
        key.window().start())) ❼
    .mapValues((trade) -> trade.computeAvgPrice()); ❽

stats.to(new TickerWindowSerde(), new TradeStatsSerde(),
    "stockstats-output"); ❾

```

- ❶ Мы начинаем с чтения событий из входной темы и выполнения операции `groupByKey()`. Несмотря на название, эта операция ничего не группирует. Она обеспечивает разделение потока событий на разделы по ключам записей. А поскольку мы записываем данные в тему с ключами и не меняем последние до вызова `groupByKey()`, то данные остаются разделенными по ключам, так что этот метод в нашем случае ничего не делает.
- ❷ Обеспечив правильное секционирование, мы приступаем к оконному 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.
- ❸ Далее мы указываем метод для собственно 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.
- ❹ Задаем окно, в данном случае пятисекундное (5000 мс), перемещающееся вперед раз в секунду.

- 5 Далее создаем объект `Serde` для сериализации и десериализации результатов агрегирования (объект `TradeStats`).
- 6 Как уже 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 название хранилища состояния. Им может быть любое уникальное название.
- 7 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 и заменяем ключ, в котором содержится определение временного окна нашим собственным ключом, содержащим только символ акции и начальное время окна. Этот метод `toStream` преобразует таблицу в поток, а также преобразует ключ в объект типа `TickerWindow`.
- 8 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.
- 9 И наконец, записываем результаты в поток `stockstats-output`.

После описания 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 сайте. Мы сгенерируем поток имитационных щелчков по ссылкам, 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 пользовательских профилях? Подобные соединения позволяют получить массу информации для анализа. На такой информации часто основаны рекомендации товаров: если пользователь искал велосипеды, щелкал по ссылкам для слова Trek, значит, ему интересны велосипедные путешествия, так что можно рекламировать ему велосипеды Trek, шлемы и велотуры в экзотические места, например, 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(Serdes.Integer(),
new PageViewSerde(), Constants.PAGE_VIEW_TOPIC); ❶
KStream<Integer, Search> searches =
builder.stream(Serdes.Integer(), new SearchSerde(),
Constants.SEARCH_TOPIC);
KTable<Integer, UserProfile> profiles =
builder.table(Serdes.Integer(), new ProfileSerde(),
Constants.USER_PROFILE_TOPIC, "profile-store"); ❷

KStream<Integer, UserActivity> viewsWithProfile = views.leftJoin(profiles, ❸
    (page, profile) -> new UserActivity(profile.getUserID(),
    profile.getUserName(), profile.getZipcode(),
    profile.getInterests(), "", page.getPage())); ❹

KStream<Integer, UserActivity> userActivityKStream =
viewsWithProfile.leftJoin(searches, ❺
    (userActivity, search) ->
    userActivity.updateSearch(search.getSearchTerms()), ❻
    JoinWindows.of(1000), Serdes.Integer(),
    new UserActivitySerde(), new SearchSerde()); ❼
```

- ❶ Прежде всего мы создаем объекты потоков для двух потоков, которые собираются объединять, — переходов по ссылкам и операций поиска.
- ❷ Создаем также таблицу типа `KTable` для профилей пользователей. `KTable` представляет собой локальный кэш, обновляемый 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 заэкшированной копии таблицы профилей. Мы выполняем левое внешнее соединение, так что в результаты попадут переходы по ссылкам, для которых выполнивший их пользователь неизвестен.
- ❹ 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, разработчик должен сам решить, как эти два значения будут объединены в единый результат. В данном случае мы создали один объект

activity, содержащий как информацию о пользователе, так и просмотренную им страницу.

- ❶ 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.
- ❷ Это и есть метод, выполняющий соединение, — мы просто добавляем ключевые слова поиска ко всем соответствующим просмотрам страниц.
- ❸ 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 по ссылкам, выполненными в течение короткого промежутка времени после поиска. Так что мы зададим размер окна соединения, равный одной секунде. То есть будем считать подходящими щелчки на ссылках, выполненные в течение не более чем одной секунды после поиска, и соответствующие поисковые ключевые слова будут включаться в запись о действиях пользователя, содержащую информацию о переходе по ссылке, и профиль пользователя. Благодаря этому появится возможность провести полный анализ операций поиска и их результатов.

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 складу данных. Второй паттерн относится к соединению двух потоков на основе временного окна. Эта операция встречается только в сфере потоковой обработки.

Как обычно, вы можете найти 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 по крайней мере 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. 11.10 показана топология для примера с подсчетом количества слов.

