

Кафка в действии

Дилан Скотт
Виктор Гамов
Дейв Клейн



MANNING

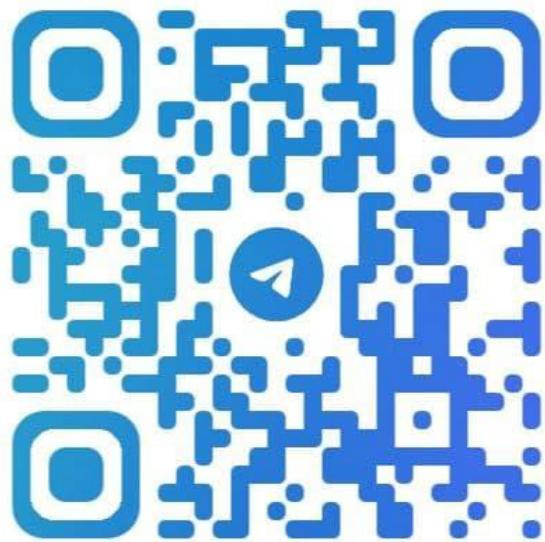


Дилан Скотт, Виктор Гамов, Дейв Клейн

Предисловие Юна Рао

Kafka в действии

Ещё больше книг в нашем телеграм канале:
<https://t.me/javalib>



@JAVALIB

Kafka в действии

ДИЛАН СКОТТ
ВИКТОР ГАМОВ
ДЕЙВ КЛЕЙН
ПРЕДИСЛОВИЕ ЮНА РАО



Москва, 2022

УДК 004.42

ББК 32.973

C44

Дилан Скотт, Виктор Гамов, Дейв Клейн

- C44** Kafka в действии / пер. с англ. А. Н. Киселева. – М.: ДМК Пресс, 2022. – 310 с.: ил.

ISBN 978-5-93700-118-4

Это практическое руководство показывает, как использовать распределенную потоковую платформу Apache Kafka для удовлетворения различных бизнес-требований. Рассказывается, как устроена Kafka и где она может пригодиться на практике; описываются характеристики проектов, в которых может пригодиться эта платформа. Рассматриваются основные ее компоненты – клиенты и кластер, представлены варианты улучшения работающего кластера.

Книга адресована разработчикам, желающим ознакомиться с идеей потоковой обработки данных. Для изучения примеров кода понадобятся базовые знания командной строки; желательно иметь навыки программирования на языке Java.

УДК 004.42

ББК 32.973

Original English language edition published by Manning Publications, USA.
Russian-language edition copyright (c) 2022 by DMK Press. All rights reserved.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN (анг.) 978-1-61729-523-2
ISBN (рус.) 978-5-93700-118-4

© 2022 by Manning Publications Co.
© Оформление, издание, перевод,
ДМК Пресс, 2022

Дилан: Я посвящаю эту работу Харпер, которой я так горжусь, и Ноэль, каждый день доставляющей радость нашей семье. Я также хотел бы посвятить эту книгу своим родителям, сестре и супруге, которые всегда оказывали мне всяческую поддержку.

Виктор: Эту работу я посвящаю своей супруге Марии за ее поддержку в процессе работы над этой книгой. Мне пришлось решать сложную задачу, выкраивая время то для того, то для другого. Без твоей поддержки у меня ничего бы не получилось. Я тебя люблю. Кроме того, хочу посвятить эту книгу (и выразить благодарность) моим детям, Эндрю и Майклу, за то, что они такие простодушные и прямолинейные. Когда люди спрашивают их, где работает папа, они отвечают: «Папа работает в Кафка».

Дейв: Я посвящаю эту книгу своей супруге Дебби и нашим детям Захарии, Эбигейл, Бенджамину, Сафе, Соломону, Ханне, Джоанне, Ребекке, Сюзанне, Ною, Самюэлю, Гидеону, Джошуа и Даниэлю. И наконец, все, что я делаю, я делаю во славу Творца и Спасителя нашего, Иисуса Христа.

Краткое оглавление

1	■ <i>Введение в Kafka</i>	26
2	■ <i>Знакомство с Kafka</i>	44
3	■ <i>Разработка проекта на основе Kafka</i>	75
4	■ <i>Производители: источники данных</i>	103
5	■ <i>Потребители: извлечение данных</i>	127
6	■ <i>Брокеры</i>	155
7	■ <i>Темы и разделы</i>	176
8	■ <i>Kafka как хранилище</i>	193
9	■ <i>Управление: инструменты и журналы</i>	211
10	■ <i>Защита Kafka</i>	236
11	■ <i>Реестр схем</i>	256
12	■ <i>Потоковая обработка с помощью Kafka Streams и ksqlDB</i>	270

Содержание

<i>Предисловие от издательства</i>	13
<i>Предисловие</i>	14
<i>Вступление.....</i>	15
<i>Благодарности</i>	16
<i>Об этой книге</i>	18
<i>Об авторах.....</i>	22
<i>Об иллюстрации на обложке</i>	23
ЧАСТЬ I. НАЧАЛО	25
1 Введение в Kafka	26
1.1. Что такое Kafka?	27
1.2. Использование Kafka.....	32
1.2.1. Kafka – разработчикам	32
1.2.2. Как преподнести Kafka вашему руководству	34
1.3. Мифы о Kafka	35
1.3.1. Kafka работает только с Hadoop®.....	35
1.3.2. Kafka ничем не отличается от других брокеров сообщений	36
1.4. Kafka в реальном мире	37
1.4.1. Ранние примеры.....	37
1.4.2. Более поздние примеры	39
1.4.3. Когда Kafka может быть неприменима.....	40
1.5. Онлайн-ресурсы	41
Итоги	42
Ссылки	42
2 Знакомство с Kafka	44
2.1. Отправка и прием сообщения	45
2.2. Что такое брокер?.....	46
2.3. Экскурсия по Kafka	51
2.3.1. Производители и потребители	51
2.3.2. Темы	55

2.3.3. ZooKeeper	56
2.3.4. Высокоуровневая архитектура Kafka.....	58
2.3.5. Журнал коммитов	59
2.4. Различные пакеты исходного кода, и что они делают	60
2.4.1. Kafka Streams	60
2.4.2. Kafka Connect.....	62
2.4.3. Пакет AdminClient	62
2.4.4. ksqlDB.....	63
2.5. Клиенты Confluent	63
2.6. Потоковая обработка и терминология	67
2.6.1. Потоковая обработка	69
2.6.2. Что означает семантика «точно один раз»	69
Итоги	70
Ссылки	70

ЧАСТЬ II. ПРАКТИЧЕСКОЕ ПРИМЕНЕНИЕ KAFKA... 73

3

<i>Разработка проекта на основе Kafka.....</i>	75
3.1. Разработка проекта на основе Kafka.....	76
3.1.1. Использование существующей архитектуры данных	76
3.1.2. Первый шаг	76
3.1.3. Встроенные возможности	77
3.1.4. Данные для наших накладных	80
3.2. События датчиков.....	82
3.2.1. Имеющиеся проблемы	82
3.2.2. Почему Kafka – правильный выбор.....	85
3.2.3. Первые мысли об архитектуре	86
3.2.4. Требования к пользовательским данным	88
3.2.5. Общий план с учетом поставленных вопросов.....	88
3.2.6. Обзор и оценка плана.....	92
3.3. Формат представления данных	93
3.3.1. План для данных	93
3.3.2. Настройка зависимостей	95
Итоги	101
Ссылки	101

4

<i>Производители: источники данных</i>	103
4.1. Пример	104
4.1.1. Примечания в отношении производителя	107
4.2. Параметры производителя	108
4.2.1. Настройка списка брокеров	109

4.2.2. Быстрее или надежнее?.....	110
4.2.3. Отметки времени	113
4.3. Генерирование кода с учетом наших требований.....	115
4.3.1. Версии клиентов и брокеров	124
Итоги	125
Ссылки	125

5 *Потребители: извлечение данных..... 127*

5.1. Пример	128
5.1.1. Параметры потребителя.....	129
5.1.2. Наши координаты в потоке событий	133
5.2. Как взаимодействуют потребители	137
5.3. Трассировка	138
5.3.1. Координатор группы.....	139
5.3.2. Стратегия назначения разделов	141
5.4. Маркировка местонахождения.....	142
5.5. Чтение из сжатой темы.....	145
5.6. Реализация в коде наших заводских требований	145
5.6.1. Варианты чтения	146
5.6.2. Требования.....	148
Итоги	151
Ссылки	151

6 *Брокеры* 155

6.1. Знакомство с брокерами.....	155
6.2. Роль ZooKeeper.....	156
6.3. Конфигурационные параметры брокеров	158
6.3.1. Другие журналы Kafka: журналы приложений.....	160
6.3.2. Журнал сервера.....	160
6.3.3. Управление состоянием	160
6.4. Ведущие реплики разделов и их роль	162
6.4.1. Потеря данных	164
6.5. Взгляд внутрь Kafka.....	165
6.5.1. Обслуживание кластера	167
6.5.2. Добавление брокера	167
6.5.3. Обновление кластера	167
6.5.4. Обновление клиентов	168
6.5.5. Резервные копии.....	168
6.6. Примечание о системах с сохранением состояния.....	169
6.7. Упражнение	171
Итоги	172
Ссылки	173

7	Темы и разделы.....	176
7.1.	Темы	176
7.1.1.	Параметры создания темы	180
7.1.2.	Коэффициенты репликации.....	182
7.2.	Разделы	183
7.2.1.	Размещение раздела.....	183
7.2.2.	Просмотр журналов.....	184
7.3.	Тестирование с помощью EmbeddedKafkaCluster.....	186
7.3.1.	Использование Kafka Testcontainers	188
7.4.	Сжатые темы.....	188
	Итоги	191
	Ссылки	191
8	Kafka как хранилище	193
8.1.	Как долго можно хранить данные	194
8.2.	Перемещение данных	195
8.2.1.	Сохранение исходных событий	195
8.2.2.	Отказ от пакетного мышления	196
8.3.	Инструменты	196
8.3.1.	Apache Flume	197
8.3.2.	Red Hat® Debezium™	199
8.3.3.	Secor	200
8.3.4.	Пример сохранения данных	201
8.4.	Возврат данных в Kafka.....	201
8.4.1.	Многоуровневое хранилище.....	203
8.5.	Архитектуры с использованием Kafka.....	203
8.5.1.	Лямбда-архитектура.....	203
8.5.2.	Каппа-архитектура	205
8.6.	Окружения с несколькими кластерами	206
8.6.1.	Масштабирование путем добавления кластеров	206
8.7.	Варианты хранения в облаке и в контейнерах	207
8.7.1.	Кластеры Kubernetes	207
	Итоги	208
	Ссылки	208
9	Управление: инструменты и журналы.....	211
9.1.	Клиенты администрирования	212
9.1.1.	Решение задач администрирования в коде с помощью AdminClient	212
9.1.2.	kcat	214
9.1.3.	Confluent REST Proxy API	216

9.2. Запуск Kafka как службы systemd	217
9.3. Журналы.....	218
9.3.1. Журналы приложений Kafka.....	219
9.3.2. Журналы ZooKeeper	220
9.4. Брандмауэры.....	221
9.4.1. Публикуемые слушатели	221
9.5. Метрики.....	222
9.5.1. Консоль JMX	222
9.6. Способы трассировки	225
9.6.1. Логика на стороне производителя.....	226
9.6.2. Логика на стороне потребителя.....	228
9.6.3. Переопределение клиентов	230
9.7. Общие инструменты мониторинга.....	231
Итоги	232
Ссылки	232

ЧАСТЬ III. ДОПОЛНИТЕЛЬНЫЕ ВОЗМОЖНОСТИ ... 235

10 Защита Kafka 236

10.1. Основы безопасности	238
10.1.1. Шифрование с помощью SSL.....	239
10.1.2. Настройка соединений SSL между брокерами и клиентами	240
10.1.3. Настройка соединений SSL между брокерами	244
10.2. Kerberos и Simple Authentication and Security Layer (SASL)	244
10.3. Авторизация в Kafka	245
10.3.1. Списки управления доступом.....	246
10.3.2. Управление доступом на основе ролей.....	247
10.4. ZooKeeper.....	248
10.4.1. Настройка Kerberos	248
10.5. Квоты	249
10.5.1. Ограничение пропускной способности сети	250
10.5.2. Ограничение частоты запросов	252
10.6. Данные в состоянии покоя.....	252
10.6.1. Управляемые варианты.....	253
Итоги	253
Ссылки	254

11 Реестр схем 256

11.1. Предлагаемая модель зрелости Kafka.....	257
11.1.1. Уровень 0	257

11.1.2. Уровень 1	258
11.1.3. Уровень 2	259
11.1.4. Уровень 3	259
11.2. Реестр схем	260
11.2.1. Установка Confluent Schema Registry	260
11.2.2. Конфигурация реестра	261
11.3. Компоненты реестра схем.....	262
11.3.1. REST API	262
11.3.2. Клиентская библиотека	263
11.4. Правила совместимости	265
11.4.1. Проверка изменений схемы.....	266
11.5. Альтернатива реестру схем	267
Итоги	268
Ссылки	268
12 <i>Потоковая обработка с помощью Kafka Streams и ksqlDB</i>	270
12.1. Kafka Streams.....	271
12.1.1. KStreams API DSL	273
12.1.2. KTable API.....	277
12.1.3. GlobalKTable API	278
12.1.4. Processor API	279
12.1.5. Настройка Kafka Streams.....	281
12.2. ksqlDB: база данных потоковой передачи событий	282
12.2.1. Запросы	284
12.2.2. Локальная разработка	284
12.2.3. Архитектура ksqlDB	286
12.3. Куда пойти дальше	287
12.3.1. Предложения по улучшению Kafka (KIP)	287
12.3.2. Проекты Kafka, которые вы можете исследовать ...	288
12.3.3. Каналы сообщества Slack	288
Итоги	288
Ссылки	289
Приложение А. Установка.....	290
Приложение В. Пример клиента	299
Предметный указатель	304

Предисловие от издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге, – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в основном тексте или программном коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательство «ДМК Пресс» очень серьезно относится к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Предисловие

Начиная с первого выпуска, вышедшего в 2011 году, технологии Apache Kafka® помогли создать новую категорию систем передачи данных, и теперь они являются основой бесчисленного множества современных приложений, управляемых событиями. В своей книге «Kafka в действии» Дилан Скотт (Dylan Scott), Виктор Гамов (Viktor Gamov) и Дэйв Клейн (Dave Klein) делятся навыками проектирования и реализации приложений на основе событий, реализованных с использованием Apache Kafka. Авторы имеют богатый опыт работы с Kafka в реальном мире, что выделяет эту книгу среди других.

Давайте на минутку зададимся вопросом: «Зачем вообще нужна платформа Kafka?» Исторически сложилось так, что большинство приложений были основаны на системах хранения данных. Когда в мире происходили какие-то интересные события, они немедленно сохранялись в этих системах, но реакция на эти события происходила позже – либо когда пользователь явно запрашивал информацию, либо в ходе выполнения некоторых заданий пакетной обработки.

В системах передачи данных приложения строятся путем предварительного определения того, что они должны делать при появлении новых событий. Когда случаются новые события, приложения автоматически реагируют на них практически мгновенно. Такие приложения, управляемые событиями, привлекательны тем, что позволяют предприятиям гораздо быстрее извлекать новую информацию из своих данных. Однако переход к приложениям, управляемым событиями, требует изменения мышления, что не всегда легко. Эта книга предлагает исчерпывающее описание событийно-ориентированного мышления, а также реалистичные практические примеры, которые вы сможете опробовать.

«Kafka в действии» объясняет, как работает Kafka, и особое внимание уделяет созданию комплексных приложений, управляемых событиями, на основе Kafka. Здесь вы познакомитесь с компонентами, необходимыми для создания простого приложения Kafka, а также узнаете, как создавать сложные приложения с использованием таких библиотек, как Kafka Streams и ksqlDB. Также в этой книге рассказывается, как после создания приложения развернуть его в промышленном окружении, и освещаются такие ключевые темы, как мониторинг и безопасность.

Я надеюсь, что вам понравится эта книга так же, как мне. Удачной передачи событий!

– Юн Рао (Jun Rao), соучредитель Confluent

Вступление

Один из вопросов, который часто задают нам, когда мы рассказываем о работе над технической книгой: почему был выбран именно формат печатной книги? Дилан, например, всегда предпочитал узнавать что-то новое, читая книги. Другой фактор – ностальгия, навеваемая воспоминаниями о первой технической книге по программированию, которую он прочитал, «Elements of Programming with Perl» Эндрю Л. Джонсона (Andrew L. Johnson), выпущенной издательством Manning в 2000 году. Эта книга особенно запомнилась ему, и он до сих пор вспоминает, насколько приятно было читать ее страницы. Мы надеемся доставить такое же удовольствие своим читателям, описывая Apache Kafka.

Предвкушение познания чего-то нового почувствовал каждый из нас, когда мы впервые начали работать с Kafka. На наш взгляд, Kafka существенно отличается от любых других брокеров сообщений или шин служб предприятия (Enterprise Service Bus, ESB), которые нам доводилось использовать раньше. Быстрота разработки производителей и потребителей сообщений, возможность повторной обработки данных и скорость, с которой независимые потребители перемещаются без удаления данных из других потребительских приложений, позволяют решать проблемы, встречавшиеся в прошлом, и впечатлили нас больше всего, когда мы стали изучать возможность применения Kafka.

Мы видим, что Kafka меняет стандарты для платформ данных; она способна помочь перенести пакетные рабочие процессы и рабочие процессы извлечения, преобразования и загрузки (Extract, Transform, Load, ETL) ближе во времени к потокам данных. Поскольку эта платформа отходит от архитектур обработки данных, использовавшихся раньше и известных многим корпоративным пользователям, мы хотели помочь пользователям, не знакомым с Kafka, научиться работать с производителями и потребителями Kafka, а также решать базовые задачи разработки и администрирования Kafka. Мы надеемся, что к концу этой книги вы почувствуете в себе готовность углубиться в исследование более сложных тем Kafka, таких как мониторинг кластеров, создание метрик и межсайтовая репликация данных.

Всегда помните, что эта книга запечатлела момент, как Кафка выглядит сегодня. Она почти наверняка будет меняться и, надеюсь, станет еще лучше к тому времени, когда вы будете читать эту работу. Мы верим, что эта книга направит вас на увлекательный путь изучения основ Apache Kafka.

Благодарности

Дилан. Прежде всего я хотел бы поблагодарить мою семью. Спасибо вам! Я никогда не устану благодарить за поддержку и любовь, которые вижу каждый день. Я люблю вас всех! Дэн и Дебби, я высоко ценю, что вы всегда были моими самыми преданными сторонниками и фанатами. Сара, Харпер и Ноэль, я не смогу на словах передать всю любовь и гордость, которую испытываю ко всем вам, и благодарность за поддержку, которую вы мне оказываете. Спасибо семье DG, что всегда были рядом со мной. Спасибо и вам, JC.

Также отдельное спасибо Виктору Гамову и Дейву Клейну, что были соавторами этой книги! В продвижении этого проекта мне также помогали мои друзья и коллеги по работе, которых я должен упомянуть: команда Team Serenity (Бекки Кэмпбелл (Becky Campbell), Адам Доман (Adam Doman), Джейсон Фер (Jason Fehr) и Дэн Рассел (Dan Russell)), Роберт Абейта (Robert Abeyta) и Джереми Кастил (Jeremy Castle). Спасибо также, Джабулани Симплезио Чибайя (Jabulani Simplizio Chibaya), не только за рецензию, но и за добрые слова.

Виктор. Я хотел бы сказать огромное спасибо моей супруге и поблагодарить ее за поддержку. Спасибо также членам команды по связям с разработчиками и сообщества Confluent: Але Мюррей (Ale Murray), Еве Байзек (Yeva Byzek), Робину Моффатт (Robin Moffatt) и Тому Берглунду (Tim Berglund). Вы все так много делаете для сообщества Apache Kafka!

Дейв. Я хотел бы поблагодарить Дилана и Виктора за то, что позволили мне поучаствовать в этом захватывающем путешествии.

Все вместе мы благодарим нашего редактора в Manning – Тони Арритола (Toni Arritola), чей опыт и наставничество помогли сделать эту книгу реальностью. Мы также выражаем благодарность Кристен Уоттерсон (Kristen Watterson), которая была нашим первым редактором до того, как на смену ей пришел Тони, нашим техническим редакторам Рафаэлю Вильеле (Raphael Villela), Ники Бакнер (Nickie Buckner), Фелипе Эстебану Вильдосо Кастильо (Felipe Esteban Vildoso Castillo), Маюру Патилу (Mayur Patil), Валентину Креттазу (Valentin Crettaz) и Уильяму Руденмальму (William Rudenmalm). Мы благодарим Чака Ларсона (Chuck Larson) за огромную помощь с графикой и Суманта Тамбе (Sumant Tambe) за техническую корректировку кода.

Сотрудники издательства Manning оказывали всемерную помощь по самым разным вопросам, от производства до продвижения, – это

очень продуктивная команда. Несмотря на все правки и обзоры, в текст книги и в исходный код могли просочиться опечатки и неточности (по крайней мере, мы никогда не видели книги без опечаток!), но эта команда, безусловно, помогла свести эти ошибки к минимуму.

Спасибо также Натану Марцу (Nathan Marz), Майклу Ноллу (Michael Noll), Джанакираму М. С. В. (Janakiram MSV), Биллу Беджеку (Bill Bejeck), Гуннару Морлингу (Gunnar Morling), Робину Моффатту (Robin Moffatt), Генри Каю (Henry Cai), Мартину Фаулеру (Martin Fowler), Александру Дину (Alexander Dean), Валентину Креттазу (Valentin Crettaz) и Ани Ли (Anyi Li). Вы очень помогли нам, рассказывая о своей работе и делясь своими замечательными предложениями и отзывами.

Юн Рао, для нас большая честь, что вы нашли время написать предисловие к этой книге. Большое спасибо!

Мы очень признательны всему сообществу Apache Kafka (включая, конечно же, Джая Крепса (Jay Kreps), Неху Наркхеде (Neha Narkhede) и Юна Рао (Jun Rao)) и команде Confluent, продвигающей Kafka вперед и давшей разрешение на использование материалов, которые помогли наполнить эту книгу. По крайней мере, мы очень надеемся, что эта книга побудит разработчиков взглянуть на Kafka.

Наконец, большое спасибо всем рецензентам, это: Брайс Дарлинг (Bryce Darling), Кристофер Бейли (Christopher Bailey), Цицеро Зандона (Cicero Zandona), Конор Редмонд (Conor Redmond), Дэн Рассел (Dan Russell), Дэвид Криф (David Krief), Фелипе Эстебан Вильдосо Кастильо (Felipe Esteban Vildoso Castillo), Финн Ньюик (Finn Newick), Флорин-Габриэль Барбучану (Florin-Gabriel Barbuceanu), Грегор Райман (Gregor Rayman), Джейсон Фер (Jason Fehr), Хавьер Колладо Кабеза (Javier Collado Cabeza), Джон Мур (Jon Moore), Хорхе Эстебан Квилкате Отоя (Jorge Esteban Quilcate Otoya), Джошуа Хорвиц (Joshua Horwitz), Мадханмохан Савадамуту (Madhanmohan Savadamuthu), Мишель Мауро (Michele Mauro), Питер Перлепес (Peter Perlepes), Роман Левченко (Roman Levchenko), Санкет Найк (Sanket Naik), Шобха Айер (Shobha Iyer), Сумант Тамбе (Sumant Tambe), Витон Витанис (Viton Vitanis) и Уильям Руденмальм (William Rudenmalm) – ваши отзывы и предложения помогли сделать эту книгу лучше.

Мы могли кого-то упустить, и если это действительно так, то просим вас простить нас за нашу ошибку. Мы ценим вас.

Об этой книге

Мы писали «Kafka в действии» как практическое руководство по началу работы с Apache Kafka. В этой книге читатели встретят небольшие примеры, объясняющие некоторые параметры и настройки, которые можно использовать для изменения поведения Kafka в соответствии с конкретными вариантами применения. Ядро Kafka специально создавалось как настраиваемое в широких пределах и легко интегрирующееся с другими продуктами, такими как Kafka Streams и ksqlDB. Мы надеемся показать, как можно использовать платформу Kafka для удовлетворения различных бизнес-требований, чтобы вы освоились с ней к концу этой книги и знали, с чего начать решение ваших задач.

Кому адресована эта книга

«Kafka в действии» адресована разработчикам, желающим познакомиться с идеей потоковой обработки данных. От читателя не требуется обладать какими-либо знаниями о Kafka, но базовые знания командной строки и умение ею пользоваться не будут лишними. В Kafka есть несколько мощных инструментов командной строки, которые мы используем, и пользователь должен уметь по крайней мере вводить команды в командной строке.

Также могут пригодиться некоторые навыки программирования на языке Java и способность распознавать идеи программирования на любом языке. Эти навыки помогут понять представленные примеры кода, которые реализованы в основном в стиле Java 11 (а также Java 8). Кроме того, хотя это и необязательно, будет полезно общее понимание архитектуры распределенных приложений. Чем больше пользователь знает о репликациях и сбоях, тем проще ему будет понять, например, как Kafka использует реплики.

Организация книги

Эта книга состоит из трех частей, разбитых на 12 глав. Часть I представляет ментальную модель Kafka и рассказывает, где может пригодиться Kafka в реальном мире:

- глава 1 содержит общее введение в Kafka, опровергает некоторые мифы и описывает примеры использования в реальных условиях;
- глава 2 исследует архитектуру Kafka в общих чертах и вводит важную терминологию.

Часть II переходит к основным компонентам Kafka – клиентам и самому кластеру:

- глава 3 рассматривает характеристики проектов, в которых с успехом можно было бы использовать Kafka, и описывает некоторые подходы к разработке новых проектов. Здесь также обсуждается необходимость схем, на которые следует обратить особое внимание на этапе создания проекта на основе Kafka, но не позже;
- глава 4 иллюстрирует некоторые детали создания клиента-производителя и параметры управления передачей ваших данных в кластер Kafka;
- глава 5 меняет фокус главы 4 и иллюстрирует приемы получения данных из Kafka с помощью клиента-потребителя. Здесь будет представлена идея смещений и повторной обработки данных, обусловленная возможностью хранения сообщений;
- глава 6 рассматривает роль брокеров в кластере и как они взаимодействуют с вашими клиентами. Здесь мы исследуем различные компоненты, такие как контроллер и реплика;
- глава 7 обсуждает понятия тем и разделов, включая возможность компактификации тем и особенности хранения разделов;
- глава 8 описывает инструменты и архитектуры для обработки данных, которые может потребоваться сохранить или обработать повторно. Необходимость хранения данных в течение нескольких месяцев или лет может привести к тому, что вам придется подумать о вариантах хранения за пределами кластера;
- глава 9 завершает часть II обзором журналов, метрик и административных функций, помогающих поддерживать работоспособность кластера.

В части III мы перейдем от знакомства с основными компонентами частей Kafka к изучению вариантов улучшения работающего кластера:

- глава 10 представляет варианты усиления защиты кластера Kafka с помощью SSL, списков управления доступом ACL и квот;
- глава 11 посвящена реестру схем Schema Registry и особенностям его использования для работы с данными и сохранения совместимости с предыдущими и будущими версиями наборов данных. Считается, что эта возможность предназначена для приложений корпоративного уровня, однако она с успехом может использоваться для обслуживания любых данных, изменяющихся с течением времени;

- глава 12, последняя, посвящена знакомству с Kafka Streams и ksqlDB. Эти продукты относятся к более высоким уровням абстракции и основаны на ядре, которому посвящена вся вторая часть книги. Kafka Streams и ksqlDB – достаточно обширные темы, поэтому в нашем введении мы представим ровно столько информации, сколько действительно необходимо, чтобы приступить к самостоятельному изучению этих продуктов.

О примерах программного кода

Эта книга содержит множество примеров исходного кода как в пронумерованных листингах, так и в обычном тексте. В обоих случаях исходный код оформлен моноширинным шрифтом, чтобы визуально отделять его от обычного текста. Во многих случаях исходный код был дополнительно отформатирован; мы добавили разрывы строк и изменили отступы, чтобы уместить примеры по ширине книжной страницы. В некоторых случаях даже этого оказалось недостаточно, и в каких-то листингах вы можете встретить символы ➔, обозначающие продолжение строк. Многие листинги сопровождаются дополнительными комментариями, поясняющими важные понятия.

Наконец, следует отметить, что многие примеры кода не предназначены для выполнения в форме самостоятельных программ, – это выдержки, иллюстрирующие наиболее важные стороны обсуждаемого. Все примеры из книги и сопровождающий их исходный код в полной форме вы найдете на GitHub по адресу <https://github.com/Kafka-In-Action-Book/Kafka-In-Action-Source-Code> и на сайте издателя www.manning.com/books/kafka-in-action. Также выполняемые фрагменты кода можно получить в онлайн-версии этой книги по адресу <https://livebook.manning.com/book/kafka-in-action>.

Живое обсуждение книги

Приобретая книгу «Kafka в действии», вы получаете бесплатный доступ к частному веб-форуму, организованному издательством Manning Publications, где можно оставлять комментарии о книге, задавать технические вопросы, а также получать помощь от автора и других пользователей. Чтобы иметь доступ к форуму и зарегистрироваться на нем, откройте в веб-браузере страницу <https://livebook.manning.com/#!/book/kafka-in-action/discussion>. Узнать больше о форумах Manning и познакомиться с правилами поведения можно по адресу <https://livebook.manning.com/#!/discussion>.

Издательство Manning обязуется предоставить своим читателям место встречи, где может состояться содержательный диалог между отдельными читателями и между читателями и автором. Но со стороны авторов отсутствуют какие-либо обязательства уделять фо-

руму какое-то определенное внимание – их присутствие на форуме остается добровольным (и неоплачиваемым). Мы предлагаем задавать авторам стимулирующие вопросы, чтобы их интерес не угасал! Форум и архив с предыдущими обсуждениями остается доступным на сайте издательства, пока книга продолжает издаваться.

Другие онлайн-ресурсы

Все изменения, происходящие в Kafka с течением времени, неизменно отражаются перечисленными ниже ресурсами. В большинстве случаев на этих сайтах можно найти документацию с описанием прошлых версий:

- документация по Apache Kafka – <http://kafka.apache.org/documentation.html>;
- документация Confluent – <https://docs.confluent.io/current>;
- портал разработчиков Confluent – <https://developer.confluent.io>.

Об авторах

Дилан Скотт (Dylan Scott) – разработчик программного обеспечения с более чем десятилетним опытом программирования на Java и Perl. После знакомства с системой обмена сообщениями Kafka как средством передачи больших объемов данных Дилан начал погружаться в мир Kafka и технологий потоковой обработки. Он имеет значительный опыт использования таких технологий и очередей, как Mule, RabbitMQ, MQSeries и Kafka.

Дилан обладает различными сертификатами, свидетельствующими об опыте работы в отрасли: PMP, ITIL, CSM, Sun Java SE 1.6, Oracle Web EE 6, Neo4j и Jenkins Engineer.

Виктор Гамов (Viktor Gamov) – пропагандист передовых практик разработки в Confluent, компании, разрабатывающей платформу потоковой передачи событий на основе Apache Kafka. За свою долгую карьеру Виктор накопил богатый опыт в сфере разработки архитектур корпоративных приложений с использованием технологий с открытым исходным кодом. Ему нравится помогать архитекторам и разработчикам проектировать и создавать масштабируемые и высокодоступные распределенные системы с малым временем реакции.

Виктор является профессиональным докладчиком на конференциях по темам распределенных систем, потоковой передачи данных, JVM и DevOps, а также регулярно посещает мероприятия, такие как JavaOne, Devoxx, OSCON, QCon и др. Является соавтором книги «Enterprise Web Development» (O'Reilly Media, Inc.).

Следуйте за Виктором в Твиттере [@gamussa](#), где он пишет о спортивной жизни, вкусной и здоровой пище, открытом исходном коде и, конечно же, о Kafka!

Дейв Клейн (Dave Klein) имеет 28-летний опыт работы разработчиком, архитектором, руководителем проекта, автором, преподавателем, организатором конференций и семейного учителя, пока недавно не получил работу своей мечты пропагандиста передовых практик разработки в Confluent. Дейв восхищается удивительным миром потоковой передачи событий Apache Kafka и всеми силами старается помочь другим исследовать его.

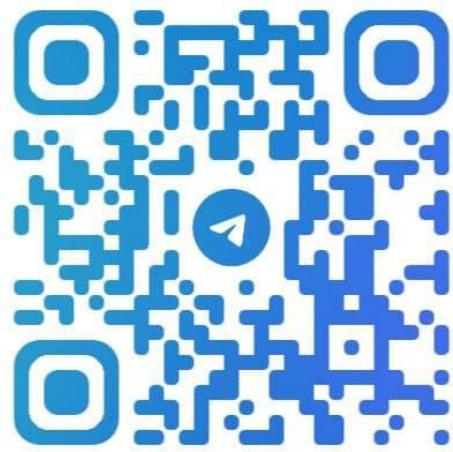
Об иллюстрации на обложке

На обложке «Kafka в действии» изображена иллюстрация, подписанная как «Femme du Madagascar» (Мадагаскарская женщина), из четырехтомного сборника изображений национальной одежды Сильвена Марешаля (Sylvain Maréchal), изданного во Франции в XIX веке. Каждая иллюстрация тщательно прорисована и раскрашена вручную. Богатое разнообразие коллекции Марешаля напоминает нам о том, насколько далекими друг от друга были культурные традиции городов и регионов мира всего 200 лет назад. Изолированные друг от друга люди говорили на разных диалектах и языках. Встретив человека на улице, по его одежду было легко определить, где он живет, чем зарабатывает на жизнь или какое положение в обществе занимает.

С тех пор стиль одежды сильно изменился и исчезло разнообразие, характеризующее различные области и страны. В настоящее время трудно отличить по одежде даже жителей разных континентов, не говоря уже об обитателях разных городов, регионов или стран. Мы заменили культурное разнообразие более разносторонней личной жизнью и, безусловно, не менее интересной интеллектуальной жизнью.

Мы в издательстве Manning славим изобретательность, предпринимчивость и радость компьютерного бизнеса обложками книг, изображающими богатство региональных различий двухвековой давности, оживших благодаря иллюстрациям Марешаля.

Ещё больше книг в нашем телеграм канале:
<https://t.me/javalib>



@JAVALIB

Часть I

Начало

В первой части этой книги мы познакомим вас с Apache Kafka и начнем рассматривать реальные случаи использования Kafka:

- в главе 1 подробно опишем преимущества Kafka и развеем некоторые мифы, которые вы, возможно, слышали о Kafka в связи с Hadoop;
- в главе 2 мы сосредоточим наше внимание на высокоуровневой архитектуре Kafka, а также на некоторых других компонентах, составляющих экосистему Kafka: Kafka Streams, Connect и ksqlDB.

К концу этой части вы будете готовы начать принимать и отправлять сообщения в Kafka. Надеемся, что вы также усвоите некоторые ключевые термины.

1

Введение в Kafka

Эта глава охватывает следующие темы:

- преимущества Kafka;
- распространенные мифы о больших данных и системах сообщений;
- реальные примеры использования, когда технологии Kafka помогли улучшить обмен сообщениями, потоковую передачу и обработку данных IoT.

Многие разработчики постоянно сталкиваются с миром, наполненным данными, то и дело льющимися со всех сторон, и нередко оказываются перед фактом, когда устаревшие системы тормозят движение вперед. Одним из основных элементов новых инфраструктур данных, занявших лидирующие позиции в ИТ-ландшафте, является Apache Kafka^{®1}. Kafka меняет стандарты платформ данных. Она подталкивает к переходу от извлечения, преобразования и загрузки (Extract, Transform, Load, ETL) данных с использованием пакетных процессов (которые обычно запускаются в одно и то же время) к потокам данных, действующим практически в режиме реального времени [1]. Пакетная обработка, которая когда-то была типичной рабочей лошадкой в сфере обработки корпоративных данных, является, пожалуй, не тем, к чему захочется вернуться по-

¹ Apache, Apache Kafka и Kafka являются товарными знаками Apache Software Foundation.

сле знакомства с мощным набором возможностей, предлагаемых Kafka. На самом деле вы просто не сможете справиться с растущим снежным комом поступающих данных, если не возьмете на вооружение что-то новое.

С таким большим количеством данных системы будут постоянно перегружены работой. Устаревшие системы могут не справиться с обработкой данных в ночной период и продолжат выполнение пакетных заданий на следующий день. Чтобы не отставать от этого постоянного потока данных, обработка информации должна производиться по мере ее поступления – это единственная возможность обеспечить актуальное состояние системы.

Kafka следует многим новейшим и наиболее практическим тенденциям в современном мире информационных технологий и упрощает повседневную работу. Например, Kafka уже нашла свое место в архитектуре микросервисов и интернете вещей (Internet of Things, IoT). Как технология, принятая на вооружение большим числом компаний, Kafka предназначена не только для фанатов или охотников за альфа-версиями. Давайте начнем знакомство с Kafka с общего обзора этой платформы и некоторых особенностей современных потоковых платформ.

1.1. Что такое Kafka?

На сайте проекта Apache Kafka (<http://kafka.apache.org/intro>) Kafka определяется как распределенная потоковая платформа, предлагающая три основные возможности:

- чтение сообщений из очереди и запись их в очередь;
- надежное хранение сообщений;
- обработку потоков данных по мере их появления [2].

Читателям, не сталкивающимся в своей практике с очередями или брокерами сообщений, может понадобиться помочь, когда мы начнем обсуждать общее назначение и принцип действия такой системы. В общем случае Kafka можно рассматривать как ИТ-эквивалент ресивера (приемника), встроенного в домашний кинотеатр. На рис. 1.1 показан поток данных между ресивером и конечными потребителями.

Как можно видеть на рис. 1.1, цифровые спутниковые, кабельные и Blu-ray™-проигрыватели могут подключаться к центральному приемнику. Эти отдельные компоненты можно рассматривать как источники данных, использующие известный им формат. Во время воспроизведения фильма или компакт-диска они порождают непрерывный поток данных. Ресивер обрабатывает этот поток и преобразует его в формат, понятный внешним устройствам, подключенным к другому концу (ресивер отправляет видеопоток

на телевизор, а звук – на декодер и на динамики). Но какое отношение это имеет к Kafka? Давайте посмотрим на те же потоки данных с точки зрения Kafka (рис. 1.2).

Kafka служит интерфейсом, обеспечивающим возможность взаимодействий клиентов с другими системами. Одни клиенты, которые называются *производителями*, отправляют потоки данных брокерам Kafka. Брокеры выполняют ту же функцию, что и ресивер на рис. 1.1. Другие клиенты Kafka называются *потребителями* – они могут читать данные из брокеров и обрабатывать их. Одни и те же данные может получать не один потребитель. Производители и потребители полностью отделены друг от друга, что позволяет каждому клиенту работать независимо. Подробнее о том, как это делается, вы узнаете в последующих главах.

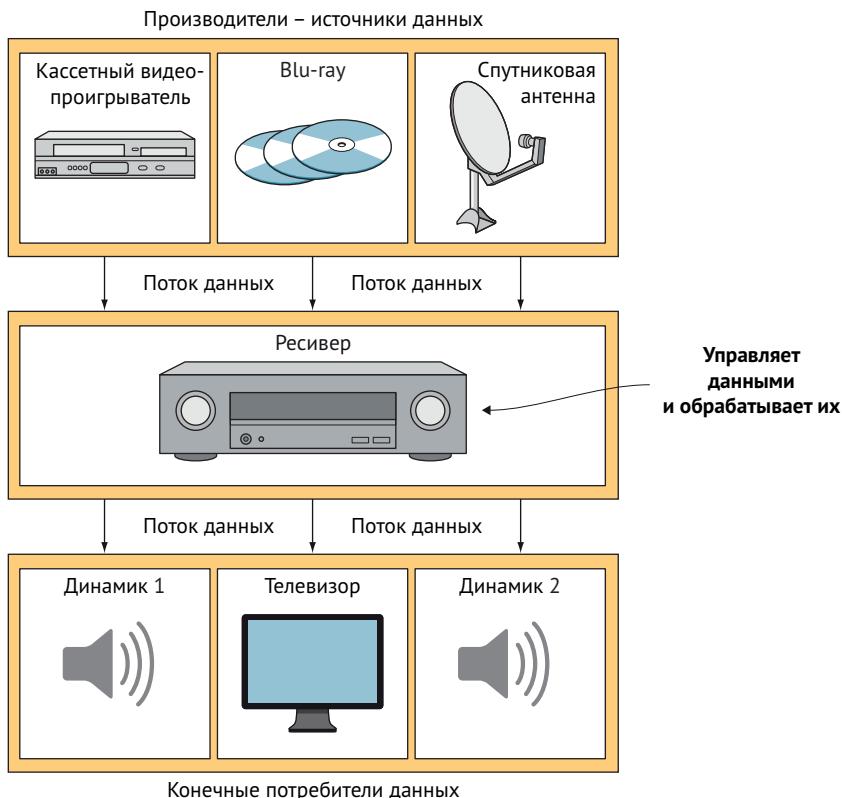


Рис. 1.1. Производители, потребители и потоки данных

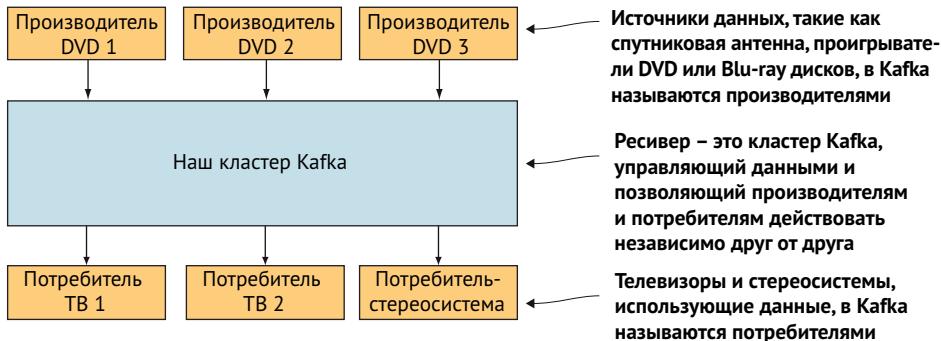


Рис. 1.2. Место Kafka в потоке данных между производителями и потребителями

Как и другие платформы обмена сообщениями, Kafka действует (если говорить упрощенно) как посредник, передавая данные, поступающие в систему (от производителей) и исходящие из системы (к потребителям или конечным пользователям). Такое отделение производителей и потребителей друг от друга позволяет обеспечить минимальную взаимозависимость (слабую связанность) между ними. Производитель может отправить любое сообщение, даже не имея ни малейшего представления о том, ожидает ли эти сообщения хоть кто-нибудь. Кроме того, Kafka поддерживает разные способы доставки сообщений, соответствующие разным бизнес-потребностям. Доставка сообщений в Kafka может осуществляться как минимум тремя способами [3]:

- *не менее одного раза* (at-least-once) – сообщение будет отправляться потребителям до тех пор, пока те не подтвердят его получение;
- *не более одного раза* (at-most-once) – сообщение отправляется только один раз и в случае сбоя не отправляется повторно;
- *точно один раз* (exactly-once) – потребитель гарантированно получит сообщение ровно один раз.