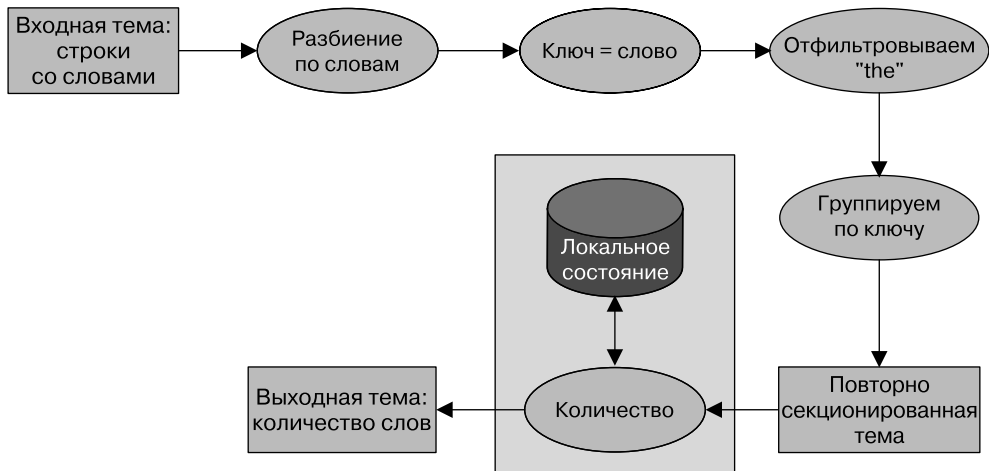


Рис. 11.10. Топология для примера подсчета числа слов с помощью потоковой обработки

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 тем и передающих их дальше, а также узлы обработки — приемники, получающие данные из предыдущих узлов обработки и генерирующие их в тему. Топология всегда начинается с одного или нескольких узлов обработки — производителей и заканчивается одним или несколькими узлами обработки — приемниками.

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 количества разделов в обрабатываемых приложением темах. 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. 11.11).

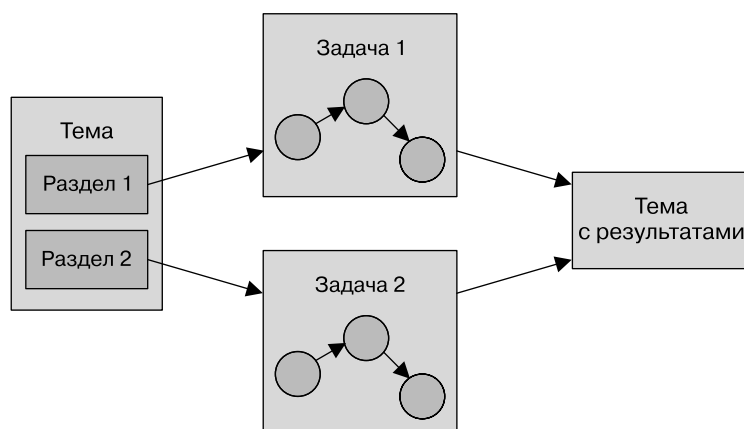


Рис. 11.11. Две задачи, реализующие одну топологию, — по одной для каждого раздела входной темы

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 темах. 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. 11.12).