Давайте разберемся, что означают эти варианты обмена сообщениями. Рассмотрим первую семантику «не менее одного раза» (рис. 1.3). Kafka можно настроить так, чтобы она позволяла производителям отправлять одно и то же сообщение многократно и передавала их брокерам. Если производитель не получил подтверждения передачи сообщения брокеру, то он может отправить сообщение повторно [3]. В случаях, когда потеря сообщений недопустима, скажем, сообщений об оплате счета, эта семантика может потребовать дополнительной фильтрации на стороне потребителя, зато это один из самых надежных способов доставки.



Рис. 1.3. Поток сообщений с использованием семантики «не менее одного раза»

При использовании семантики «не более одного раза» (рис. 1.4) производитель отправляет сообщение только один раз и никогда не повторяет попытку. В случае сбоя производитель движется дальше, не пытаясь отправить сообщение [3]. Но разве потеря сообщений может быть допустима? Представьте популярный веб-сайт, отслеживающий количество просмотров страниц. Для такого сайта вполне возможно потерять несколько событий просмотра из миллионов, которые он обрабатывает каждый день. Увеличение производительности за счет отказа от ожидания подтверждений может перевесить любые потери данных.

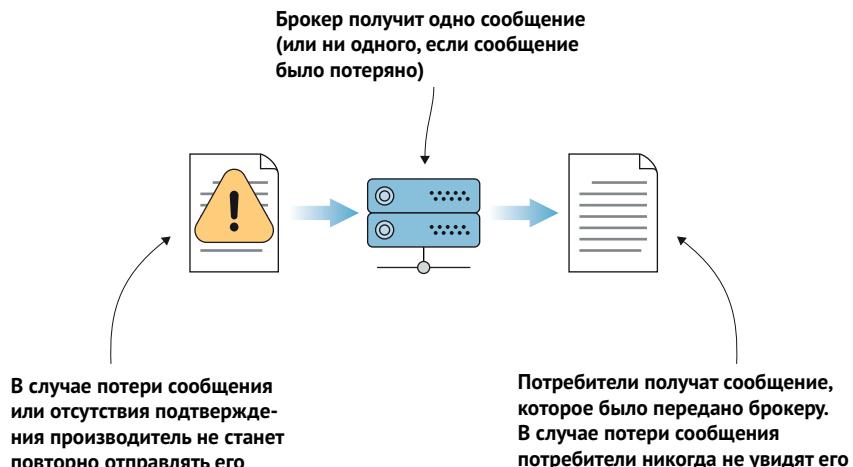


Рис. 1.4. Поток сообщений при использовании семантики «не более одного раза»

Наконец, в версии Kafka 0.11.0 появилась поддержка семантики «точно один раз» (Exactly-Once Semantic, EOS). Появление поддержки этой семантики вызвало множество споров [3]. С одной стороны, семантика «точно один раз» (рис. 1.5) идеально подходит для многих случаев использования. Она выглядит как логическая гарантия удаления дубликатов сообщений. Но большинству разработчиков больше по душе другая логика: отправка одного сообщения и получение того же сообщения на стороне потребителя.



Рис. 1.5. Поток сообщений при использовании семантики «точно один раз»

Также после выхода поддержки EOS развернулась еще одна дискуссия о возможности вообще семантики «точно один раз». Корнями эта дискуссия уходит глубоко в теорию информатики, и все же вам будет полезно знать, как Kafka определяет данную семантику [4]. Если производитель отправит сообщение более одного раза, оно будет доставлено конечному потребителю только один раз. Поддержка семантики «точно один раз» затрагивает все уровни Kafka – производителей, темы, брокеров и потребителей, – и мы кратко рассмотрим их далее в этой книге, а пока продолжим наше обсуждение.

Помимо различных семантик доставки, есть еще одно общее преимущество использования брокера сообщений – если приложение-потребитель потерпело аварию или остановлено для технического обслуживания, то производитель может не ждать, пока его сообщение будет обработано. Когда потребители возобновят работу и вернутся в сеть, они смогут продолжить с того места, на котором остановились, и обработать ожидающие сообщения.

1.2. Использование Kafka

Многие традиционные компании сталкиваются со сложностями, обусловленными все возрастающей ролью технического и программного обеспечения. По этой причине возникает вопрос: как им подготовиться к будущему? Один из возможных ответов: использовать Kafka. Kafka известна как высокопроизводительная рабочая лошадка для доставки сообщений, которая по умолчанию поддерживает репликацию и обеспечивает высокую надежность.

Kafka способна удовлетворить самые взыскательные потребности в обработке данных в промышленном окружении [5]. И это – инструмент версии 1.0, которого не существовало до 2017 года! Однако, отложим эти яркие факты и зададимся вопросом: что может дать Kafka пользователям? Попробуем ответить на него дальше.

1.2.1. Kafka – разработчикам

Что может дать Kafka разработчикам? Распространение Kafka продолжает наращивать темпы, но вопросов у разработчиков не становится меньше [6]. Необходим сдвиг в традиционном подходе к обработке данных. Обмен опытом использования или устранения болевых точек может помочь разработчикам понять, почему Kafka можно считать шагом вперед в их архитектурах данных.

Одним из аспектов, способствующих переходу разработчиков на Kafka, является возможность применения прежнего опыта для познания нового. Например, многие разработчики на Java® используют привычные им концепции Spring®, поэтому в механизм внедрения зависимостей (Dependency Injection, DI) в Spring была добавлена поддержка Kafka (<https://projects.spring.io/spring-kafka>) и уже претерпела два выпуска. Поддерживающие проекты, а также сама платформа Kafka имеют свою расширяющуюся экосистему инструментов.

Большинство программистов сталкивалось в своей практике с проблемами, обусловленными тесной связанностью компонентов. Например, вы хотите внести изменения в одно приложение, но обнаруживаете, что они повлияют на множество других приложений, напрямую связанных с этим. Или вы начинаете модульное тестирование и оказываетесь перед необходимостью создания большого количества фиктивных объектов. Kafka, если применять ее вдумчиво, может помочь в таких ситуациях.

Возьмем, к примеру, систему управления персоналом, которая используется для оформления отпусков. Если вы привыкли к системам с поддержкой операций создания, чтения, обновления и удаления (create, read, update, delete – CRUD), то запрос на добавление записи об отпуске, скорее всего, будет обрабатываться не только бухгалтерской системой, но также системой планиро-

вания проектов для прогнозирования выполнения этапов работ. Вы свяжете эти две системы вместе? И что вы будете делать, если бухгалтерская система рухнет? Должно ли это повлиять на работу системы планирования?

Kafka позволяет отделить друг от друга приложения, которые в старых проектах часто оказывались связанными. (Более подробно об усовершенствовании текущих моделей данных мы поговорим в главе 11.) Ее можно вставить в середину рабочего процесса [7], и она будет служить единым интерфейсом к данным вместо многочисленных API и баз данных.

Некоторые говорят, что есть более простые решения. Представим себе использование процесса извлечения, преобразования и загрузки данных в базы данных для каждого приложения. Это тоже единый интерфейс для каждого приложения, и реализовать его было бы несложно, верно? Но что, если первоначальный источник данных прекратил работать или обновляться? Как часто вы проверяете наличие обновлений, и какие задержки допускаете? Кроме того, хранимые вами копии данных тоже в какой-то момент устареют или разойдутся с источником настолько далеко, что будет трудно восстановить этот поток и получить желаемые результаты. Что должно играть роль источника истины? Kafka может помочь избежать этих проблем.

Еще одна интересная тема, знакомство с которой может повысить доверие к Kafka: насколько она сама использует свои же механизмы. Например, когда мы углубимся в обсуждение потребителей в главе 5, то увидим, что Kafka использует внутренние темы для управления смещениями потребителей. В версии 0.11 семантика «точно один раз» тоже использует внутренние темы. Возможность иметь много потребителей данных, использующих одно и то же сообщение, дает множество допустимых результатов.

Еще один вопрос разработчиков может заключаться в том, почему бы не изучить Kafka Streams, ksqlDB, Apache Spark™ Streaming или другие платформы и миновать изучение ядра Kafka? Количество приложений, внутренне использующих Kafka, действительно впечатляет. Пользоваться слоями абстракции часто удобнее (а иногда без них вообще не обойтись с таким количеством движущихся частей), и все же мы считаем, что изучение самой Kafka совершенно необходимо.

Есть разница между знанием того, что Kafka – это вариант канала для Apache Flume™, и пониманием всех параметров конфигурации. Kafka Streams может упростить примеры, представленные в этой книге, но Kafka добилась успеха задолго до того, как появилась Kafka Streams. Ядро Kafka образует мощный фундамент, и его изучение, я надеюсь, поможет вам понять, почему она используется в некоторых приложениях и что происходит внутри. Если вы

хотите стать экспертом в потоковой передаче данных, то обязаны знать устройство основных распределенных частей ваших приложений и все параметры, которые можно использовать для точной настройки. С чисто технической точки зрения существует множество интересных тем информатики, применяемых на практике. Наиболее обсуждаемым, пожалуй, является понятие распределенных журналов коммитов, которое мы подробно обсудим в главе 2, и моя любимая тема – иерархические колеса синхронизации [8]. Эти примеры показывают, как Kafka решает проблему масштаба, применяя интересную структуру данных для решения практической задачи.

Мы также хотели бы отметить, что платформа распространяется с открытым исходным кодом, а это позволяет исследовать ее исходный код в поисках примеров и ответов на вопросы. Ресурсы не ограничиваются только внутренними знаниями, основанными исключительно на конкретном рабочем месте.

1.2.2. Как преподнести Kafka вашему руководству

Как это часто бывает, нередко члены высшего руководства, услышав слово *Kafka*, приходят в смущение, не задумываясь о фактическом предназначении этой платформы. Было бы неплохо объяснить им ценность этого продукта. Кроме того, нам с вами тоже полезно сделать шаг назад и посмотреть более широко на реальную ценность этого инструмента.

Одна из наиболее важных функций Kafka – возможность принимать большие объемы данных и делать их доступными для использования различными бизнес-подразделениями. Такая магистраль данных, которая делает информацию, поступающую на предприятие, доступной для всех подразделений, обеспечивает гибкость и открытость в масштабе всей компании. Потенциальным результатом является расширение доступа к данным. Большинство руководителей также знает, что, когда поступает очень много данных, возникает проблема организации максимально быстрого доступа к ним. Чтобы платить за хранение постепенно устаревающих данных на диске, их можно получать по мере их поступления и с максимальной выгодой использовать эту оперативность. Kafka дает возможность отказаться от обработки информации с помощью пакетных заданий, ограничивающих скорость превращения данных в ценность. *Быстрые данные* – более новый термин, намекающий на то, что реальная ценность кроется не только в больших объемах данных, но и в оперативности их получения.

Использование виртуальной машины Java JVM® – хорошо знакомое и привычное дело для многих центров корпоративной разработки. Возможность запуска в локальном окружении является

решающим фактором для тех, чьи данные должны управляться на месте. Неплохими вариантами также являются облачные и управляемые платформы. Платформа Kafka может масштабироваться не только вертикально, но также горизонтально, что особенно важно, учитывая существование физического предела вертикального масштабирования.

Возможно, одними из самых веских аргументов, которые можно было привести в пользу Kafka, могли бы стать яркие примеры стартапов и других компаний, действующих в той же отрасли и сумевших преодолеть когда-то непомерно высокую стоимость вычислительной мощности. Вместо крупных и мощных серверов или мейнфреймов, стоящих миллионы долларов, можно использовать менее дорогие распределенные приложения и архитектуры, которые при этом не ухудшают конкурентоспособность.

1.3. Мифы о Kafka

Приступая к изучению любой новой технологии, первым естественным желанием часто бывает сопоставить существующие знания с новыми концепциями. Этот прием тоже можно использовать при изучении Kafka, однако мы хотели бы в первую очередь отметить некоторые из наиболее распространенных заблуждений, с которыми приходилось сталкиваться в нашей работе. Мы рассмотрим их в следующих разделах.

1.3.1. Kafka работает только с Hadoop®

Как уже упоминалось, Kafka – это мощный инструмент, широко используемый в различных ситуациях. Однако многие познакомились с этой платформой в связке с экосистемой Hadoop, а кто-то – как с инструментом в составе пакета Cloudera™ или Hortonworks™. Нередко можно услышать миф о том, что Kafka работает только с Hadoop. На чем основывается такое мнение? Одна из причин заключается в том, что в Kafka применяют различные инструменты, такие как Spark Streaming и Flume, которые вместе с тем используют (или когда-то использовали) Hadoop. Еще один инструмент – Apache ZooKeeper™ – тоже часто встречается в кластерах Hadoop и может еще больше подкреплять этот миф о Kafka.

Другой часто встречающийся миф – для работы Kafka нужна распределенная файловая система Hadoop (Hadoop Distributed Filesystem, HDFS). В действительности это далеко не так. Начав знакомиться с внутренними механизмами Kafka, мы увидим, что скорость работы Kafka и обработки событий падает, когда NodeManager находится внутри процесса. Кроме того, репликация блоков, обычно являющаяся частью HDFS, выполняется иначе. Например, в Kafka реплики не восстанавливаются по умолчанию. Но даже при

том, что оба продукта используют репликацию по-разному, надежность Kafka можно легко объединить с готовностью Hadoop к сбоям по умолчанию (и, следовательно, планированием его преодоления), что является общей целью для Hadoop и Kafka.

1.3.2. *Kafka ничем не отличается от других брокеров сообщений*

Еще один большой миф заключается в том, что Kafka – это лишь еще один брокер сообщений. Прямые сравнения функций различных инструментов (таких как RabbitMQ™ компании Pivotal или MQSeries® компании IBM) с Kafka часто сопровождаются звездочками (или мелким шрифтом) и не всегда точно отражают варианты использования, для которых эти инструменты подходят лучше всего. Некоторые инструменты со временем приобрели или планируют приобрести новые возможности, как, например, семантика «точно один раз» в Kafka. Также можно изменить конфигурацию по умолчанию, чтобы сблизить возможности разных инструментов в том же пространстве. В целом ниже перечислены некоторые из наиболее интересных и выдающихся особенностей, которые мы вкратце рассмотрим:

- возможность повторной передачи сообщений по умолчанию;
- параллельная обработка данных.

Платформа Kafka изначально была ориентирована на работу с несколькими потребителями. Это означает, что приложение, читающее сообщение из брокера сообщений, не делает это сообщение недоступным для других приложений, которые также могут захотеть его получить и использовать. Как следствие, потребитель, уже прочитавший сообщение, сможет снова прочитать его (и другие сообщения). В некоторых архитектурных моделях, таких как лямбда (обсуждается в главе 8), ошибки программиста считаются таким же ожидаемым явлением, как и аппаратные сбои. Представьте, что вы потребляете миллионы сообщений и забываете использовать определенное поле из исходного сообщения. В некоторых очередях прочитанные сообщения удаляются или сохраняются в другом месте. Однако Kafka дает потребителям возможность отыскать конкретный момент и прочитать сообщение снова (пусть и с некоторыми ограничениями), просто просматривая более ранние позиции в теме.

Как уже было сказано, Kafka позволяет обрабатывать данные параллельно и может обслуживать нескольких потребителей в одной и той же теме. Но в Kafka также есть понятие группы потребителей, которое подробно рассматривается в главе 5. Объединяя потребителей в группу, можно определить, какие из них будут получать те или иные сообщения и как будет обслуживаться

эта группа. Группы потребителей действуют независимо друг от друга и позволяют запустить столько приложений, потребляющих сообщения в своем собственном темпе, сколько потребуется для своевременного обслуживания потока. Обработка может происходить разными способами: потреблением многими потребителями, работающими в одном приложении, и потреблением многими потребителями, работающими в нескольких приложениях. А теперь оставим в стороне другие брокеры сообщений и сосредоточимся на испытанных вариантах использования, сделавших Kafka одним из инструментов, к которому обращаются многие разработчики.

1.4. **Kafka в реальном мире**

Практическое использование Kafka – вот главная цель этой книги. Одна из особенностей Kafka, которую следует отметить, – про эту платформу нельзя сказать, что она решает какую-то одну конкретную задачу; она прекрасно справляется с множеством задач. Да, она у нее есть некоторые базовые идеи, и их желательно осветить в первую очередь, но не менее полезно обсудить в общих чертах некоторые реальные случаи использования Kafka. На сайте Apache Kafka перечислены основные сферы использования Kafka в реальном мире, и мы обязательно исследуем их в этой книге [9].

1.4.1. **Ранние примеры**

Некоторые пользователи (как и я сам) начинали с применения Kafka в роли инструмента обмена сообщениями. Лично мне после многих лет использования других инструментов, таких как IBM® WebSphere® MQ (ранее MQ Series), Kafka (в то время это была версия 0.8.3) казалась простым и удобным средством передачи сообщений из пункта A в пункт B. Kafka воздерживается от использования популярных протоколов и стандартов, таких как расширяемый протокол обмена сообщениями о присутствии (Extensible Messaging and Presence Protocol, XMPP), Java Message Service (JMS) API (ныне часть Jakarta EE) или расширенный протокол организации очереди сообщений (Advanced Message Queuing Protocol, AMQP) компании OASIS®, отдавая предпочтение нестандартному двоичному протоколу на основе TCP. Далее в книге мы рассмотрим некоторые варианты его использования.

Для конечного пользователя, разрабатывающего клиента Kafka, основная работа заключается в том, чтобы определить конфигурацию, следя относительно простой логике (например, «Я хочу отправить сообщение в эту тему»). Наличие надежного канала для отправки сообщений также является причиной использования Kafka.

Часто хранение данных в оперативной памяти не гарантирует сохранности информации; если такой сервер отключится, то сообщения исчезнут после перезагрузки. Платформа Kafka изначально создавалась с прицелом на высокую доступность и долговременное хранение. Apache Flume предоставляет вариант канала Kafka, поддержка репликации и высокая доступность которой позволяют сделать события Flume немедленно доступными для других потребителей в случае сбоя агента Flume (или сервера, на котором он работает) [10]. Kafka позволяет создавать надежные приложения и помогает преодолевать ожидаемые сбои, с которыми рано или поздно сталкиваются распределенные приложения.

Во многих ситуациях, в том числе при попытке выбрать события, записанные распределенными приложениями, может пригодиться возможность агрегирования журналов (рис. 1.6). На рисунке показано, как журнальные записи отправляются в виде сообщений в Kafka, при этом разные приложения имеют по одной логической теме для использования этой информации. Благодаря способности Kafka обрабатывать большие объемы данных сбор событий с различных серверов или источников превратился в одну из ключевых функций. В зависимости от содержимого журнала некоторые организации используют эту функцию для аудита и выявления причин сбоев. Kafka также используется в различных инструментах журналирования (или как инструмент ввода).



Рис. 1.6. Агрегирование журналов в Kafka

Как хранение всех этих журналов не вызывает нехватки ресурсов и не мешает Kafka поддерживать высокую производитель-

ность? Передача большого количества коротких сообщений иногда может привести к перегрузке системы, потому что обработка каждого метода требует времени и дополнительных ресурсов. Поэтому для отправки и записи данных Kafka использует пакетную обработку. Особенность журналов, когда данные всегда записываются в конец, тоже помогает сэкономить ресурсы. Подробнее о форматах журналов сообщений мы поговорим в главе 7.

1.4.2. Более поздние примеры

Раньше микросервисы взаимодействовали друг с другом посредством API, таких как REST, но теперь асинхронные сервисы могут обмениваться событиями с помощью Kafka [11]. Микросервисы могут использовать Kafka как интерфейс для своих взаимодействий вместо прямых вызовов API. Kafka зарекомендовала себя как надежная платформа, позволяющая разработчикам быстро получать данные. В настоящее время многие проекты используют Kafka Streams по умолчанию, но еще до выхода Streams API в 2016 году Kafka успела зарекомендовать себя как успешное решение. Streams API можно рассматривать как слой передачи информации, обертывающий производителей и потребителей. Этот уровень абстракции реализован в виде клиентской библиотеки, позволяющей интерпретировать работу с данными как с неограниченным потоком.

В выпуске Kafka 0.11 была реализована семантика «точно один раз». Мы рассмотрим, как она работает, позже, когда поближе познакомимся с основами. Однако уже сейчас можно отметить, что сквозные рабочие нагрузки, действующие через Kafka Streams API, могут воспользоваться усиленными гарантиями доставки. Библиотека Streams упрощает этот вариант использования еще больше и избавляет пользовательскую логику приложения от любых непроизводительных расходов, гарантируя обработку сообщения только один раз от начала до конца транзакции.

Ожидается, что с течением времени количество устройств для интернета вещей (рис. 1.7) будет только увеличиваться. Все эти устройства отправляют сообщения, иногда пачками, когда подключаются к Wi-Fi или сотовой сети, и другая сторона, принимающая эти сообщения, должна иметь возможность эффективно обрабатывать их. Как вы наверняка уже поняли, огромные объемы данных – одна из областей, в которых Kafka предстает во всем блеске. Как отмечалось выше, небольшие сообщения не являются проблемой для Kafka. Датчики, сенсоры, автомобили, телефоны, умные дома и т. д. будут отправлять данные, и должно быть что-то, что обработает поток данных и сделает их доступными для выполнения тех или иных действий [12].

Это лишь небольшая подборка примеров применения Kafka на практике. Как будет показано в следующих главах, Kafka может при-

меняться в самых разных практических областях. А чтобы увидеть другие возможности практического применения этой платформы, необходимо изучать основополагающие концепции.



Рис. 1.7. Интернет вещей (Internet of Things, IoT)

1.4.3. Когда Kafka может быть неприменима

Kafka – отличный инструмент, который с успехом можно использовать во многих разных случаях, но нужно отметить также, что в некоторых ситуациях это не лучший выбор. Давайте рассмотрим несколько вариантов, когда полезнее использовать другие инструменты или специализированный код.

Представьте, что вам нужно получить сводные данные только раз в месяц или даже раз в год. Представьте также, что вам не требуется получать данные по запросу или повторно обрабатывать их. В таких случаях может не понадобиться, чтобы Kafka работала в течение всего года (особенно если пакет данных не особенно велик и его можно обработать целиком). Как обычно, понятие «не особенно велик» может иметь разное значение для разных пользователей.

Если в вашем случае основной операцией с данными является произвольный поиск, то Kafka будет не лучшим выбором. Линейное чтение и запись – вот основная сфера, где Kafka показывает себя с лучшей стороны и обеспечивает максимально быстрое перемещение данных. Возможно, вы слышали, что в Kafka есть индексы, но имейте в виду, что в действительности это совсем не те ин-

дексы, что используются в реляционных базах данных и конструируются на основе полей и первичных ключей.

Точно так же если нужно соблюсти точный порядок следования сообщений в Kafka для всей темы, то вам придется посмотреть, насколько практична ваша рабочая нагрузка в этой ситуации. Чтобы избежать любых нарушений в порядке следования сообщений, понадобится позаботиться о том, чтобы в системе всегда выполнялся только один поток, опрашивающий производителя, и чтобы в теме был только один раздел. Существуют разные обходные решения, но если требуется обрабатывать большие объемы данных в строго определенном порядке, то потребление данных будет ограничено одним потребителем на группу за раз.

Еще один практический момент, который приходит на ум: большие сообщения. Размер сообщения по умолчанию составляет около 1 Мбайт [13]. При передаче больших сообщений вы начнете замечать увеличение нагрузки на память. Иначе говоря, уменьшение количества сообщений, умещающихся в кеше страниц, может стать проблемой. Если вы планируете рассыпать огромные архивы, то я бы посоветовал поискать более подходящие способы управления такими сообщениями. Имейте в виду, что с Kafka вы, вероятно, сможете достичь конечной цели в описанных ситуациях (в принципе, это возможно), но эта платформа может оказаться не лучшим выбором.

1.5. Онлайн-ресурсы

Сообщество Kafka проделало большую работу и создало одну из лучших (на наш взгляд) подборку документации. Kafka – часть фонда Apache (включена в Apache Incubator в 2012 году), и текущая ее документация хранится на веб-сайте проекта по адресу <https://kafka.apache.org>.

Еще один отличный информационный ресурс – Confluent® (<https://www.confluent.io/resources>). Проект Confluent был основан создателями Kafka и активно влияет на направление работы. В рамках этого проекта разрабатываются корпоративные функции и осуществляется поддержка компаний с целью помочь им в разработке их потоковых платформ. Участники проекта продолжают поддерживать открытую природу исходного кода Kafka и предоставляют презентации и лекции, в которых обсуждаются производственные задачи и достижения.

Когда в следующих главах мы начнем углубляться в другие API и параметры конфигурации, эти ресурсы станут для вас полезным справочником, где вы сможете почерпнуть дополнительные подробности. В главе 2 мы еще вернемся к теме использования Kafka и начнем знакомиться с практической стороной Apache Kafka.

Итоги

- Apache Kafka – это платформа потоковой передачи, которую можно использовать для быстрой обработки большого количества событий.
- Kafka может использоваться только в качестве шины сообщений, но в этом случае останутся неиспользованными возможности обработки данных в реальном времени.
- Для многих Kafka ассоциировалась в прошлом с другими решениями для работы с большими данными, однако в действительности Kafka является самостоятельной, масштабируемой и надежной системой. В ней используются те же системные методы обеспечения отказоустойчивости и распределенной обработки, поэтому она способна удовлетворить самые разные потребности современной инфраструктуры данных, предоставляя свои средства кластеризации.
- При использовании для потоковой передачи большого количества событий, таких как данные интернета вещей, Kafka позволяет быстро обрабатывать данные. Когда появляется дополнительная информация для ваших приложений, Kafka быстро предоставляет результаты для ваших данных, которые когда-то обрабатывались в пакетном режиме.

Ссылки

- 1 R. Moffatt. «The Changing Face of ETL». Confluent blog (September 17, 2018). <https://www.confluent.io/blog/changing-face-etl/> (дата публикации: 10 мая 2019).
- 2 «Introduction». Apache Software Foundation (n.d.). <https://kafka.apache.org/intro> (доступно по состоянию на 30 мая 2019).
- 3 Документация. Apache Software Foundation (n.d.). <https://kafka.apache.org/documentation/#semantics> (доступно по состоянию на 30 мая 2020).
- 4 N. Narkhede. «Exactly-once Semantics Are Possible: Here's How Apache Kafka Does It». Блог Confluent (30 июня 2017). <https://www.confluent.io/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it> (доступно по состоянию на 27 декабря 2017).
- 5 N. Narkhede. «Apache Kafka Hits 1.1 Trillion Messages Per Day – Joins the 4 Comma Club». Блог Confluent (1 сентября 2015). <https://www.confluent.io/blog/apache-kafka-hits-1-1-trillion-messages-per-day-joins-the-4-comma-club/> (доступно по состоянию на 20 октября 2019).

- 6 L. Dauber. «The 2017 Apache Kafka Survey: Streaming Data on the Rise». Блог Confluent (4 мая 2017). <https://www.confluent.io/blog/2017-apache-kafka-survey-streaming-data-on-the-rise/> (доступно по состоянию на 23 декабря 2017).
- 7 K. Waehner. «How to Build and Deploy Scalable Machine Learning in Production with Apache Kafka». Блог Confluent (29 сентября 2017) <https://www.confluent.io/blog/build-deploy-scalable-machine-learning-production-apache-kafka/> (доступно по состоянию на 11 декабря 2018).
- 8 Y. Matsuda. «Apache Kafka, Purgatory, and Hierarchical Timing Wheels». Блог Confluent (28 октября 2015). <https://www.confluent.io/blog/apache-kafka-purgatory-hierarchical-timing-wheels> (доступно по состоянию на 20 декабря 2018).
- 9 «Use cases». Apache Software Foundation (n.d.). <https://kafka.apache.org/uses> (доступно по состоянию на 30 мая 2017).
- 10 «Flume 1.9.0 User Guide». Apache Software Foundation (n.d.). <https://flume.apache.org/FlumeUserGuide.html> (доступно по состоянию на 27 мая 2017).
- 11 B. Stopford. «Building a Microservices Ecosystem with Kafka Streams and KSQL». Блог Confluent (9 ноября 2017). <https://www.confluent.io/blog/building-a-microservices-ecosystem-with-kafka-streams-and-ksql/> (доступно по состоянию на 1 мая 2020).
- 12 «Real-Time IoT Data Solution with Confluent». Документация Confluent. (n.d.). <https://www.confluent.io/use-case/internet-of-things-iot/> (доступно по состоянию на 1 мая 2020).
- 13 Документация. Apache Software Foundation (n.d.). https://kafka.apache.org/documentation/#brokerconfigs_message.max.bytes (доступно по состоянию на 30 мая 2020).



Знакомство с Kafka

Эта глава охватывает следующие темы:

- высокоуровневую архитектуру Kafka;
- знакомство с возможностями клиента;
- порядок взаимодействий приложений с брокером;
- отправку и прием вашего первого сообщения;
- использование клиентов Kafka с приложением на Java.

Теперь, получив общее представление о Kafka и в каких случаях эту платформу можно использовать, давайте углубимся в компоненты Kafka, из которых состоит система. Apache Kafka по своей сути является распределенной системой, но ее также можно установить и запустить на одном хосте. Это позволяет нам начать исследование примеров использования. Как это часто бывает, настоящие вопросы начинают возникать, только когда руки касаются клавиатуры. К концу этой главы вы сможете отправить и получить свое первое сообщение Kafka из командной строки. Давайте начнем наше более близкое знакомство с Kafka, а затем перейдем к изучению деталей ее архитектуры.

ПРИМЕЧАНИЕ. Если у вас нет кластера Kafka или вы хотите запустить систему на локальном компьютере, обратитесь к прило-

жению A, где описывается, какие изменения в конфигурацию по умолчанию нужно внести, чтобы запустить три брокера Apache Kafka, которые мы будем использовать в наших примерах. Убедитесь, что все экземпляры запущены и работают, прежде чем пытаться опробовать какие-либо примеры из этой книги! Если какие-то примеры не работают, проверьте исходный код на GitHub, где приводятся советы и рекомендации по исправлению проблем.

2.1. Отправка и прием сообщения

Сообщение, также называемое *записью*, является основной частью данных, проходящих через Kafka. Сообщения – это представление ваших данных в Kafka. Каждое сообщение имеет отметку времени, значение и необязательный ключ. При желании также можно добавлять свои заголовки [1]. Вот простой пример сообщения: машина с идентификатором хоста «1234567» (*ключ сообщения*) вышла из строя с сообщением «Предупреждение: внутренняя ошибка» (*значение сообщения*) в «2020-10-02T10:34:11.654Z» (*отметка времени сообщения*). В главе 9 мы покажем пример добавления своего заголовка, определяющего пару *ключ/значение*, с целью трассировки.

На рис. 2.1 показаны наиболее важные части сообщения, с которыми пользователи работают напрямую. Основное внимание в этой главе мы будем уделять ключам и значениям, которые требуют анализа при разработке сообщений. Каждый ключ и значение могут взаимодействовать по-своему, выполняя сериализацию или десериализацию заключенных в них данных. Подробнее о сериализации мы поговорим в главе 4, когда приступим к обсуждению вопросов создания сообщений.



Рис. 2.1. Сообщения Kafka состоят из ключа и значения (отметка времени и необязательные заголовки здесь не показаны)

Но как, создав запись, передать ее в Kafka? Для этого сгенерированное сообщение нужно передать так называемым *брокерам*.

2.2. Что такое брокер?

Брокер – это серверный компонент Kafka [1]. До появления виртуальных машин и Kubernetes® вы могли иметь один физический сервер, на котором размещается один брокер. Поскольку почти все кластеры имеют несколько серверов (или узлов), в большинстве наших примеров мы будем использовать три сервера Kafka. Это локальное тестовое окружение позволит нам видеть вывод команд, выполняемых несколькими брокерами, обеспечивая сходство с окружением, в котором работает несколько брокеров на разных машинах.

Для нашего первого примера мы создадим тему и отправим первое сообщение в Kafka из командной строки. Следует отметить, что Kafka изначально создавалась с возможностью управления из командной строки. Мы не будем использовать графический интерфейс и все действия будем производить в интерфейсе командной строки операционной системы. Команды вводятся в строке приглашения к вводу. Можете использовать любой текстовый редактор – vi, Emacs, Nano или какой-то другой, – главное, чтобы вы чувствовали себя комфортно.

ПРИМЕЧАНИЕ. Kafka совместима со многими операционными системами, но чаще всего она развертывается в Linux, а, как известно, при работе с этой операционной системой желательно иметь навыки работы с командной строкой.

Вспомогательные средства командной оболочки

Если вы часто пользуетесь командной строкой и хотели бы получить в свое распоряжение улучшенные возможности автодополнения команд (и для получения справки о доступных аргументах), то мы рекомендуем обратиться к проекту поддержки автодополнения команд Kafka по адресу <http://mng.bz/K48O>. Если вы пользуетесь командной оболочкой Zsh, то также можете установить плагин Kafka Zsh-completion, доступный по адресу <https://github.com/Dabz/kafka-zsh-completions>.

Чтобы отправить сообщение, необходимо иметь место, куда оно будет отправлено. Таким местом в Kafka является *тема*. Чтобы создать тему, выполним в терминале команду `kafka-topics.sh` с параметром `--create` (листинг 2.1). Этот сценарий находится в каталоге установки Kafka, путь к которому может выглядеть так: `~/kafka_2.13-2.7.1/bin`. Обратите внимание, что пользователи Windows могут использовать файлы `.bat` с теми же именами. Например, для сценария `kafka-topics.sh` в Windows имеется аналогичный сценарий с именем `kafka-topics.bat`, который должен находиться в каталоге `<каталог_установки_kafka>/bin/windows`.

ПРИМЕЧАНИЕ. Ссылки `kinaction` и `ka` (например, в используемом имени `kaProperties`) представляют различные сокращения от названия книги «*Kafka in Action*» («*Kafka в действии*») и не связаны с какими-либо продуктами или компаниями.

Листинг 2.1. Создание темы `kinaction_helloworld`

```
bin/kafka-topics.sh --create --bootstrap-server localhost:9094  
--topic kinaction_helloworld --partitions 3 --replication-factor 3
```

После этого в консоли должен появиться текст: `Created topic kinaction_helloworld` (Создана тема `kinaction_helloworld`). В листинге 2.1 мы дали нашей теме имя `kinaction_helloworld`. Конечно, мы могли бы использовать любое другое имя, но, вообще, при выборе имен принято следовать соглашениям, принятым в Unix/Linux, в том числе не использовать пробелы. Вы сможете избежать многих неприятных ошибок и предупреждений, если не будете использовать пробелы или специальные символы в именах. Они не всегда хорошо сочетаются с интерфейсом командной строки и механизмом автодополнения.

В команде есть еще пара параметров, смысл которых, возможно, вам пока не ясен, поэтому, чтобы продолжить исследование, давайте кратенько определим их. Более подробно параметры будут рассматриваться в главе 6.

Параметр `--partitions` определяет, на сколько частей будет делиться тема. Например, мы предполагаем использовать три брокера, поэтому, разбив тему на три раздела, дадим по одному разделу каждому брокеру. Для наших тестовых рабочих нагрузок такого количества может не понадобиться. Однако создание нескольких разделов на этом этапе поможет нам увидеть, как работает распределение данных по разделам. В параметре `--replication-factor` мы также передали число 3. Оно говорит, что в каждом разделе мы хотим иметь по три реплики (копии). Создание копий является важным шагом для увеличения надежности и отказоустойчивости. Параметр `--bootstrap-server` задает сетевой адрес брокера Kafka (в данном случае на локальном компьютере). Вот почему брокер должен быть запущен перед вызовом этого сценария. Для нас сейчас самое важное – получить общее представление об устройстве. А подробнее о том, как правильно выбрать числа в конкретных ситуациях, мы поговорим при обсуждении брокеров.

Также из командной строки можно запросить список всех имеющихся тем и убедиться, что наша новая тема присутствует в списке. Для этого предназначен параметр `--list`. Снова запустим сценарий, как показано в листинге 2.2.

Листинг 2.2. Проверка наличия созданной темы

```
bin/kafka-topics.sh --list --bootstrap-server localhost:9094
```

Чтобы получить представление о том, как выглядит наша новая тема, в листинге 2.3 показана еще одна команда, которую можем выполнить, дающая более полное представление о кластере. Обратите внимание, что наша тема не похожа на традиционную одиночную тему в других системах обмена сообщениями: в ней есть реплики и разделы. Цифры, которые выводятся рядом с метками полей `Leader`, `Replicas` и `Isr`, – это идентификаторы брокеров, соответствующие значениям параметров настройки наших трех брокеров, которые мы установили в файлах конфигурации. Взглянув на вывод, можно заметить, что наша тема состоит из трех разделов: `Partition 0`, `Partition 1` и `Partition 2`. Каждый раздел скопирован три раза, как мы и задали при создании темы.

Листинг 2.3. Описание темы `kinaction_helloworld`

<pre>bin/kafka-topics.sh --bootstrap-server localhost:9094 \ --describe --topic kinaction_helloworld</pre>	Параметр -describe позволяет увидеть харак- теристики соз- данной темы
--	---

```
Topic:kinaction_helloworld PartitionCount:3 ReplicationFactor:3 Configs:
Topic: kinaction_helloworld Partition: 0 Leader: 0 Replicas: 0,1,2 Isr: 0,1,2
Topic: kinaction_helloworld Partition: 1 Leader: 1 Replicas: 1,2,0 Isr: 1,2,0
Topic: kinaction_helloworld Partition: 2 Leader: 2 Replicas: 2,0,1 Isr: 2,0,1
```

В первой строке в листинге 2.3 приводятся краткие сведения об общем количестве разделов и реплик на каждый раздел. В следующих строках показаны характеристики всех разделов темы. Вторая строка в выводе относится к разделу с меткой 0 и т. д. Давайте рассмотрим подробнее раздел 0, имеющий лидера и реплику в брокере 0. В этом разделе также есть реплики, которые в брокерах 1 и 2. Имя последнего столбца – `Isr` – расшифровывается как *in-sync replicas* – синхронизированные реплики. Синхронизированные реплики показывают, какие брокеры актуальны и не отстают от лидера. Наличие устаревшей копии реплики раздела (отстающей от лидера) – это проблема, которую мы рассмотрим позже, а пока важно запомнить, что работоспособность реплики в распределенной системе – это та характеристика, за которой мы должны следить. На рис. 2.2 показано, как устроен брокер с идентификатором 0.

Брокер 0 управляет разделом 0.
Остальные реплики копируются из других брокеров

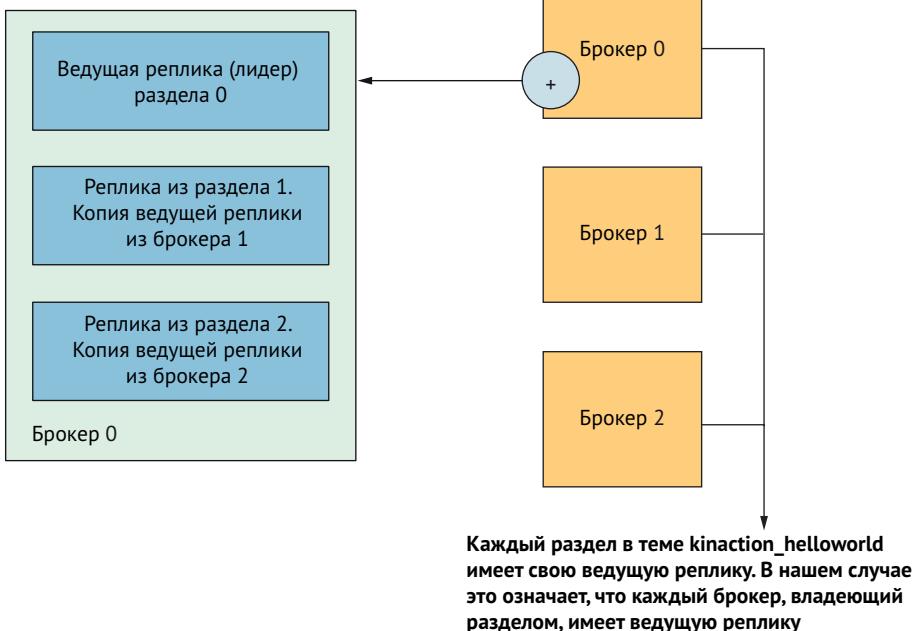


Рис. 2.2. Устройство брокера

Брокер 0 в нашей теме `kinaction_helloworld` содержит ведущую реплику для раздела 0. Он также содержит копии реплик для разделов 1 и 2, для которых он не является ведущей репликой. В случае его копии раздела 1 данные для этой реплики будут скопированы с брокера 1.

ПРИМЕЧАНИЕ. Мы еще не раз будем ссылаться на *лидера* – ведущую реплику раздела. Важно знать, что раздел может иметь одну и более реплик, но только одна реплика будет ведущей. Роль лидера предполагает получение обновлений от внешних клиентов, в то время как простые реплики получают обновления только от соответствующего лидера.

Теперь, создав тему и убедившись, что она существует, можно начать отправлять настоящие сообщения! Имеющие прежний опыт использования Kafka могут спросить, почему мы вручную создали тему перед отправкой сообщения, если есть настройка, включающая и выключающая автоматическое создание тем. Ответ прост: всегда лучше контролировать создание тем, выполняя конкретное действие, чтобы избежать случайного появления новых тем, если в коде допущена опечатка в названии темы, или создания тем заново при повторных попытках производителя отправить сообщение.

Чтобы отправить сообщение, откройте окно терминала, в котором мы запустим производителя как консольное приложение, принимающее ввод пользователя [2]. Команда в листинге 2.4 запускает интерактивную программу, которая берет управление на себя; чтобы вернуться обратно в командную оболочку, нужно нажать комбинацию **Ctrl-C**. После запуска программы просто начните вводить какой-нибудь текст с префиксом `kinaction` (от оригинального названия книги «Kafka In Action»). Мы в нашем примере ввели текст `kinaction_helloworld`, следуя духу примера «hello, world», который можно найти в книге «The C Programming Language» [3].

Листинг 2.4. Команда запуска производителя Kafka

```
bin/kafka-console-producer.sh --bootstrap-server localhost:9094 \
--topic kinaction_helloworld
```

Обратите внимание, что в команде в листинге 2.4 мы сослались на тему для взаимодействий, определив параметр `bootstrap-server`. Этот параметр может ссылаться только на существующего брокера в кластере. Используя эту информацию, кластер сможет извлечь метаданные, необходимые для работы с темой.

Теперь запустите новое окно терминала, где мы опробуем потребителя, тоже реализованного как консольное приложение. Команда в листинге 2.5 запускает программу потребителя, которая также берет на себя управление оболочкой [2]. В этом окне вы должны увидеть сообщение, введенное в консоли с приложением производителя. Обязательно используйте одно и то же значение параметра `topic` в обеих командах – иначе ничего не увидите.

Листинг 2.5. Команда запуска потребителя Kafka

```
bin/kafka-console-consumer.sh --bootstrap-server localhost:9094 \
--topic kinaction_helloworld --from-beginning
```

Следующий пример (листинг 2.6) демонстрирует вывод, который получили мы у себя.

Листинг 2.6. Пример вывода потребителя для темы kinaction_helloworld

```
bin/kafka-console-consumer.sh --bootstrap-server localhost:9094 \
--topic kinaction_helloworld --from-beginning
```

```
kinaction_helloworld
...
```

При отправке последующих сообщений и подтверждении доставки в приложении-потребителе можно опустить параметр `--from-beginning`. Обратите внимание, что при этом вы не уви-

те ранее отправленных сообщений – отображаться будут только сообщения, созданные после запуска потребителя. Способность определять, какие сообщения должны быть прочитаны следующими, и возможность задавать определенное смещение – вот те инструменты, которые мы будем использовать в главе 5 при обсуждении потребителей. Теперь после опробования простого примера у нас появилась дополнительная информация для обсуждения.

2.3. Экскурсия по Kafka

В табл. 2.1 перечислены основные компоненты и их роли в архитектуре Kafka, а в следующих разделах мы рассмотрим их более подробно, чтобы заложить прочную основу для следующих глав.

Таблица 2.1. Архитектура Kafka

Компонент	Роль
Производитель	Посыпает сообщения в Kafka
Потребитель	Извлекает сообщения из Kafka
Темы	Логические имена очередей в брокерах, куда помещаются сообщения
Ансамбль ZooKeeper	Помогает поддерживать согласованность в кластере
Брокер	Обслуживает журнал коммитов на диске, в котором сохраняются сообщения

2.3.1. Производители и потребители

Давайте задержимся ненадолго на первой остановке нашей экскурсии и поговорим о производителях и потребителях. На рис. 2.3 показано, что производители и потребители различаются направлением передачи данных по отношению к кластеру.



Рис. 2.3. Производители и потребители

Производитель – это компонент, отправляющий сообщения в темы Kafka [1]. Как отмечалось в примерах использования в главе 1, хорошей иллюстрацией могут служить файлы журналов, производимых приложениями. Эти файлы не являются частью системы Kafka, пока не будут собраны и отправлены в Kafka. Но когда вы рассматриваете входные данные, поступающие в Kafka, видите перед собой производителя, вовлеченного во внутренние операции.

Производителей по умолчанию не существует, но есть API, взаимодействующие с Kafka, которые используют производителей в собственной реализации. Некоторые пути входа в Kafka могут включать использование отдельного инструмента, такого как Flume, или других API в Kafka, таких как Connect и Streams. Одним из примеров, когда используется производитель внутри реализации, может служить `WorkerSourceTask` в исходном коде Apache Kafka Connect (начиная с версии 1.0). Он предоставляет свой высокоуровневый API. Этот конкретный код в версии 1.0 доступен по лицензии Apache 2 (<https://github.com/apache/kafka/blob/trunk/LICENSE>) на GitHub (<http://mng.bz/9N4r>). Производители также используются для отправки сообщений внутри самой Kafka. Например, читая данные из определенной темы и отправляя их в другую тему, мы также будем использовать производителя.

Чтобы получить представление о том, как будет выглядеть наш собственный производитель, полезно заглянуть в код, по своей концепции напоминающий `WorkerSourceTask` – класс Java, упомянутый выше. В листинге 2.7 показан код нашего примера. Здесь показан не весь код метода `main`, только логика отправки сообщения с помощью стандартного `KafkaProducer`. Не обязательно полностью понимать весь код следующего примера. Просто постарайтесь увидеть, как используется производитель.

Листинг 2.7. Отправка сообщений с помощью производителя

```
Alert alert = new Alert(1, "Stage 1", "CRITICAL", "Stage 1 stopped");
ProducerRecord<Alert, String> producerRecord =
    new ProducerRecord<Alert, String>
        ("kinaction_alert", alert, alert.getAlertMessage()); □

producer.send(producerRecord,
    new AlertCallback()); □
producer.close();
```

For asynchronous message delivery, you can use return callbacks

Выполняет фактическую отправку сообщения брокерам

ProducerRecord хранит все сообщения, отправленные в Kafka

Для отправки данных в Kafka мы создали в листинге 2.7 экземпляр `ProducerRecord`. Этот объект позволяет определить содержимое сообщения и указать тему (в данном случае `kinaction_alert`), куда должно быть отправлено сообщение. Мы использовали свой объект `Alert` в качестве ключа в сообщении. Затем вызвали метод `send` для отправки экземпляра `ProducerRecord`. Мы можем дождаться отправки сообщения или использовать обратный вызов для отправки в асинхронном режиме и сохранить при этом возможность обрабатывать любые ошибки. Этот пример более подробно обсуждается в главе 4.

На рис. 2.4 показано взаимодействие с пользователем, запускающее процесс отправки данных производителю. Пользователь щелкает на кнопке на веб-странице и генерирует событие щелчка, которое затем передается в кластер Kafka.

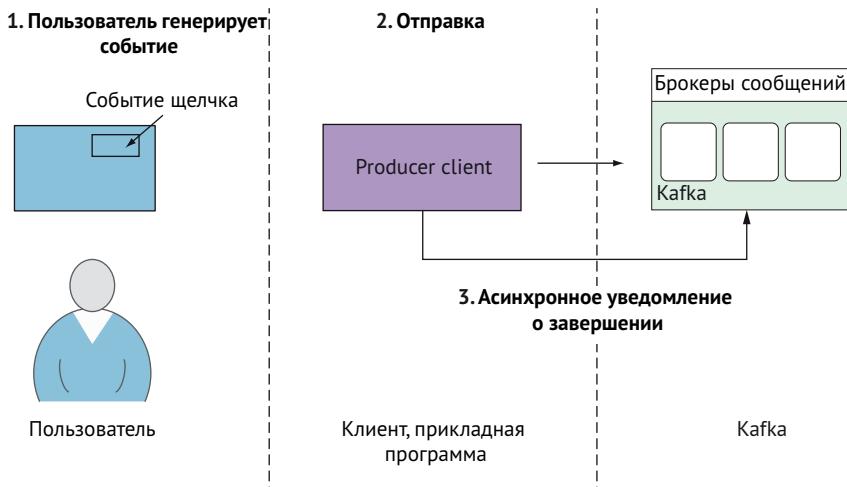


Рис. 2.4. Пример производителя для события, генерируемого пользователем

В отличие от производителя *потребитель* – это инструмент для получения сообщений из Kafka [1]. По аналогии с производителями, говоря о получении данных из Kafka, мы подразумеваем потребителя, вовлеченного в работу прямо или косвенно. `WorkerSinkTask` – это еще один класс в исходном коде Apache Kafka Connect версии 1.0, который может служить примером использования потребителя (<http://mng.bz/WrRW>). Приложения-потребители подписываются на интересующие их темы и постоянно запрашивают данные. Действующим примером потребителя может служить класс `WorkerSinkTask`, который используется для извлечения записей из тем в Kafka. В листинге 2.8 показан пример потребителя, который мы создадим в главе 5. Он иллюстрирует идеи, аналогичные `WorkerSinkTask.java`.

Листинг 2.8. Потребление сообщений

```

...
consumer.subscribe(List.of("kinaction_audit"));
while (keepConsuming) {
    var records = consumer.
        poll(Duration.ofMillis(250));
    for (ConsumerRecord<String, String> record : records) {
        log.info("kinaction_info offset = {}, kinaction_value = {}",
            record.offset(), record.value());
    }
    OffsetAndMetadata offsetMeta =
        new OffsetAndMetadata(++record.offset(), "");
    Map<TopicPartition, OffsetAndMetadata> kaOffsetMap = new HashMap<>();
    kaOffsetMap.put(new TopicPartition("kinaction_audit",
        record.partition()), offsetMeta);
    consumer.commitSync(kaOffsetMap);
}
}
...

```

В листинге 2.8 показано, как объект-потребитель вызывает метод `subscribe` и передает ему список тем, откуда он хотел бы получать данные (в данном случае `kinaction_audit`). Затем потребитель опрашивает темы (рис. 2.5) и обрабатывает любые данные, полученные в виде экземпляров `ConsumerRecord`.

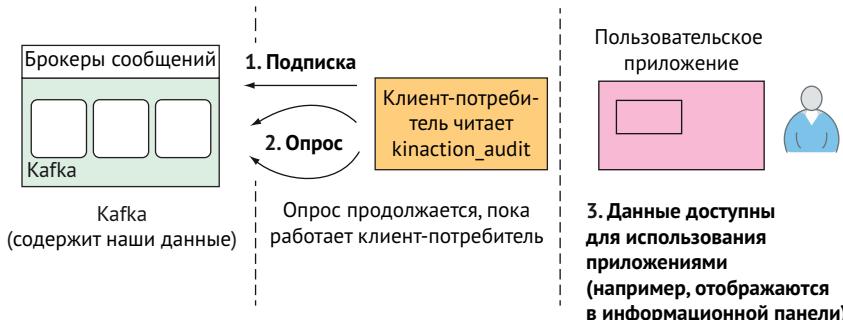


Рис. 2.5. Пример работы потребителя

Листинги 2.7 и 2.8 представляют две части конкретного примера использования Kafka, изображенные на рис. 2.4 и 2.5. Допустим, компания хочет знать, сколько раз пользователи щелкнули на кнопке, чтобы выполнить некоторое действие. События щелчков, сгенерированные пользователями, будут служить данными, поступающими в экосистему Kafka. Потребителями данных будет приложение, которое может использовать свои функции для обработки данных.

Ввод данных в Kafka и получение их из Kafka с помощью своего кода, как было показано выше (или даже с помощью Kafka Connect), позволяет пользователям работать с данными, которые могут повлиять на их бизнес-требования и цели. Kafka не участвует в обработке данных для приложений, это прерогатива приложений-потребителей, которые действительно извлекают из данных пользу для бизнеса. Теперь, узнав, как передавать данные в Kafka и извлекать их из нее, давайте посмотрим, как эти данные попадают в наш кластер.

2.3.2. Темы

Темы – это место, с которого большинство пользователей начинает думать о том, какие сообщения и куда нужно отправить. Темы состоят из блоков, называемых *разделами* [1]. Другими словами, одна тема может состоять из одного или нескольких разделов. Что касается фактической реализации Kafka, то она по большей части работает именно с разделами.

ПРИМЕЧАНИЕ. Реплика с одним разделом существует только в одном брокере и не может разбиваться между брокерами.

Как показано на рис. 2.6, каждая ведущая реплика раздела существует в одном брокере Kafka и не может разбиваться на более мелкие части. Вспомните наш первый пример, тему `kinaction_helloworld`. Если вы стремитесь к надежности и хотите иметь три копии данных, то сама тема – это не одна сущность (или один файл), которая копируется; в действительности трижды копируются различные разделы.

Тема `kinaction_helloworld` состоит из трех разделов, которые, возможно, будут распределены между разными брокерами

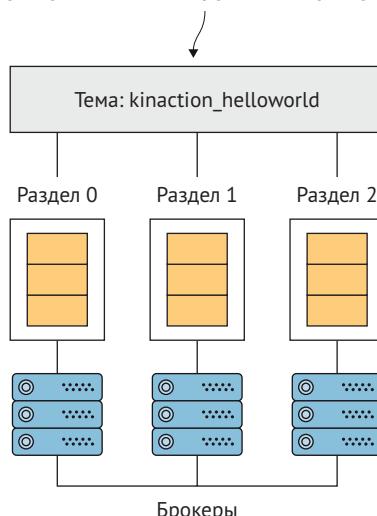


Рис. 2.6. Темы состоят из разделов

ПРИМЕЧАНИЕ. Разделы в свою очередь разбиваются на файлы сегментов и записываются на диск. Структуру этих файлов и их расположение мы рассмотрим в следующих главах, когда будем говорить о брокерах. Даже при том, что файлы сегментов составляют разделы, вы, скорее всего, никогда не будете взаимодействовать с этими файлами напрямую, и их следует рассматривать как внутреннюю деталь реализации.

Одной из наиболее важных идей, которую нужно понять на данном этапе, – одна из копий раздела (реплика) будет называться *ведущей*, или *лидером*. Например, если есть тема, состоящая из трех разделов и трех копий каждого раздела, то каждый раздел будет иметь реплику, избранную лидером. Этот лидер будет одной из копий раздела, а две другие (на рис. 2.6 они не показаны) будут *подписчиками* лидера и обновлять свою информацию, получая ее от своего лидера (ведущей реплики) раздела [1]. В сценариях, когда не возникает исключений или сбоев (известных также как сценарии «счастливого пути»), производители и потребители взаимодействуют только с ведущей репликой раздела, с которым они связаны. Но как производитель или потребитель узнает, какая реплика раздела является ведущей? В распределенных вычислениях, когда возможны случайные сбои, для ответа на этот вопрос часто используют ZooKeeper, о котором мы поговорим далее.

2.3.3. ZooKeeper

ZooKeeper – один из старейших источников дополнительной сложности в экосистеме Kafka. Apache ZooKeeper (<http://zookeeper.apache.org/>) – это распределенное хранилище, предлагающее высокодоступные службы обнаружения, настройки и синхронизации. В версиях Kafka, начиная с 0.9, в ZooKeeper были внесены изменения, позволяющие потребителю не хранить информацию о том, как далеко он продвинулся в чтении сообщений (так называемые *смещения*). О важности смещений мы поговорим в последующих главах. Однако это упрощение не избавило от необходимости достижения консенсуса и координации в распределенных системах.

Удаление ZooKeeper

Для упрощения требований к Kafka было предложено заменить ZooKeeper собственным управляемым кворумом [4]. Поскольку на момент публикации книги эта работа еще не была завершена, мы рассмотрим ZooKeeper. Почему ZooKeeper по-прежнему важен?

В этой книге рассматривается версия 2.7.1, а в ваших промышленных окружениях наверняка встретятся более старые версии, в которых какое-то время еще будет использоваться ZooKeeper. Кроме того, несмотря на замену ZooKeeper режимом метаданных Kafka Raft (KRaft)

в будущем, концепции координации в распределенной системе не потеряют своей актуальности, и мы надеемся, что знание роли ZooKeeper заложит основу для понимания этих концепций. Kafka обеспечивает отказоустойчивость и надежность, но что-то должно обеспечивать координацию, и этим «чем-то» является ZooKeeper. Мы не будем подробно рассматривать внутреннее устройство ZooKeeper, но коснемся его использования в Kafka в следующих главах.

Как вы уже видели, наш кластер для Kafka включает несколько брокеров (серверов). Чтобы брокеры действовали как одно согласованное приложение, они должны не только общаться друг с другом, но и достигать *согласия*. Согласование того, какой из них является ведущей репликой раздела, – один из примеров практического применения ZooKeeper в экосистеме Kafka. Рассмотрим для сравнения пример из реального мира: все мы видели примеры рассинхронизации часов и как порой невозможно определить правильное время, если показания нескольких часов разнятся. Согласование – слишком сложная задача для отдельных брокеров. Необходимо, чтобы Kafka координировала их действия и сохраняла работоспособность как в сценариях успеха, так и в сценариях неудач.

Следует отметить, что в любом варианте промышленного использования ZooKeeper работает как ансамбль, но мы будем запускать только один сервер в нашем локальном окружении [5]. На рис. 2.7 показан кластер ZooKeeper и как Kafka взаимодействует с брокерами. В KIP-500 это называется «текущей» архитектурой кластера [4].

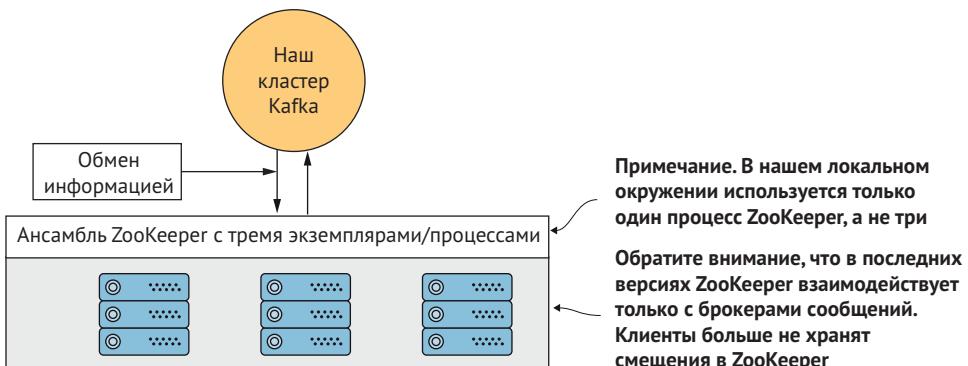


Рис. 2.7. Взаимодействие с ZooKeeper

СОВЕТ. Если вы знакомы с *znodes* или уже имеете опыт работы с ZooKeeper, то хорошим местом внутри исходного кода Kafka для изучения взаимодействий вам послужит *ZkUtils.scala*.

Знание основ описанных выше концепций поможет вам эффективнее применять Kafka на практике. Кроме того, эти знания помогут вам увидеть, как существующие системы, использующие Kafka, могут взаимодействовать для решения реальных задач.

2.3.4. Высокоуровневая архитектура Kafka

В общем случае ядро Kafka можно рассматривать как процессы приложений на языке Scala, которые выполняются в виртуальной машине Java (JVM). Как известно, Kafka способна быстро обрабатывать миллионы сообщений, но какая особенность архитектуры Kafka делает это возможным? Одна из ключевых особенностей Kafka – использование кеша страниц операционной системы (как показано на рис. 2.8). Избегая кеширования в куче JVM, брокеры могут предотвратить некоторые проблемы, свойственные большим кучам (например, длительные или частые паузы на сборку мусора) [6].

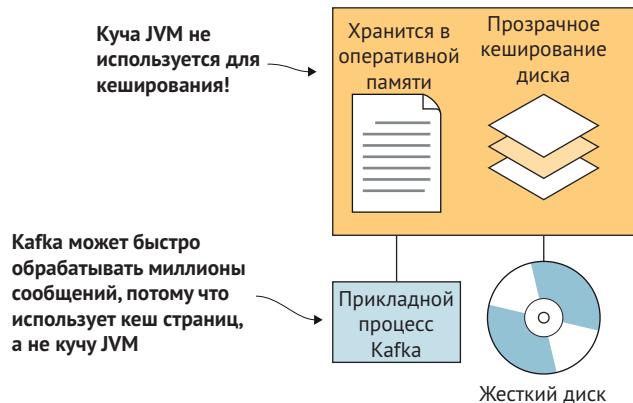


Рис. 2.8. Кеш страниц операционной системы

Еще одной архитектурной особенностью является схема доступа к данным. Когда поступают новые сообщения, высока вероятность, что пришедшие последними представляют больший интерес для потребителей, и, соответственно, предпочтительнее обслуживать их из кеша. Обслуживание из кеша страниц, а не с диска в большинстве случаев происходит намного быстрее. В некоторых особых случаях увеличение объема ОЗУ помогает большей части рабочей нагрузки попасть в кеш страниц.

Как упоминалось выше, Kafka использует свой протокол [7], потому что, как отмечают создатели Kafka, использование существующего протокола, такого как AMQP (Advanced Message Queuing Protocol – расширенный протокол организации очередей сообщений), оказывает слишком большое влияние на реальную реализацию. Например, в версии 0.11 в заголовок сообщения были добавлены новые поля для реализации семантики «точно один раз».

Кроме того, в той же версии был переработан формат сообщений, обеспечивающий более эффективное их сжатия. Используя свой протокол, создатели Kafka могут менять его в соответствии со своими потребностями.

Мы почти закончили нашу экскурсию. Осталось сделать еще одну остановку и познакомиться поближе с брокерами и журналами коммитов (фиксаций).

2.3.5. Журнал коммитов

Еще один важный компонент, понимание которого поможет вам освоить основы Kafka, – журнал коммитов (фиксаций). Журнал коммитов основан на простой, но мощной идее. Многое станет очевиднее, если понять значение этого архитектурного выбора. Итак, журнал, о котором мы говорим, в корне отличается от привычных нам журналов, в которые приложения записывают разного рода сообщения, например, с помощью `LOGGER.логг` в Java.

На рис. 2.9 показано, насколько простой может быть концепция журнала коммитов, куда последовательно добавляются сообщения [8]. Несмотря на наличие других механизмов, используемых, например, для обработки сбоев брокера, эта базовая концепция является чрезвычайно важной для понимания Kafka. Журнал, применяемый в Kafka, – это не просто скрытая деталь, как в других системах, которые могут использовать что-то подобное (например, журнал упреждающей записи в базах данных). Он находится впереди и в центре, и его пользователи применяют смещения, чтобы знать, где они находятся в этом журнале.

Пример добавления двух сообщений (7 и 8) в тему, например `kinaction_alert` (см. главу 4)



Рис. 2.9. Журнал коммитов

Особенным журнал коммитов делает его доступность только для добавления, т. е. события всегда добавляются в конец журнала. Хранение сообщений в журнале – это основная черта, отличающая Kafka от других брокеров сообщений. Чтение сообщения не удаляет его из системы и не исключает из других источников.

В связи с этим возникает типичный вопрос: как долго могут храниться данные в Kafka? В настоящее время в различных компаниях нередко можно увидеть, что, когда размеры журналов превысят некоторый порог или будет достигнут предельный срок хранения, данные перемещаются в постоянное хранилище. Пороговые значения во многом зависят от емкости диска и особенностей рабочих процессов. В *New York Times*, например, используется всего один раздел размером менее 100 Гбайт [9]. Kafka устроена так, чтобы поддерживать высокую производительность и одновременно обеспечивать сохранность сообщений. Мы еще вернемся к деталим хранения сообщений в главе 6, когда будем говорить о брокерах. А сейчас просто запомните, что хранением данных в журнале можно управлять по возрасту или размеру с помощью свойств конфигурации.

2.4. Различные пакеты исходного кода, и что они делают

Kafka часто упоминается в названиях различных API. Есть также определенные компоненты, которые описываются как самостоятельные продукты. Далее мы рассмотрим некоторые из них, чтобы увидеть, что есть в нашем распоряжении. Пакеты, перечисленные в следующих разделах, – это API-интерфейсы, находящиеся в одном репозитории с исходным кодом ядра Kafka, за исключением ksqlDB [10].

2.4.1. Kafka Streams

Kafka Streams привлекает больше, чем даже ядро Kafka. Этот API находится в каталоге *streams* в исходном коде проекта Kafka, и большая его часть написана на Java. Одно из преимуществ Kafka Streams – отсутствие необходимости в отдельном кластере. Это легковесная библиотека, прекрасно подходящая для использования в приложениях. Она не требует установки дополнительного программного обеспечения для управления кластером или ресурсами, такого как Apache Hadoop, и обладает мощными возможностями, включая надежное хранение локального состояния, обработку сообщений по одному и поддержку семантики «точно один раз» [10]. Чем дальше вы будете продвигаться по этой книге, тем лучше будете понимать, как Kafka Streams API использует ядро Kafka для выполнения своей работы.

Этот API был создан с целью максимально упростить создание потоковых приложений и поддерживает цепочечный стиль написания кода, подобно Stream API в Java 8, также называемый предметно-ориентированным языком (Domain-Specific Language, DSL). Kafka Streams работает поверх ядра Kafka и добавляет, например, обработку с трассировкой состояния и распределенные соединения без особых сложностей или накладных расходов [10].

Организация микросервисов тоже оказалась под влиянием этого API. Вместо изоляции данных в различных приложениях они загружаются в приложения, которые могут использовать данные независимо. На рис. 2.10 показаны способы реализации системы микросервисов с применением и без применения Kafka (см. видео на YouTube «Microservices Explained by Confluent» [11]).

В варианте, изображенном в верхней части на рис. 2.10 (без Kafka), каждое приложение напрямую взаимодействует с другими приложениями через несколько интерфейсов, в нижней части показано решение с Kafka. Применение Kafka не только позволяет передавать данные всем приложениям без предварительной обработки какой-либо службой, но также предоставляет единый интерфейс для всех приложений. Главное преимущество отсутствия прямых связей между приложениями при использовании Kafka – ослабление зависимостей между конкретными приложениями.

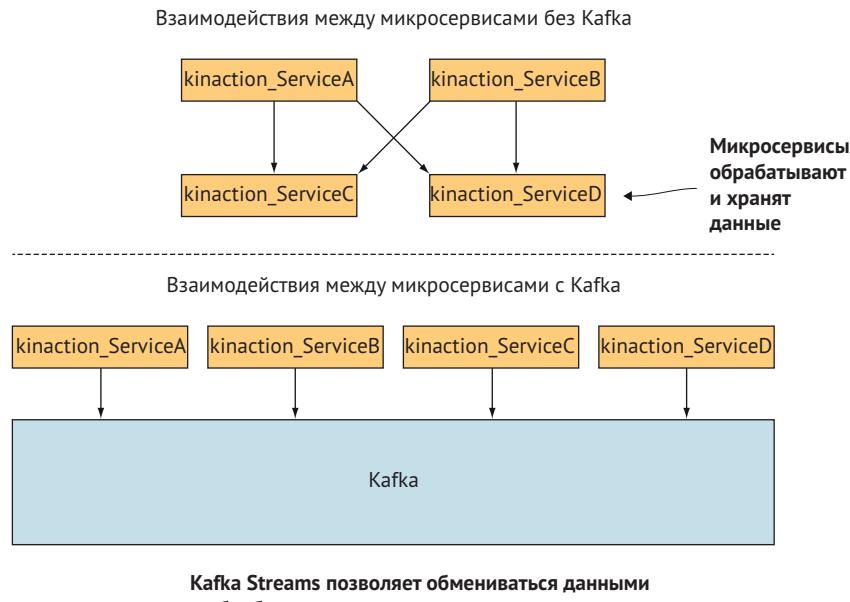


Рис. 2.10. Организация микросервисов

2.4.2. Kafka Connect

Исходный код Kafka Connect находится в папке *connect* в исходных кодах Kafka и тоже в основном написан на Java. Этот фреймворк был создан для упрощения интеграции с другими системами [10]. Во многих случаях он способен заменить другие инструменты, такие как Apache Gobblin™ и Apache Flume. Если вы знакомы с Flume, то некоторые используемые термины, вероятно, покажутся вам знакомыми.

Для импорта данных в Kafka используются так называемые коннекторы-источники. Допустим, чтобы переместить данные из таблиц MySQL® в темы Kafka, нужно использовать Connect для преобразования этих данных в сообщения Kafka. С другой стороны, имеются также коннекторы-приемники, используемые для экспорта данных из Kafka в другие системы. Например, переместить сообщения из какой-то темы в долговременное хранилище можно с помощью коннектора-приемника, который будет извлекать сообщения из темы и записывать их, скажем, в облачное хранилище. На рис. 2.11 для примера показана организация потока данных из базы данных в Connect, а затем в облачное хранилище, как описано в статье «The Simplest Useful Kafka Connect Data Pipeline in the World...or Thereabouts – Part 1» [12].



Рис. 2.11. Пример использования Connect

Следует отметить, что прямая замена Apache Flume, вероятно, не была намерением или основной целью Kafka Connect. Kafka Connect не имеет агента для настройки каждого узла Kafka и предназначен для интеграции с платформами потоковой обработки данных. В целом Kafka Connect – отличный выбор для создания быстрых и простых конвейеров данных, связывающих системы.

2.4.3. Пакет AdminClient

Недавно Kafka представила AdminClient API. До появления этого API сценарии и другие программы, выполняющие определенные административные действия, должны были запускать сценарии командной оболочки (также входящие в состав Kafka) или вызы-

вать внутренние классы, часто используемые этими сценариями оболочки. Этот API является частью файла *kafka-clients.jar*, который отличается от других JAR-файлов, обсуждавшихся ранее. AdminClient API – отличный инструмент для администрирования Kafka [10]. Он использует аналогичную конфигурацию, которую применяют производители и потребители. Исходный код можно найти в пакете *org/apache/kafka/clients/admin*.

2.4.4. *ksqldb*

В конце 2017 года в Confluent выпустили предварительную версию для разработчиков нового механизма поддержки SQL для Kafka с названием KSQL. Впоследствии он был переименован в ksqlDB. Этот инструмент дает разработчикам и аналитикам данных, использующим в своей работе SQL, возможность использовать потоки с помощью привычных и хорошо знакомых интерфейсов. Хотя синтаксис может показаться знакомым, в нем все же есть существенные различия.

Большинство запросов, хорошо знакомых пользователям реляционных баз данных, предполагают выполнение поиска и носят пакетный характер. Сдвиг в сторону запросов, возвращающих потоки данных, требует иного мышления от разработчиков. По аналогии с Kafka Streams API ksqlDB упрощает использование непрерывных потоков данных. Интерфейс будет выглядеть как знакомая инженерам данных SQL-подобная грамматика, однако эта грамматика порождает постоянно действующие и обновляющиеся запросы. Эти запросы с успехом можно использовать, например, в информационных панелях, которые смогут заменить приложения, выполняющие операторы SELECT через регулярные интервалы времени.

2.5. Клиенты Confluent

Благодаря растущей популярности Kafka выбор языка программирования для взаимодействия с Kafka перестал быть проблемой. В наших упражнениях и примерах мы будем использовать клиентов на Java, созданных в рамках основного проекта Kafka. Однако множество других клиентов можно найти в проекте Confluent [13].

Поскольку клиенты отличаются своими возможностями, на сайте проекта Confluent приводится таблица поддерживаемых функций в разных языках программирования: <https://docs.confluent.io/current/clients/index.html>. Кстати, изучение клиентов с открытым исходным кодом может помочь вам разработать собственного клиента или даже выучить новый язык.

Поскольку использование клиентов является наиболее предпочтительным способом взаимодействия с Kafka в приложениях, рассмотрим использование клиента на Java (листинг 2.9). Он реализу-

ет тот же процесс создания и потребления данных, который был показан в примере с использованием командной строки выше. Добавив немного шаблонного кода (здесь не приводится, чтобы не отвлекать внимание от особенностей работы с Kafka), вы сможете запустить этот код в основном методе Java для создания сообщения.

Листинг 2.9. Клиент-производитель на Java

```
public class HelloWorldProducer {
    public static void main(String[] args) {
        Properties kaProperties =
            new Properties(); ←
        kaProperties.put("bootstrap.servers",
            "localhost:9092,localhost:9093,localhost:9094"); ←
        kaProperties.put("key.serializer",
            "org.apache.kafka.common.serialization.StringSerializer");
        kaProperties.put("value.serializer",
            "org.apache.kafka.common.serialization.StringSerializer");

        try (Producer<String, String> producer =
            new KafkaProducer<>(kaProperties)) ←
            ProducerRecord<String, String> producerRecord =
                new ProducerRecord<>("kinaction_helloworld",
                    null, "hello world again!"); ←
            producer.send(producerRecord); ←
        } ←
    }
}
```

Производитель принимает массив пар имя/значение для настройки различных параметров

Это свойство может принимать список брокеров Kafka

Определяет формат для сериализации ключа сообщения и значения

Создает экземпляра производителя. Производители реализуют интерфейс, который автоматически закрывается средой выполнения Java

Собственно сообщение

Отправляет запись брокеру Kafka

В листинге 2.9 представлен простой производитель. Первый шаг в создании производителя включает настройку конфигурационных свойств. Свойства настроены так, чтобы было удобно всем, кто привык пользоваться ассоциативными массивами.

Параметр `bootstrap.servers` является важным элементом конфигурации, и его назначение может быть неочевидным на первый взгляд. В нем определяется список брокеров Kafka. Однако этот список не обязательно должен включать все имеющиеся серверы, потому что после подключения клиент сам найдет информацию об остальных брокерах в кластере и не будет зависеть от этого списка.

Параметры `key.serializer` и `value.serializer` также играют важную роль. В них мы должны указать класс, который будет отвечать за сериализацию данных при их перемещении в Kafka. Ключи и значения могут использовать разные сериализаторы.

На рис. 2.12 показан поток, в котором производитель отправляет сообщение. Созданный нами производитель принимает свойства

конфигурации в виде аргументов конструктора. После создания производителя мы можем отправлять сообщения. `ProducerRecord` содержит фактическое сообщение, которое мы хотим отправить. В наших примерах `kinaction_helloworld` – это название темы. Другие аргументы представляют ключ сообщения и его значение. Подробнее о ключах мы поговорим в главе 4, а пока просто запомните, что они действительно могут иметь значение `null`. Это делает наш пример менее сложным.

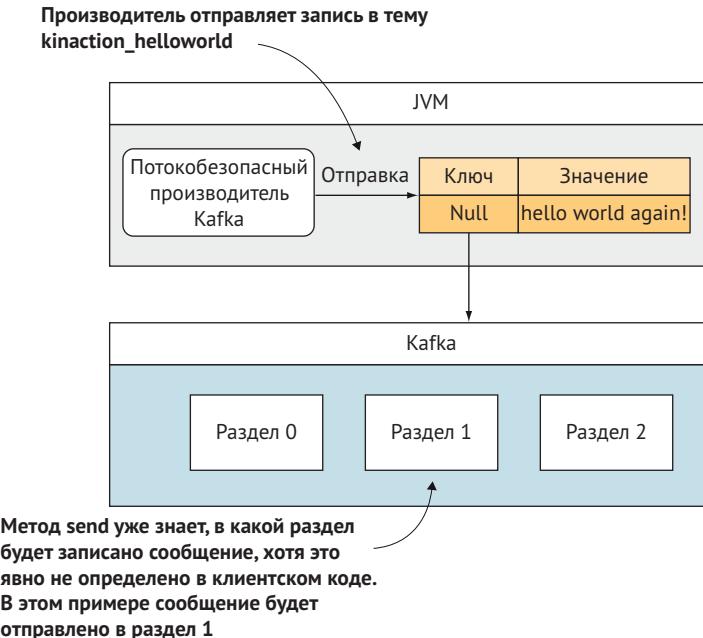


Рис. 2.12. Поток производителя

Сообщение, отправляемое в последнем аргументе, отличается от первого, которое мы отправили нашему производителю из консоли. Знаете, почему мы решили послать сообщение с другим текстом? Мы продолжаем использовать ту же тему, и, поскольку далее будем использовать нового потребителя, он извлечет старое сообщение, отправленное нами ранее. Подготовив сообщение, мы асинхронно отправляем его с помощью производителя. В этом случае, поскольку отправляется только одно сообщение, производитель дождется завершения передачи ранее отправленных запросов, а затем завершит работу.

Прежде чем запускать эти примеры Java-клиентов, нужно убедиться, что в нашем файле `root.xml` [14] присутствуют настройки, показанные в листинге 2.10. Все примеры в этой книге собираются с помощью Apache Maven™.

Листинг 2.10. Настройки Java-клиента

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>2.7.1</version>
</dependency>
```

Теперь, создав новое сообщение, воспользуемся нашим Java-клиентом, представленным в листинге 2.11, чтобы создать потребителя, извлекающего сообщение. Этот код можно вызвать из метода `main` и завершить программу после чтения сообщений.

Листинг 2.11. Клиент-потребитель

```
public class HelloWorldConsumer {

    final static Logger log =
        LoggerFactory.getLogger(HelloWorldConsumer.class);

    private volatile boolean keepConsuming = true;

    public static void main(String[] args) {
        Properties kaProperties = new Properties(); ←
        kaProperties.put("bootstrap.servers",
            "localhost:9092,localhost:9093,localhost:9094");
        kaProperties.put("group.id", "kinaction_helloconsumer");
        kaProperties.put("enable.auto.commit", "true");
        kaProperties.put("auto.commit.interval.ms", "1000");
        kaProperties.put("key.deserializer",
            "org.apache.kafka.common.serialization.StringDeserializer");
        kaProperties.put("value.deserializer",
            "org.apache.kafka.common.serialization.StringDeserializer");

        HelloWorldConsumer helloWorldConsumer = new HelloWorldConsumer();
        helloWorldConsumer.consume(kaProperties);
        Runtime.getRuntime().addShutdownHook(new Thread(helloWorldConsumer::shutdown));
    }

    private void consume(Properties kaProperties) {
        try (KafkaConsumer<String, String> consumer =
            new KafkaConsumer<>(kaProperties)) {
            consumer.subscribe(
                List.of(
                    "kinaction_helloworld" ←
                )
            );

            while (keepConsuming) {
                ConsumerRecords<String, String> records =
                    consumer.poll(Duration.ofMillis(250)); ←
                Проверка появления
                Проверка появления
                новых сообщений
            }
        }
    }
}
```

Конфигурационные
свойства устанавлива-
ются так же, как в про-
изводителе

Потребитель сообщает Kafka, какие
темы его интересуют

Проверка появления
новых сообщений

```

for (ConsumerRecord<String, String> record :
    Для нагляд-
    ности каждая
    полученная
    запись выво-
    дится в консоль
    records) {
    log.info("kinaction_info offset = {}, kinaction_value = {}",
            record.offset(), record.value());
}
}