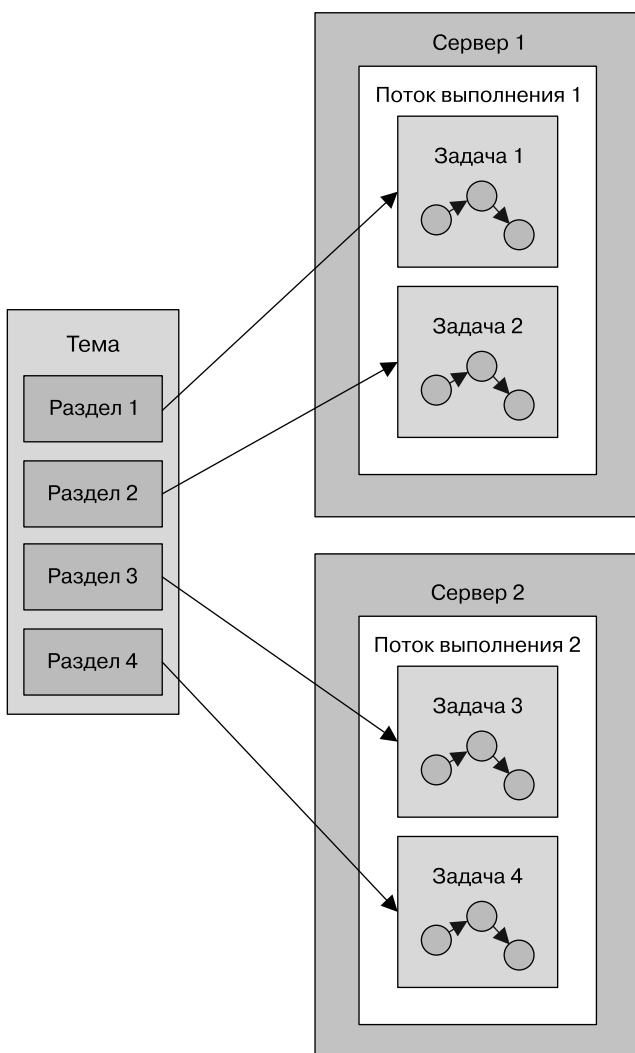


Рис. 11.12. Задачи потоковой обработки могут выполняться в нескольких потоках и на нескольких серверах

Возможно, вы обратили внимание на то, что для шага обработки иногда требуются результаты из нескольких разделов, вследствие чего между задачами могут возникать зависимости. Например, при соединении двух потоков данных (как в примере из раздела «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 темах было одинаковое количество разделов и чтобы они были секционированы по ключу соединения.

Еще один пример возникновения зависимостей между задачами — случай, когда для приложения требуется повторное секционирование. Например, 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 придется повторно разделить данные по почтовому индексу и выполнить 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 *перетасовка* (`shuffle`), а значит, придется отправлять события другим задачам для обработки. В отличие от других фреймворков потоковой обработки Kafka Streams выполняет повторное разделение путем записи событий в новую тему с новыми ключами и разделами. Далее иной набор задач читает эти события из новой темы 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 тему 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. 11.13).

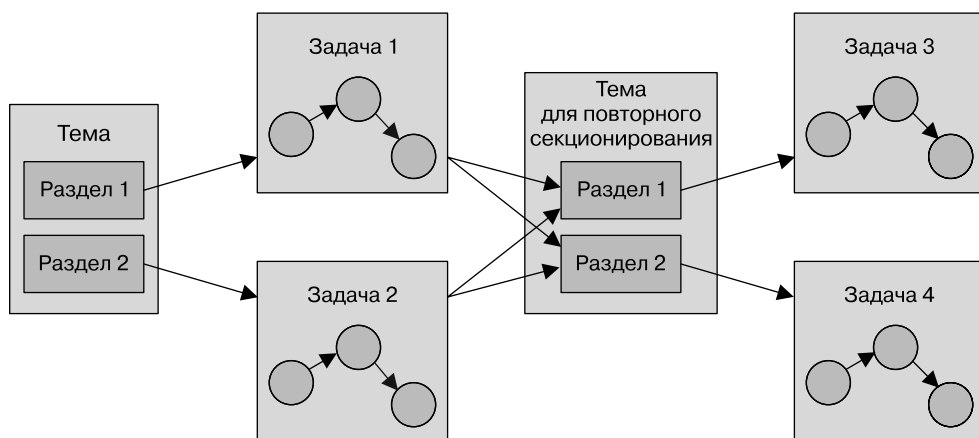


Рис. 11.13. Два набора задач, обрабатывающих события, с темой для повторного разбиения на разделы событий между ними

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.

Сценарии использования потоковой обработки

На протяжении данной 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 обработка, 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, но слишком расплывчато. Рассмотрим несколько настоящих задач, решаемых с помощью потоковой обработки.

- ❑ *Обслуживание клиентов.* Допустим, вы только что забронировали комнату в большой сети отелей и ожидаете получения подтверждения по электронной почте и платежной квитанции. Через несколько минут после бронирования, когда подтверждение все еще не прибыло, вы звоните 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 меня 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 я бы получал 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 мои 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 за качеством 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. Такими паттернами могут быть потерянные пакеты в случае сетевого коммутатора, повышение усилий, необходимых для закручивания гаек, в фабричном производстве или частоты перезагрузки тюнера кабельного ТВ.
- *Обнаружение мошенничества.* Известно также под названием «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 под названием *сигнализация* (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, лучше 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, лучше также 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 вам понадобится система потоковой обработки, хорошо интегрируемая с выбранной вами шиной сообщений (надемся, что ею будет Kafka), обладающая возможностями захвата изменений для удобной передачи произошедших далее по конвейеру изменений в локальный кэш 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 соображений по поводу сценариев использования следует учитывать несколько общих.