private void shutdown() {
    keepConsuming = false;
}
}

```

Первое, что бросается в глаза в листинге 2.11, – это бесконечный цикл. Намеренное применение бесконечного цикла выглядит странным, но давайте не будем забывать, что мы намереваемся обрабатывать бесконечный поток данных. Потребитель, подобно производителю, использует ассоциативный массив конфигурационных свойств. Однако, в отличие от производителя, клиент-потребитель не является потокобезопасным [15]. Это необходимо учитывать при масштабировании и использовании нескольких потребителей, как будет показано в последующих разделах. Наш код должен гарантировать синхронизацию доступа: один простой вариант – создавать не более одного потребителя в каждом потоке выполнения Java. Кроме того, подобно тому, как мы указали производителю, куда отправлять сообщения, мы указываем темы, на которые должен подписаться потребитель. В вызове `subscribe` можно указать сразу несколько тем.

Также в листинге 2.11 следует обратить внимание на вызов метода `poll`. Этим вызовом мы активно пытаемся извлечь сообщения для передачи в приложение. Вызов `poll` может не вернуть ни одного, или одно, или несколько сообщений, и наша логика должна учитывать это.

Наконец, после получения всех сообщений мы можем нажать **Ctrl-C** в программе-потребителе и на этом закончить. Следует отметить, что эти примеры основаны на многих конфигурационных свойствах со значениями по умолчанию. У нас еще будет возможность покопаться в них в следующих главах.

2.6. Потоковая обработка и терминология

Мы не собираемся оспаривать теорию распределенных систем или отдельные определения, которые могут иметь различный смысл, а просто посмотрим, как работает Kafka. Начав думать о применении Kafka в своей работе, вы столкнетесь со следующими терминами и, надеюсь, сможете использовать следующие описания для переосмыслиния своего понимания обработки.

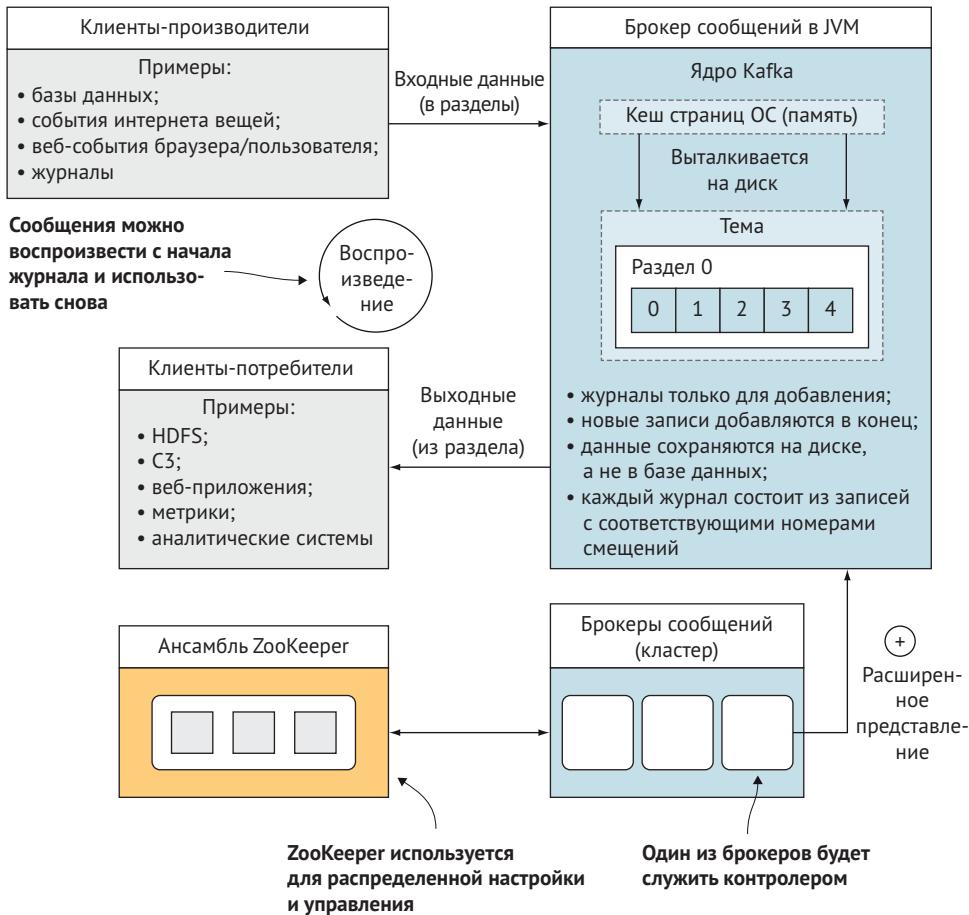


Рис. 2.13. Обзор Kafka

На рис. 2.13 в общем виде показано, что делает Kafka. В Kafka много движущихся частей, которые зависят от данных, поступающих в ядро и исходящих из него, и обеспечивают ценность этой платформы для пользователей. Производители отправляют данные в Kafka, которая для надежности и масштабируемости действует как распределенная система и использует журналы как основные хранилища. Как только данные попадут в экосистему Kafka, потребители смогут помочь пользователям получить их в других приложениях и сценариях использования. Наши брокеры объединяются в кластер и координируют свои действия с помощью ZooKeeper. Поскольку Kafka хранит данные на диске, поддержка воспроизведения сообщений в случае сбоя приложения также является частью набора функций Kafka. Все это позволяет Kafka стать основой мощных приложений потоковой обработки.

2.6.1. Потоковая обработка

Потоковая обработка имеет разные определения в разных проектах. Основной принцип потоковой передачи данных заключается в том, что данные продолжают поступать и не заканчиваются [16]. Кроме того, наш код должен постоянно обрабатывать эти данные, а не ждать запроса или начала некоторого периода времени для запуска. Как мы видели выше, на этот постоянный поток данных намекает бесконечный цикл в нашем коде, который не имеет определенной точки завершения.

Этот подход не объединяет данные в пакеты для последующей обработки и не предполагает запуск пакетной обработки по ночам или раз в месяц. Если представить нескончаемый водопад, то здесь действуют те же принципы. Иногда требуется передать огромное количество данных, а иногда не так много, но в обоих случаях они передаются между пунктами назначения непрерывно.

На рис. 2.14 показано, что Kafka Streams API зависит от ядра Kafka. Пока сообщения продолжают поступать в кластер, приложение-потребитель может постоянно передавать конечному пользователю обновленную информацию, а не ждать запроса на получение статического снимка событий. Больше не нужно обновлять веб-страницу каждые пять минут, чтобы пользователи могли увидеть последние события!

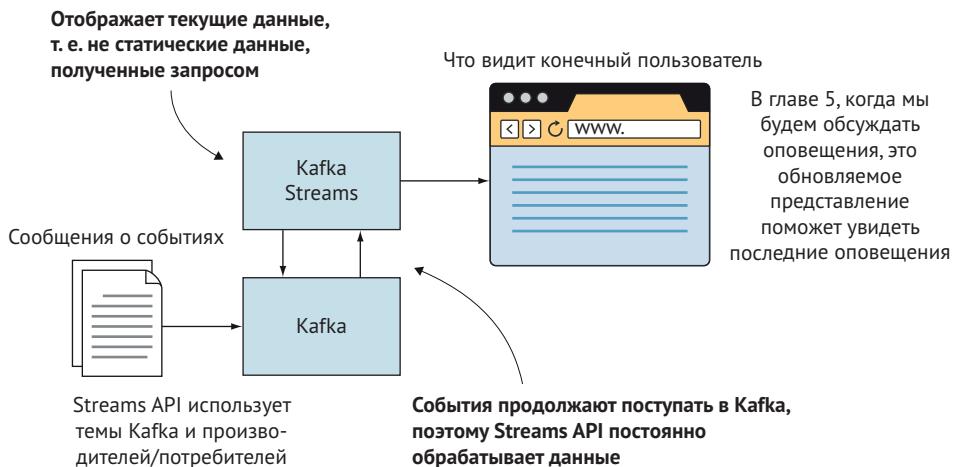


Рис. 2.14. Потоковая обработка

2.6.2. Что означает семантика «точно один раз»

Одной из самых захватывающих и, пожалуй, наиболее обсуждаемых особенностей Kafka является семантика «точно один раз».

Мы не будем обсуждать теорию, стоящую за ней, но коснемся ее значения для повседневного использования Kafka.

Важно отметить, что самый простой способ поддержки семантики «точно один раз» – оставаться в стенах (и темах) Kafka. Закрытая система может действовать как транзакция. Вот почему использование Streams API – один из самых простых путей к семантике «точно один раз». Различные коннекторы Kafka Connect тоже поддерживают семантику «точно один раз» и являются отличными примерами извлечения данных из Kafka, поскольку они не всегда будут играть роль конечных точек для всех данных во всех сценариях.

Итоги

- Сообщения представляют ваши данные в Kafka. Кластер брокеров Kafka обрабатывает эти данные и взаимодействует с внешними системами и клиентами.
- Использование журнала фиксаций в Kafka объясняет особенности работы системы в целом.
- Сообщения добавляются в конец журнала. Это определяет способы хранения и повторного использования данных. Имея возможность начать извлекать сообщения с начала журнала, приложения могут повторно обрабатывать данные в определенном порядке.
- Производители – это клиенты, помогающие перемещать данные в экосистему Kafka. Передача существующей информации в Kafka из других источников, таких как базы данных, может помочь экспорттировать данные, которые когда-то были разбросаны по разным системам, предоставившим интерфейс доступа к данным для других приложений.
- Потребители – это клиенты, получающие сообщения из Kafka. Разные потребители могут читать одни и те же данные одновременно. Способность разных потребителей начинать читать с разных позиций также показывает гибкость потребления, возможную благодаря темам Kafka.
- Непрерывный поток данных между пунктами назначения, передаваемый с помощью Kafka, может помочь перепроектировать системы, которые раньше были ограничены пакетными или отложенными рабочими процессами.

Ссылки

- 1 «Main Concepts and Terminology». Apache Software Foundation (n.d.). https://kafka.apache.org/documentation.html#intro_concepts_and_terms (доступно по состоянию на 22 мая 2019).

- 2 «Apache Kafka Quickstart». Apache Software Foundation (2017). <https://kafka.apache.org/quickstart> (доступно по состоянию на 15 июля 2020).
- 3 B. Kernighan and D. Ritchie. «The C Programming Language, 1st ed.». Englewood Cliffs, NJ, USA: Prentice Hall, 1978².
- 4 KIP-500: «Replace ZooKeeper with a Self-Managed Metadata Quorum». Wiki for Apache Kafka. Apache Software Foundation (9 июля 2020). <https://cwiki.apache.org/confluence/display/KAFKA/KIP-500%3A+Replace+ZooKeeper+with+a+Self-Managed+Metadata+Quorum> (доступно по состоянию на 22 августа 2020).
- 5 «ZooKeeper Administrator’s Guide». Apache Software Foundation. (n.d.). <https://zookeeper.apache.org/doc/r3.4.5/zookeeperAdmin.html> (доступно по состоянию на 10 июня 2020).
- 6 «Kafka Design: Persistence». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/kafka/design.html#persistence> (доступно по состоянию на 19 ноября 2020).
- 7 «A Guide To The Kafka Protocol: Some Common Philosophical Questions». Wiki for Apache Kafka. Apache Software Foundation (n.d.). <https://cwiki.apache.org/confluence/display/KAFKA/A+Guide+To+The+Kafka+Protocol#AGuideToTheKafkaProtocol-SomeCommonPhilosophicalQuestions> (доступно по состоянию на 21 августа 2019).
- 8 «Documentation: Topics and Logs». Apache Software Foundation (n.d.). https://kafka.apache.org/23/documentation.html#intro_topics (доступно по состоянию на 25 мая 2020).
- 9 B. Svingen. «Publishing with Apache Kafka at The New York Times». Блог Confluent (6 сентября 2017). <https://www.confluent.io/blog/publishing-apache-kafka-new-york-times/> (доступно по состоянию на 25 сентября 2018).
- 10 «Documentation: Kafka APIs». Apache Software Foundation (n.d.). https://kafka.apache.org/documentation.html#intro_apis (доступно по состоянию на 15 июня 2021).
- 11 «Microservices Explained by Confluent». Confluent. Веб-презентация (23 августа 2017). <https://youtu.be/aWI7iU36qv0> (доступно по состоянию на 9 августа 2021).
- 12 R. Moffatt. «The Simplest Useful Kafka Connect Data Pipeline in the World...or Thereabouts – Part 1». Блог Confluent (11 августа 2017). <https://www.confluent.io/blog/simplest-useful-kafka-connect-data-pipeline-world-thereabouts-part-1/> (доступно по состоянию на 17 декабря 2017).

² Ритчи Деннис М., Керниган Брайан У., «Язык программирования С», Вильямс, 2017, ISBN: 978-5-8459-1874-1, 0-13-110362-8, 978-5-8459-1975-5. – Прим. перев.

- 13 «Kafka Clients». Confluent documentation (n.d.). <https://docs.confluent.io/current/clients/index.html> (доступно по состоянию на 15 июня 2020).
- 14 «Kafka Java Client». Confluent documentation (n.d.). <https://docs.confluent.io/clients-kafka-java/current/overview.html> (доступно по состоянию на 21 июня 2021).
- 15 «Class KafkaConsumer<K,V>». Apache Software Foundation (9 ноября 2019). <https://kafka.apache.org/24/javadoc/org/apache/kafka/clients/consumer/KafkaConsumer.html> (доступно по состоянию на 20 ноября 2019).
- 16 «Streams Concepts». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/streams/concepts.html> (доступно по состоянию на 17 июня 2020).

Часть II

Практическое применение Kafka

Во второй части мы будем опираться на нашу ментальную модель Kafka, сформированную в первой части, и пробовать применять знания на практике. Мы рассмотрим основы Kafka и начнем с обзора клиентов производителей и потребителей. Даже если вы планируете разрабатывать только приложения с Kafka Streams или ksqlDB, информация из части II все равно пригодится. Основные обсуждение в этой части будут сосредоточено на высокогорневых библиотеках и абстракциях, используемых в экосистеме Kafka:

- в главе 3 мы начнем разработку примера проекта и узнаем, как применить Kafka на практике. Хотя схемы подробно рассматриваются в главе 11, наши проектные требования явно указывают на необходимость их разработки уже на ранних стадиях проектирования данных;
- в главе 4 мы подробно расскажем, как можно использовать производителей для перемещения данных в Kafka. Здесь же обсудим важные конфигурационные параметры и их влияние на источники данных;

- в главе 5 углубимся в потребление данных из Kafka с помощью потребителей. Здесь мы определим сходства и различия между клиентами-потребителями и производителями из главы 4;
- в главе 6 начнем рассматривать роль брокеров. В этой главе рассматриваются роли лидеров и контролеров, а также их отношения с клиентами;
- в главе 7 мы рассмотрим особенности сочетания тем и разделов для предоставления данных. Здесь также будут представлены сжатые темы;
- в главе 8 мы исследуем инструменты и архитектуры, реализующие разные варианты обработки данных, которые необходимо сохранить или повторно обработать;
- в главе 9, завершающей часть II, мы рассматриваем основные журналы и метрики, использование которых поможет администраторам поддерживать работоспособность кластеров.

К концу части II вы получите четкое представление об основных элементах Kafka и как использовать эти элементы в своих сценариях. А теперь приступим!

3

Разработка проекта на основе Kafka

Эта глава охватывает следующие темы:

- разработку практического проекта на основе Kafka;
- определение используемого формата данных;
- существующие проблемы использования данных;
- выбор решения о преобразовании данных;
- как Kafka Connect помогает перейти на потоковую передачу данных.

В предыдущей главе мы увидели, как работать с Kafka из командной строки и использовать клиентов на Java. Теперь расширим эти первые идеи и рассмотрим приемы разработки различных решений с помощью Kafka. Мы обсудим некоторые вопросы, которые необходимо учитывать, начав разработку примера проекта. Когда приступим к разработке наших решений, имейте в виду, что, как и в большинстве проектов, мы можем вносить небольшие изменения по ходу работы и просто искать место, откуда начать разработку. Прочитав эту главу, вы познакомитесь с реальными сценариями использования и одновременно с приемами проектирования, что облегчит вам дальнейшее изучение Kafka в оставшейся части этой книги. Давайте начнем этот увлекательный путь!

3.1. Разработка проекта на основе Kafka

Новые компании и проекты могут использовать Kafka с самого начала, но это относится не ко всем компаниям. Для тех из нас, кто работал в корпоративной среде или с устаревшими системами (а в наши дни все, что старше пяти лет, можно считать устаревшим), начинать с нуля не всегда возможно. И все же работа с существующими архитектурами имеет свои преимущества, и одно из них заключается в том, что мы имеем список проблем, которые можно разрешить. Контраст также помогает нам подчеркнуть сдвиг в восприятии данных в нашей работе. В этой главе мы будем работать над проектом для компании, готовой отказаться от своего текущего способа обработки данных и применить новый инструмент с названием Kafka.

3.1.1. Использование существующей архитектуры данных

Давайте рассмотрим предысторию вопроса, чтобы заложить фундамент нашего будущего примера и понять причины постоянно растущей популярности Kafka. В статье о Confluent (<https://www.confluent.io/use-case/internet-of-things-iot/>), упоминавшейся в главе 1, а также в замечательной статье Джанакирама МСВ (Janakiram MSV) под названием «Apache Kafka: The Cornerstone of an Internet-of-Things Data Platform» рассказывается об использовании Kafka для сбора информации с датчиков [1]. Далее мы возьмем тему датчиков в качестве варианта использования и углубимся в вымышленный пример проекта.

Наша вымышленная консалтинговая компания только что выиграла тендер на помощь в реконструкции завода, выпускающего электровелосипеды. По всей линии производства велосипедов установлено множество датчиков, постоянно сообщающих о состоянии агрегатов, узлов и механизмов. Однако датчики генерируют так много событий, что текущая система просто не справляется с их потоком и теряет большинство сообщений. Нас попросили помочь владельцам завода раскрыть потенциал этих данных для их использования в различных приложениях. Кроме того, текущая инфраструктура данных включает большие и кластеризованные системы реляционных баз данных. Как, имея такое количество датчиков и существующую базу данных, создать новую архитектуру на основе Kafka, не останавливая производство?

3.1.2. Первый шаг

Очевидно, что подход «большого взрыва» не лучший выбор. Мы не должны перемещать сразу все данные в Kafka. Если сегодня используется база данных, а завтра мы решим дать зеленый свет потоковым данным, то проще всего начать с внедрения Kafka Con-

nect. Этот фреймворк вполне способен справиться с промышленными нагрузками. Мы возьмем одну таблицу базы данных и запустим в работу нашу новую архитектуру, позволив существующим приложениям работать без изменений. Но сначала рассмотрим несколько примеров, чтобы поближе познакомиться с Kafka Connect.

3.1.3. Встроенные возможности

Цель Kafka Connect – помочь в перемещении данных в Kafka или из нее без создания своих производителей и потребителей. Con-nect – это фреймворк, уже являющийся частью Kafka, что упрощает переход на работу с потоковыми данными с применением готовых компонентов. Эти компоненты называются *коннекторами* и предназначены для надежной работы с другими источниками данных [2].

В главе 2 приводились примеры кода производителей и потребителей на Java из реального мира, показывающие, как Connect абстрагирует и использует эти понятия. Поэтому мы пойдем простым путем: посмотрим, как Connect берет обычный файл журнала приложения и перемещает его в тему Kafka. Для простоты мы запустим и протестируем Connect на локальном компьютере. Проблемой масштабирования можно заняться позже, если нам понравится, как работает решение в автономном режиме! В папке установки Kafka найдите следующие файлы в каталоге *config*:

- *connect-standalone.properties*;
- *connect-file-source.properties*.

Заглянув внутрь файла *connect-standalone.properties*, вы увидите некоторые конфигурационные ключи и значения – они должны выглядеть похожими на свойства, которые мы использовали при создании своих клиентов на Java в главе 2. Знание основ производителей и потребителей поможет нам понять, как Connect использует ту же конфигурацию в своей работе, перечисляющую такие настройки, как *bootstrap.servers*.

В нашем примере мы будем извлекать данные из единственного источника и помещать их в Kafka, чтобы потом можно было обрабатывать их как полученные из файла. Используя шаблон файла *connect-file-source.properties*, входящий в состав дистрибутива Kafka в качестве примера, создадим файл с именем *alert-source.properties* и заменим его содержимое кодом из листинга 3.1. Этот файл определяет параметры, необходимые для настройки файла *alert.txt* и передачи данных в конкретную тему *kinaction_alert_connect*. Обратите внимание, что в этом примере выполняются те же шаги, что описаны в прекрасном руководстве по быстрому запуску Connect (<https://docs.confluent.io/3.1.2/connect/quickstart.html>),

к которому вы можете обращаться за дополнительными подробностями. Еще больше подробностей вы найдете в замечательной презентации Рэндалла Хауха (Randall Hauch; одного из разработчиков Apache Kafka и PMC), продемонстрированной им на саммите Kafka (Сан-Франциско, 2018) и доступной по адресу <http://mng.bz/8WeD>.

С помощью конфигураций (не написав ни строчки программного кода) мы можем переместить данные в Kafka из любого файла. Поскольку чтение из файла является обычной задачей, можно использовать готовые классы Connect. В данном случае это класс `FileStreamSource` [2]. Рассматривая листинг 3.1, представьте, что у нас есть приложение, которое записывает уведомления в текстовый файл.

Листинг 3.1. Настройка Connect для извлечения данных из файла

```
name=alert-source
connector.class=FileStreamSource ← Определяет класс, обрабатывающий
tasks.max=1 ← наш файл-источник
file=alert.txt ← Для работы тестового
topic=kinaction_alert_connect ← примера в автономном
                                режиме значения 1 более
                                чем достаточно
                                Имя темы для
                                передачи данных
```

Значение свойства `topic` играет важную роль. Мы используем его позже, чтобы убедиться, что сообщения извлекаются из файла в конкретную тему `kinaction_alert_connect`. Файл `alert.txt` постоянно проверяется на наличие изменений и появление в нем новых сообщений. И наконец, мы выбрали 1 как значение свойства `tasks.max`, потому что нам достаточно одной задачи для нашего коннектора, и нет необходимости использовать возможности параллельной обработки.

ПРИМЕЧАНИЕ. Запуская ZooKeeper и Kafka локально, убедитесь, что ваши брокеры Kafka все еще выполняются (на случай, если вы остановили их, закончив читать предыдущую главу).

Теперь, выполнив необходимые настройки, нужно запустить Connect и отправить нашу конфигурацию. Запустить процесс Connect можно с помощью сценария командной оболочки `connect-standalone.sh`, передав ему наш файл конфигурации в качестве параметра. Чтобы запустить Connect в терминале, выполните команду, как показано в листинге 3.2, и оставьте ее выполняться [2].

Листинг 3.2. Запуск Connect для обработки файла-источника

```
bin/connect-standalone.sh config/connect-standalone.properties \
alert-source.properties
```

Откройте другое окно терминала, создайте текстовый файл с именем *alert.txt* в каталоге, в котором вы запустили службу Connect, и добавьте в этот файл пару строк с помощью текстового редактора; текст может быть каким угодно. Теперь выполним команду `console-consumer`, чтобы убедиться, что Connect выполняет свои обязанности. Для этого откройте еще одно окно терминала и проверьте содержимое темы *kinaction_alert_connect*, как показано в листинге 3.3. Connect должен принять содержимое этого файла и передать данные в Kafka [2].

Листинг 3.3. Проверка передачи содержимого файла в Kafka

```
bin/kafka-console-consumer.sh \
--bootstrap-server localhost:9094 \
--topic kinaction_alert_connect --from-beginning
```

Прежде чем перейти к использованию коннектора другого типа, давайте кратко рассмотрим коннектор приемника и поговорим о том, как он переносит сообщения из Kafka в другой файл. Поскольку местом назначения (или приемником) для наших данных является другой файл, мы должны определить настройки еще в одном файле. Его содержимое показано в листинге 3.4. Он определяет параметры записи данных в файл-приемник. Назначим на роль коннектора-приемника класс `FileStreamSink`. Тема *kinaction_alert_connect* теперь будет служить нам источником данных. Поместите текст из листинга 3.4 в новый файл с именем *alert-sink.properties*, чтобы настроить нашу новую конфигурацию [2].

Листинг 3.4. Настройка Connect для записи данных в файл

```
name=alert-sink
connector.class=FileStreamSink
tasks.max=1
file=alert-sink.txt
topics=kinaction_alert_connect
```

The diagram shows the `alert-sink.properties` file with annotations pointing to specific parameters:

- `name=alert-sink`: Определяет класс, выполняющий передачу данных из Kafka в файл-приемник.
- `connector.class=FileStreamSink`: Для работы тестового примера в автономном режиме значения 1 более чем достаточно.
- `tasks.max=1`: Имя темы, откуда должны извлекаться данные.
- `file=alert-sink.txt`: Файл, куда должны копироваться данные из нашей темы в Kafka.
- `topics=kinaction_alert_connect`: Тема, откуда должны извлекаться данные.

Если экземпляр Connect все еще работает в терминале, то закройте это окно терминала или остановите процесс, нажав **Ctrl-C**. Затем вновь запустите его с конфигурационными файлами *alert-source.properties* и *alert-sink.properties*, как показано в листинге 3.5 [2]. В результате данные из файла-источника должны переместиться в Kafka, а оттуда – в файл-приемник.

Листинг 3.5. Запуск Connect для обработки файлов источника и приемника

```
bin/connect-standalone.sh config/connect-standalone.properties \
    alert-source.properties alert-sink.properties
```

Чтобы убедиться, что Connect скопировал данные из одного файла в другой, откройте файл приемника, указанный в конфигурации *alert-sink.txt*, и проверьте, присутствуют ли в нем те же сообщения, что вы сохранили в файле-источнике, а также в теме Kafka.

3.1.4. Данные для наших накладных

Рассмотрим еще одно требование, связанное с накладными на заказы велосипедов. Connect позволяет тем, кто хорошо разбирается в создании настраиваемых коннекторов, совместно использовать их с другими (и помочь тем, кто не является экспертом в этих системах). Теперь, получив небольшой опыт использования коннектора (листинги 3.4 и 3.5), вы сможете без особого труда настроить еще один коннектор, поскольку все взаимодействия Connect с другими системами стандартизированы.

Чтобы использовать Connect в нашем примере, рассмотрим применение готового коннектора-источника, который передает изменения из локальной базы данных в тему Kafka. И снова наша цель не в том, чтобы сразу изменить всю архитектуру обработки данных, а просто посмотреть, как можно копировать изменения из реляционной базы данных и параллельно разрабатывать новое приложение, позволяя существующей системе продолжать работать по-старому. Обратите внимание, что в этом примере выполняются шаги, описанные в руководстве, доступном по адресу <https://docs.confluent.io/kafka-connect-jdbc/current/source-connector/index.html>.

Первым шагом настроим базу данных для наших примеров. Для простоты используем коннекторы из Confluent для работы с SQLite. Если вы можете прямо сейчас запустить команду `sqlite3` в своем терминале и получить приглашение к вводу, значит подделка уже сделана. Иначе установите версию SQLite для своей операционной системы.

СОВЕТ. Прочтайте содержимое файла *Commands.md* в исходном коде примеров для этой главы. Там вы найдете инструкции по установке интерфейса командной строки Confluent, а также коннектора JDBC с использованием `confluent-hub`. В остальных примерах мы будем использовать команды только из каталога установки Confluent, а не из каталога установки Kafka.

Чтобы создать базу данных, запустите команду `sqlite3 kafkatest.db` из командной строки. Затем в строке приглашения к вводу этой базы данных выполните код из листинга 3.6, чтобы создать табли-

цу накладных (*invoices*) и добавить в нее некоторые тестовые данные. Проектируя таблицы, всегда следует подумать о том, как будут фиксироваться изменения в Kafka. В большинстве случаев после первичной загрузки нет необходимости копировать всю базу данных, достаточно скопировать только изменения. Определить, какие данные изменились и должны быть отправлены в Kafka, можно с помощью полей с отметкой времени, порядковым номером или идентификатором. В нашем случае отличить изменившиеся записи нам помогут поля `ID` и `modified`, как показано в листинге 3.6 [3].

Листинг 3.6. Создание таблицы invoices

Создав файл `etc/kafka-connect-jdbc/kafka-test-sqlite.properties` и внося небольшие изменения в таблицу в базе данных, вы сможете увидеть, как добавление новых и изменение существующих записей вызывают отправку сообщений в Kafka. Те из вас, кому необходимы подробные инструкции по поиску и созданию файлов с настройками для коннектора JDBC в каталоге установки Confluent, смогут найти их в примерах исходного кода к главе 3 в репозитории Git. Имейте в виду, что этот коннектор не является частью Apache Kafka, в отличие от коннекторов для работы с файлами. Кроме того, если при изменении формата поля с отметкой времени начнут возникать ошибки, то обязательно проверьте другие параметры в исходном коде для этой главы.

Теперь, определив новую конфигурацию, запустим Connect и передадим ему файл с настройками *kafka-test-sqlite.properties*.

Листинг 3.7. Запуск Connect для обработки таблицы в базе данных в качестве источника

```
confluent-hub install confluentinc/kafka-connect-jdbc:10.2.0  
confluent local services connect start
```

3

```
# Описание других шагов смотрите в файле Commands.md
confluent local services connect connector config jdbc-source
--config etc/kafka-connect-jdbc/kafkatest-sqlite.properties
```

Используется новый файл с настройками для обработки базы данных

В листинге 3.7 показано, как запустить Connect с помощью инструмента командной строки Confluent. Также можно было бы использовать сценарий `connect-standalone.sh` [3]. Фреймворк Kafka Connect отлично подходит для переноса существующих таблиц базы данных в Kafka, однако для наших датчиков (которые не поддерживаются базой данных) необходим другой метод.

3.2. События датчиков

Поскольку для наших суперсовременных датчиков нет готовых коннекторов, мы можем организовать прямые взаимодействия с их системой событий, используя свои реализации производителей. Приемы реализации и подключения своих производителей для отправки данных в Kafka мы рассмотрим в следующих разделах.

На рис. 3.1 показано, что существует критическая последовательность этапов производства. Один из шагов, которые мы должны выполнить при обработке этапа, – дополнительная проверка работоспособности датчика. Обработку датчика можно пропустить, чтобы избежать задержек, если вдруг он окажется неработоспособным в период технического обслуживания или из-за отказа. Датчики присутствуют во всех агрегатах и механизмах на всех этапах производства (обозначены шестеренками на рис. 3.1) и отправляют сообщения на серверы в кластере текущей системы. Имеется также административная консоль для удаленного обновления ПО и отправки команд датчикам.

3.2.1. Имеющиеся проблемы

Прежде чем продолжить, обсудим некоторые проблемы, свойственные большинству предыдущих вариантов использования. Получение данных и организация их доступности для пользователей – две большие и сложные задачи. Их можно решить, организовав хранение исходных данных и реализовав процедуры восстановление в случае сбоев.

Работа с исходными данными

На нашем заводе данные и их обработка принадлежат приложению. Если другие захотят использовать эти данные, им нужно связаться с владельцем приложения и договориться об экспорте данных. И каковы шансы, что данные будут экспортированы в формате, удобном для потребителя? А если владелец приложения решит вообще не экспорттировать данные?

Каждый значок шестеренки представляет важный этап в производственном процессе. На каждом этапе имеется датчик

В настоящее время датчики отправляют свои события в кластерную базу данных. Это одна из сторон архитектуры данных, которую мы должны изменить!

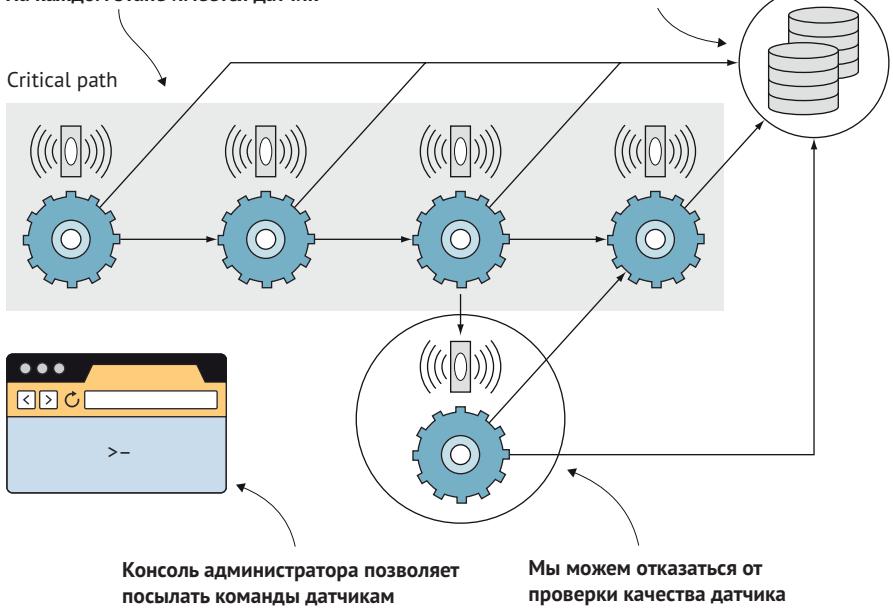


Рис. 3.1. Структура системы

Отход от традиционного «мышления о данных» позволяет сделать исходные данные доступными для всех. Если у вас есть доступ к данным на этапе их поступления, то вам не нужно беспокоиться о том, что API приложения экспортирует их не в том формате или в преобразованном виде. А что, если приложение, предоставляющее API, содержит ошибку и экспортирует неверные данные? Чтобы распутать этот клубок, может потребоваться некоторое время, если нам придется воссоздавать исходные данные из измененной версии.

Восстановление

Одним из замечательных преимуществ распределенной системы, такой как Kafka, является ее готовность к встрече со сбоями: их планируют и обрабатывают! Однако кроме системных сбоев есть еще и человеческий фактор. Если в приложении таится ошибка, уничтожающая данные, то можно ли устраниТЬ ее последствия после исправления? С Kafka это легко реализовать, просто начав потреблять данные с самого начала темы, как в примере с флагом `--from-beginning` консольного потребителя в главе 2. Кроме того,

сохранение данных делает их доступными для использования снова и снова. Возможность повторной обработки данных для исправления ошибок бесцenna. Но если исходное событие недоступно, то модифицировать существующие данные может оказаться сложной задачей.

Поскольку события посылаются датчиком конкретному экземпляру только один раз, брокер сообщений может играть решающую роль в нашем шаблоне потребления. Если сообщение будет удаляться брокером из очереди после того, как подписчик прочитает его, как это делается в версии 1.0 приложения на рис. 3.2, то оно навсегда исчезнет из системы. Если дефект в логике приложения будет обнаружен постфактум, то потребуется провести серьезный анализ, чтобы выяснить, можно ли исправить данные, используя информацию, оставшуюся после обработки исходного события, потому что его нельзя будет обработать повторно. К счастью, брокеры Kafka допускают другой вариант.

Начиная с версии 1.1, приложение может воспроизводить уже использованные сообщения с помощью новой логики. Наш новый код приложения с исправленной логической ошибкой, обнаруженной в версии 1.0, может повторно обрабатывать все события. Возможность повторной обработки событий упрощает совершенствование приложений, избавляя от страха потери или повреждения данных.

Повторное воспроизведение событий также может показать нам, как меняется значение с течением времени. Было бы полезно провести параллель между повторным чтением темы Kafka и идеей журнала упреждающей записи (Write-Ahead Log, WAL). Используя журнал WAL, можно сказать, какое значение имел тот или иной параметр раньше и как его значение менялось с течением времени, потому что изменения значений записываются в журнал до того, как они будут применены. Журналы WAL широко используются в системах баз данных и помогают им восстанавливаться, если в ходе выполнения транзакции произошел сбой. Проследив за событиями от начала до конца, можно увидеть, как меняются данные от начального значения к текущему.

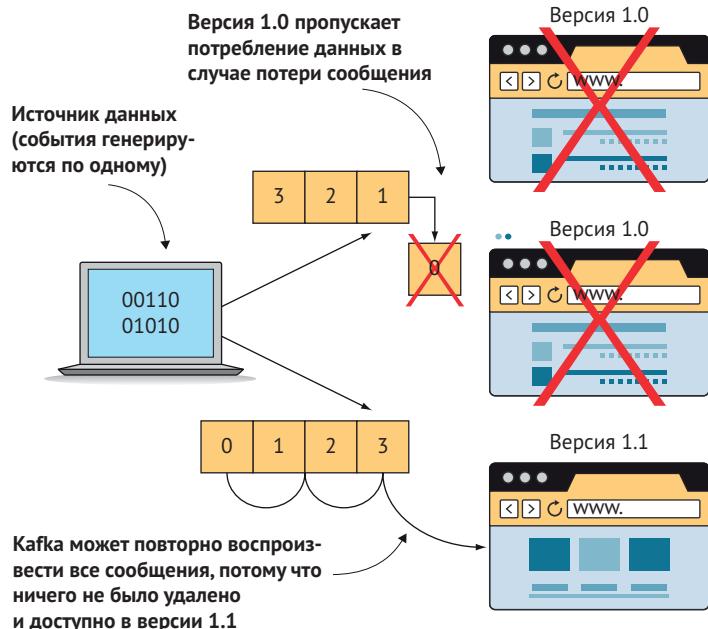


Рис. 3.2. Результат ошибки, допущенной разработчиком

Когда следует изменять данные?

Независимо от того, поступает ли информация из базы данных или из журнала событий, предпочтительнее сначала поместить данные в Kafka; тогда они будут доступны в чистом виде. Но каждый шаг перед сохранением данных в Kafka – это лишняя возможность изменить данные или внести различные логические ошибки. Имейте в виду, что аппаратное и программное могут и будут давать сбои в распределенных вычислениях, поэтому всегда полезно сначала загрузить данные в Kafka, что получить возможность воспроизвести данные в случае любого из этих сбоев.

3.2.2. Почему Kafka – правильный выбор

Есть ли смысл использовать Kafka в нашем вымышленном примере с датчиками? Конечно! Эта книга о Kafka, верно? И тем не менее попробуем определить пару веских причин использовать Kafka.

Наши клиенты ясно дали понять, что вертикальное масштабирование их текущей базы данных обходится слишком дорого. Под вертикальным масштабированием подразумевается увеличение таких ресурсов, как вычислительная мощность процессора, объем ОЗУ и дисковых накопителей на существующей машине. (Для динамического масштабирования мы рассмотрим возможность добавления дополнительных серверов в наше окружение.) Возможность горизонтального масштабирования кластера позволяет нам надеяться получить больше

общих преимуществ за вложенные средства. Конечно, серверы, на которых будут работать наши брокеры, являются не самыми дешевыми машинами, что можно купить, но, чтобы справиться с промышленными нагрузками, их достаточно оснастить 32 или 64 Гбайт ОЗУ [4].

Еще одно обстоятельство, которое наверняка бросилось вам в глаза, – предполагается, что в нашей системе постоянно будут генерироваться события. Это очень похоже на потоковую обработку, о которой мы говорили выше. Постоянный поток данных не имеет определенного времени окончания или точки остановки, поэтому наши системы должны быть готовы к постоянной обработке сообщений. Еще один интересный момент, на который стоит обратить внимание, – размер сообщений в нашем примере обычно не превышает 10 Кбайт. Чем меньше размер сообщения и чем больше объем памяти, который можно выделить для кеша страниц, тем более высокую производительность мы сможем обеспечить.

В ходе этого обзора требований к нашему сценарию некоторые разработчики, беспокоящиеся о безопасности, могли заметить, что в брокерах отсутствует встроенное шифрование диска (хранимых данных). Однако это не является обязательным требованием для текущей системы. Сначала мы сосредоточимся на том, чтобы наша система просто заработала, а затем подумаем о безопасности.

3.2.3. Первые мысли об архитектуре

Давайте кратко посмотрим, какие возможности доступны в конкретных версиях Kafka. В этом примере мы используем последнюю версию (на момент написания этой статьи это была версия 2.7.1), но некоторые разработчики могут не иметь возможности использовать текущие версии брокера и клиента из-за ограничений существующей у них инфраструктуры. Поэтому было бы полезно знать, когда появились те или иные функции и API, которые мы могли бы использовать. В табл. 3.1 перечислены некоторые основные возможности, но в ней указаны не все версии [5].

Таблица 3.1. Наиболее значимые версии Kafka

Версия Kafka	Функция
2.0.0	Списки управления доступом (ACL) с поддержкой префикса и проверкой имени хоста (по умолчанию для SSL)
1.0.0	Поддержка Java 9 и улучшенная обработка отказов дисковых массивов JBOD
0.11.0.0	API администрирования
0.10.2.0	Улучшенная совместимость с клиентами
0.10.1.0	Поиск по времени
0.10.0.0	Потоки Kafka, отметки времени, поддержка топологий стоек (rack awareness)
0.9.0.0	Различные функции безопасности (ACL, SSL), Kafka Connect и новый клиент потребителя

В следующих нескольких главах мы сосредоточимся на клиентах, поэтому обратите внимание на улучшенную совместимость с клиентами. Брокеры, начиная с версии 0.10.0, могут работать с более новыми версиями клиентов. Это важно, потому при использовании этой версии Kafka мы можем обновить версии клиентов и оставить прежние версии брокеров, пока не решим, что настала пора обновить и их. Это особенно удобно для тех, кто работает с уже существующим кластером.

Теперь, когда мы решили попробовать Kafka, самое время выбрать, как хранить данные. Следующие вопросы помогут нам задуматься о том, как лучше обрабатывать наши данные. Ответы на эти вопросы повлияют на различные части нашей архитектуры, но наше основное внимание мы сосредоточим на выяснении структуры данных, а реализацией займемся в последующих главах. Этот список не претендует на полноту, но является хорошей отправной точкой при планировании архитектуры.

- *Допустима ли потеря каких-либо сообщений в системе?* Например, может ли одно пропущенное событие, связанное с выплатой ипотечного кредита, доставить массу неприятностей вашему клиенту и подорвать его доверие к вашему бизнесу? Или это незначительная проблема, как, например, отсутствие в RSS-канале в вашей учетной записи некоторого сообщения? Последнее, конечно, досадно, но воспримут ли его ваши клиенты как конец мира?
- *Нужно ли каким-то образом группировать ваши данные?* Связаны одни события с другими? Например, будут ли учитываться изменения? Если да, то хотелось бы связать различные изменения в учетной записи с клиентом, чья учетная запись меняется. Предварительная группировка событий может также избавить приложения от необходимости координировать сообщения от нескольких потребителей при чтении темы.
- *Должны ли данные доставляться в определенном порядке?* Что, если сообщение будет доставлено не в том порядке, в каком оно было получено? Например, критично ли получить уведомление об отмене заказа до уведомления о приеме заказа на рассмотрение. Поскольку товар отправляется только после оформления заказа, можно утверждать, что очередность уведомлений по заказу действительно важна. Однако строгое соблюдение очередности требуется не во всех случаях. Например, если вы просматриваете данные о стоимости своих акций, то порядок их следования в списке не так важен, как итоговая сумма в конце.
- *Нужно только последнее значение определенного элемента, или важна история его изменения?* Вас волнует, как менялись ваши данные? Представьте обновление информации в традицион-

ной таблице реляционной базы данных. Она изменяется на месте (старое значение исчезает, а новое значение заменяет его). Прежние значения, имевшие место день или даже месяц тому назад, теряются.

- *Сколько потребителей предполагается иметь?* Будут ли они независимыми друг от друга, или для них важен какой-то определенный порядок чтения сообщений? При большом количестве данных, которые желательно использовать как можно быстрее, ответ на этот вопрос поможет определить, как разбивать сообщения на конечных этапах обработки.

Теперь, определив несколько вопросов об организации системы для завода, перейдем к следующему разделу и попробуем ответить на них, следуя реальным требованиям и используя диаграмму.

3.2.4. Требования к пользовательским данным

Наша новая архитектура должна обеспечивать пару конкретных возможностей. Во-первых, нужна возможность перехватывать сообщения, даже если потребляющая служба не работает. Например, если одно из приложений-потребителей не работает, мы должны гарантировать, что оно сможет обработать события потом. Кроме того, когда приложение остановлено на техническое обслуживание или восстанавливается после сбоя, оно должно иметь возможность получить данные, поступившие в этот период. Нам также нужно предусмотреть признак работоспособности датчиков (своего рода предупреждение) и реализовать возможность видеть, может ли какая-либо часть процесса производства велосипедов привести к полному отказу.

Также мы должны сохранить историю предупреждений от датчиков. Эти данные можно использовать для определения тенденций и прогнозирования отказов до того, как реальные события приведут к поломке оборудования. Также нам нужно вести журнал истории действий пользователей, которые обновляют данные или посылают запросы непосредственно к датчикам. Наконец, мы должны фиксировать, кто и какие административные действия выполнял на самих датчиках.

3.2.5. Общий план с учетом поставленных вопросов

Давайте сосредоточимся на наших требованиях к созданию журнала аудита. Как определено выше, все, что приходит на уровне API управления, должно фиксироваться в журнале. Мы должны гарантировать, что только уполномоченные пользователи смогут выполнять действия с датчиками, при этом никакие сообщения не должны теряться, иначе журнал аудита будет неполным. В данном случае нам не требуется группировать сообщения, поскольку каждое событие можно рассматривать как независимое.

Порядок событий в журнале аудита не имеет значения, потому что каждое сообщение будет сопровождаться отметкой времени в самих данных. Наша основная задача состоит в том, чтобы обеспечить доступность всех данных для обработки. Отметим, однако, что сама платформа Kafka позволяет сортировать сообщения по времени, но данные внутри сообщений тоже могут включать время. Тем не менее рассматриваемый нами конкретный вариант не гарантирует такого использования.

На рис. 3.3 показано, как пользователь может сгенерировать два события аудита из консоли веб-администрирования, отправив одну команду на датчик 1, а другую – на датчик 3. Обе команды должны зафиксироваться в Kafka как отдельные события. Для большей ясности в табл. 3.2 представлен приблизительный контрольный список того, что мы должны учесть в отношении данных для каждого требования. Этот краткий обзор поможет при определении параметров конфигурации наших клиентов-производителей.

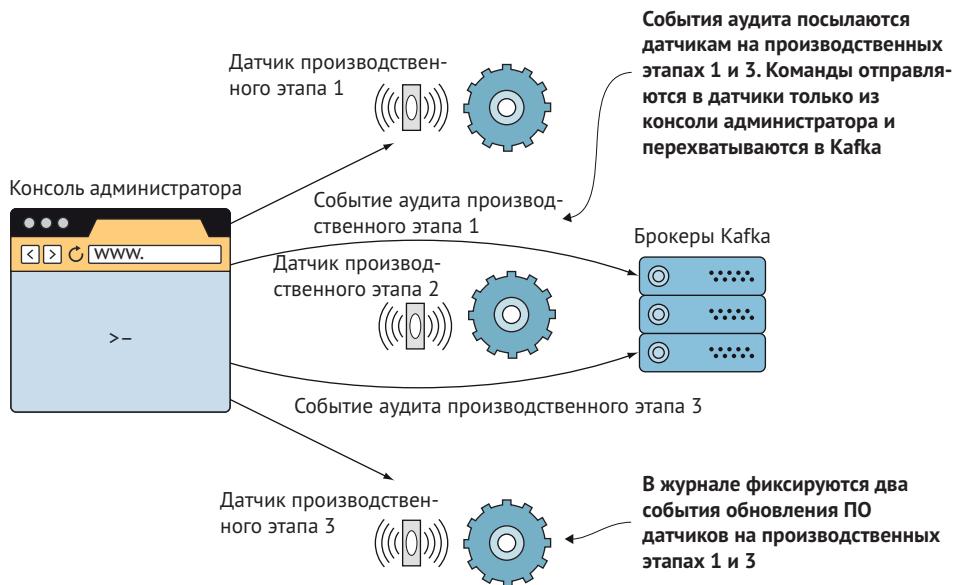


Рис. 3.3. Диаграмма аудита

Таблица 3.2. Контрольный список для аудита

Требования к Kafka	Важно?
Сохранность сообщений	Да
Группировка	Нет
Упорядоченность	Нет
Хранить только последнее значение	Нет
Независимость потребителей	Да

При создании этого производителя событий аудита нам важно обеспечить сохранность всех данных, но порядок и координация следования сообщений не важны. Кроме того, нам будет интересно выявить тенденции предупреждений, порождаемых датчиками на каждом производственном этапе, чтобы дать возможность предотвратить выход оборудования из строя. Для этого желательно сгруппировать предупреждения с помощью ключа. Мы не рассматривали определение термина *ключ*, но в первом приближении его можно интерпретировать как способ группировки связанных событий.

Скорее всего, мы будем использовать идентификаторы деталей велосипедов на каждом этапе, где установлены датчики, потому что они лучше обеспечат уникальность идентификации, чем любые другие имена. Нам также нужна возможность просматривать ключевые события на каждом этапе, чтобы выявлять тенденции с течением времени. Используя один и тот же ключ для каждого датчика, мы сможем легко извлекать соответствующие события. Поскольку сообщения с информацией о работоспособности отправляются каждые пять секунд, мы можем пренебречь вероятностью потери сообщений, так как вскоре должно поступить следующее. Если датчик отправляет сообщение «Требуется техническое обслуживание» каждые пару дней, то мы обязательно должны фиксировать их для выявления тенденций к отказу оборудования.

На рис. 3.4 показан процесс наблюдения за датчиками на всех этапах производственного процесса. События от датчиков попадают в Kafka. Хотя это и не является нашей непосредственной задачей, Kafka позволяет переносить эти данные в другую систему хранения или обработки данных, такую как Hadoop.



Рис. 3.4. Выявление тенденций к отказу

Как показано в табл. 3.3, наша цель – сгруппировать предупреждения по этапам, но нас не беспокоит периодическая потеря сообщений.

Таблица 3.3. Контрольный список для выявления тенденций к отказу

Требования к Kafka	Важно?
Сохранность сообщений	Нет
Группировка	Да
Упорядоченность	Нет
Хранить только последнее значение	Нет
Независимость потребителей	Да

Оповещения о состоянии работоспособности желательно также сгруппировать по ключу, идентифициирующему производственный этап. Однако нас волнуют не прошлые состояния датчиков, а текущие. Иначе говоря, текущее состояние – это все, что нам важно и нужно для удовлетворения обозначенных требований. Новое состояние заменяет старое, и нет необходимости вести историю. Слово *заменяет* здесь употреблено не совсем верно (по крайней мере, оно не совсем точно отражает происходящее). Получив сообщение с состоянием, Kafka добавит новое событие в конец своего журнала, как и любое другое событие. В конце концов, журнал неизменяем и поддерживает только добавление в конец. Но как в Kafka реализовано то, что выглядит как замена? Для этого используется процесс, называемый *уплотнением журнала*, который мы рассмотрим в главе 7.

Еще одно отличие, связанное с этим требованием, – использование потребителями определенных разделов предупреждений. Критические оповещения должны обрабатываться в первую очередь из-за требования к времени безотказной работы, и максимально быстро. На рис. 3.5 показан пример, как критические оповещения можно отправлять в Kafka, а затем использовать для вывода на дисплей оператора, чтобы привлечь его внимание. Контрольный список в табл. 3.4 подтверждает идею о группировке предупреждений по производственным этапам и достаточности иметь только последнее состояние.

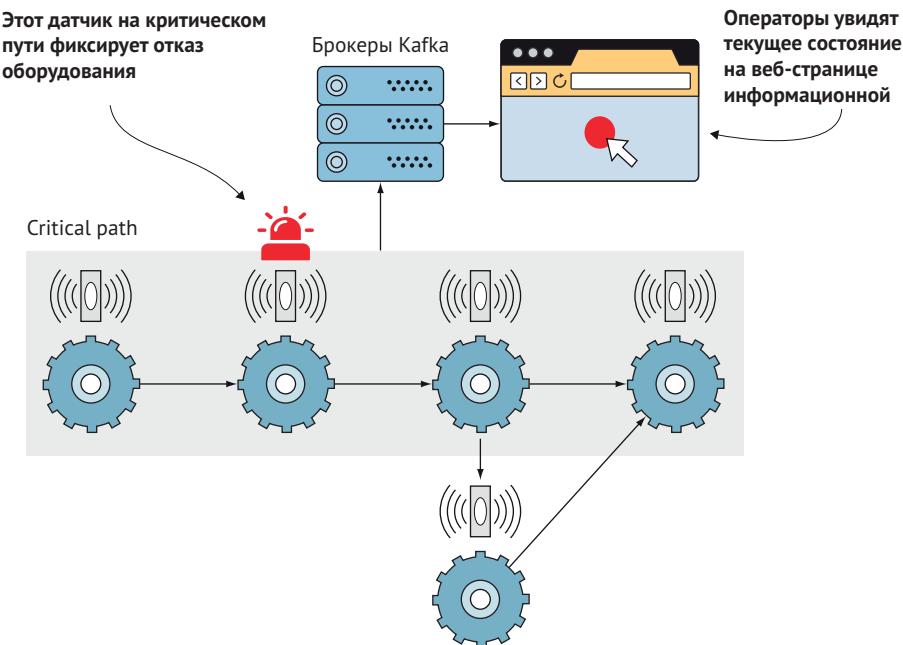


Рис. 3.5. Обработка критических оповещений

Таблица 3.4. Контрольный список для критических оповещений

Требования к Kafka	Важно?
Сохранность сообщений	Нет
Группировка	Да
Упорядоченность	Нет
Хранить только последнее значение	Да
Независимость потребителей	Нет

Потратив время на планирование наших требований к данным, мы не только проясним требования к нашим приложениям, но и, надеюсь, подтвердим верность выбора Kafka для использования в нашем проекте.

3.2.6. Обзор и оценка плана

Наконец, мы должны подумать об организации этих групп данных. Логические данные можно разделить на следующие группы:

- данные аудита;
- данные для выявления тенденций к отказу;
- предупреждения.

Забегая вперед, могу сказать, что мы можем использовать данные для выявления тенденций к отказу как источник инфор-

мации для темы с предупреждениями; Kafka позволяет использовать одну тему как источник для заполнения другой темы. Однако на первом этапе мы будем записывать разные типы событий от датчиков в соответствующие им логические темы, чтобы упростить первую попытку для реализации и понимания. Иначе говоря, все события аудита будут записываться в тему аудита, все события с данными для выявления тенденций к отказу – в тему тенденций, а все предупреждения – в тему предупреждений. Такое отношение «один к одному» поможет нам сосредоточиться на текущих требованиях.

3.3. Формат представления данных

Один из аспектов, которые легко упустить из виду, но важные для проекта, – формат представления данных. XML и JSON – широко используемые стандартные форматы, помогающие определить некоторую структуру данных. Однако даже при использовании четко установленного формата в данных может отсутствовать важная информация. Что означает первый столбец или третий? Какой тип данных имеет поля во втором столбце? Знания о том, как правильно анализировать данные, могут быть скрыты в приложениях, которые извлекают данные из хранилища. Решить эту проблему помогают схемы, предоставляющие некоторую важную информацию так, чтобы ее можно было использовать в программном коде или в других приложениях, которым могут потребоваться те же данные.

Если заглянуть в документацию Kafka, можно заметить ссылки на систему сериализации под названием Apache Avro. Avro обеспечивает возможность определения схем и их хранения в файлах Avro [6]. Как мне кажется, Avro – это тот инструмент, который вы часто будете встречать в коде Kafka и в реальном мире, поэтому мы выберем его. Давайте подробнее рассмотрим, почему этот формат обычно используется в Kafka.

3.3.1. План для данных

Одним из значительных преимуществ использования Kafka является отсутствие прямой связи между производителями и потребителями. Кроме того, по умолчанию Kafka не выполняет никакой проверки данных. Однако, вероятно, каждый процесс или приложение должны понимать, что означают эти данные и какой формат используется. Используя схему, мы можем дать разработчикам приложения возможность понять структуру и назначение данных. Определение схемы не обязательно размещать в файле *README*, чтобы другие сотрудники организации могли

определить типы данных, не прибегая к попыткам реконструировать их из дампов данных.

В листинге 3.8 показан пример схемы Avro в формате JSON. Поля могут сопровождаться такой информацией, как имя, тип и значения по умолчанию. Например, глядя на поле `daysOverDue` в схеме, можно сказать, что количество дней опоздания готовности книги к печати выражается целым числом и имеет значение по умолчанию 0. Знание того, что это числовое значение, а не текст (например, одна неделя), помогает сформировать четкое представление для производителей и потребителей данных.

Листинг 3.8. Пример схемы Avro