- ❑ *Удобство эксплуатации системы.* Легко ли развертывать 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 хранения локальных кэшей, 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, относительно 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. Мы дали формальное ее определение и обсудили характерные черты парадигмы потоковой обработки. Мы также сравнили ее с другими парадигмами программирования.

Далее мы обсудили важнейшие понятия 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.

Приложение. Установка Kafka на 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 на Windows

В Windows 10 можно запускать Kafka двумя способами. Традиционный способ — применить естественную Java-установку. 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)

(<http://www.bit.ly/2r6HnN7>). После этого необходимо установить JDK с помощью утилиты `apt-get`:

```
$ sudo apt-get install openjdk-7-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 задействовать 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 8, которую можно найти на странице загрузки Oracle Java SE (<http://www.bit.ly/TEA7iC>). 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 1.8 update 121 выбрать путь `C:\Java\jdk1.8.0_121`.

После установки Java необходимо настроить переменные среды. Это можно сделать в панели управления Windows, хотя точное место зависит от используемой вами версии Windows. В Windows 10 нужно выбрать Система и безопасность (System and Security), затем Система (System), после чего нажать кнопку Изменить параметры (Change settings) в разделе Имя компьютера, имя домена и параметры рабочей группы (Computer name, domain and workgroup settings). При этом откроется окно Свойства системы (System Properties), в котором нужно выбрать вкладку Дополнительно (Advanced) и, наконец, нажать кнопку Переменные среды (Environment Variables).

В этом разделе вы сможете добавить новую пользовательскую переменную `JAVA_HOME` (рис. П.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.

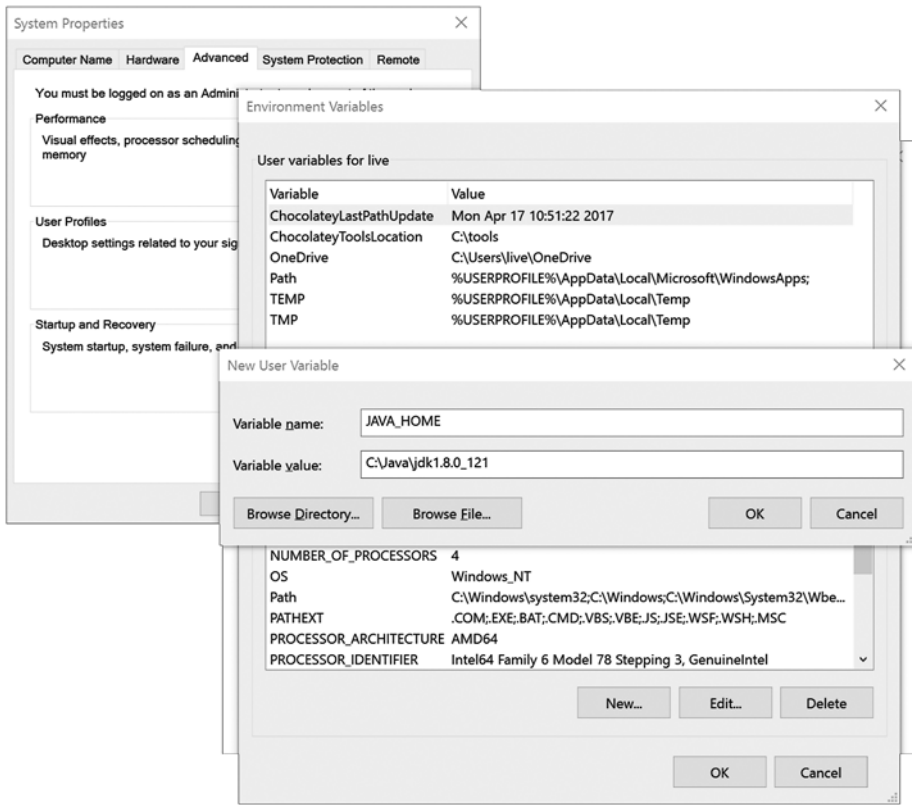


Рис. П.1. Добавление переменной `JAVA_HOME`

Теперь можно приступить к установке Apache Kafka. Установочный пакет включает ZooKeeper, так что устанавливать его отдельно не нужно. На момент выхода данной книги актуальной является версия 0.10.2.0, работающая со Scala 2.11.0¹. 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 8zip. Как и при установке на операционной системе Linux, у вас будет возможность выбрать, куда выполнять разархивирование.

¹ Речь идет об англоязычном издании книги. По состоянию на июль 2018 года новейшей является версия 1.1.1, работающая со Scala 2.12.0 или 2.12.6. — *Примеч. пер.*

В данном примере мы будем считать, что установочный пакет Kafka разархивирован в каталог `C:\kafka_2.11-0.10.1.0`.

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_2.11-0.10.2.0
PS C:\kafka_2.11-0.10.2.0> bin\windows\zookeeper-server-start.bat C:\
\kafka_2.11-0.10.2.0\config\zookeeper.properties
[2017-04-26 16:41:51,529] INFO Reading configuration from: C:\
\kafka_2.11-0.10.2.0\config\zookeeper.properties (org.apache.zoo
keeper.server.quorum.QuorumPeerConfig)
[... ]
[2017-04-26 16:41:51,595] INFO minSessionTimeout set to -1 (org.apache.zoo
keeper.server.ZooKeeperServer)
[2017-04-26 16:41:51,596] INFO maxSessionTimeout set to -1 (org.apache.zoo
keeper.server.ZooKeeperServer)
[2017-04-26 16:41:51,673] INFO binding to port 0.0.0.0/0.0.0.0:2181
(org.apache.zookeeper.server.NIOServerCnxnFactory)
```