```
{
  "type" : "record",
  "name" : "kinaction_libraryCheckout", ← Схема Avro
  ...                                     в формате JSON
  "fields" : [{"name" : "materialName",
    "type" : "string",
    "default" : ""},
    {"name" : "daysOverDue", ← Имя поля,
     "type" : "int", ← Тип поля
     "default" : 0}, ← Значение
    {"name" : "checkoutDate",
     "type" : "int",
     "logicalType": "date",
     "default" : "-1"},

    {"name" : "borrower",
     "type" : {
       "type" : "record",
       "name" : "borrowerDetails",
       "fields" : [
         {"name" : "cardNumber",
          "type" : "string",
          "default" : "NONE"}]}},
    "default" : {}
  ]
}
```

Глядя на схему Avro в листинге 3.8, разработчик с легкостью сможет ответить на вопрос «Как интерпретировать поле `cardNumber` – как число или как строку?» (Ответ: в данном случае как строку.) Приложения могут автоматически использовать эту информацию для создания объектов, представляющих эти данные, чтобы избежать ошибок синтаксического анализа данных.

Схемы могут использоваться такими инструментами, как Apache Avro, для обработки изменяющихся данных. Большинство из нас имело дело с операторами или инструментами преобразования, такими как Liquibase, для обработки изменений в структуре реляционных баз данных. Имея схему, мы начинаем осознавать, что структура наших данных может измениться в будущем.

Нужна ли схема, когда мы только начинаем проектировать структуру данных? Одна из основных проблем заключается в сохранении контроля над данными с увеличением масштаба системы. Чем больше потребителей у нас будет, тем большая нагрузка ляжет на тестирование, которое нам нужно будет провести. Если не считать роста числа, то мы можем даже не знать всех потребителей этих данных.

3.3.2. Настройка зависимостей

Теперь, обсудив некоторые преимущества использования схем, возникает вопрос: зачем нам рассматривать Avro? Во-первых, Avro всегда выполняет сериализацию с использованием своей схемы [7]. Хотя сам по себе Avro не является схемой, он поддерживает возможность использования схем при чтении и записи данных и может обрабатывать схемы, изменяющиеся с течением времени. Кроме того, знакомые с форматом JSON без труда освоят Avro. Помимо данных, сам язык схемы также определяется в формате JSON. Если схема изменится, вы все равно сможете обрабатывать данные [7]. Старые данные используют схему, существовавшую как часть этих данных. С другой стороны, любые новые форматы будут использовать схему, присутствующую в их данных. Наибольшую выгоду от использования Avro получают клиенты.

Еще одно преимущество Avro – популярность. Мы впервые увидели его применение в различных проектах Hadoop, но его можно использовать и во многих других приложениях. Confluent тоже имеет встроенную поддержку Avro в большинстве своих инструментов [6]. Существуют библиотеки подключения Avro к приложениям для многих языков программирования, и их нетрудно найти. Те, кто имеет прошлый «плохой» опыт и предпочитает избегать генерируемого кода, могут использовать Avro как динамическую библиотеку, без генерации кода.

А теперь давайте сделаем Avro доступным в нашем приложении, добавив зависимость в файл *pom.xml*, как показано в листинге 3.9 [8]. Для тех, кто еще не освоился с *pom.xml* или системой сборки Maven, подскажем, что этот файл находится в корневом каталоге нашего проекта.

Листинг 3.9. Добавление Avro в файл pom.xml

```
<dependency>
    <groupId>org.apache.avro</groupId> ←
    <artifactId>avro</artifactId>
    <version>${avro.version}</version>
</dependency>
```

Добавьте эту запись с описанием зависимости в файл pom.xml проекта

Раз уж мы взялись изменять файл РОМ, давайте заодно добавим плагин, генерирующий исходный код на Java для наших определений схемы. Стоит отметить, что точно так же можно генерировать исходный код из отдельного файла JAR, *avro-tools*, если у вас нет желания использовать плагин для Maven. Для тех, кому не нравится присутствие генерированного кода в проектах, это не является жестким требованием [9].

В листинге 3.10 показано, как добавить зависимость *avro-maven-plugin* в файл *pom.xml*; именно так предлагается поступать в документе «Apache Avro Getting Started» на сайте проекта Apache Avro [8]. Код в этом листинге не включает XML-блок конфигурации. Добавление необходимой конфигурации также позволяет Maven узнать, что мы хотим генерировать исходный код для файлов Avro и сохранить его в указанный каталог. Если хотите, то можете изменить пути к каталогам с исходной схемой и со генерированным кодом, чтобы привести их в соответствие со структурой вашего конкретного проекта.

Листинг 3.10. Добавление плагина Avro Maven в pom.xml

```
<plugin>
    <groupId>org.apache.avro</groupId>
    <artifactId>avro-maven-plugin</artifactId> ← Идентификатор артефакта плагина, который должен указываться в файлах pom.xml
    <version>${avro.version}</version>
    <executions>
        <execution>
            <phase>generate-sources</phase> ← Настройка этапа сборки в Maven
            <goals>
                <goal>schema</goal> ← Настройка цели Maven
            </goals>
            ...
        </execution>
    </executions>
</plugin>
```

Теперь приступим к определению схемы с типами данных, которые мы будем использовать, и начнем со сценария оповещения о состоянии. Прежде всего создадим новый файл с именем *kinaction_alert.avsc* в текстовом редакторе. В листинге 3.11 показано его содержимое. Назовем наш Java-класс *Alert*; мы будем взаимодействовать с ним после генерации исходного кода на основе этого файла.

Листинг 3.11. Схема Alert: kinaction_alert.avsc

```
{
  ...
  "type": "record",           ← Имя создаваемого
  "name": "Alert",            ← Java-класса
  "fields": [                ← Определение типов данных и документирующие примечания
    {
      "name": "sensor_id",
      "type": "long",
      "doc": "The unique id that identifies the sensor"
    },
    {
      "name": "time",
      "type": "long",
      "doc":
        "Time alert generated as UTC milliseconds from epoch"
    },
    {
      "name": "status",
      "type": {
        "type": "enum",
        "name": "AlertStatus",
        "symbols": [
          "Critical",
          "Major",
          "Minor",
          "Warning"
        ]
      },
      "doc":
        "Allowed values sensors use for current status"
    }
  ]
}
```

В листинге 3.11, где показано определение схемы для предупреждений, следует отметить, что "doc" не является обязательной частью определения. Тем не менее всегда полезно добавить детали, которые прямо сообщают разработчикам производителей или потребителей назначение данных, чтобы им не приходилось догадываться. Например, поле "time" всегда вызывает беспокойство у разработчика: в каком виде хранится это самое время? В виде строки? Или в каком-то другом формате? Содержит ли значение времени информацию о часовом поясе? Включает ли оно высокосные секунды? Всю эту информацию можно сообщить в поле "doc". Поле, определяющее пространство имен (в листинге 3.11 не показано), превращается в имя пакета Java для генерированного класса. Полный пример вы найдете в исходном коде для книги. Также определения полей включают их имена и типы.

Теперь, определив схему, запустим сборку проекта, чтобы посмотреть, что у нас получилось. Команды `mvn generate-sources` и `mvn install` могут генерировать исходный код в папках нашего проекта. Они должны дать нам пару классов, `Alert.java` и `AlertStatus.java`, которые теперь можно использовать.

До сих пор мы сосредоточились на самом Avro, но также нужно внести изменения в код наших клиентов производителя и потребителя, чтобы использовать созданную нами схему. Всегда можно написать собственный компонент сериализации для Avro, но у нас уже есть отличный пример, созданный в Confluent. Чтобы получить доступ к получившимся классам, нужно добавить в нашу сборку зависимость `kafka-avro-serializer` [10]. В листинге 3.12 показан элемент из файла `pom.xml`, который мы добавим. Это необходимо, чтобы не создавать свой компонент сериализации и десериализации для ключей и значений наших событий.

Листинг 3.12. Добавление поддержки компонента сериализации Kafka в pom.xml

```
<dependency>
    <groupId>io.confluent</groupId>
    <artifactId>kafka-avro-serializer</artifactId> ←
    <version>${confluent.version}</version>
</dependency>
```

1. Добавление зависимости в файл pom.xml проекта

Если вы используете Maven, то добавьте ссылку на репозиторий Confluent в свой файл `pom.xml`. Это необходимо, чтобы сообщить Maven, где брать конкретные зависимости [11].

```
<repository>
    <id>confluent</id>
    <url>https://packages.confluent.io/maven/</url>
</repository>
```

Теперь, когда сборка настроена и объект Avro готов к использованию, возьмем пример производителя `HelloWorldProducer` из предыдущей главы и немного изменим класс, чтобы он использовал Avro. В листинге 3.13 показаны соответствующие изменения в классе производителя (исключая инструкции импорта). Обратите внимание на значение `io.confluent.kafka.serializers.KafkaAvroSerializer` свойства `value.serializer`. Этот компонент обрабатывает объекты `Alert`, отправляемые в нашу новую тему `kinaction_schematest`.

Раньше мы могли использовать компонент сериализации строк, но при использовании Avro нужно определить конкретный механизм сериализации значений, чтобы сообщить клиенту, как обращаться с нашими данными. Пример с использованием объекта `Alert` вместо строки покажет, как можно использовать нестандартные типы при наличии возможности сериализовать их. В этом

примере также используется реестр схем. Подробнее о реестре схем мы поговорим в главе 11. Этот реестр поддерживает историю версий схем, помогающую управлять эволюцией схемы.

Листинг 3.13. Производитель, использующий компонент сериализации с поддержкой Avro

```
public class HelloWorldProducer {

    static final Logger log =
        LoggerFactory.getLogger(HelloWorldProducer.class);

    public static void main(String[] args) {
        Properties kaProperties = new Properties();
        kaProperties.put("bootstrap.servers",
            "localhost:9092,localhost:9093,localhost:9094");
        kaProperties.put("key.serializer",
            "org.apache.kafka.common.serialization.LongSerializer");
        kaProperties.put("value.serializer",
            "io.confluent.kafka.serializers.KafkaAvroSerializer"); ←
        kaProperties.put("schema.registry.url",
            "http://localhost:8081"); ← Настойка использования
                                         класса сериализации
                                         KafkaAvroSerializer в свой-
                                         стве value.serializer для
                                         поддержки нестандартного
                                         типа Alert

        try (Producer<Long, Alert> producer =
            new KafkaProducer<>(kaProperties)) { ← Создание предупрежде-
                                         ния с критическим уров-
                                         нем важности
            Alert alert =
                new Alert(12345L,
                    Instant.now().toEpochMilli(),
                    Critical); ←

            log.info("kinaction_info Alert -> {}", alert);
            ProducerRecord<Long, Alert> producerRecord =
                new ProducerRecord<>("kinaction_schematest",
                    alert.getSensorId(),
                    alert);
            producer.send(producerRecord);
        }
    }
}
```

Различия довольно незначительны. Изменились типы `Producer` и `ProducerRecord`, а также параметр конфигурации `value.serializer`.

Далее, создав сообщения с помощью `Alert`, мы должны изменить код потребителя. Чтобы потребитель мог получить значения из вновь созданной темы, ему придется использовать компонент десериализации для восстановления значений; в данном случае `KafkaAvroDeserializer` [10]. Этот компонент использует для восстановления значения из сериализованной формы, отправленной производителем. Этот код может ссылаться на тот же класс `Alert`, созданный в проекте. В листинге 3.14 показаны основные изменения в классе потребителя `HelloWorldConsumer`.

Листинг 3.14. Потребитель, использующий компонент сериализации с поддержкой Avro

```

public class HelloWorldConsumer {

    final static Logger log =
        LoggerFactory.getLogger(HelloWorldConsumer.class);

    private volatile boolean keepConsuming = true;

    public static void main(String[] args) {
        Properties kaProperties = new Properties();
        kaProperties.put("bootstrap.servers", "localhost:9094");
        ...
        kaProperties.put("key.deserializer",
            "org.apache.kafka.common.serialization.LongDeserializer");
        kaProperties.put("value.deserializer",
            "io.confluent.kafka.serializers.KafkaAvroDeserializer"); ←
        kaProperties.put("schema.registry.url", "http://localhost:8081");

        HelloWorldConsumer helloWorldConsumer = new HelloWorldConsumer();
        helloWorldConsumer.consume(kaProperties); ←
        Настройка использова-
        ния класса сериализа-
        ции KafkaAvroSerializer
        в свойстве value.
        serializer для поддер-
        жки нестандартного
        типа Alert

        Runtime.getRuntime()
            .addShutdownHook(
                new Thread(helloWorldConsumer::shutdown)
            );
    }

    private void consume(Properties kaProperties) { ←
        try (KafkaConsumer<Long, Alert> consumer = ←
            new KafkaConsumer<>(kaProperties)) {
            consumer.subscribe(
                List.of("kinaction_schematest")
            );

            while (keepConsuming) { ←
                ConsumerRecords<Long, Alert> records =
                    consumer.poll(Duration.ofMillis(250));
                for (ConsumerRecord<Long, Alert> record : ←
                    records) {
                    log.info("kinaction_info offset = {}, kinaction_value = {}", ←
                        record.offset(),
                        record.value());
                }
            }
        }
    }

    private void shutdown() {
        keepConsuming = false;
    }
}

```

Потребитель KafkaConsumer, обрабатывающий значения типа Alert

Изменения в ConsumerRecord для обработки экземпляров Alert

Так же как клиент производителя, клиент потребителя не требует значительных изменений благодаря использованию компонента десериализации и Avro! Теперь, когда у нас есть некоторое представление о том, *чего* мы хотим достичь, и определен конкретный формат данных, мы готовы узнать, *как* это реализовать, чем мы и займемся в следующей главе. Более детально поговорим о схемах в главе 11 и там же рассмотрим другой способ обработки нестандартных типов объектов, взяв за основу пример проекта из глав 4 и 5. Вообще говоря, отправка данных в Kafka реализуется просто, но существуют разные способы, зависящие от конфигурации, которые можно использовать для удовлетворения конкретных потребностей.

Итоги

- Проектирование решения Kafka требует в первую очередь понимания имеющихся данных, в том числе: допустима ли потеря данных, должны ли упорядочиваться сообщения и требуется ли их группировать.
- Необходимость группировки данных определяет необходимость добавления ключей в сообщения.
- Использование определения схемы помогает не только сгенерировать код, но также обрабатывать будущие изменения в структуре данных. Кроме того, мы можем использовать схемы с нашими собственными клиентами Kafka.
- Kafka Connect предоставляет готовые коннекторы для записи и извлечения данных из различных источников.

Ссылки

- 1 J. MSV. «Apache Kafka: The Cornerstone of an Internet-of-Things Data Platform» (15 февраля 2017). <https://thenewstack.io/apache-kafka-cornerstone-iot-data-platform/> (доступно по состоянию на 10 августа 2017).
- 2 «Quickstart». Confluent documentation (n.d.). <https://docs.confluent.io/3.1.2/connect/quickstart.html> (доступно по состоянию на 22 ноября 2019).
- 3 «JDBC Source Connector for Confluent Platform». Confluent documentation (n.d.). <https://docs.confluent.io/kafka-connect-jdbc/current/source-connector/index.html> (доступно по состоянию на 15 октября 2021).
- 4 «Running Kafka in Production: Memory». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/kafka/deployment.html#memory> (доступно по состоянию на 16 июня 2021).

- 5 «Download». Apache Software Foundation (n.d.). <https://kafka.apache.org/downloads> (доступно по состоянию на 21 ноября 2019).
- 6 J. Kreps. «Why Avro for Kafka Data?» Confluent blog (25 февраля 2015). <https://www.confluent.io/blog/avro-kafka-data/> (доступно по состоянию на 23 ноября 2017).
- 7 «Apache Avro 1.8.2 Documentation». Apache Software Foundation (n.d.). <https://avro.apache.org/docs/1.8.2/index.html> (доступно по состоянию на 19 ноября 2019).
- 8 «Apache Avro 1.8.2 Getting Started (Java): Serializing and deserializing without code generation» Apache Software Foundation (n.d.). https://avro.apache.org/docs/1.8.2/gettingstartedjava.html#download_install (доступно по состоянию на 19 ноября 2019).
- 9 «Apache Avro 1.8.2 Getting Started (Java): Serializing and deserializing without code generation» Apache Software Foundation (n.d.). <https://avro.apache.org/docs/1.8.2/gettingstartedjava.html#Serializing+and+deserializing+without+code+generation> (доступно по состоянию на 19 ноября 2019).
- 10 «Application Development: Java» Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/app-development/index.html#java> (доступно по состоянию на 20 ноября 2019).
- 11 «Installation: Maven repository for jars» Confluent documentation (n.d.). <https://docs.confluent.io/3.1.2/installation.html#maven-repository-for-jars> (доступно по состоянию на 20 ноября 2019).



Производители: источники данных

Эта глава охватывает следующие темы:

- производителей и отправку сообщений;
- создание собственных компонентов сериализации и фрагментирования на разделы для производителей;
- исследование параметров конфигурации для удовлетворения требований компании.

В предыдущей главе мы познакомились с требованиями, которые организация может предъявлять к своим данным. Некоторые принятые нами архитектурные решения оказывают практическое влияние на способ отправки данных в Kafka. Давайте теперь перейдем в мир платформ потоковой передачи событий через портал производителя Kafka. Прочитав эту главу, вы сможете выбрать правильный путь к решению основных требований проектов Kafka путем производства данных несколькими разными способами.

Производитель, несмотря на свою важность, является лишь частью системы. Мы уже познакомились с некоторыми параметрами конфигурации производителя и научились их настраивать на уровне брокера или темы. А теперь обсудим эти параметры подробнее, сделав основной упор на передаче данных в Kafka.

4.1. Пример

Производитель в нашем примере проекта реализует передачу данных в систему Kafka. Чтобы освежить память, посмотрим на рис. 4.1: здесь показано, какое место в Kafka занимают производители.

В этой главе мы сосредоточимся на производителе предупреждений

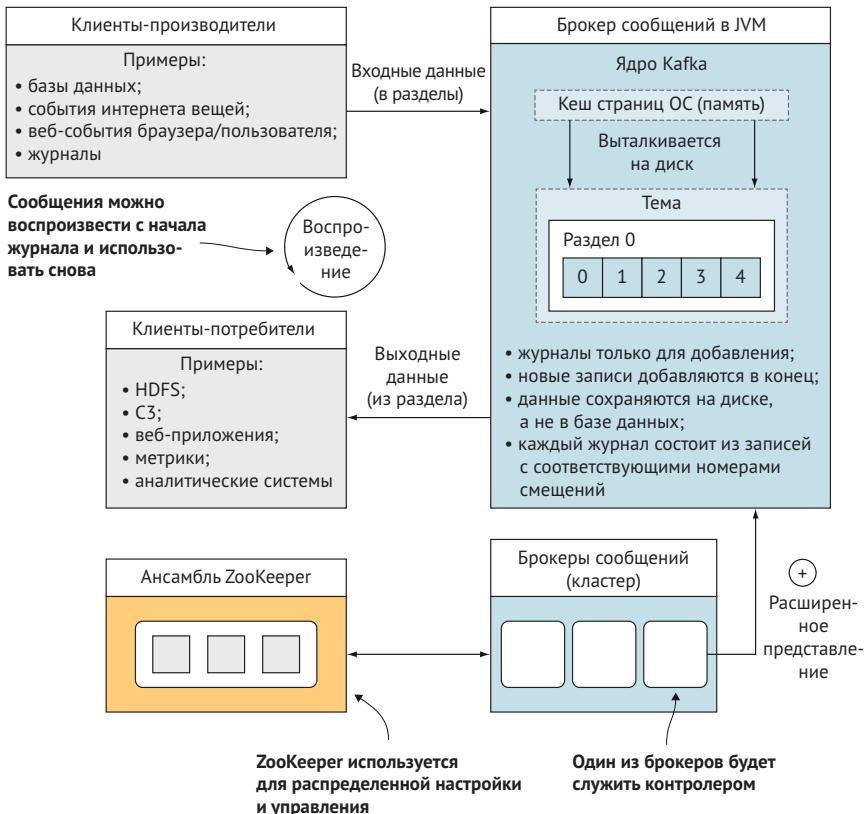


Рис. 4.1. Производители Kafka

Взгляните на список слева вверху на рис. 4.1 (клиенты-производители), где перечислены примеры данных, создаваемых в Kafka, и представьте, что эти данные отражают события интернета вещей (IoT), используемых в нашей вымышленной компании. Чтобы конкретизировать идею создания данных, рассмотрим практический пример, который мы могли бы написать для одного из наших проектов. Пусть это будет приложение, принимающее отзывы пользователей о работе веб-сайта.

В настоящее время пользователь заполняет форму на веб-сайте, которая генерирует электронное письмо и отправляет его на адрес службы поддержки или чат-бота. Время от времени один из наших сотрудников проверяет почтовый ящик, чтобы узнать, какие предложения внесли клиенты или с какими проблемами они столкну-

лись. В будущем мы хотели бы сохранить возможность получать отзывы, но так, чтобы доступ к этой информации был проще и не требовалось проверять почтовый ящик. Если отзывы пересыпать в тему Kafka, мы сможем получать отзывы оперативнее и надежнее, чем по электронной почте. Дополнительную гибкость в этом отношении дает возможность использования событий Kafka любыми приложениями-потребителями.

Давайте сначала посмотрим, как используется электронная почта в нашем конвейере данных. Взгляните на рис. 4.2 и особое внимание уделите формату, в котором сохраняются данные, когда пользователь отправляет форму с отзывом.



Рис. 4.2. Отправка отзыва по электронной почте

Для передачи традиционных электронных писем использует простой протокол передачи почты (Simple Mail Transfer Protocol, SMTP), и далее мы посмотрим, как это обстоятельство отражается на представлении и сохранении событий, пересылаемых по электронной почте. Для быстрого получения электронных писем часто используются почтовые клиенты, такие как Microsoft® Outlook®, но как еще можно извлекать данные из этой системы, кроме чтения электронной почты? Для этого нередко бывают задействованы операции копирования и вставки, выполняемые вручную, а также сценарии анализа электронной почты. (При применение сценариев предполагает использование инструментов

или языков программирования, а также библиотек или фреймворков.) Для сравнения: хотя Kafka использует свой уникальный протокол, она не навязывает конкретного формата представления сообщений и позволяет записывать данные в любом формате, выбранном нами.

ПРИМЕЧАНИЕ. В предыдущей главе мы рассмотрели одни из распространенных форматов, используемых сообществом Kafka, – формат Apache Avro. Также большой популярностью пользуются форматы Protobuf и JSON [1].

Еще один шаблон использования, который приходит на ум, – рассматривать уведомления о проблемах клиентов или сбоях веб-сайта как временные предупреждения, которые можно удалить после отправки ответа клиенту. Однако информация, отправленная клиентом, может служить нескольким целям. С ее помощью можно выявлять, например, закономерности сбоев; замедление работы сайта после массовой рассылки кодов купонов в рекламных электронных письмах; нехватку функциональных возможностей, востребованных клиентами; затруднения пользователей с поиском настроек конфиденциальности в их учетных записях и т. д. Передача этих данных в тему с возможностью их воспроизведения или чтения несколькими приложениями с разными целями может повысить их ценность, по сравнению с поддержкой отправки электронных писем, которые после прочтения обычно удаляются.

Кроме того, если электронные письма потребуется сохранять, то эта потребность будет контролироваться командами, управляющими инфраструктурой электронной почты, а не параметрами конфигурации Kafka. А теперь взгляните на рис. 4.3, где изображен новый конвейер обработки отзывов пользователей. Приложение все так же предоставляет форму HTML, но сведения, отправляемые пользователями, передаются в тему Kafka, а не серверу электронной почты. При таком подходе можно извлекать важную информацию в любом нужном нам формате и использовать ее для разных целей. Приложения-потребители могут использовать схемы для работы с данными и не зависеть от одного формата протокола. Мы можем сохранить и повторно обработать эти сообщения для новых вариантов использования, потому что полностью контролируем хранение этих событий. Теперь, когда мы узнали, для каких целей можно использовать производителей, давайте кратко обсудим некоторые детали взаимодействия производителя с брокерами Kafka.



Рис. 4.3. Отправка данных в Kafka

4.1.1. Примечания в отношении производителя

Работа производителя включает получение метаданных о кластере [2]. Поскольку производители могут передавать данные только в ведущую реплику того раздела, которому они назначены, метаданные помогают определить, какому брокеру послать сообщение, так как пользователь может указать только имя темы без любых других подробностей. Это хорошо, потому что избавляет конечного пользователя производителя от выполнения отдельного вызова, чтобы получить эту информацию. Конечный пользователь, однако, должен иметь возможность подключения хотя бы к одному действующему брокеру, а все остальное вычислит клиентская библиотека Java.

Распределенная система Kafka учитывает возможность появления кратковременных ошибок, таких как сбой сети, поэтому логика повторных попыток уже встроена в нее. Однако если порядок сообщений имеет значение, то, помимо установки числа повторных попыток в параметре `retries` (например, 3), мы должны также установить параметр `max.in.flight.requests.per.connection` равным 1 и параметр `acks` (количество брокеров, которые возвращают подтверждение) равным `all` [3] [4]. На наш взгляд, это один из самых надежных способов гарантировать поступление сообщений от нашего производителя в требуемом порядке [4]. Мы можем установить значения для `acks` и `retries` в виде конфигурационных параметров.

Еще одна особенность, о которой следует знать, – это идемпотентность производителя. Под *идемпотентностью* в данном случае понимается создание сообщения только один раз независимо от количества попыток отправить его. Чтобы обеспечить идемпотентность производителя, можно установить свойство конфигурации `enable.idempotence=true` [5]. Однако в следующих примерах мы не будем использовать его.

Еще одна вещь, о которой нет нужды беспокоиться, – взаимовлияние производителей. Безопасность в многопоточном окружении не является проблемой, потому что данные не затираются, а обрабатываются самим брокером и добавляются в журнал [6]. Теперь пришло время посмотреть, как определить в коде такие значения, как `max.in.flight.requests.per.connection`.

4.2. Параметры производителя

Одним из интересных наблюдений, которые мы сделали, когда начали работать с отправкой данных в Kafka, была простота установки параметров с помощью клиентов Java, на которых мы сосредоточимся в этой книге. Если вам довелось работать с другими системами очередей или обмена сообщениями, то вы могли заметить, что они предоставляют списки удаленных и локальных очередей, имена хостов диспетчеров, начальные соединения, фабрики соединений, сеансы и многое другое.

Хотя настройка не вызывает особых проблем, производитель, работая с конфигурацией, самостоятельно пытается получить большую часть необходимой ему информации, например список всех брокеров Kafka. Используя значение свойства `bootstrap.servers` в качестве отправной точки, производитель извлекает метаданные о брокерах и разделах, которые он будет использовать во всех последующих операциях записи.

Как упоминалось выше, Kafka позволяет радикально менять поведение простым изменением некоторых значений конфигурации. Один из способов справиться с многочисленными параметрами конфигурации производителя при его разработке – использовать константы, определяемые классом `ProducerConfig` (<http://mng.bz/ZYdA>) и подробно описанные на сайте Confluent (особое внимание обращайте на параметры, снабженные меткой **Importance: high** (Важность: высокая)) [7]. Однако в наших примерах для ясности мы будем использовать сами имена свойств.

В табл. 4.1 перечислены некоторые из наиболее важных параметров производителей, используемые в наших конкретных примерах. В следующих разделах рассмотрим, что нам нужно для завершения работы.

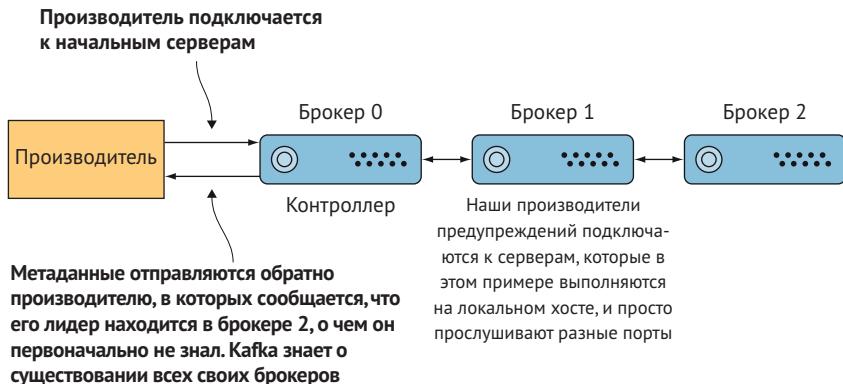
Таблица 4.1. Важные параметры настройки производителя

Параметр	Назначение
acks	Количество подтверждений реплик, которое должен получить производитель, чтобы убедиться в успехе отправки сообщения
bootstrap.servers	Список брокеров, с которыми должно устанавливаться соединение на запуске
value.serializer	Класс, используемый для сериализации значения
key.serializer	Класс, используемый для сериализации ключа

4.2.1. Настройка списка брокеров

Как видно в наших примерах передачи сообщений в Kafka, мы должны сообщить производителю тему, куда он должен отправлять сообщения. Напомним, что темы состоят из разделов, но как Kafka узнает, где находится раздел темы? Самое замечательное, что при отправке сообщений от нас не требуется знать детали организации разделов. Возможно, иллюстрация поможет прояснить эту загадку. Одним из обязательных параметров конфигурации производителей является `bootstrap.servers`. На рис. 4.4 показан пример производителя, в списке серверов которого присутствует только брокер 0, но он сможет узнать о существовании всех трех брокеров в кластере, подключившись только к одному из них.

Свойство `bootstrap.servers` может содержать список с несколькими брокерами или только одним, как показано на рис. 4.4. Подключившись к брокеру, клиент может получить необходимые ему метаданные, в том числе данные о других брокерах в кластере [8].

**Рис. 4.4.** Начальное соединение с серверами

Такая организация помогает производителю найти брокера, с которым он должен взаимодействовать. Подключившись к кластеру, производитель сможет получить метаданные со всей не-

обходимой ему информацией (например, где на диске находится ведущая реплика раздела), которую мы не могли предоставить заранее. Производители также могут обработать сбой лидера раздела, которому они посылают данные, используя информацию о кластере для поиска нового лидера. Выше вы могли заметить, что информация ZooKeeper не является частью конфигурации. Любые метаданные, необходимые производителю, будут автоматически получены производителем без предоставления сведений о кластере ZooKeeper.

4.2.2. Быстрее или надежнее?

Возможность асинхронного обмена сообщениями – одна из причин, почему многие используют системы очередей, и эта мощная возможность доступна также в Kafka. Мы в своем коде можем подождать результата отправки запроса производителем или асинхронно обработать успех или неудачу с помощью обратных вызовов или объектов Future. Чтобы двигаться вперед быстрее и не ждать ответа, можно организовать обработку результатов позже с помощью своей логики.

Еще одно конфигурационное свойство, актуальное в нашем сценарии, – это ключ `acks` (сокращенно от *acknowledgments* – подтверждения). Он определяет, сколько подтверждений должен получить производитель от копий ведущей реплики раздела, прежде чем запрос будет считаться отправленным. Допустимые значения для этого свойства: `all`, `-1`, `1` и `0` [9].

На рис. 4.5 показано поведение производителя с параметром `acks = 0`. Значение `0`, как предполагается, дает самую низкую задержку, но за счет надежности. Кроме того, не дается никаких гарантий, что какой-либо брокер получит сообщение, а также не будут предприняты попытки повторной передачи [9]. В качестве примера предположим, что у нас есть платформа, которая отслеживает щелчки мышью на странице и передает эти события в Kafka. В такой ситуации потеря одного-двух щелчков не имеет большого значения. Если какое-то из событий потерянется, то это не окажет никакого реального влияния на бизнес.

По сути, как показано на рис. 4.5, производитель отправляет событие и тут же забывает о нем. Сообщение может так и не попасть в раздел. Если сообщение попало в ведущую реплику, то производитель не будет знать, было ли оно успешно передано ведомым репликам.

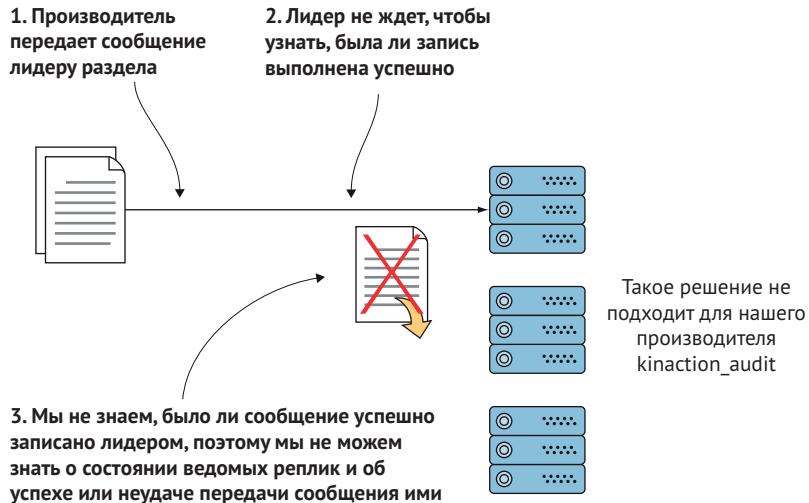


Рис. 4.5. Поведение при установке свойства `acks = 0`

Полной противоположностью предыдущей настройке является настройка параметра `acks` значением `all` или `-1`. Значения `all` и `-1` определяют самую строгую настройку этого параметра. Как показано на рис. 4.6, значение `all` требует, чтобы ведущая реплика дождалась подтверждения от всех синхронизированных с нею реплик (In-Sync Replicas, ISR) [9]. Другими словами, производитель не получит подтверждения успеха, пока все реплики раздела не запишут сообщение. Очевидно, что такая операция будет выполнена с задержкой из-за зависимости от других брокеров. Но во многих случаях лучше поступиться производительностью ради предотвращения потери данных. При наличии множества брокеров в кластере необходимо знать, сколько брокеров должен ждать лидер. Брокер, ответ от которого занимает больше всего времени, определяет, как долго производитель не получит сообщение об успешном завершении.

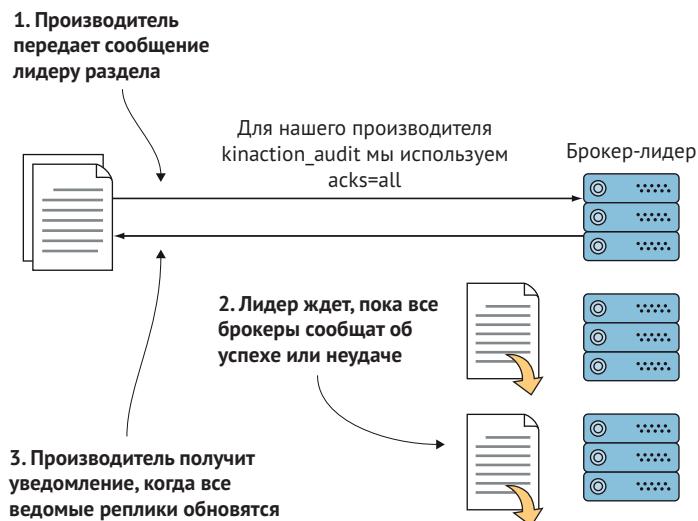


Рис. 4.6. Свойство `acks = all`

На рис. 4.7 показано влияние установки значения 1 в параметре `acks`. Такая настройка предполагает, что получатель сообщения (ведущая реплика определенного раздела) отправляет подтверждение обратно производителю. Клиент-производитель ожидает только этого подтверждения. Однако ведомые реплики могут не успеть получить сообщение до того, как лидер потерпит сбой. В такой ситуации сообщение никогда не появится в ведомых репликах этого раздела [9]. На рис. 4.7 показано, что несмотря на подтверждение получения сообщения ведущей репликой, сбой в лидере до того, как он успеет скопировать сообщение в ведомые реплики, приведет к ситуации, как если сообщение никогда не попадало в кластер.

ПРИМЕЧАНИЕ. Параметр `acks` тесно связан с семантиками «не менее одного раза» и «не более одного раза», которые мы рассмотрели в главе 1 [10]. Он является составной частью этой более широкой картины.



Рис. 4.7. Свойство acks = 1

4.2.3. Отметки времени

Последние версии записей, создаваемых производителем, содержат поле с временем события. Посылая объект `ProducerRecord`, пользователь может передать в конструктор это время как значение типа `long` или текущее системное время. Фактическое время в сообщении может устанавливаться в это значение или задаваться брокером в момент записи сообщения в журнал. При установке параметра `message.timestamp.type` конфигурации темы в значение `CreateTime` используется время, заданное клиентом, а при установке в значение `LogAppendTime` время задается брокером [11].

Когда лучше выбирать тот или иной вариант установки времени? Значение `CreateTime` следует устанавливать, когда желательно знать время выполнения транзакции (например, оформления заказа на продажу), а не время, когда сообщение дошло до брокера. Использование времени брокера (`LogAppendTime`) может пригодиться в случаях, когда время события находится в теле самого сообщения или когда фактическое время не имеет отношения к бизнесу или порядку следования событий.

Как всегда, работа с отметками времени сопряжена с некоторыми сложностями. Например, мы можем получить запись с более ранней отметкой времени, чем в предшествующей записи. Это может произойти, например, когда произошел сбой и до завершения

повторной попытки передать первую запись было зафиксировано другое сообщение с более поздним временем. Данные упорядочиваются в журнале в порядке их поступления, а не по отметкам времени. Хотя чтение данных с отметками времени часто рассматривается как задача клиента-потребителя, она так же затрагивает и производителя, потому что именно производитель делает первый шаг в обеспечении упорядоченности сообщений.

Как обсуждалось выше, именно по этой причине важно учитывать значение параметра `max.in.flight.requests.per.connection`, рассматривая вопрос о включении поддержки повторных попыток при наличии вероятности получения нескольких запросов одновременно. Если происходит повторная попытка и имеются другие запросы, выполняющиеся с первой попытки, то более ранние сообщения могут оказаться позади более поздних. На рис. 4.8 показан пример, когда порядок сообщений может нарушиться. Несмотря на то что сообщение 1 было отправлено первым, оно попало в журнал с нарушением порядка из-за того, что его не удалось отправить с первой попытки.

Напоминаем, что в версиях Kafka до 0.10 информация об отметках времени недоступна из-за отсутствия соответствующей поддержки в более ранних выпусках. Тем не менее мы можем добавлять отметки времени, но внутри самих сообщений.

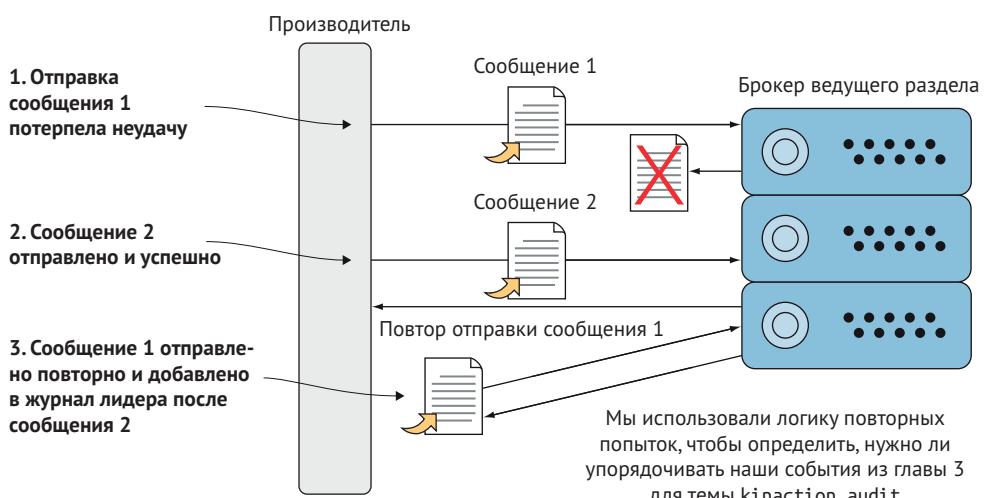


Рис. 4.8. Повторные попытки отправки могут влиять на порядок сообщений

Другой вариант использования производителя – создание перехватчиков. Запрос на их реализацию был зарегистрирован под номером KIP-42 (Kafka Improvement Proposal – предложение по улучшению Kafka). Основная цель перехватчиков – помочь в поддержке мониторинга [12]. По сравнению с использованием типичного рабочего процесса Kafka Streams для фильтрации или агрегирова-

ния данных или даже создания выделенных разделов для измененных данных, использование перехватчиков может быть не лучшим выбором. В настоящее время нет перехватчиков по умолчанию, которые выполняются в жизненном цикле. В главе 9 мы покажем вариант использования перехватчиков для трассировки передачи сообщений от производителей потребителям, добавляющих идентификатор трассировки.

4.3. Генерирование кода с учетом наших требований

Попробуем использовать собранную нами информацию о работе производителей в наших решениях. Начнем с контрольного списка аудита, который мы написали в главе 3, когда обсуждали вопрос применения Kafka на заводе по производству электровелосипедов. Как отмечалось в главе 3, мы должны гарантировать, что не будут потеряны никакие контрольные сообщения при отправке операторами команд датчикам. Одним из требований было отсутствие необходимости группировать какие-либо события. Еще одним требованием была гарантия сохранности всех сообщений. В листинге 4.1 показана конфигурация нашего производителя, гарантирующая надежное подтверждение сообщений установкой параметра `acks = all`.

Листинг 4.1. Настройка производителя аудита

Обратите внимание, что для решения проблемы потери сообщений нам достаточно только определить конфигурацию производителя. Настройка параметра `acks` – небольшой, но действенный шаг, оказывающий существенное влияние на гарантии доставки сообщений. Поскольку не нужно группировать какие-либо события, мы не используем ключи для них. Однако есть еще одно очень важное обстоятельство: мы должны дождаться результата отправки,

прежде чем двигаться дальше. В листинге 4.2 показан метод `get`, с помощью которого можно дождаться ответа, прежде чем продолжить выполнение. Обратите внимание, что листинг 4.2 основан на примерах, доступных по адресу <https://docs.confluent.io/2.0.0/clients/producer.html#examples>.

Листинг 4.2. Ожидание результата