После успешного запуска ZooKeeper можете открыть еще одно окно для запуска Kafka:

```
PS C:\> cd kafka_2.11-0.10.2.0
PS C:\kafka_2.11-0.10.2.0> .\bin\windows\kafka-server-start.bat C:\
\kafka_2.11-0.10.2.0\config\server.properties
[2017-04-26 16:45:19,804] INFO KafkaConfig values:
[... ]
[2017-04-26 16:45:20,697] INFO Kafka version : 0.10.2.0 (org.apache.kafka.com
mon.utils.AppInfoParser)
[2017-04-26 16:45:20,706] INFO Kafka commitId : 576d93a8dc0cf421
(org.apache.kafka.common.utils.AppInfoParser)
[2017-04-26 16:45:20,717] INFO [Kafka Server 0], started (kafka.server.Kafka
Server)
```

Установка на MacOS

MacOS основан на Darwin — Unix-подобной операционной системе, ведущей свое происхождение в том числе и от FreeBSD. Это значит, что многое из того, что мы можем ожидать от нее, можно ожидать и от любой Unix-подобной операционной системы, так что установка на 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 произведена установка Java, а потом установлена Apache Kafka 0.10.2.0 (версия по состоянию на момент написания данной книги)¹.

Если вы еще не установили Homebrew, сделайте это, следуя указаниям, приведенным на странице описания установки (<http://docs.brew.sh/Installation.html>). Затем можно будет установить саму Kafka. Система управления пакетами Homebrew гарантирует, что сначала будут установлены все зависимости, включая Java:

```
$ brew install kafka
==> Installing kafka dependency: zookeeper
[...]
==> Summary
/usr/local/Cellar/kafka/0.10.2.0: 132 files, 37.2MB
$
```

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
JMX enabled by default
Using config: /usr/local/etc/zookeeper/zoo.cfg
Starting zookeeper ... STARTED
$ kafka-server-start.sh /usr/local/etc/kafka/server.properties
[...]
[2017-02-09 20:48:22,485] INFO [Kafka Server 0], started (kafka.server.Kafka
Server)
```

Установка вручную

Аналогично установке вручную на операционной системе Windows, при установке Kafka на MacOS необходимо сначала установить JDK. Для получения нужной версии для MacOS можно воспользоваться той же страницей загрузки Oracle Java SE,

¹ См. предыдущую сноску. — *Примеч. пер.*

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_2.11-0.10.2.0`.

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"
$ echo $JAVA_HOME
/Library/Java/JavaVirtualMachines/jdk1.8.0._131.jdk/Contents/Home
$ /usr/local/kafka_2.11-0.10.2.0/bin/zookeeper-server-start.sh -daemon /usr/
local/kafka_2.11-0.10.2.0/config/zookeeper.properties
$ /usr/local/kafka_2.11-0.10.2.0/bin/kafka-server-start.sh /usr/local/etc/kafka/
server.properties
[2017-04-26 16:45:19,804] INFO KafkaConfig values:
[...]
[2017-04-26 16:45:20,697] INFO Kafka version : 0.10.2.0 (org.apache.kafka.com
mon.utils.AppInfoParser)
[2017-04-26 16:45:20,706] INFO Kafka commitId : 576d93a8dc0cf421
(org.apache.kafka.common.utils.AppInfoParser)
[2017-04-26 16:45:20,717] INFO [Kafka Server 0], started (kafka.server.Kafka
Server)
```