```
RecordMetadata result =
    producer.send(producerRecord).get(); ← Ждет ответа по-
log.info("kinaction_info offset = {}, topic = {}, timestamp = {}",
        result.offset(), result.topic(), result.timestamp());
producer.close();
```

Ожидание ответа в синхронном режиме гарантирует, что код обработает результат отправки одного сообщения, прежде чем приступит к отправке другого. В данном случае для нас важна надежность доставки сообщений, а не скорость!

В предыдущих главах мы использовали пару готовых сериализаторов. Для сериализации обычных текстовых сообщений наш производитель будет использовать класс `StringSerializer`. Обсуждая Avro в главе 3, мы познакомились с классом `io.confluent.kafka.serializers.KafkaAvroSerializer`. А как быть, если мы намерены использовать свой определенный формат? Такая необходимость часто возникает, когда требуется передавать нестандартные объекты. Давайте попробуем использовать свой механизм сериализации для преобразования данных в формат, который можно передавать, хранить и затем извлекать, чтобы получить копию наших исходных данных. В листинге 4.3 показано определение нашего класса `Alert`.

Листинг 4.3. Класс Alert

```
public class Alert implements Serializable {

    private final int alertId;
    private String stageId;
    private final String alertLevel;
    private final String alertMessage;

    public Alert(int alertId,
                String stageId,
                String alertLevel,
                String alertMessage) { ← Содержит идентификатор предупрежде-
this.alertId = alertId;
this.stageId = stageId;
this.alertLevel = alertLevel;
this.alertMessage = alertMessage;
}
```

```

public int getAlertId() {
    return alertId;
}

public String getStageId() {
    return stageId;
}

public void setStageId(String stageId) {
    this.stageId = stageId;
}

public String getAlertLevel() {
    return alertLevel;
}

public String getAlertMessage() {
    return alertMessage;
}
}

```

Код в листинге 4.3 создает bean-компонент `Alert` для хранения информации, которую нужно отправить. Знакомые с Java заметят, что в определении класса присутствуют только методы доступа к свойствам и конструктор класса `Alert`. Теперь, когда у нас есть определение объекта с данными, можно реализовать класс `AlertKeySerde` для его сериализации. Определение этого класса показано в листинге 4.4.

Листинг 4.4. Класс для сериализации экземпляров Alert

```

public class AlertKeySerde implements Serializer<Alert>,
                                    Deserializer<Alert> {

    public byte[] serialize(String topic, Alert key) { ←
        if (key == null) {
            return null;
        }
        return key.getStageId()
            .getBytes(StandardCharsets.UTF_8); ←
    }

    public Alert deserialize
        (String topic, byte[] value) {
        // в будущем может возвращать Alert, если потребуется
        return null;
    }
    //...
}

```

Принимает тему и объект Alert

Преобразует объект в последовательность байтов (наша конечная цель)

Остальные методы интерфейса нам пока не нужны, поэтому в них отсутствует какая-либо конкретная логика

В листинге 4.5 мы используем этот класс только для сериализации ключей, а для сериализации значений применяем класс `StringSerializer`. Обратите внимание на интересную возможность сериализации ключей и значений в одном и том же сообщении с помощью разных сериализаторов. Правда, она требует помнить об использовании разных сериализаторов и настраивать конфигурационные значения для обоих. Класс `AlertKeySerde` реализует интерфейс `Serializer` и извлекает только поле `stageId`, которое служит ключом нашего сообщения. Это довольно простой пример, потому что основное внимание уделяется технике использования классов `serde`. Другими часто используемыми вариантами таких классов являются реализации JSON и Avro.

ПРИМЕЧАНИЕ. Термин *serde* означает *serializer/deserializer* (сериализатор/десериализатор) – класс, реализующий и сериализацию, и десериализацию [13]. В практике, однако, интерфейсы сериализации и десериализации реализуются разными классами. Но взгляните, как используются `StringSerializer` и `StringDeserializer`; разницу трудно заметить!

Также следует иметь в виду, что для десериализации значений потребитель должен знать, как эти значения были сериализованы производителем. Соответственно, должны быть выработаны соглашения по формату данных между потребителями и производителями, хотя платформе Kafka все равно, какие данные она хранит в брокерах.

Еще одна наша цель заключалась в том, чтобы зафиксировать статус уведомлений на производственных этапах, чтобы их можно было отслеживать с течением времени. Поскольку нам важна информация для каждого этапа отдельно (а не для всех датчиков одновременно), будет полезно подумать о группировке событий. В данном случае каждый этап имеет уникальный идентификатор, поэтому в качестве ключа можно использовать его. В листинге 4.5 показан пример установки свойства `key.serializer`, а также отправки уведомления с уровнем важности CRITICAL.

Листинг 4.5. Производитель с поддержкой трассировки сообщений

```
public class AlertTrendingProducer {
    private static final Logger log =
        LoggerFactory.getLogger(AlertTrendingProducer.class);

    public static void main(String[] args)
        throws InterruptedException, ExecutionException {
        Properties kaProperties = new Properties();
        kaProperties.put("bootstrap.servers",
            "localhost:9092,localhost:9093,localhost:9094");
    }
}
```

```

kaProperties.put("key.serializer",
    AlertKeySerde.class.getName()); ← | Сообщает клиенту-производи-
kaProperties.put("value.serializer",
    "org.apache.kafka.common.serialization.StringSerializer"); | телю, как сериализовать объ-
|ект Alert в ключ

try (Producer<Alert, String> producer =
    new KafkaProducer<>(kaProperties)) {

    Alert alert = new Alert(0, "Stage 0", "CRITICAL", "Stage 0 stopped");
    ProducerRecord<Alert, String> producerRecord =
        new ProducerRecord<>("kinaction_alerttrend",
            alert, alert.getAlertMessage()); ← |

    RecordMetadata result = producer.send(producerRecord).get();
    log.info("kinaction_info offset = {}, topic = {}, timestamp = {}",
        result.offset(), result.topic(), result.timestamp());
}
}
}

1. 2. Вместо null во втором па-
метре передается фактический
объект для заполнения ключа

```

Как правило, один и тот же ключ должен обеспечивать привязку к одному и тому же разделу, и ничего менять не нужно. Другими словами, одни и те же идентификаторы этапов (ключи) будут группироваться вместе только при использовании правильного ключа. В будущем мы проследим за размерами разделов и оценим, насколько неравномерно они распределены, но пока согласимся с таким решением. Также обратите внимание, что для наших конкретных классов свойства устанавливаются иначе. Это сделано для того, чтобы показать другой способ. Вместо полного пути к классу можно использовать что-то вроде `AlertKeySerde.class.getName()` или даже `AlertKeySerde.class`.

Нашим последним требованием была быстрая обработка уведомлений, чтобы операторы могли оперативно оповещаться о любых критических сбоях. При этом уведомления тоже должны группироваться по идентификатору этапа. Одна из причин такого требования – дать возможность определить, вышел ли датчик из строя или восстановился, наблюдая только последнее событие для этого этапа. Нас не интересует история проверок состояния – только текущее положение вещей. Кроме того, мы должны распределить уведомления по разделам.

До сих пор в наших примерах записи в Kafka данные направлялись в тему без дополнительных метаданных, предоставляемых клиентом. Поскольку темы состоят из разделов, размещенных в брокерах, Kafka поддерживает возможность отправки сообщений в определенные разделы по умолчанию. Так, по умолчанию к сообщениям без ключа (которые использовались в примерах до сих пор) в Kafka версии 2.4 применялась циклическая стратегия

распределения по разделам. В версиях выше 2.4 – стратегия закрепления за разделами [14]. Однако иногда может возникать необходимость распределения сообщений некоторым определенным способом. Одно из возможных решений этой задачи – написать свой уникальный класс, реализующий распределение по разделам.

Клиент также имеет возможность выбирать раздел для записи данных, настроив уникальный класс распределения по разделам. Один из примеров, где может понадобиться такое разделение, – это уровни уведомлений от нашей службы мониторинга датчиков, которые обсуждались в главе 3. Информация с одних датчиков может быть важнее, чем с других; они могут находиться на критическом пути сборки электровелосипеда, сбои на котором могут привести к простою всего конвейера, если их не устраниТЬ вовремя. Допустим, у нас есть четыре уровня уведомлений: Critical (критический), Major (серьезный), Minor (незначительный) и Warning (предупреждение). Мы могли бы создать класс распределения по разделам, помещающий сообщения с разными уровнями в разные разделы, а клиенты-потребители могли бы проверять сначала раздел с критическими уведомлениями и только потом переходить к обработке других.

Если наши потребители будут следить за журналируемыми сообщениями, то своевременная обработка критических уведомлений, вероятно, не будет большой проблемой. Однако в листинге 4.6 показано, что мы можем изменить привязку к разделам с помощью класса, чтобы гарантировать отправку критических оповещений в конкретный раздел (например, раздел 0). (Обратите внимание, что другие уведомления тоже могут оказаться в разделе 0 из-за особенностей нашей логики, но критические уведомления всегда будут попадать туда.) Логика отражает пример реализации `DefaultPartitioner`, используемой в самой Kafka [15].

Листинг 4.6. Класс для распределения уведомлений по разделам

```
public int partition(final String topic  ← AlertLevelPartitioner должен
                    # ...                                реализовать метод выбора
                                                 раздела

int criticalLevelPartition = findCriticalPartitionNumber(cluster, topic);
    return isCriticalLevel(((Alert) objectKey).getAlertLevel()) ?
        criticalLevelPartition :
        findRandomPartition(cluster, topic, objectKey);   ←
    }
//...
```

Критические уведомления должны помещаться в раздел, возвращаемый методом `findCriticalPartitionNumber`

Реализация интерфейса `Partitioner` может использовать метод `partition` для возврата определенного раздела, куда должен записать сообщение производитель. В этом случае значение ключа гарантирует, что любое событие с уровнем важности `CRITICAL` попадет в определенное место; например, можно представить, что метод `findCriticalPartitionNumber` возвращает раздел 0. Помимо определения самого класса в листинге 4.7 показано, как можно установить конфигурационный ключ `partitioner.class`, чтобы производитель использовал созданный нами конкретный класс. Конфигурация, которая управляет Kafka, предписывает использовать наш новый класс.

Листинг 4.7. Настройка класса `partitioner`

```
Properties kaProperties = new Properties();
//...
kaProperties.put("partitioner.class", <-- Дополнить конфигурацию
                  производителя ссылкой на
                  класс распределения по раз-
                  делам AlertLevelPartitioner
                  AlertLevelPartitioner.class.getName());
```

Этот пример, в котором всегда возвращается определенный номер раздела, можно расширить или сделать еще более динамичным. Например, можно использовать свой код для выполнения конкретной логики наших бизнес-потребностей.

В листинге 4.8 показана конфигурация производителя, добавляющая значение `partitioner.class` для использования нашего конкретного класса распределения по разделам. Цель состоит в том, чтобы обеспечить доступность данных в определенном разделе и потребители, обрабатывающие данные, могли извлекать критические уведомления отдельно и отдельно обращаться к другим уведомлениям (в других разделах).

Листинг 4.8. Производитель уведомлений `Alert`

```
public class AlertProducer {
    public static void main(String[] args) {
        Properties kaProperties = new Properties();
        kaProperties.put("bootstrap.servers",
                        "localhost:9092,localhost:9093");
        kaProperties.put("key.serializer",
                        AlertKeySerde.class.getName()); <-- Повторно использовать се-
                                                       риализатор ключей Alert
        kaProperties.put("value.serializer",
                        "org.apache.kafka.common.serialization.StringSerializer");
        kaProperties.put("partitioner.class",
                        AlertLevelPartitioner.class.getName()); <-- Использовать свойство
                                                       partitioner.class для вы-
                                                       бора конкретного класса
                                                       распределения по раз-
                                                       делам
        try (Producer<Alert, String> producer =
             new KafkaProducer<>(kaProperties)) {
            Alert alert = new Alert(1, "Stage 1", "CRITICAL", "Stage 1 stopped");
            ProducerRecord<Alert, String>
```

```

producerRecord = new ProducerRecord<>
    ("kinaction_alert", alert, alert.getAlertMessage());

    producer.send(producerRecord,
        new AlertCallback()); <-- | Это первый раз, когда мы
    }                                | использовали обратный
}                                | вызов для обработки
}                                | удачной или неудачной
}                                | отправки сообщения
}
}
}

```

Одно из новшеств, которые можно видеть в листинге 4.8, – это использование обратного вызова для получения результата отправки сообщения. Мы уже говорили, что 100%-ная гарантия отправки нужна не для всех сообщений, но иногда желательно убедиться, что частота сбоев, которая может служить признаком ошибки в приложении, не слишком высокая. В листинге 4.9 показан пример реализации интерфейса `Callback`. В случае ошибки обратный вызов посыпает сообщение. Обратите внимание, что этот листинг основан на примерах, доступных по адресу <https://docs.confluent.io/2.0.0/clients/producer.html#examples>.

Листинг 4.9. Обратный вызов для обработки результата отправки уведомления Alert

```

public class AlertCallback implements Callback { <-- | Реализует интер-
    private static final Logger log = <-- | фейс Callback из
        LoggerFactory.getLogger(AlertCallback.class); <-- | Kafka

    public void onCompletion
        (RecordMetadata metadata, <-- | Результат может сообщать
         Exception exception) { <-- | об успехе или неудаче

        if (exception != null) {
            log.error("kinaction_error", exception);
        } else {
            log.info("kinaction_info offset = {}, topic = {}, timestamp = {}",
                    metadata.offset(), metadata.topic(), metadata.timestamp());
        }
    }
}

```

В большей части книги мы представляем короткие примеры, но считаем, что вам будет полезно увидеть, как использовать производителя в реальном проекте. Как упоминалось выше, для предоставления различных возможностей обработки данных вместе с Kafka можно использовать Apache Flume. Когда Kafka используется в качестве приемника, Flume помещает данные в Kafka. Кто-то из вас знаком с Flume, кто-то не знаком, но нас не интересует набор возможностей этого фреймворка. Мы лишь хотим показать, как он использует код производителя Kafka в реальной ситуации.

В следующих примерах используется Flume версии 1.8 (если у вас появится желание заглянуть в исходный код фреймворка, то вы найдете его по адресу <https://github.com/apache/flume/tree/flume-1.8>). В листинге 4.10 показан фрагмент конфигурации, определяющей настройки агента Flume.

Листинг 4.10. Конфигурация приемника Flume

```
a1.sinks.k1.kafka.topic = kinaction_helloworld
a1.sinks.k1.kafka.bootstrap.servers = localhost:9092
a1.sinks.k1.kafka.producer.acks = 1
a1.sinks.k1.kafka.producer.compression.type = snappy
```

Некоторые конфигурационные свойства из листинга 4.10 могут показаться знакомыми: `topic`, `acks`, `bootstrap.servers`. В предыдущих примерах мы определяли настройки через свойства внутри нашего кода. Однако в листинге 4.10 показан пример приложения, которое выносит конфигурационные значения во внешний файл (в наших проектах мы могли бы поступить так же). Исходный код KafkaSink из Apache Flume (доступен по адресу <http://mng.bz/JvpZ>) демонстрирует пример получения и передачи данных в Kafka с помощью производителя. Листинг 4.11 демонстрирует еще один пример производителя, использующего аналогичную идею. Он берет файл конфигурации, аналогичный представленному в листинге 4.10, и загружает значения из него в экземпляр производителя.

Листинг 4.11. Чтение конфигурации производителя Kafka из файла

```
...
Properties kaProperties = readConfig();
String topic = kaProperties.getProperty("topic");
kaProperties.remove("topic");

try (Producer<String, String> producer =
      new KafkaProducer<>(kaProperties)) {
    ProducerRecord<String, String> producerRecord =
        new ProducerRecord<>(topic, null, "event");
    producer.send(producerRecord,
                  new AlertCallback()); ← Уже знакомый нам producer.
}                                         send с обратным вызовом

private static Properties readConfig() {
    Path path = Paths.get("src/main/resources/kafkasink.conf");

    Properties kaProperties = new Properties(); ← Читает внешний файл
    try (Stream<String> lines = Files.lines(path)) ← с конфигурацией
        lines.forEachOrdered(line ->
            determineProperty(line, kaProperties));
    } catch (IOException e) {
        System.out.println("kinaction_error" + e);
    }
}
```

```

    return kaProperties;
}

private static void determineProperty
    (String line, Properties kaProperties) {
    if (line.contains("bootstrap")) {
        kaProperties.put("bootstrap.servers", line.split("=")[1]);
    } else if (line.contains("acks")) {
        kaProperties.put("acks", line.split("=")[1]);
    } else if (line.contains("compression.type")) {
        kaProperties.put("compression.type", line.split("=")[1]);
    } else if (line.contains("topic")) {
        kaProperties.put("topic", line.split("=")[1]);
    }
    ...
}

```

Разбирает конфигурационные свойства и устанавливает значения

Часть кода в листинге 4.11 опущена, однако основные элементы производителя Kafka могут показаться вам знакомыми. Настройка конфигурации и метод `send` производителя выглядят так же, как в примерах, представленных выше в этой главе. И теперь, надеемся, вы сможете с уверенностью сказать, какие конфигурационные свойства необходимо настраивать и какое влияние они оказывают.

Теперь за вами остается одно упражнение: сравнить, как `AlertCallback.java` сочетается с классом обратного вызова `SinkCallback` из Kafka Sink (исходный код доступен по адресу <http://mng.bz/JvpZ>). В обоих примерах для получения дополнительной информации об успешных вызовах используется объект `RecordMetadata`. Эта информация может помочь узнать больше о том, куда было записано сообщение производителя, включая раздел и смещение в этом конкретном разделе.

Вы также можете использовать такие приложения, как Flume, даже не копаясь в их исходном коде, и при этом добиваться успеха. Однако мы считаем, что вам стоит узнать, как они работают, чтобы уметь устранять неполадки. Теперь, получив новые базовые знания о производителях, вам должно быть очевидно, что можно самостоятельно создавать мощные приложения, используя эти методы.

4.3.1. Версии клиентов и брокеров

Важно отметить, что версии брокеров и клиентов Kafka не всегда должны совпадать. Если вы используете брокер Kafka версии 0.10.0 и клиента-производителя версии 0.10.2, то брокер будет учитывать различия в версиях сообщений [16]. Однако такая возможность не означает, что вы должны поступать так во всех случаях. Чтобы узнать больше о совместимости версий, загляните в KIP-97 (<http://mng.bz/7jAQ>).

Мы продвинулись далеко вперед, начав загружать данные в Kafka. Теперь, углубившись в экосистему Kafka, мы должны обсудить некоторые другие концепции, прежде чем закончим создание нашего

комплексного решения. Следующий вопрос, который следует рассмотреть: как извлекать записанные данные, чтобы иметь возможность использовать их в других приложениях? У нас уже есть некоторое представление о записи данных в Kafka, поэтому далее мы постараемся узнать больше о том, как сделать эти данные полезными для других приложений, извлекая их правильными способами. Клиенты-потребители являются жизненно важной частью системы и, так же как в случае с производителями, могут реализовать различные модели поведения, определяемые конфигурацией и помогающие удовлетворить различные требования к потреблению.

Итоги

- Клиенты-производители предоставляют возможность передавать данные в Kafka.
- Для управления поведением клиента можно использовать множество конфигурационных параметров.
- Данные хранятся в брокерах в так называемых разделах.
- Клиент может определять, в какой раздел записываются данные, предоставляя свою логику реализации интерфейса `Partitioner`.
- Kafka обычно рассматривает данные как последовательность байтов. Однако для работы с некоторыми форматами данных можно использовать пользовательские сериализаторы.

Ссылки

- 1 J. Kreps. «Why Avro for Kafka Data?» Confluent blog (25 февраля 2015). <https://www.confluent.io/blog/avro-kafka-data/> (доступно по состоянию на 23 ноября 2017).
- 2 «Sender.java». Apache Kafka. GitHub (n.d.). <https://github.com/apache/kafka/blob/299eea88a5068f973dc055776c7137538ed01c62/clients/src/main/java/org/apache/kafka/clients/producer/internals/Sender.java> (доступно по состоянию на 20 августа 2021).
- 3 «Producer Configurations: Retries». Confluent documentation (n.d.). https://docs.confluent.io/platform/current/installation/configuration/producer-configs.html#producerconfigs_retries (доступно по состоянию на 29 мая 2020).
- 4 «Producer Configurations: max.in.flight.requests.per.connection». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/installation/configuration/producer-configs.html#max.in.flight.requests.per.connection> (доступно по состоянию на 29 мая 2020).
- 5 «Producer Configurations: enable.idempotence». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/>

- installation/configuration/producer-configs.html#producerconfigs_enable.idempotence (доступно по состоянию на 29 мая 2020).
- 6 «KafkaProducer». Apache Software Foundation (n.d.). <https://kafka.apache.org/10/javadoc/org/apache/kafka/clients/producer/KafkaProducer.html> (доступно по состоянию на 7 июля 2019).
 - 7 «Producer Configurations». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/installation/configuration/producer-configs.html> (доступно по состоянию на 29 мая 2020).
 - 8 «Producer Configurations: bootstrap.servers». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/installation/configuration/producer-configs.html#bootstrap.servers> (доступно по состоянию на 29 мая 2020).
 - 9 «Producer Configurations: acks». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/installation/configuration/producer-configs.html#acks> (доступно по состоянию на 29 мая 2020).
 - 10 «Documentation: Message Delivery Semantics». Apache Software Foundation (n.d.). <https://kafka.apache.org/documentation/#semantics> (доступно по состоянию на 30 мая 2020).
 - 11 «Topic Configurations: message.timestamp.type». Confluent documentation (n.d.). https://docs.confluent.io/platform/current/installation/configuration/topicconfigs.html#topicconfigs_message.timestamp.type (доступно по состоянию на 22 июля 2020).
 - 12 KIP-42: «Add Producer and Consumer Interceptors», Wiki for Apache Kafka, Apache Software Foundation. <https://cwiki.apache.org/confluence/display/KAFKA/KIP-42%3A+Add+Producer+and+Consumer+Interceptors> (доступно по состоянию на 15 апреля 2019).
 - 13 «Kafka Streams Data Types and Serialization». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/streams/developer-guide/datatypes.html> (доступно по состоянию на 21 августа 2021).
 - 14 J. Olshan. «Apache Kafka Producer Improvements with the Sticky Partitioner». Confluent blog (18 декабря 2019). <https://www.confluent.io/blog/apache-kafka-producer-improvements-sticky-partitioner/> (доступно по состоянию на 21 августа 2021).
 - 15 «DefaultPartitioner.java», Apache Software Foundation. GitHub (n.d.). <https://github.com/apache/kafka/blob/trunk/clients/src/main/java/org/apache/kafka/clients/producer/internals/DefaultPartitioner.java> (доступно по состоянию на 22 марта 2020).
 - 16 C. McCabe. «Upgrading Apache Kafka Clients Just Got Easier». Confluent blog (18 июля 2017). <https://www.confluent.io/blog/upgrading-apache-kafka-clients-just-got-easier/> (доступно по состоянию на 21 августа 2021).

5

Потребители: извлечение данных

Эта глава охватывает следующие темы:

- потребители: особенности работы;
- использование групп потребителей для координации чтения данных из тем;
- смещения и способы их использования;
- различные варианты настроек, меняющих поведение потребителей.

В предыдущей главе мы начали записывать данные в Kafka. Однако, как вы понимаете, это только одна часть истории. Потребители извлекают данные из Kafka и предоставляют их другим системам или приложениям. Поскольку потребители являются клиентами, существующими вне брокеров, они могут быть написаны на разных языках программирования (как и клиенты-производители). Обратите внимание, что при знакомстве с примерами извлечения данных в этой главе мы постараемся использовать настройки по умолчанию везде, где это возможно. Прочитав эту главу, мы сможем приступить к решению наших предыдущих бизнес-требований, используя данные несколькими различными способами.

5.1. Пример

Клиент-потребитель – это программа, которая подписывается на интересующую ее тему или несколько тем [1]. Так же как в случае с клиентами-производителями, фактические процессы-потребители могут выполняться на отдельных машинах и необязательно на конкретном сервере. На самом деле большинство клиентов-потребителей в промышленных окружениях выполняются на отдельных хостах. Если клиенты имеют возможность подключаться к брокерам Kafka, они смогут читать сообщения. На рис. 5.1 снова перечислены возможности Kafka и показаны потребители, работающие вне брокеров и получающие данные из Kafka.

Почему важно понимать, что потребитель подписывается на темы (сам извлекает сообщения), а не получает их по инициативе брокера? В этой ситуации управление обработкой данных целиком и полностью возлагается на потребителя. На рис. 5.1 показано, как клиенты-потребители вписываются в общую экосистему Kafka. Клиенты несут ответственность за чтение данных из тем и их передачу приложениям (например, информационным панелям или аналитическим системам) или сохранение в других системах. Потребители сами управляют скоростью потребления.

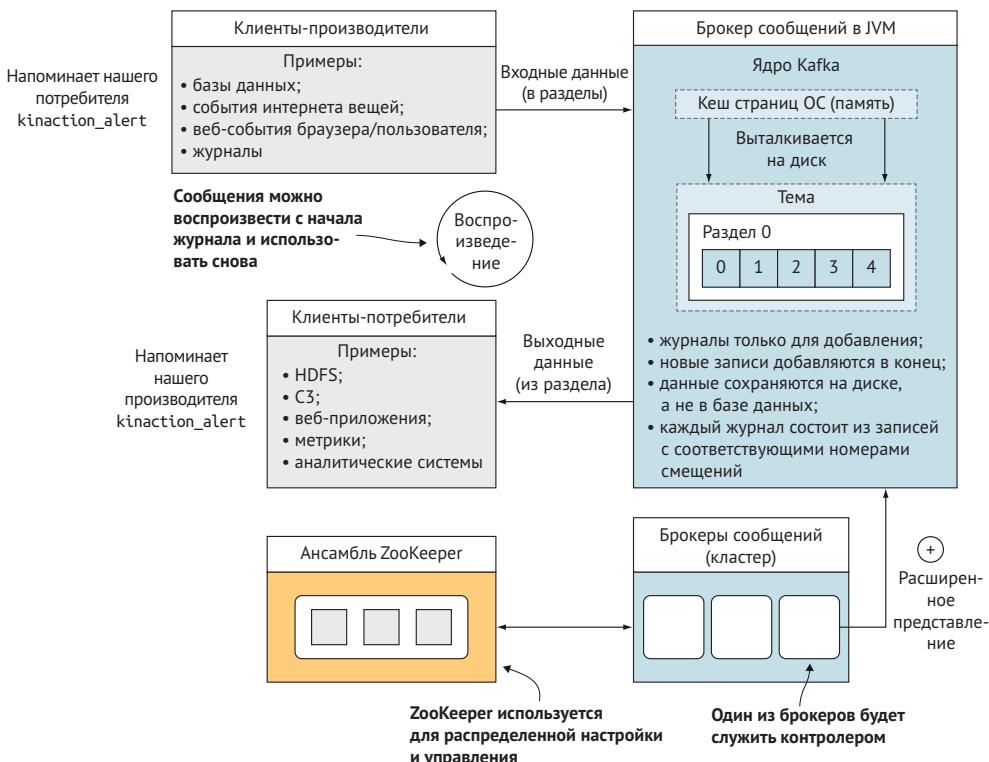


Рис. 5.1. Потребители Kafka

Если в приложении с потребителями в кресле водителя произойдет сбой, то после восстановления работоспособности оно сможет начать извлекать данные с того места, на котором остановилось. Нет необходимости держать потребителей постоянно работающими для обработки уведомлений. Хотя вы можете разработать приложение, способное обрабатывать постоянный поток данных и даже оказывать обратное давление в моменты, когда приложение не будет успевать обрабатывать данные, но тогда это получится не потребитель брокеров; потребитель – это тот, кто извлекает данные сам, а не получает их автоматически. Однако те из вас, кто имеет опыт использования Kafka, могут знать, что есть причины, почему бывает нежелательно отключать потребителей на длительные периоды времени. Когда мы начнем более подробно обсуждать темы, вы увидите, что данные могут удаляться из Kafka из-за ограничений по размеру или времени, которые могут определить пользователи.

5.1.1. Параметры потребителя

В нашем обсуждении вы заметите несколько свойств, связанных со свойствами клиентов-производителей. При создании потребителя мы всегда должны знать, к каким брокерам подключаться на запуске. Также важно знать, какие десериализаторы использовать для извлечения ключей и значений из сообщений. Например, если сообщение было произведено с помощью `StringSerializer`, то при попытке использовать `LongDeserializer` вы получите исключение, которое вам нужно будет обработать.

В табл. 5.1 перечислены некоторые конфигурационные параметры – их следует знать, приступая к разработке потребителей [2].

Таблица 5.1. Важные параметры настройки потребителя

Параметр	Назначение
<code>bootstrap.servers</code>	Список брокеров, с которыми должно устанавливаться соединение на запуске
<code>value.deserializer</code>	Класс, используемый для десериализации значения
<code>key.deserializer</code>	Класс, используемый для десериализации ключа
<code>group.id</code>	Имя, используемое для присоединения к группе потребителей
<code>client.id</code>	Идентификатор пользователя (будет использоваться в главе 10)
<code>heartbeat.interval.ms</code>	Интервал проверки связи потребителя с координатором группы

Один из способов справиться с многочисленными параметрами конфигурации потребителя при его разработке – использовать константы, определяемые классом `ConsumerConfig` (<http://mng.bz/oGgy>) и подробно описанные на сайте Confluent (особое внимание обращайте на параметры, снабженные меткой **Importance: high** (Важность: высокая)) (<http://mng.bz/drdf>). Однако в наших примерах для

ясности мы будем использовать сами имена свойств. В листинге 5.1 показан пример использования четырех из этих параметров. Значения параметров, перечисленных в табл. 5.1, определяют особенности взаимодействия потребителя с брокерами и другими потребителями.

Теперь давайте переключимся на чтение из темы с одним потребителем, как мы делали это в главе 2. Для этого примера рассмотрим приложение, демонстрирующее, с чего могло бы начаться применение Kafka в LinkedIn, и обрабатывающее события действий пользователей (как упоминалось в главе 1) [3]. Допустим, у нас есть определенная формула, позволяющая оценить время, проведенное пользователем на странице, а также количество выполненных им операций. Эти данные отправляются в виде значения в тему для прогнозирования щелчков мышью в будущем после начала новой акции. Представьте, что мы запускаем потребителя и обрабатываем все сообщения, имеющиеся в теме, и нас вполне устраивает используемая формула (в данном случае она просто выполняет умножение на магическое число).

В листинге 5.1 показан пример просмотра сообщений из раздела `kinaction_promos` и вывода значения на основе данных из всех событий. Этот листинг во многом похож на код производителя, который мы написали в главе 4, где свойства используются для настройки поведения потребителя. Такое использование десериализаторов для ключей и значений отличается от использования сериализаторов в производителях и может меняться в зависимости от темы.

ПРИМЕЧАНИЕ. Листинг 5.1 содержит не весь код, а только ту его часть, которая подчеркивает интересующие нас особенности потребителей. Помните, что потребитель может подписаться на несколько тем, но в данном случае нас интересует только тема `kinaction_promos`.

Цикл в листинге 5.1 используется для опроса разделов тем, назначенных потребителю для обработки сообщений. Этот цикл переключается логическим значением. Подобные циклы могут вызвать ошибки, особенно у начинающих программистов! Зачем он нужен? Согласно потоковому мышлению, события интерпретируются как непрерывный поток, и это отражено в логике. Обратите внимание, что в этом примере продолжительность опроса ограничивается 250 мс. Этот тайм-аут указывает, как долго может блокироваться основной поток приложения в ожидании ответа, который, впрочем, может вернуться немедленно, если записи готовы к доставке [4]. Это значение можно точно настроить в зависимости от потребностей приложений. Ссылку (и более подробную информацию) на описание стиля оформления тайм-аутов в Java 8 с использованием `addShutdownHook`, который мы используем в примере, можно найти по адресу <https://docs.confluent.io/platform/current/streams/developer-guide/write-streams.html>.

Листинг 5.1. Потребитель информации для рекламных акций

```

...
private volatile boolean keepConsuming = true;      Определяет group.id.
                                                    (Мы коснемся этого
public static void main(String[] args) {          параметра при обсуж-
    Properties kaProperties = new Properties();     дении групп потреби-
    kaProperties.put("bootstrap.servers",
                     "localhost:9092,localhost:9093,,localhost:9094");
    kaProperties.put("group.id",
                     "kinaction_webconsumer");           ←
    kaProperties.put("enable.auto.commit", "true");
    kaProperties.put("auto.commit.interval.ms", "1000");
    kaProperties.put("key.deserializer",
                     "org.apache.kafka.common.serialization.StringDeserializer");
    kaProperties.put("value.deserializer",
                     "org.apache.kafka.common.serialization.StringDeserializer");
    WebClickConsumer webClickConsumer = new WebClickConsumer();
    webClickConsumer.consume(kaProperties);           Определяют десериа-
                                                    лизаторы для ключей
Runtime.getRuntime()                           и значений
    .addShutdownHook(
        new Thread(webClickConsumer::shutdown)
    );
}

private void consume(Properties kaProperties) {      Передает свой-
    try (KafkaConsumer<String, String> consumer =         ства в конструктор
        new KafkaConsumer<>(kaProperties)) {           ←
        consumer.subscribe(
            List.of("kinaction_promos")           ←
        );                                         Подписывается на одну тему
                                                    kinaction_promos
        while (keepConsuming) {                ←
            ConsumerRecords<String, String> records =   Использует цикл для
            consumer.poll(Duration.ofMillis(250));       опроса сообщений
            for (ConsumerRecord<String, String> record : records) {
                log.info("kinaction_info offset = {}, key = {}",
                          record.offset(),
                          record.key());
                log.info("kinaction_info value = {}",
                          Double.parseDouble(record.value()) * 1.543);
            }
        }
    }
}

private void shutdown() {
    keepConsuming = false;
}

```

The diagram consists of several callout boxes with arrows pointing to specific parts of the code:

- An arrow points from the line `kaProperties.put("group.id", "kinaction_webconsumer");` to the note: "Определяет group.id. (Мы коснемся этого параметра при обсуждении групп потребителей.)".
- Two arrows point from the lines `key.deserializer` and `value.deserializer` to the note: "Определяют десериализаторы для ключей и значений".
- An arrow points from the line `new KafkaConsumer<>(kaProperties)` to the note: "Передает свойства в конструктор KafkaConsumer".
- An arrow points from the line `List.of("kinaction_promos")` to the note: "Подписывается на одну тему kinaction_promos".
- An arrow points from the line `while (keepConsuming)` to the note: "Использует цикл для опроса сообщений в теме".

Сгенерировав значение для каждого сообщения темы в листинге 5.1, мы обнаруживаем, что наша формула моделирования неверна! И что теперь делать? Попытаться пересчитать данные, которые у нас есть, из последних результатов (при условии, что исправление будет сложнее, чем в примере), а затем применить новую формулу?

Здесь мы можем использовать наши знания о поведении потребителей в Kafka и повторно получить уже обработанные сообщения. Благодаря сохранности исходных данных можно не беспокоиться о воссоздании исходных данных. Ошибки разработчика, ошибки логики приложения и даже сбои зависимых приложений можно исправить, потому что данные не удаляются из тем после их потребления. Это также объясняет, как стало возможным путешествие во времени для Kafka.

Давайте посмотрим, как остановить нашего потребителя. Вы уже видели, что в терминале можно нажать комбинацию **Ctrl-C**, чтобы завершить обработку или остановить процесс. Однако правильнее будет вызвать метод `close` потребителя [23].

В листинге 5.2 показан потребитель, выполняющийся в отдельном потоке. Его завершением управляет другой класс. После запуска код в листинге 5.2 запустит еще один поток выполнения с экземпляром потребителя. Вызывая публичный метод `shutdown`, другой класс может изменить логическое значение и заставить потребителя прекратить получение новых сообщений. Переменная `stopping` – это наша защита. Она используется для решения, когда прекратить обработку. Вызов метода `wakeup` также приводит к возникновению исключения `WakeUpException` и закрытию потребительского ресурса в блоке `finally` [5]. Код в листинге 5.2 написан на основе справочной документации, доступной по адресу <https://kafka.apache.org/26/javadoc/index.html?org/apache/kafka/clients/consumer/KafkaConsumer.html>.

Листинг 5.2. Закрытие потребителя

```
public class KinactionStopConsumer implements Runnable {  
    private final KafkaConsumer<String, String> consumer;  
    private final AtomicBoolean stopping =  
        new AtomicBoolean(false);  
    ...  
  
    public KinactionStopConsumer(KafkaConsumer<String, String> consumer) {  
        this.consumer = consumer;  
    }  
  
    public void run() {  
        try {  
            consumer.subscribe(List.of("kinaction_promos"));  
        } catch (Exception e) {  
            consumer.close();  
        }  
        while (!stopping.get()) {  
            consumer.poll(Duration.ofMillis(100));  
        }  
    }  
}
```

```

while (!stopping.get()) {
    ConsumerRecords<String, String> records =
        consumer.poll(Duration.ofMillis(250));
    ...
}
} catch (WakeupException e) {
    if (!stopping.get()) throw e;
} finally {
    consumer.close();
}
}

public void shutdown() {
    stopping.set(true);
    consumer.wakeup();
}
}

```

Переменная `stopping` определяет необходимость прекращения обработки

Обработчик события завершения работы клиента вызывает исключение `WakeupException`

Останавливает клиента и информирует брокера о завершении работы

Вызывает `shutdown` из другого потока, чтобы правильно остановить клиента

Прежде чем продолжить обсуждение, нам нужно разобраться с понятием смещений: как их можно использовать для управления чтения данных потребителями.

5.1.2. Наши координаты в потоке событий

Один из аспектов, упоминавшийся лишь вскользь, – это понятие *смещений*. Смещения играют роль индексов сообщений в журнале, которые потребитель посыпает брокеру.

Индексы позволяют брокеру понять, какие сообщения хочет получить потребитель. Вспомните, как в примере с консольным потребителем мы использовали флаг `--from-beginning`. Этот флаг устанавливает конфигурационный параметр потребителя `auto.offset.reset` в значение `earliest`. С такой настройкой параметра потребитель получит все сообщения из заданной темы, на которую он подписан, даже если они были отправлены до того, как потребитель был запущен. Вверху на рис. 5.2 показано, что чтение журнала начинается с начала каждый раз, когда потребитель запускается в этом режиме.



Рис. 5.2. Смещения в Kafka [6]

По умолчанию параметр `auto.offset.reset` принимает значение `latest`. Работа в этом режиме тоже показана на рис. 5.2. В этом случае вы не увидите никаких сообщений от производителя, отправленных до запуска потребителя. Это значение параметра требует игнорировать сообщения, уже находящиеся в разделе темы, откуда их читает потребитель, и обрабатывать только поступившие после того, как клиент-потребитель начнет опрашивать тему. Тему можно рассматривать как бесконечный массив, нумерация сообщений в котором начинается с 0. При этом существующие сообщения не могут изменяться. Любые изменения должны добавляться в конец журнала.

Обратите внимание, что смещения всегда только увеличиваются. После того как в разделе темы появится сообщение со смещением 0, этот номер смещения больше никогда не будет использоваться, даже если позже соответствующее сообщение будет удалено. Кто-то из вас, возможно, сталкивался с проблемой исчерпания диапазона представления чисел, которые продолжают увеличиваться до достижения верхней границы типа данных. Каждый раздел имеет свою последовательность смещений, поэтому есть надежда, что риск исчерпания будет невелик.

Но как найти сообщение в теме? По каким координатам его искать? Сначала нужно найти раздел в теме, куда это сообщение было записано, а затем его смещение на основе индекса. Как показано на рис. 5.3, потребители обычно читают сообщения из ведущей реплики раздела потребителя. Ведущая реплика потребителя может отличаться от ведущей реплики (лидера) производителя из-за смены лидерства с течением времени; однако концептуально они схожи.

Topic: 3 partitions, 2 replicas

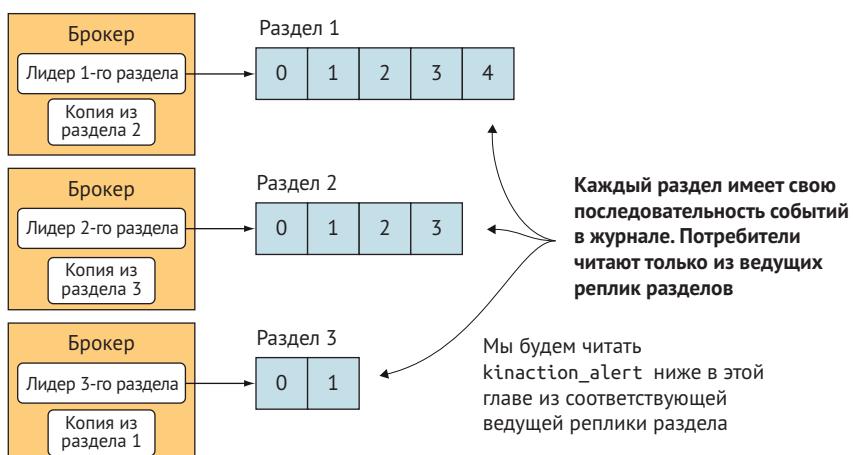


Рис. 5.3. Лидеры разделов

Кроме того, рассуждая о разделах, необходимо учитывать, что все разделы могут иметь одинаковый номер смещения. Чтобы различать сообщения, необходимо уточнять, о каком разделе в теме идет речь, а также смещение.

Отметим, что иногда может понадобиться получить данные из ведомой реплики из-за проблем, таких как высокая задержка в сети (такое может случиться, например, если кластер охватывает несколько центров обработки данных). В KIP-392 описана эта возможность, и она была реализована в версии 2.4.0 [7]. Для тех, кто только приступает к работе со своими первыми кластерами, мы рекомендуем начинать с поведения по умолчанию и использовать эту возможность, только когда это действительно необходимо. Если у вас нет кластера, охватывающего несколько разных физических сайтов, то, скорее всего, вам не понадобится эта возможность.

Разделы играют важную роль в обработке сообщений. Тема определяет логическое имя того, что интересует потребителей, но читать сообщения они будут из ведущих реплик назначенных им разделов. Но как потребители узнают к какому разделу подключаться? И не только раздел, но и ведущую реплику этого раздела? Для каждой группы потребителей конкретный брокер принимает на себя роль координатора группы [8]. Клиент-потребитель соединяется с этим координатором и получает от него назначенный раздел вместе с другими деталями, необходимыми для чтения сообщений.

Когда речь заходит о потреблении, немаловажную роль играет также количество разделов. Некоторые потребители не получат ничего, если потребителей больше, чем разделов, например когда имеется четыре потребителя и только три раздела. Почему это нормальная ситуация? Иногда бывает желательно сохранить постоянной скорость потребления, если один из потребителей неожиданно завершится. *Координатор группы* отвечает за назначение разделов потребителям не только в начале запуска группы, но и когда они добавляются в группу или терпят неудачу и покидают ее [8]. А в случае, когда разделов больше, чем потребителей, потребители могут обслуживать несколько разделов, если это необходимо.

На рис. 5.4 показана ситуация, когда четыре потребителя читают все данные из брокеров, подписавшись на тему с равномерно распределенными ведущими репликами разделов, по одному в каждом из трех брокеров. На этом рисунке данные имеют примерно одинаковый объем, однако в реальности так бывает не всегда. Один потребитель простояивает без работы, потому что каждая ведущая реплика раздела обрабатывается только одним потребителем.

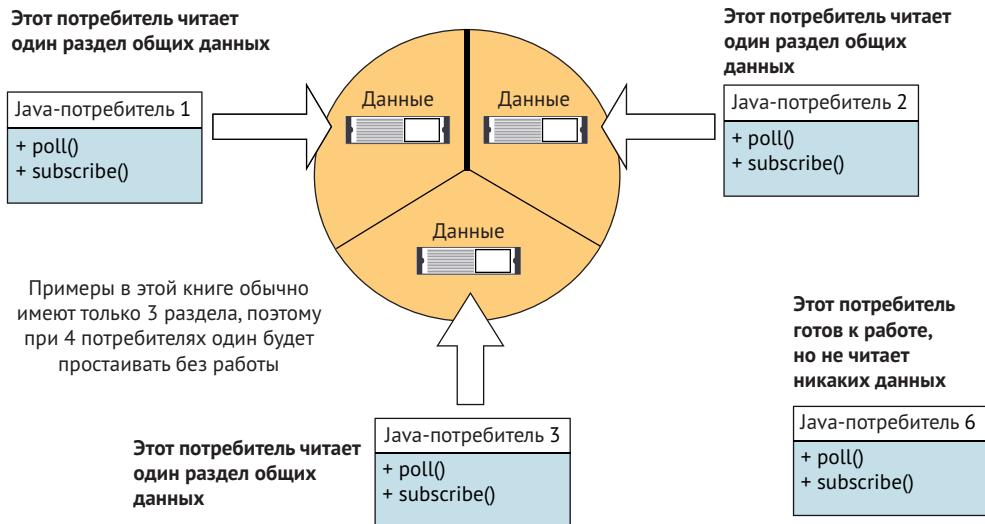


Рис. 5.4. Лишний потребитель Kafka

Поскольку количество разделов определяет количество потребителей, активно действующих параллельно, возникает резонный вопрос: почему бы не создать большое количество разделов, например 500? Дело в том, что такой способ увеличить пропускную способность имеет свои недостатки [9]. Именно поэтому старайтесь выбирать такую конфигурацию, которая лучше всего соответствует форме вашего потока данных.

Один из ключевых недостатков конфигурации с большим количеством разделов является увеличение сквозной задержки в разделах. Если в вашем приложении счет идет на миллисекунды, то может случиться так, что при отправке сообщения вы не дождитесь в отведенный интервал времени, пока раздел будет скопирован между брокерами [9]. Это особенно относится к синхронизированным репликам и влияет на время, через которое сообщение будет доступно потребителям. Также необходимо следить за объемом памяти, занимаемым вашими потребителями. Если у вас не производится однозначное сопоставление разделов потребителям «1-к-1», то требования к памяти каждого потребителя могут возрасти, потому что им будет назначаться больше разделов [9].

В старой документации по Kafka можно заметить описание конфигурации клиента-потребителя для Apache ZooKeeper. Если вы не используете старых клиентов-потребителей, то Kafka не позволит потребителям напрямую использовать ZooKeeper. Старые потребители использовали ZooKeeper для хранения смещений, теперь же смещения обычно хранятся во внутренней теме Kafka [10]. Дополнительно отметим, что клиенты-потребители не обязаны сохранять свои смещения ни в одном из этих мест, но нередко делают это.

Если у вас появится желание организовать свое хранилище смещений, то этому нет никаких препятствий! Смещения можно хранить в локальном файле, в облачном хранилище у поставщика, такого как AWS™, или в базе данных. Одно из преимуществ отказа от хранилища ZooKeeper – уменьшение зависимости клиентов от ZooKeeper.

5.2. Как взаимодействуют потребители

Почему идея организации потребителей в группы имеет первостепенное значение? Вероятно, самая важная причина заключается в масштабировании, на которое влияет добавление или удаление потребителей из группы. Потребители, не являющиеся частью одной и той же группы, не координируют информацию о смещении.

В листинге 5.3 показан пример группы с именем `kinaction_team0group`. Если создать новую группу с другим идентификатором в `group.id` (например, выбрав случайный GUID), то вы запустите нового потребителя, не имеющего сохраненных смещений и других потребителей в этой группе [11]. Если присоединить нового потребителя к существующей группе (или к той, в которой уже имеются сохраненные смещения), то новый потребитель может снять часть работы с существующих потребителей или даже продолжить чтение с того места, на котором он остановился в прошлый раз [1].

Листинг 5.3. Конфигурация потребителя для включения в группу

```
Properties kaProperties = new Properties();
kaProperties.put("group.id", "kinaction_team0group");
```

group.id определяет поведение потребителя по отношению к другим потребителям

Часто бывает так, что несколько потребителей читают одну и ту же тему. Поэтому важно решить, нужен ли новый идентификатор группы – работают ли потребители как части одного приложения, или это отдельные логические потоки. Почему это важно?

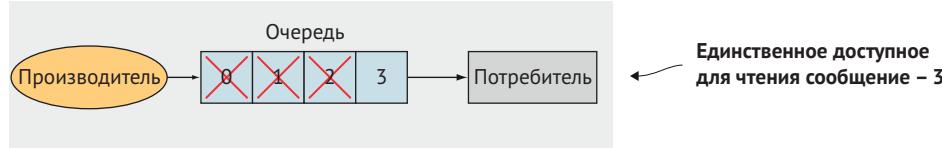
Представим два варианта использования данных, поступающих из системы управления персоналом. Одна команда интересуется количеством нанятых сотрудников из конкретных областей и краев, а другую больше интересуют затраты на поездки на собеседование. Будет ли кто-то из первой команды интересоваться тем, что делает другая команда, и устроит ли обе команды тот факт, что они получат только часть сообщений? Скорее всего, нет! Как правильно разделить их? Для этого можно присвоить каждому приложению свой идентификатор `group.id`. Все потребители, использующие один тот же идентификатор `group.id`, будут считаться работающими вместе, совместно использующими разделы и смещения в теме и составляющими одно логическое приложение.

5.3. Трассировка

До сих пор, обсуждая наши модели использования, мы почти ничего не говорили об учете того, что прочитал каждый клиент. Давайте кратко разберем, как брокеры обрабатывают сообщения в некоторых других системах. В некоторых системах потребители не оставляют на месте то, что они прочитали. Они извлекают сообщение, которое после подтверждения получения удаляется из очереди. Такая организация хорошо подходит для одиночных сообщений, что должны обрабатываться только одним приложением. Некоторые системы используют темы, в которых публикуют сообщения для всех подписчиков. И часто подписчики не получают опубликованных сообщений, потому что отсутствовали в списке получателей на момент публикации.

На рис. 5.5 показаны сценарии работы брокера сообщений, отличного от Kafka, в которых сообщения часто удаляются после их получения потребителями. Здесь также показан второй сценарий, когда сообщение может поступать из источника, а затем копироваться в несколько очередей. В системах, где сообщение доступно только одному потребителю, такой подход необходим, чтобы обеспечить доставку копии каждому отдельному приложению.

Чтение выполняется один раз и подтверждается



Несколько потребителей должны получить одно и то же сообщение

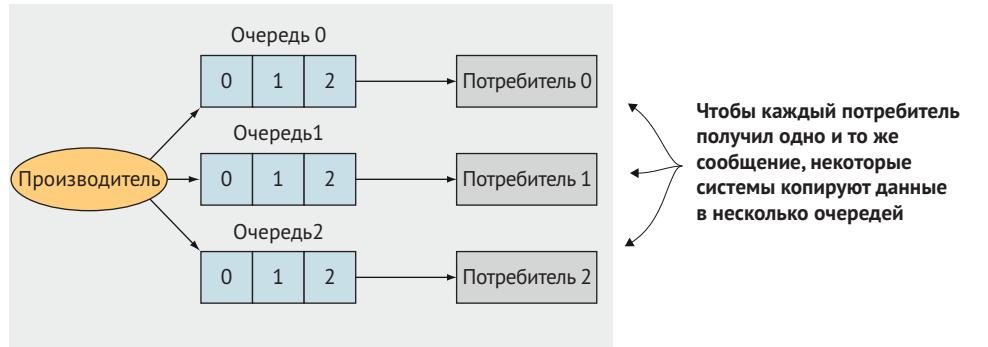


Рис. 5.5. Другие сценарии работы брокеров

Теперь представьте, что число копий растет с ростом востребованности события. Вместо создания копий целых очередей (кроме копий

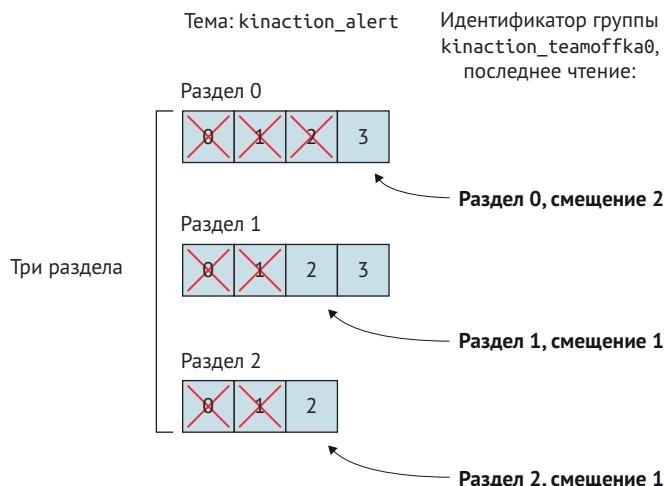
для репликации или отработки отказов) Kafka может обслуживать несколько приложений из одной и той же ведущей реплики раздела.

Kafka, отмечалось в первой главе, не ограничивается одним потребителем. Даже если приложение-потребитель неактивно в момент создания сообщения, оно все равно сможет получить его, если Kafka сохранит сообщение в своем журнале. Поскольку сообщения не удаляются при получении другими потребителями и могут доставляться многократно, клиенты-потребители должны вести учет того, что они прочитали из темы. Кроме того, поскольку одну и ту же тему могут читать несколько приложений, важно, чтобы смещения и разделы были специфичны для определенной группы потребителей. Ключевые координаты, позволяющие клиентам-потребителям работать вместе, представляют собой уникальное сочетание идентификатора группы, темы и номера раздела.

5.3.1. Координатор группы

Как упоминалось выше, координатор группы обслуживает клиентов-потребителей и ведет учет сообщений, прочитанных из темы конкретной группой [8]. Координаты раздела и идентификатор группы определяют значение смещения.

Взгляните на рис. 5.6 и обратите внимание, что в качестве координат можно использовать смещения сообщений, чтение которых подтверждено, и использовать их для чтения следующих сообщений. Например, потребитель на рис. 5.6 из группы `kinaction_teamoffka0`, которому назначен раздел 0, будет готов прочитать следующее сообщение со смещением 3.



Эта информация о смещении сообщает, где вы находитесь и что потреблять дальше!

Рис. 5.6. Координаты

На рис. 5.7 показан сценарий, в котором одни и те же разделы, представляющие интерес, находятся в трех отдельных брокерах и назначены двум разным группам потребителей: `kinaction_teamoffka0` и `kinaction_teamsetka1`. Потребители в каждой группе получают свои копии данных из разделов каждого брокера. Они работают порознь, если не являются частью одной группы. Правильное оформление членства в группах играет важную роль, позволяя правильно управлять метаданными.

Потребители из разных групп никак не влияют друг на друга и получают свои копии данных

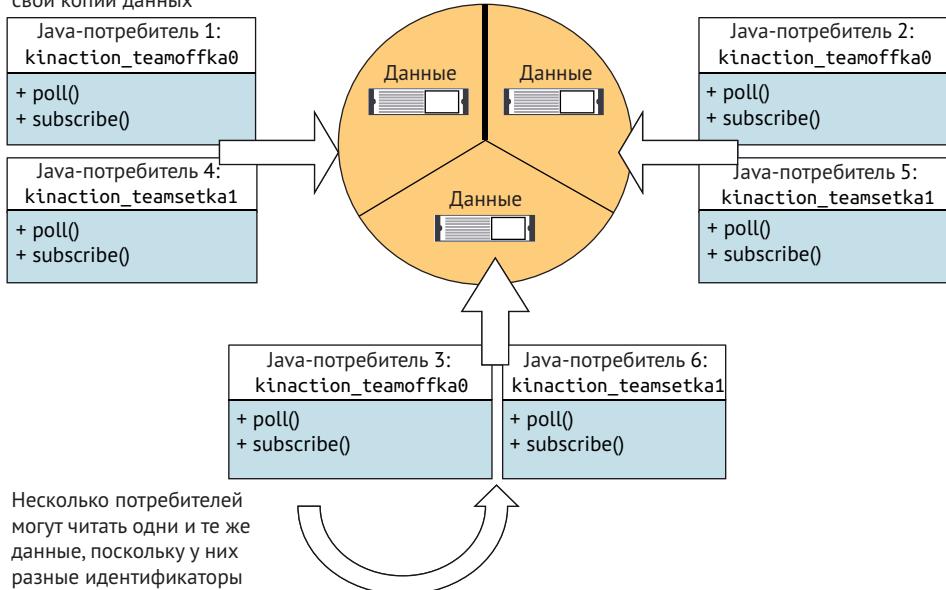


Рис. 5.7. Потребители в разных группах [12]

Как правило, только один потребитель в группе может читать один раздел. Другими словами, раздел может читаться многими потребителями, но только одним потребителем из каждой группы в каждый момент времени. На рис. 5.8 показано, как один потребитель может читать ведущие реплики из двух разделов, тогда как второй – данные только из ведущей реплики третьего раздела [8]. Реплика с одним разделом не может быть разделена или совместно использована более чем одним потребителем с одним и тем же идентификатором.

Одна из приятных особенностей использования групп потребителей заключается в том, что при сбое одного из потребителей его разделы, которые он читал, переназначаются другим потребителям в той же группе [8]. Существующий потребитель начинает читать разделы, откуда читал потребитель, выпавший из группы.

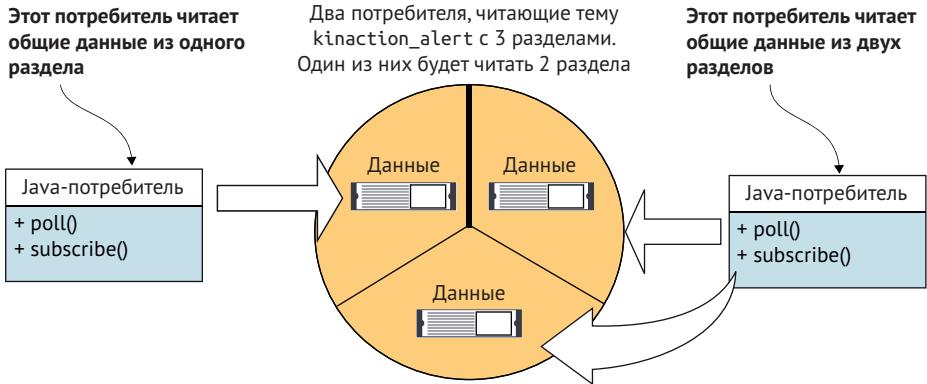


Рис. 5.8. Потребители Kafka в группе

В табл. 5.1 приведен параметр `heartbeat.interval.ms`, определяющий интервал проверки связи с координатором группы [13]. Эти проверки представляют способ взаимодействия потребителя с координатором, чтобы сообщить последнему, что потребитель по-прежнему работоспособен [8].

Отсутствие связи с потребителем в течение определенного периода времени может быть обусловлено разными причинами, например остановкой клиента-потребителя, завершением процесса или сбоем из-за неустранимого исключения. Если клиент не запущен, он не сможет подтверждать свою работоспособность координатору группы [8].

5.3.2. Стратегия назначения разделов

Еще один момент, о котором мы должны знать, – как назначаются разделы потребителям. Это важно, потому что помогает выяснить, сколько разделов будет назначено для обработки каждому из ваших потребителей. Свойство `partition.assignment.strategy` определяет стратегию назначения разделов каждому потребителю [14]. Доступны стратегии `Range`, `RoundRobin`, `Sticky` и `CooperativeSticky` [15].

Стратегия назначения диапазонов `Range` определяет количество разделов в одной теме (упорядочивая их по номерам), а затем делит на количество потребителей. Если разделов больше, чем потребителей, то лишние разделы будут распределены между первыми потребителями в списке (упорядоченном по алфавиту) [16]. Применяя эту стратегию, убедитесь, что разделы более или менее равномерно распределяются между потребителями, и подумайте о переходе на другую стратегию, если одни клиенты-потребители используют все свои ресурсы для обработки разделов, а другие остаются недостаточно загруженными. На рис. 5.9

показано, как эта стратегия распределит семь разделов между тремя клиентами – здесь первый клиент получит больше разделов, чем остальные два.

Стратегия *циклического перебора RoundRobin* пытается равномерно распределить разделы между потребителями [1]. На рис. 5.9 приводится измененная схема из статьи «What I have learned from Kafka partition assignment strategy» с примером циклического назначения семи разделов из одной темы трем потребителям, входящим в одну группу [17]. Первый потребитель получает первый раздел, второй потребитель – второй и т. д., пока не будут назначены все разделы.

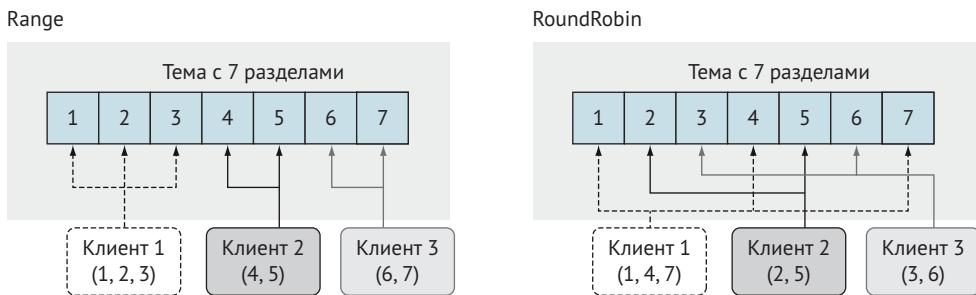


Рис. 5.9. Назначение разделов

Стратегия *фиксированного назначения Sticky* была добавлена в версии 0.11.0 [18]. Однако в большинстве наших примеров используется стратегия назначения диапазонов, и мы уже познакомились со стратегией циклического перебора, поэтому не будем углубляться в стратегии *Sticky* и *CooperativeSticky*.

5.4. Маркировка местонахождения

Еще одна важная деталь, о которой не следует забывать, – мы должны гарантировать, что наши приложения читают все сообщения из темы. Допустимо ли пропустить несколько сообщений, и нужно ли подтверждать прочтение каждого? Ответы на эти вопросы зависят от конкретных требований и любых компромиссов, на которые вы готовы пойти. Согласны ли вы пожертвовать скоростью, чтобы гарантировать просмотр каждого сообщения? Все это обсуждается в этом разделе.

Один из вариантов – присвоить параметру `enable.auto.commit` значение `true`, по умолчанию используемое для клиентов-потребителей [19]. В этом случае будет происходить автоматическая фиксация смещений. Одна из замечательных сторон этой настройки – отсутствие необходимости выполнять какие-либо другие вызовы для фиксации использованных смещений.

Брокеры Kafka повторно возвращают одни и те же сообщения, если их получение не было зафиксировано автоматически клиентом-потребителем из-за сбоя. А есть ли минусы у этого варианта? Есть. Представьте, что мы обрабатываем сообщения, получаемые в ходе опроса, который выполняется, скажем, в отдельном потоке. В таком случае при использовании автоматической фиксации сообщение может быть отмечено как прочитанное, даже если на самом деле его обработка не завершилась. Что, если во время обработки произошла ошибка и было бы желательно повторить попытку? В следующем цикле опроса мы могли бы получить следующий набор смещений, уже после зафиксированного и отмеченного как полученное [8]. Сообщения могут теряться, хотя будут выглядеть как прочитанные, несмотря на то что не были обработаны вашей потребительской логикой.

Выполняя фиксацию, имейте в виду, что время не всегда подходит для этого. Если вы вызываете метод фиксации для потребителя с метаданными, не определяющими однозначно конкретное фиксируемое смещение, то остается возможность для некоторого неопределенного поведения, зависящего от интервалов опроса, срабатывающих таймеров или даже вашей собственной логики потоковой передачи. Если сообщение обязательно нужно зафиксировать в определенное время, соответствующее времени его обработки, или с определенным смещением, то не забудьте передать в метод фиксации метаданные, определяющие смещение.

Давайте рассмотрим этот вопрос подробнее и обсудим использование фиксации на конкретном примере кода с параметром `enable.auto.commit = false`. Этот метод можно использовать для организации максимального контроля над тем, когда приложение фактически получает сообщение и фиксирует его. С помощью этого шаблона можно обеспечить семантику «не менее одного раза».

Возьмем за основу пример, в котором каждое полученное сообщение вызывает создание файла в Hadoop в определенном месте. Итак, вы получаете сообщение со смещением 999. Во время обработки потребитель останавливается из-за ошибки. Поскольку он не зафиксировал смещение 999, то в следующий раз, когда другой потребитель из той же группы начнет чтение из этого же раздела, он снова получит сообщение со смещением 999. Получив его дважды, клиент выполнит задачу, не пропустив сообщения. С другой стороны, вы получили его дважды! Если по какой-то причине обработка завершилась, а сбой произошел непосредственно перед фиксацией, то ваш код должен быть готов определить и обработать факт повторного получения одного и того же сообщения.

Теперь давайте посмотрим на часть кода, который мы будем использовать для управления смещениями. Так же как в случае с производителем, отправляющим сообщения, мы можем фиксировать

смещения синхронно или асинхронно. В листинге 5.4 показана синхронная фиксация. Найдите в этом листинге вызов `commitSync` и обратите внимание, что на время его работы выполнение кода блокируется, пока он не вернет признак успеха или неудачи [20].

Листинг 5.4. Синхронное подтверждение

```
consumer.commitSync();           ← Код ждет завершения
// Любой последующий код будет ждать завершения
// вызова в предыдущей строке
```

Код ждет завершения
вызыва `commitSync` с
признаком успеха или
неудачи

Так же как в случае с производителями, мы опять же можем использовать обратный вызов. В листинге 5.5 показано, как организовать асинхронную фиксацию с обратным вызовом путем реализации интерфейса `OffsetCommitCallback` (метод `onComplete`) в лямбда-выражении [21]. Этот интерфейс позволяет записывать сообщения в журнал, чтобы потом определить успех или неудачу, и избавляет от необходимости ждать завершения фиксации перед переходом к следующей инструкции.

Листинг 5.5. Асинхронная фиксация с обратным вызовом

```
public static void commitOffset(long offset,
                                int partition,
                                String topic,
                                KafkaConsumer<String, String> consumer) {
    OffsetAndMetadata offsetMeta = new OffsetAndMetadata(++offset, "");
    Map<TopicPartition, OffsetAndMetadata> kaOffsetMap = new HashMap<>();
    kaOffsetMap.put(new TopicPartition(topic, partition), offsetMeta);

    consumer.commitAsync(kaOffsetMap, (map, e) -> {           ← Лямбда-выраже-
        if (e != null) {                                         ние, создающее
            for (TopicPartition key : map.keySet()) {           экземпляр Offset-
                log.info("kinaction_error: offset {}", map.get(key).offset());   CommitCallback
            }
        } else {
            for (TopicPartition key : map.keySet()) {
                log.info("kinaction_info: offset {}", map.get(key).offset());
            }
        }
    });
}
```

Этот код очень похож на асинхронную отправку сообщений с обратным вызовом для подтверждения, показанную в главе 4. Для реализации обратного вызова необходимо использовать интерфейс `OffsetCommitCallback`. Вы также можете определить метод `onComplete` для обработки исключений или успеха, если это необходимо.

В каких случаях следует использовать синхронную, а в каких – асинхронную фиксацию? Имейте в виду, что при синхронной фиксации задержка выше, так как приходится ждать завершения блокирующего вызова. Эта задержка может быть оправдана, если в число ваших требований входит необходимость обеспечения согласованности данных [21]. Оценка требований поможет вам определить уровень контроля, который необходимо использовать при информировании Kafka о том, какие сообщения ваша логика считает обработанными.

5.5. Чтение из сжатой темы

Потребители должны знать, когда выполняют чтение из сжатой темы. Kafka сжимает журнал разделов в фоновом режиме и может удалить записи с одним и тем же ключом, кроме последней. В главе 7 мы подробнее рассмотрим, как работают такие темы, но если говорить коротко, то нам нужно обновить записи с одинаковым значением ключа. Если вам нужна не история сообщений, а только последнее значение, то вас может заинтересовать, как эта концепция может быть применима к неизменяемому журналу, доступному только для добавления в конец. Самая большая проблема для потребителей, которая может вызвать ошибку, заключается в том, что при чтении записей из сжатой темы потребители могут получить несколько записей для одного ключа [22]! Как такое возможно? Поскольку сжатие выполняется с файлами журналов, находящимися на диске, процесс сжатия может не увидеть сообщения, которые во время его работы еще находятся в памяти.

Клиенты должны уметь правильно обрабатывать ситуацию, когда для каждого ключа возвращается более одного значения, – игнорировать все значения, кроме последнего. Чтобы пробудить ваш интерес к сжатым темам, обратите внимание, что Kafka использует свою сжатую внутреннюю тему `_consumer_offsets`, которая на прямую связана со смещениями потребителей [23]. Сжатие этой темы имеет определенный смысл, потому что для определенной комбинации группы потребителей, раздела и темы требуется только самое последнее значение, поскольку оно будет иметь самое последнее использованное смещение.

5.6. Реализация в коде наших заводских требований

Давайте попробуем использовать собранную нами информацию о работе потребителей для реализации наших решений, разработанных в главе 3, и использования Kafka на нашем заводе по производству электровелосипедов, но уже с точки зрения клиента-потребителя. Как отмечалось в главе 3, мы должны гарантировать

получение всех управляющих сообщений, посредством которых операторы посылают команды датчикам. Для начала рассмотрим имеющиеся у нас варианты чтения смещений.

5.6.1. Варианты чтения

В Kafka нет поиска сообщений по ключам, зато есть поиск по определенному смещению. Учитывая, что наш журнал сообщений представляет собой постоянно растущий массив, в котором каждое сообщение имеет индекс, у нас есть несколько вариантов организации чтения: начать все с начала, сразу перейти к концу или найти смещения, соответствующие определенному моменту времени. Давайте рассмотрим эти варианты.

Одна проблема, которая может встретиться, заключается в том, что нам может понадобиться начать чтение с самого начала темы, даже если прежде мы уже сделали это. Причины могут быть разные: желание повторно прочитать весь журнал после устранения логических ошибок или сбой в конвейере данных после запуска с Kafka. Для реализации этого варианта важно установить конфигурационный параметр `auto.offset.reset` в значение `earliest` [24]. Другой способ начать чтение с начала – запустить ту же логику, но использовать другой идентификатор группы. По сути, это означает, что Kafka, не найдя зафиксированных смещений в теме, начнет возвращать сообщения с самого первого найденного.

В листинге 5.6 показан пример установки свойства `auto.offset.reset` в значение `"earliest"`, чтобы отыскать определенное смещение [24]. Выбор в качестве идентификатора группы случайного UUID тоже позволит начать работу без использования истории смещений для группы потребителей. Это вариант чтения мы могли бы взять для просмотра `kinaction_alerttrend` с использованием другого кода, пытающегося определить тенденции по всем данным в этой теме.

Листинг 5.6. Чтение с начала

```
Properties kaProperties = new Properties();
kaProperties.put("group.id",
                  UUID.randomUUID().toString());
kaProperties.put("auto.offset.reset", "earliest");
```

Создает идентификатор группы, для которого в Kafka отсутствует сохраненное смещение

Использует самое раннее смещение, хранящееся в журнале

Иногда бывает желательно просто начать обработку с того места, на котором она была прекращена в прошлый раз, и забыть о ранее прочитанных сообщениях [24]. Прошлые данные могут быть слишком старыми, чтобы иметь ценность для бизнеса. В ли-

стинге 5.7 показаны свойства, которые следует установить, чтобы начать чтение с последнего смещения. Если нужно гарантировать, что потребитель не получит ранее прочитанных сообщений и начнет чтение с первого непрочитанного, смещение которого хранит Kafka, то вам не потребуется использовать UUID, за исключением случаев тестирования. Если вас интересуют только новые уведомления, поступающие в тему `kinaction_alert`, потребитель может увидеть только эти уведомления.

Листинг 5.7. Чтение с последнего смещения

```
Properties kaProperties = new Properties();
kaProperties.put("group.id",
    UUID.randomUUID().toString()); <--  
kaProperties.put("auto.offset.reset", "latest"); <--
```

Создает идентификатор группы, для которого в Kafka отсутствует сохраненное смещение

Использует самое последнее зафиксированное смещение

Один из самых сложных методов поиска смещений – `offsetsForTimes`. Этот метод позволяет передать ассоциативный массив тем и разделов, а также отметку времени для каждого элемента массива, чтобы получить обратно массив смещений и отметок времени для заданных тем и разделов [25]. Это может пригодиться, когда логическое смещение неизвестно, но известна отметка времени. Например, если имело место исключение, связанное с зарегистрированным событием, то вы можете использовать потребителя, чтобы определить данные, обрабатывавшиеся в конкретный момент времени. Определение местонахождения события аудита по времени можно использовать для нашей темы `kinaction_audit`, чтобы обнаружить выполнявшиеся команды.

Как показано в листинге 5.8, у нас есть возможность получить смещение и отметки времени для каждой темы или раздела, передав исключуюю отметку времени. После получения массива метаданных из вызова `offsetsForTimes` можем сразу перейти к интересующему смещению для соответствующего ключа, отыскав его в возвращаемом массиве.

Листинг 5.8. Поиск смещения по отметке времени

```
...
Map<TopicPartition, OffsetAndTimestamp> kaOffsetMap =
consumer.offsetsForTimes(timeStampMapper); <--  
...
// Использовать полученный массив
consumer.seek(partitionOne,
    kaOffsetMap.get(partitionOne).offset()); <--
```

Поиск первого смещения с отметкой времени больше или равной указанной в timeStampMapper

Поиск первого смещения, указанного в kaOffsetMap

Следует помнить, что возвращаемое смещение – это первое сообщение с отметкой времени, соответствующей заданным критериям. Однако из-за того, что производитель может выполнять повторные попытки отправить сообщение в случае ошибки, или из-за различий во времени при добавлении отметок (возможно, потребителями) смещения могут следовать не по порядку.

Kafka также дает возможность искать другие смещения, как описывается в документации с описанием потребителей [26]. Теперь, кратко познакомившись с доступными вариантами, давайте посмотрим, как применить их к нашему сценарию использования.

5.6.2. Требования

Одним из требований в нашем примере аудита было отсутствие необходимости сопоставлять (или группировать) какие-либо события. Это означает, что нарушение порядка следования событий в разделях или их чтения потребителями не является проблемой. Еще одним требованием была гарантия невозможности потери сообщений. Надежный способ гарантировать обработку каждого сообщения аудита нашей логикой – специально фиксировать смещение сообщения после его обработки. Для управления фиксацией из кода потребителя нужно установить `enable.auto.commit` в значение `false`.

В листинге 5.9 показан пример реализации синхронной фиксации после обработки каждого сообщения аудита. Подробная информация о следующем смещении в теме и разделе поддерживается в цикле обхода сообщений. Следует отметить, что прибавление 1 к текущему смещению может показаться странным, однако это легко объяснимо: смещение, отправляемое брокеру, должно быть вашим будущим индексом. Для его отправки вызывается метод `commitSync`, которому передается массив, содержащий смещение только что обработанной записи [20].

Листинг 5.9. Логика потребителя сообщений аудита

```
...
kaProperties.put("enable.auto.commit", "false"); ← Запретить автоматическую фиксацию
try (KafkaConsumer<String, String> consumer =
    new KafkaConsumer<>(kaProperties)) {
    consumer.subscribe(List.of("kinaction_audit"));

    while (keepConsuming) {
        var records = consumer.poll(Duration.ofMillis(250));
        for (ConsumerRecord<String, String> record : records) {
            // обработка сообщения аудита...
            OffsetAndMetadata offsetMeta = ← Увеличить смещение текущей записи, чтобы определить следующее смещение для чтения
            ...
        }
    }
}
```

```

        new OffsetAndMetadata(++record.offset(), "");

    Map<TopicPartition, OffsetAndMetadata> kaOffsetMap =
        new HashMap<>();
    kaOffsetMap.put(
        new TopicPartition("kinaction_audit", <-- Связать тему и ключ
                           record.partition()), offsetMeta);

    consumer.commitSync(kaOffsetMap); <-- Зафиксировать смещения
}
}
...

```

Еще одна цель разработки новой архитектуры для нашей фабрики по производству электровелосипедов заключалась в том, чтобы перехватывать предупреждения, отображать в информационной панели и выявлять тенденции их появления с течением времени. Несмотря на то что в наших сообщениях имеется ключ, идентифицирующий производственный этап, мы можем не беспокоиться об организации групп для потребления по отдельным этапам или в определенном порядке. В листинге 5.10 показано, как установить свойство `key.deserializer`, чтобы потребитель знал, как обращаться с двоичными данными, отправленными в Kafka при создании сообщения. В этом примере для десериализации ключа используется `AlertKeySerde`. Поскольку в этом сценарии потеря сообщений не является серьезной проблемой, мы можем разрешить автоматическую фиксацию.

Листинг 5.10. Потребитель уведомлений для дальнейшего анализа тенденций в их появлении

```

...
kaProperties.put("enable.auto.commit", "true"); <-- Использовать автоматическую фиксацию, потому что потеря отдельных сообщений не является проблемой
kaProperties.put("key.deserializer",
    AlertKeySerde.class.getName()); <-- Использовать AlertKeySerde для десериализации ключей
kaProperties.put("value.deserializer",
    "org.apache.kafka.common.serialization.StringDeserializer");

KafkaConsumer<Alert, String> consumer =
    new KafkaConsumer<Alert, String>(kaProperties);
consumer.subscribe(List.of("kinaction_alerttrend"));

while (true) {
    ConsumerRecords<Alert, String> records =
        consumer.poll(Duration.ofMillis(250));
    for (ConsumerRecord<Alert, String> record : records) {
        // ...
    }
}
...

```

Еще одним важным требованием является скорость обработки любых предупреждений, чтобы операторы как можно быстрее узнавали о критических проблемах. Поскольку производитель в главе 4 использовал нестандартный подход к назначению разделов, мы прямо закрепим потребителя за соответствующим разделом, чтобы максимально быстро получать предупреждения о критических проблемах. Так как задержка при обработке таких предупреждений нежелательна, фиксация каждого смещения будет выполняться асинхронно.

В листинге 5.11 показана логика клиента-потребителя, обрабатывающего критические уведомления, которые записываются в определенную тему и раздел с помощью пользовательского класса `AlertLevelPartitioner`, в данном случае в раздел 0 и тему `kinaction_alert`.

Мы используем объекты `TopicPartition`, чтобы сообщить Kafka, какие именно разделы в данной теме нас интересуют. Передача объектов `TopicPartition` в метод `assign` позволяет назначить раздел потребителю в обход координатора группы [27].

Получение каждого сообщения фиксируется потребителем асинхронным способом с использованием обратного вызова. Информация о фиксации смещения отправляется брокеру, и эта операция не должна блокировать обработку следующей записи потребителем, что соответствует нашим требованиям к дизайну из главы 3.

Листинг 5.11. Потребитель критических уведомлений

```
kaProperties.put("enable.auto.commit", "false");

KafkaConsumer<Alert, String> consumer =
    new KafkaConsumer<Alert, String>(kaProperties);
TopicPartition partitionZero =
    new TopicPartition("kinaction_alert", 0); ← Использовать TopicPartition для чтения критических сообщений
consumer.assign(List.of(partitionZero)); ← Потребитель сам назначает себе раздел, а не подписывается на тему

while (true) {
    ConsumerRecords<Alert, String> records =
        consumer.poll(Duration.ofMillis(250));
    for (ConsumerRecord<Alert, String> record : records) {
        // ...
        commitOffset(record.offset(),
            record.partition(), topicName, consumer); ← Прочитанные смещения фиксируются асинхронно
    }
}
...
public static void commitOffset(long offset,int part, String topic,
    KafkaConsumer<Alert, String> consumer) {
    OffsetAndMetadata offsetMeta = new OffsetAndMetadata(++offset, "");
```

```
Map<TopicPartition, OffsetAndMetadata> kaOffsetMap =  
    new HashMap<TopicPartition, OffsetAndMetadata>();  
    kaOffsetMap.put(new TopicPartition(topic, part), offsetMeta);  
  
    OffsetCommitCallback callback = new OffsetCommitCallback() {  
        ...  
    };  
    consumer.commitAsync(kaOffsetMap, callback); ↗  
}
```

Методу асинхронной фиксации передаются аргументы `kaOffsetMap` и `callback`

Потребитель может быть чуть ли не самой сложной частью взаимодействий с Kafka. Некоторые варианты можно реализовать с помощью конфигурационных свойств, а для реализации других могут понадобиться все ваши знания о темах, разделах, смещениях и способах поиска нужных данных.

Итоги

- Клиенты-потребители дают разработчикам возможность получать данные из Kafka. Подобно клиентам-производителям, клиенты-потребители имеют большое количество конфигурационных параметров, которые можно использовать для реализации разных сценариев без программирования конкретной логики.
- Группы потребителей позволяют нескольким клиентам вместе обрабатывать сообщения. Объединяя клиентов в группы, можно организовать параллельную обработку данных.
- Смещения представляют позицию записи в журнале. Используя смещения, потребители могут определять, с какого места они начнут чтение данных.
- Смещение может быть смещением предыдущего сообщения, которое потребители уже видели, что дает возможность повторно обрабатывать записи.
- Потребители могут подтверждать чтение данных синхронно или асинхронно.
- При асинхронном подтверждении чтения потребитель может использовать обратные вызовы для совершения дополнительных действий после получения данных.

Ссылки

- 1 S. Kozlovski. «Apache Kafka Data Access Semantics: Consumers and Membership». Confluent blog (n.d.). <https://www.confluent.io/blog/apache-kafka-data-access-semantics-consumers-and-membership> (доступно по состоянию на 20 августа 2021).

- 2 «Consumer Configurations». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/installation/configuration/consumer-configs.html> (доступно по состоянию на 19 июня 2019).
- 3 N. Narkhede. «Apache Kafka Hits 1.1 Trillion Messages Per Day – Joins the 4 Comma Club». Confluent blog (1 сентября 2015). <https://www.confluent.io/blog/apache-kafka-hits-1-1-trillion-messages-per-day-joins-the-4-comma-club/> (доступно по состоянию на 20 октября 2019).
- 4 «Class KafkaConsumer<K,V>». Kafka 2.7.0 API. Apache Software Foundation (n.d.). <https://kafka.apache.org/27/javadoc/org/apache/kafka/clients/consumer/KafkaConsumer.html#poll-java.time.Duration-> (доступно по состоянию на 24 августа 2021).
- 5 «Class WakeupException». Kafka 2.7.0 API. Apache Software Foundation (n.d.). <https://kafka.apache.org/27/javadoc/org/apache/kafka/common/errors/WakeupException.html> (доступно по состоянию на 22 июня 2020).
- 6 «Documentation: Topics and Logs». Confluent documentation (n.d.). <https://docs.confluent.io/5.5.1/kafka/introduction.html#topics-and-logs> (доступно по состоянию на 20 октября 2021).
- 7 «KIP-392: Allow consumers to fetch from closest replica». Wiki for Apache Kafka. Apache Software Foundation (5 ноября 2019). <https://cwiki.apache.org/confluence/display/KAFKA/KIP-392%3A+Allow+consumers+to+fetch+from+closest+replica> (доступно по состоянию на 10 декабря 2019).
- 8 J. Gustafson. «Introducing the Kafka Consumer: Getting Started with the New Apache Kafka 0.9 Consumer Client». Confluent blog (21 января 2016). <https://www.confluent.io/blog/tutorial-getting-started-with-the-new-apache-kafka-0-9-consumer-client/> (доступно по состоянию на 01 июня 2020).
- 9 J. Rao. «How to choose the number of topics/partitions in a Kafka cluster?». Confluent blog (12 марта 2015). <https://www.confluent.io/blog/how-choose-number-topics-partitions-kafka-cluster/> (доступно по состоянию на 19 мая 2019).
- 10 «Committing and fetching consumer offsets in Kafka». Wiki for Apache Kafka. Apache Software Foundation (24 марта 2015). <https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=48202031> (доступно по состоянию на 15 декабря 2019).
- 11 «Consumer Configurations: group.id». Confluent documentation (n.d.). https://docs.confluent.io/platform/current/installation/configuration/consumer-configs.html#consumerconfigs_group.id (доступно по состоянию на 11 мая 2018).

- 12 «Documentation: Consumers». Apache Software Foundation (n.d.). https://kafka.apache.org/23/documentation.html#intro_consumers (доступно по состоянию на 11 декабря 2019).
- 13 «Consumer Configurations: heartbeat.interval.ms». Confluent documentation (n.d.). https://docs.confluent.io/platform/current/installation/configuration/consumer-configs.html#consumerconfigs_heartbeat.interval.ms (доступно по состоянию на 11 мая 2018).
- 14 «Consumer Configurations: partition.assignment.strategy». Confluent documentation (n.d.). https://docs.confluent.io/platform/current/installation/configuration/consumer-configs.html#consumerconfigs_partition.assignment.strategy (доступно по состоянию на 22 декабря 2020).
- 15 S. Blee-Goldman. «From Eager to Smarter in Apache Kafka Consumer Rebalances». Confluent blog (n.d.). <https://www.confluent.io/blog/cooperative-rebalancing-in-kafka-streams-consumer-ksqldb/> (доступно по состоянию на 20 августа 2021).
- 16 «RangeAssignor.java». Apache Kafka GitHub (n.d.). <https://github.com/apache/kafka/blob/c9708387bb1dd1fd068d6d8cec2394098d-5d6b9f/clients/src/main/java/org/apache/kafka/clients/consumer/RangeAssignor.java> (доступно по состоянию на 25 августа 2021).
- 17 A. Li. «What I have learned from Kafka partition assignment strategy». Medium (1 декабря 2017). <https://medium.com/@anyili0928/what-i-have-learned-from-kafka-partition-assignment-strategy-799fdf15d3ab> (доступно по состоянию на 20 октября 2021).
- 18 «Release Plan 0.11.0.0». Wiki for Apache Kafka. Apache Software Foundation (26 июня 2017). <https://cwiki.apache.org/confluence/display/KAFKA/Release+Plan+0.11.0.0> (доступно по состоянию на 14 декабря 2019).
- 19 «Consumer Configurations: enable.auto.commit». Confluent documentation (n.d.). https://docs.confluent.io/platform/current/installation/configuration/consumer-configs.html#consumerconfigs_enable.auto.commit (доступно по состоянию на 11 мая 2018).
- 20 «Synchronous Commits». Confluent documentation (n.d.). <https://docs.confluent.io/3.0.0/clients/consumer.html#synchronous-commits> (доступно по состоянию на 24 августа 2021).
- 21 «Asynchronous Commits». Confluent documentation (n.d.). <https://docs.confluent.io/3.0.0/clients/consumer.html#asynchronous-commits> (доступно по состоянию на 24 августа 2021).
- 22 «Kafka Design». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/kafka/design.html> (доступно по состоянию на 24 августа 2021).

- 23 «Kafka Consumers». Confluent documentation (n.d.). <https://docs.confluent.io/3.0.0/clients/consumer.html> (доступно по состоянию на 24 августа 2021).
- 24 «Consumer Configurations: auto.offset.reset». Confluent documentation (n.d.). https://docs.confluent.io/platform/current/installation/configuration/consumer-configs.html#consumerconfigs_auto.offset.reset (доступно по состоянию на 11 мая 2018).
- 25 «offsetsForTimes». Kafka 2.7.0 API. Apache Software Foundation (n.d.). <https://kafka.apache.org/27/javadoc/org/apache/kafka/clients/consumer/Consumer.html#offsetsForTimes-java.util.Map-> (доступно по состоянию на 22 июня 2020).
- 26 «seek». Kafka 2.7.0 API. Apache Software Foundation (n.d.). <https://kafka.apache.org/27/javadoc/org/apache/kafka/clients/consumer/Consumer.html#seek-org.apache.kafka.common.TopicPartition-long-> (доступно по состоянию на 22 июня 2020).
- 27 «assign». Kafka 2.7.0 API. Apache Software Foundation (n.d.). <https://kafka.apache.org/27/javadoc/org/apache/kafka/clients/consumer/KafkaConsumer.html#assign-java.util.Collection-> (доступно по состоянию на 24 августа 2021).

Брокеры

Эта глава охватывает следующие темы:

- роль брокеров и их обязанности;
- оценку влияния различных конфигурационных значений брокеров;
- функционирование реплик и как они поддерживают актуальность.

До сих пор мы рассматривали Kafka с точки зрения разработчика приложений, взаимодействующих с внешними приложениями и процессами. Однако Kafka – это распределенная система, которая сама по себе заслуживает пристального внимания. В этой главе мы рассмотрим компоненты, обеспечивающие работу брокеров Kafka.

6.1. Знакомство с брокерами

До сих пор все наше внимание было сосредоточено на стороне клиентов Kafka, теперь мы переключимся на другие мощные компоненты экосистемы: брокеры, которые образуют ядро системы.

По мере углубления в изучение Kafka знакомые с концепциями больших данных или имеющие опыт использования Hadoop могут увидеть знакомые термины, такие как *осведомленность о стойке* (rack awareness – знание, в какой физической серверной стойке размещена машина) и *разделы*. Kafka поддерживает осведомленность о стойке, благодаря чему реплики раздела физически распределяются по

отдельным стойкам [1]. Использование знакомых терминов должно помочь вам почувствовать себя как дома, потому что мы будем проводить новые параллели между тем, с чем вы работали раньше, и тем, что может предложить Kafka. При настройке собственного кластера Kafka важно знать, что есть еще один кластер, о котором следует помнить: Apache ZooKeeper. Вот с него и начнем.

6.2. Роль ZooKeeper

ZooKeeper является ключевой частью брокеров и обязательным условием для запуска Kafka. Поскольку Kafka должна запускаться до запуска брокеров, мы начнем наше обсуждение с этого.

ПРИМЕЧАНИЕ. Для упрощения требований к запуску Kafka в главе 2 предлагалось заменить ZooKeeper ее собственным механизмом управления кворумом [2]. Но так как на момент публикации работы над этим механизмом еще не была завершена, мы обсудим роль и значение ZooKeeper. Тем не менее имейте в виду, что в версии 2.8.0 уже доступна бета-версия механизма управления кворумом.

Поскольку механизм управления кластерами ZooKeeper требует наличия минимального количества брокеров для выбора лидеров и принятия решения, он действительно важен для наших брокеров [3]. Сам ZooKeeper тоже хранит в нашем кластере свою информацию, такую как темы [4], и помогает брокерам, координируя назначение и уведомления [5].

Для взаимодействия с брокерами важно, чтобы ZooKeeper был запущен до запуска брокеров. Состояние кластера ZooKeeper влияет на состояние брокеров Kafka. Например, если экземпляры ZooKeeper выйдут из строя, то метаданные темы и конфигурация могут быть потеряны.

Обычно нет необходимости раскрывать детали организации кластера ZooKeeper (IP-адреса и порты) приложениям производителей и потребителей. Некоторые устаревшие фреймворки, которые мы используем, тоже могут поддерживать возможность подключения клиентских приложений к нашему кластеру ZooKeeper. Одним из примеров может служить версия Spring Cloud Stream 3.1.x, которая позволяет установить свойство `zkNodes` [6]. По умолчанию это свойство принимает значение `localhost`, и в большинстве случаев его следует оставить без изменений, чтобы избежать зависимости от ZooKeeper. Свойство `zkNodes` отмечено как устаревшее, но вы никогда не знаете, столкнетесь ли вы с более старым кодом, который придется поддерживать, поэтому за ним приходится следить. Почему это не нужно сейчас и не понадобится в будущем? Помимо того, что Kafka не всегда будет требовать наличия ZooKeeper, нам также

важно избегать ненужных внешних зависимостей в своих приложениях. Кроме того, это требует открывать меньше портов при использовании брандмауэра для защиты Kafka и клиента.

Используя инструмент Kafka `zookeeper-shell.sh`, который находится в папке `bin` в каталоге установки Kafka, мы можем подключиться к хосту ZooKeeper в нашем кластере и посмотреть, как хранятся данные [7]. Один из способов найти пути, которые использует Kafka, – заглянуть в исходный код класса `ZkData.scala` [8]. В этом файле вы найдете, например, такие пути, как `/controller`, `/controller_epoch`, `/config` и `/brokers`. Если заглянуть в папку `/brokers/topics`, то можно увидеть список созданных нами тем. К настоящему моменту у вас, как мы надеемся, в списке должна присутствовать тема `kinaction_helloworld` как минимум.

ПРИМЕЧАНИЕ. Чтобы увидеть список тем и получить те же результаты, можно также использовать еще один инструмент Kafka – `kafka-topics.sh`! В листинге 6.1 показаны команды, которые подключаются к ZooKeeper и Kafka соответственно и получают свои данные, но делают это с помощью другого командного интерфейса. Вывод должен включать тему, созданную нами в главе 2: `[kinaction_helloworld]`.

Листинг 6.1. Список тем

```
bin/zookeeper-shell.sh localhost:2181
ls /brokers/topics ← Вывод списка всех тем с
# или
bin/kafka-topics.sh --list \
→ --bootstrap-server localhost:9094 ← kafka-topics подключается
                                         к ZooKeeper и выводит список тем
```

Подключается к нашему локальному экземпляру ZooKeeper

Даже когда ZooKeeper станет ненужным для запуска Kafka, нам все равно может понадобиться работать со старыми кластерами, и, вероятно, мы еще какое-то время будем видеть ссылки на ZooKeeper в документации и справочных материалах. В целом знание задач, в решении которых Kafka раньше полагалась на ZooKeeper, и переход на использование кластера Kafka с внутренними узлами метаданных дают представление о движущихся частях всей системы.

Быть брокером Kafka означает иметь возможность координировать свои действия с другими брокерами, а также взаимодействовать с ZooKeeper. При тестировании или работе с экспериментальными кластерами у нас может быть только один узел-брокер. Однако в промышленном окружении почти всегда будет несколько брокеров.

Но отвлекитесь от ZooKeeper и взгляните на рис. 6.1, где показано место брокеров в кластере и что они являются домом для журна-

лов данных Kafka. Клиенты будут взаимодействовать с брокерами, пытаясь получить и передать информацию в Kafka, и требовать их внимания [9].

6.3. Конфигурационные параметры брокеров

Конфигурация – важная часть работы с клиентами, темами и брокерами Kafka. Загляните в приложение A, где описываются этапы настройки наших первых брокеров. Там мы представили файл *server.properties*, который передается в качестве аргумента командной строки сценарию запуска брокера. Использование этого файла является распространенным способом передачи конкретной конфигурации экземпляру брокера. Например, конфигурационное свойство *log.dirs* в этом файле всегда должно указывать местоположение журнала, имеющее смысл для вашего окружения.

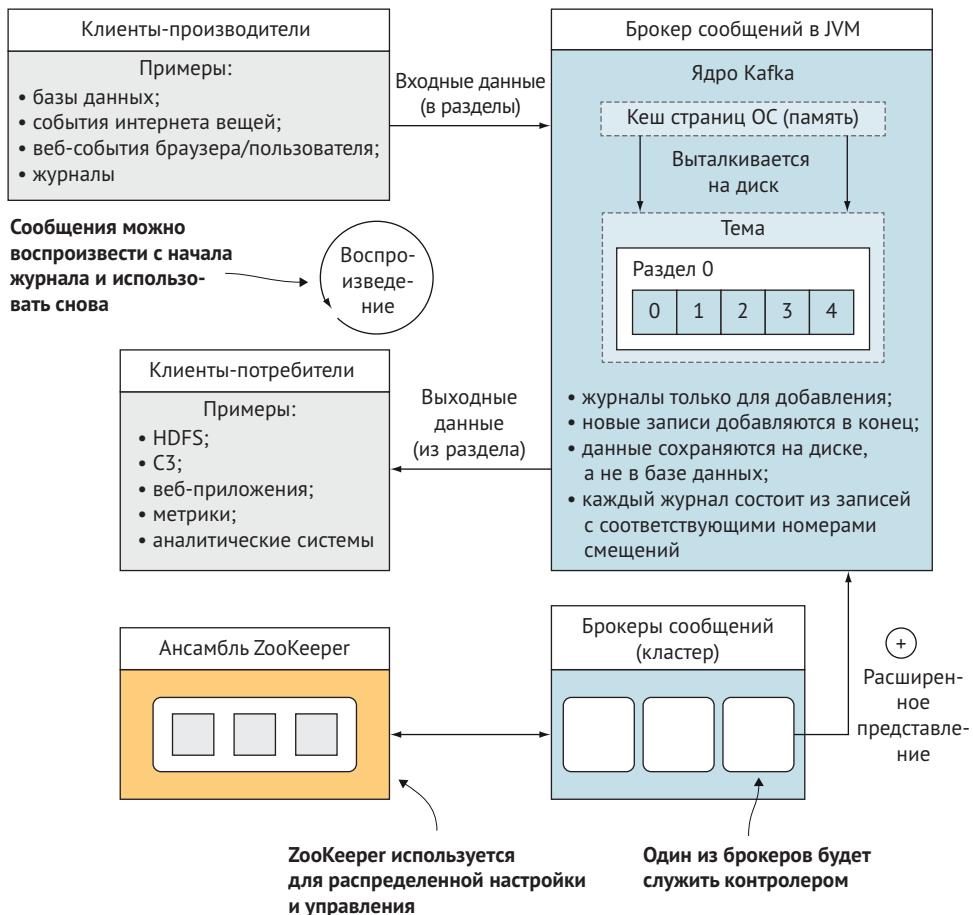


Рис. 6.1. Брокеры

Этот файл определяет также конфигурацию слушателей, расположения журналов, хранения журналов, настройки ZooKeeper и координатора группы [10]. Как и в случае с конфигурациями производителя и потребителя, просматривая документацию, доступную по адресу <http://mng.bz/p9p2>, особое внимание обращайте на параметры, снабженные меткой **Importance: high** (Важность: высокая).

В листинге 6.2 показано, что происходит, когда имеется только одна копия данных и брокер, хранящий ее, выходит из строя. Такое может случиться при использовании значений по умолчанию в конфигурации брокера. Для начала убедитесь, что в вашем локальном тестовом кластере Kafka имеется три узла, и создайте тему, как показано в листинге 6.2.

Листинг 6.2. Список тем

```
bin/kafka-topics.sh --create \
--bootstrap-server localhost:9094 \
--topic kinaction_one_replica
```

Создать тему с одним разделом
и одной репликой


```
bin/kafka-topics.sh --describe --bootstrap-server localhost:9094 \
--topic kinaction_one_replica
```

←


```
Topic: one-replica PartitionCount: 1 ReplicationFactor: 1 Configs:
      Topic: kinaction_one_replica Partition: 0
Leader: 2 Replicas: 2 Isr: 2
```

Получить описание темы kinaction_one_replica со
всеми данными, расположенными в брокере с ID 2

Запустив команды из листинга 6.2 для создания и описания темы `kinaction_one_replica`, вы увидите, что в полях `Partition`, `Leader`, `Replicas` и `Isr` (in-sync replicas – синхронизированных реплик) присутствуют только по одному значению. Кроме того, брокер использует то же значение идентификатора. Это означает, что доступность темы целиком и полностью зависит от работоспособности этого брокера.

Если теперь остановить брокера с идентификатором 2, а затем попытаться получить сообщение из этой темы, то мы получим ответ «1 partitions have leader brokers without a matching listener» (1 раздел имеет ведущих брокеров без соответствующего слушателя). Из-за отсутствия копий раздела темы мы не сможем производить и потреблять сообщения, используя эту тему, не восстановив работоспособность брокера. Этот простой пример иллюстрирует важность настройки брокеров, когда пользователи создают свои темы вручную, как в листинге 6.2.

Еще одно важное конфигурационное свойство, которое необходимо определить, задает местоположение журналов приложений и ошибок во время нормальной работы. Давайте рассмотрим его дальше.

6.3.1. Другие журналы Kafka: журналы приложений

Как и большинство приложений, Kafka предоставляет журналы, позволяющие узнать, что происходит внутри приложения. Далее под термином *журналы приложений* мы будем подразумевать журналы, о которых обычно думаем при работе с любым приложением. Они не связаны с журналами сообщений, которые составляют основу Kafka.

Кроме того, журналы приложений хранятся в совершенно другом месте, отдельно от журналов сообщений. После запуска брокера его журнал приложения можно найти в каталоге установки Kafka в папке `logs/`. Это местоположение можно изменить, скорректировав свойство `kafka.logs.dir` в файле `config/log4j.properties` [11].

6.3.2. Журнал сервера

Многие ошибки и неожиданное поведение, возникающие при запуске, могут быть связаны с проблемами конфигурации. Ошибки на запуске, или исключения, вызывающие завершение брокера, нужно искать в файле журнала сервера `server.log`. Это самое первое место, куда следует заглянуть при поиске причин появления проблем. Ищите (например, с помощью команды `grep`) значения под заголовком `KafkaConfig`.

Содержимое каталога, в котором находится этот файл, может ошеломить при первом знакомстве с ним. В нем, скорее всего, вы увидите множество других файлов, таких как `controller.log` (если брокер когда-либо играл эту роль) и более старые файлы с тем же именем. Одним из инструментов, который можно использовать для ротации и сжатия журналов, является `logrotate` (<https://linux.die.net/man/8/logrotate>), но есть и другие, помогающие управлять старыми журналами сервера.

Также следует упомянуть, что такие журналы имеются для каждого брокера. По умолчанию они не объединены в одном месте. Однако существуют различные платформы, такие как Splunk™ (<https://www.splunk.com/>), позволяющие организовать их совместное хранение. Это особенно важно для анализа журналов при использовании чего-то вроде облачной среды.

6.3.3. Управление состоянием

Как отмечалось в главе 2, каждый раздел имеет одну ведущую реплику (лидера). В любой момент времени существует один лидер в одном брокере. Брокер может хранить ведущие реплики нескольких разделов, и ведущие реплики могут храниться в любом брокере в кластере. Однако только один брокер в кластере действует как контроллер. Роль контроллера заключается в управлении кластером [12]. Контроллер также выполняет другие административные действия, такие как переназначение разделов [13].

Когда мы производим последовательное обновление кластера, останавливая и запуская по одному брокеру за раз, то обновлять контроллер лучше всего последним [14]. Иначе может потребоваться остановить и запустить контроллер несколько раз.

Выяснить, какой брокер является контроллером, можно с помощью сценария `zookeeper-shell.sh`, который найдет и выведет идентификатор брокера-контроллера, как показано в листинге 6.3. Путь `/controller` существует в ZooKeeper, и в листинге 6.3 мы запускаем одну команду, чтобы получить текущее значение. Выполнив эту команду у себя, я получил ответ, сообщающий, что роль контроллера в моем кластере играет брокер с идентификатором 0.

Листинг 6.3. Вывод идентификатора текущего контроллера

```
bin/zookeeper-shell.sh localhost:2181
get /controller
```

← Выполнить команду `get`,
передав ей путь `/con-`
`troller`

Подключиться к экзем-
пляру ZooKeeper

На рис. 6.2 показаны все выходные данные ZooKeeper, включая значение `brokerid: "brokerid":0`. Если мы решим перенести или обновить этот кластер, то этого брокера мы должны перенести/обновить последним из-за его особой роли.

Журнал приложения для контроллера хранится в файле `controller.log`, в данном случае он служит журналом приложения для брокера 0. Этот файл журнала может пригодиться для анализа действий и сбоев брокера.

```
Connecting to localhost:2181
Welcome to ZooKeeper!
JLine support is disabled

WATCHER:::

WatchedEvent state:SyncConnected type:None path:null
get /controller
{"version":1,"brokerid":0,"timestamp":"1540874053577"}
cZxid = 0x2f
ctime = Mon Oct 29 23:34:13 CDT 2018
mZxid = 0x2f
mtime = Mon Oct 29 23:34:13 CDT 2018
pZxid = 0x2f
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x166c33ffa650000
dataLength = 54
numChildren = 0
■
```

Рис. 6.2. Пример вывода контроллера

6.4. Ведущие реплики разделов и их роль

Напомним, что темы состоят из разделов, а разделы могут иметь реплики (копии) для обеспечения отказоустойчивости. Разделы хранятся на дисках брокеров Kafka. Одна из реплик раздела будет играть роль лидера. Лидер отвечает за прием сообщений от внешних клиентов-производителей в этот раздел. Поскольку лидер является единственной репликой, хранящей самые свежие сообщения, он также должен передавать эти сообщения ведомым репликам [15]. А поскольку список ISR поддерживается лидером, он знает, какие реплики обновлены и хранят все текущие сообщения. Реплики действуют как потребители ведущей реплики раздела и будут получать сообщения из нее [15].

На рис. 6.3 показан кластер из трех узлов, где брокер 3 играет роль лидера для темы `kinaction_helloworld`, а брокеры 2 и 1 действуют как ведомые. Брокер 3 содержит ведущую реплику для раздела 2. Будучи лидером, брокер 3 обрабатывает все запросы на чтение и запись от внешних производителей и потребителей. Он также обрабатывает запросы, которые получает от брокеров 2 и 1, когда они загружают новые сообщения в свои реплики. Список ISR [3,2,1] включает лидера в первой позиции (3), а затем ведомых брокеров (2,1), которые обновляют свои копии сообщений, обращаясь к лидеру.

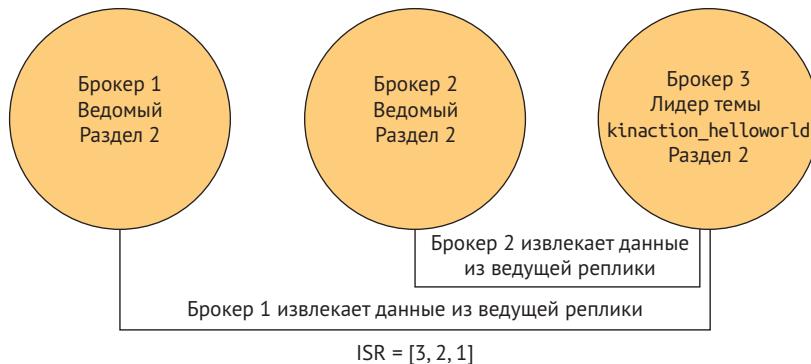


Рис. 6.3. Лидер

В некоторых случаях брокер, хранящий ведущую реплику раздела, может потерпеть сбой. На рис. 6.4 показана ситуация сбоя лидера из примера на рис. 6.3. Поскольку брокер 3 стал недоступен, выбирается новый лидер. В данном случае, как показано на рис. 6.4, новым ведущим выбирается брокер 2, чтобы Kafka могла продолжать обслуживать и получать данные для этого раздела. Список ISR теперь включает две реплики [2,1], первая позиция в котором отражает новую ведущую реплику, размещенную в брокере 2.

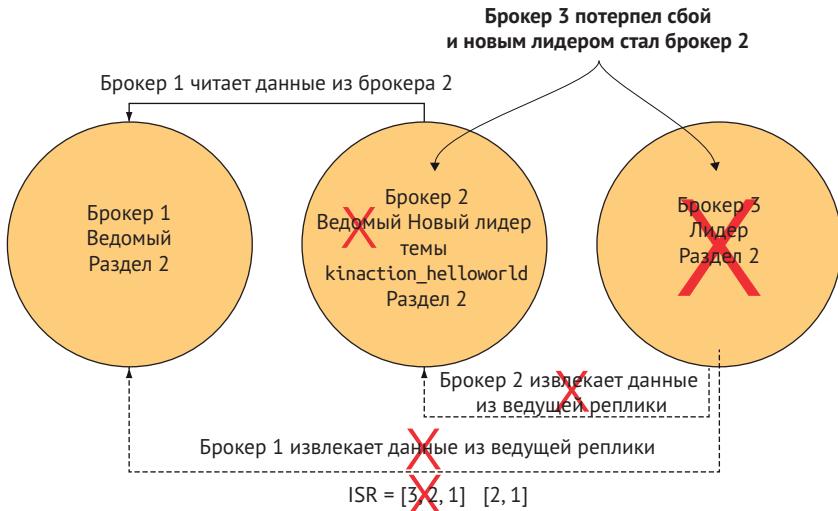


Рис. 6.4. Выбор нового лидера

ПРИМЕЧАНИЕ. В главе 5 мы обсуждали предложение по улучшению Kafka KIP-392, которое позволяет клиентам-потребителям получать данные из ближайшей реплики [16]. Чтение из более предпочтительной ведомой реплики, а не из ведущей может иметь смысл, если брокеры охватывают несколько физических центров обработки данных. Однако при обсуждении ведущих и ведомых реплик в этой книге мы будем подразумевать стандартное поведение, когда для чтения и записи используется лидер, если явно не указано иное.

Синхронизированные реплики (In-Sync Replicas, ISR) – это ключ к пониманию Kafka. Для каждой новой темы создается определенное количество реплик, которые добавляются в первоначальный список ISR [17]. Это количество может определяться конфигурационным параметром или значением по умолчанию из конфигурации брокера.

Одна из особенностей Kafka, на которую следует обратить внимание, заключается в том, что по умолчанию реплики не пытаются восстановить свою работоспособность самостоятельно. Если брокер, хранящий одну из реплик раздела, потерпел сбой, то Kafka (в настоящее время) не создаст новую копию. Мы упоминаем об этом, потому что некоторые пользователи привыкли к файловым системам, таким как HDFS, которые сохраняют свой номер репликации (выполняя самовосстановление), если блок был поврежден или оказался неисправным. Важным элементом, на который следует обратить внимание при мониторинге работоспособности наших систем, является соответствие количества синхронизированных реплик желаемому числу.

Почему так важно следить за этим количеством? Потому что всегда полезно знать, сколько копий у вас в запасе, прежде чем их количество достигнет 0! Допустим, у нас есть тема, состоящая из одного раздела, и этот раздел реплицируется три раза. В лучшем случае у нас будет две копии данных, находящиеся в ведущей реплике раздела. Конечно, ведомые реплики нагонят лидера. Но что, если мы потеряем еще одну синхронизированную реплику?

Также важно отметить, что, если реплика начинает слишком сильно отставать от лидера, ее можно удалить из списка ISR. Лидер замечает, что обновление ведомой реплики занимает слишком много времени, и удаляет ее из списка ведомых [17]. Затем лидер продолжает работу с новым списком ISR. Результат «медлительности» реплики для списка ISR такой же, как на рис. 6.4, где произошел сбой брокера.

6.4.1. Потеря данных

Что случится, если у нас не осталось синхронизированных реплик и мы потеряли ведущую реплику из-за сбоя? Когда параметр `unclean.leader.election.enable` имеет значение `true`, контроллер выберет лидера для раздела, даже если он не успел обновиться, чтобы система продолжала работать [15]. Проблема в том, что при этом могут быть потеряны данные, потому что ни одна из реплик не хранила всех данных на момент сбоя лидера.

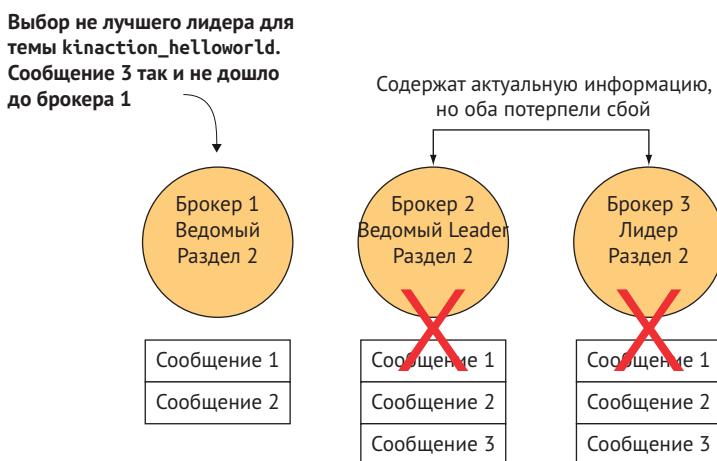


Рис. 6.5. Выбор не лучшего лидера

На рис. 6.5 показано, как могут теряться данные при использовании раздела с тремя репликами. В этом случае сразу два брокера, 3 и 2, потерпели сбой. Поскольку был разрешен выбор не лучшего лидера, брокер 1 становится новым лидером даже при том, что он не успел синхронизироваться с другими брокерами. Брокер 1 ни-

когда не получит сообщение 3 и не сможет предоставить эти данные клиентам. Ценой потери данных этот вариант позволяет нам продолжать обслуживать клиентов.

6.5. Взгляд внутрь Kafka

Существует множество инструментов, которые можно использовать для сбора и просмотра данных из наших приложений. В качестве примеров мы рассмотрим Grafana® (<https://grafana.com/>) и Prometheus® (<https://prometheus.io/>) и покажем, как на их основе создать простой стек мониторинга, который можно использовать для Confluent Cloud [18]³. Инструмент Prometheus мы используем для извлечения и хранения метрик Kafka. Затем полученные данные отправим в Grafana для создания удобных графических представлений. Чтобы понять, зачем нужны все инструменты, описываемые ниже, давайте кратко рассмотрим компоненты и работу каждого из них (рис. 6.6).

Информация из таких тем,
как `kinaction_alert`

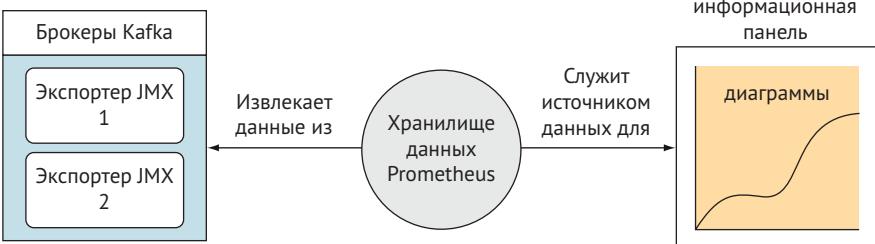


Рис. 6.6. Диаграмма потока данных

Как показано на рис. 6.6, мы используем JMX, чтобы заглянуть внутрь приложений Kafka. Экспорт JMX принимает уведомления JMX и экспортирует их в формате Prometheus. Prometheus получает данные от экспортера и сохраняет их. Затем различные инструменты могут получать информацию из Prometheus и отображать ее в информационных панелях.

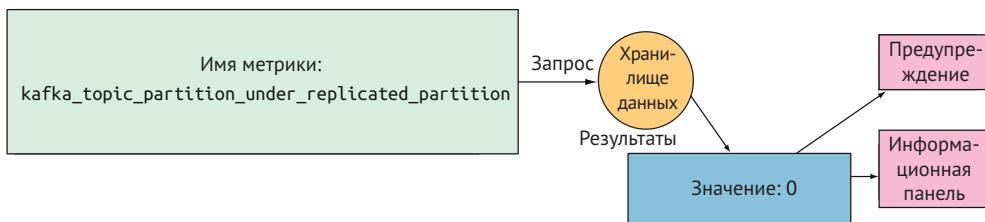
Существует множество образов Docker™ и файлов Docker Compose, объединяющих все эти инструменты, но при желании вы можете установить каждый инструмент на локальный компьютер, чтобы изучить этот процесс более подробно.

Отличный вариант экспортера Kafka доступен по адресу https://github.com/danielqsj/kafka_exporter. Мы выбрали этот инструмент

³ Grafana Labs Marks – торговая марка, принадлежащая Grafana Labs, используется в этой книге с разрешения Grafana Labs. Мы не связаны с Grafana Labs или ее филиалами, эта организация не поддерживает и не спонсирует нас.

за его простоту. Чтобы воспользоваться им, достаточно передать ему список серверов Kafka для наблюдения. Он прекрасно подходит для самых разных вариантов использования. Обратите внимание, что мы получим много метрик, характерных для клиентов и брокеров, потому что у нас довольно много параметров, которые хотелось бы отслеживать, но имейте в виду, что это далеко не полный список доступных метрик.

На рис. 6.7 показан запрос к локальному хранилищу данных, такому как локальный экземпляр Prometheus, куда собираются метрики из экспортёров Kafka. Как мы уже говорили, реплики в Kafka не восстанавливаются автоматически, поэтому всегда желательно отслеживать не до конца реплицированные разделы. Если это число больше 0, то это веский повод посмотреть, что происходит в кластере, чтобы определить причину проблемы с копированием. Мы можем отобразить данные из этого запроса на диаграмме или в информационной панели или, может быть, отправить предупреждение.



kafka_topic_partition_under_replicated_partition{instance="localhost:9308",job="kafka_exporter",partition="0",topic=kinaction_helloworld}0

Рис. 6.7. Пример запроса метрик

Как отмечалось выше, экспортёр Kafka предоставляет не все метрики JMX. Чтобы их получить больше, можно установить переменную окружения `JMX_PORT` при запуске процессов Kafka [19]. Доступны и другие инструменты, использующие агентов на Java для передачи метрик через конечные точки, которые Prometheus может прочитать.

В листинге 6.4 показано, как установить переменную `JMX_PORT` при запуске брокера [19]. Если брокер уже запущен и этот порт не был открыт, то нужно перезапустить брокера, чтобы применить это изменение. Настройку этой переменной можно автоматизировать, чтобы гарантировать ее действие при всех запусках брокеров в будущем.

Листинг 6.4. Запуск брокера с открытым портом JMX

```
JMX_PORT=$JMX_PORT bin/kafka-server-start.sh \ <--  
→ config/server0.properties
```

Добавляет переменную `JMX_PORT` при запуске кластера

6.5.1. Обслуживание кластера

При переносе в промышленное окружение нам потребуется настроить несколько серверов. Следует также отметить, что различные части экосистемы, такие как клиенты Kafka и Connect, реестр Schema Registry и REST Proxy, обычно работают на серверах, отличных от тех, на которых работают брокеры. Конечно, мы можем запустить все это на ноутбуке для тестирования (и все это можно сделать на одном сервере), но в промышленном окружении для большей надежности и эффективности все эти процессы должны выполняться на разных серверах. Подобно инструментам из экосистемы Hadoop, Kafka хорошо масштабируется по горизонтали с большим количеством серверов. Давайте посмотрим, как добавить новый сервер в кластер.

6.5.2. Добавление брокера

Начать с небольшого кластера – отличный способ приступить к работе, так как всегда можно добавить брокеров, чтобы расширить наши возможности. Чтобы добавить брокера Kafka в кластер, нужно просто запустить нового брокера Kafka с уникальным идентификатором. Этот идентификатор может быть создан установкой конфигурационного параметра `broker.id` или `broker.id.generation.enable` в значение `true` [10]. Вот и все. Но в этой ситуации новому брокеру не будут назначены никакие разделы! Любые разделы, созданные перед добавлением нового брокера, по-прежнему будут храниться в брокерах, существовавших на момент их создания [20]. Если новый брокер создавался только для обслуживания новых тем, то больше ничего делать не нужно.

6.5.3. Обновление кластера

Любое программное обеспечение рано или поздно приходится обновлять. Однако не все системы можно отключить одновременно, чтобы обновить их. Для предотвращения простоев приложений Kafka можно использовать *последовательный перезапуск* [14]. Под этими словами подразумевается простое обновление по одному брокеру за раз. На рис. 6.8 показано, как происходит последовательное обновление брокеров в кластере.

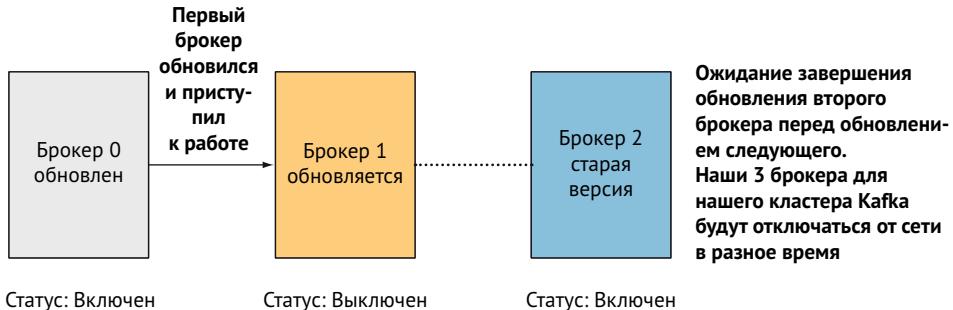


Рис. 6.8. Последовательное обновление и перезапуск

Важным свойством конфигурации брокера для последовательного перезапуска является `controlled.shutdown.enable`. Установка его в значение `true` позволяет передать лидерство в разделе до остановки брокера [21].

6.5.4. Обновление клиентов

Как упоминалось в главе 4, Kafka делает все возможное, чтобы отделить клиентов от брокера, однако иногда полезно знать версии клиентов, работающих с брокерами. Эта поддержка двусторонней совместимости с клиентами появилась в Kafka 0.10.2, и брокеры версии 0.10.0 или выше имеют эту поддержку [22]. Клиенты обычно можно обновить после обновления всех брокеров Kafka в кластере. Однако, так же как в случае любого обновления, обязательно ознакомьтесь с примечаниями к версии, чтобы убедиться, что более новые версии совместимы.

6.5.5. Резервные копии

В Kafka отсутствует стратегия резервного копирования, которую можно было бы использовать для сохранения резервной копии базы данных; эта платформа не делает моментальных снимков или резервных копий диска. Поскольку журналы Kafka существуют на диске, то, наверное, можно просто скопировать все каталоги с разделами? Ничто не мешает нам поступить так, но есть одна проблема – довольно сложно организовать одномоментное копирование всех каталогов с данными, находящихся в разных местах. Вместо ручного копирования и координации брокеров одним из предпочтительных вариантов является поддержка второго кластера [23]. Эти два кластера могут копировать события между собой. Одним из первых инструментов, которые можно встретить в промышленных окружениях, является `MirrorgMaker`. Новейшая версия этого инструмента (`MirrorgMaker 2.0`) была выпущена вместе с Kafka версии 2.4.0 [24]. В папке `bin` в каталоге установки Kafka находится сценарий командной оболочки с име-

нем *kafka-mirror-maker*, а также сценарий *connect-mirror-maker* для MirrorMaker 2.0.

Существуют также ряд других решений с открытым исходным кодом и корпоративные решения для зеркалирования данных между кластерами. Кроме того, имеются Confluent Replicator (<http://mng.Bz/yw7k>) и Cluster Linking (<http://mng.bz/OQZo>) [25].

6.6. Примечание о системах с сохранением состояния

Kafka – это приложение, которое работает с хранилищами данных, сохраняющими свое состояние. В этой книге мы используем не облачные, а наши собственные узлы. Есть несколько отличных ресурсов, в том числе сайт Confluent по использованию Kubernetes Confluent Operator API (<https://www.confluent.io/confluent-operator/>), а также образы Docker, включающие все, что нужно. Еще один интересный вариант – Strimzi™ (<https://github.com/stimzi/stimzi-kafka-operator>) для тех, кто желает запустить свой кластер в Kubernetes. На момент написания этой книги Strimzi представлял собой изолированный проект Cloud Native Computing Foundation® (<https://www.cncf.io/>). Эти инструменты можно найти на Docker Hub и использовать для опробования свежих идей, которые могут у вас зародиться. Однако универсального рецепта по организации инфраструктуры не существует.

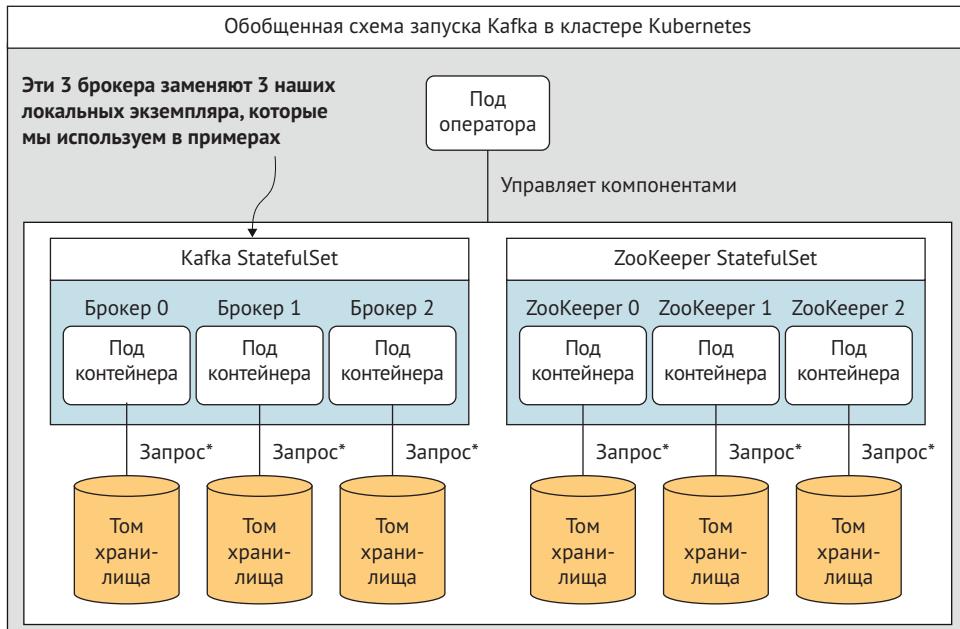
Одним из выдающихся преимуществ фреймворка Kubernetes является его способность быстро создавать новые кластеры с различными способами обмена данными между хранилищами и службами, которые Гвен Шапира (Gwen Shapira) исследует в своей статье «Recommendations for Deploying Apache Kafka on Kubernetes» [26]. Некоторым компаниям проще организовать для каждого продукта свой кластер, чем поддерживать один огромный кластер для всего предприятия. Возможность быстро развернуть кластер без добавления физических серверов может обеспечить быстрое развертывание продуктов, необходимых бизнесу.

На рис. 6.9 в общих чертах показано, как можно настроить брокеры Kafka в Kubernetes с помощью пода⁴ оператора⁵, такого как операторы Confluent и Strimzi. На рисунке используются терми-

⁴ Под (англ. *pod*) – это группа из одного или нескольких контейнеров приложений (например, Docker), включающая общие используемые хранилища (тома), IP-адрес и конфигурацию для запуска. – Прим. перев.

⁵ Оператор осуществляет упаковку, развертывание и управление приложениями Kubernetes, где под приложением Kubernetes подразумевается приложение, которое управляет с помощью API-интерфейсов Kubernetes и инструментов `kubectl` и развертывается в Kubernetes. – Прим. перев.

ны, характерные для Kubernetes, и мы не даем особых пояснений к ним, потому что не хотим смещать фокус с изучения самой платформы Kafka. Мы даем лишь общий обзор, поэтому имейте в виду, что это лишь обобщенная схема, описывающая, как Kafka может работать в кластере, а не конкретное описание его настройки.



* Запрос на выделение тома для хранилища

Рис. 6.9. Kafka в Kubernetes

Оператор Kubernetes – это отдельный под внутри кластера Kubernetes. Кроме того, каждый брокер находится в своем собственном поде, входящем в состав логической группы, называемой StatefulSet. Группы StatefulSet обеспечивают управление подами Kafka и гарантируют их упорядоченность и идентичность. Если, например, произойдет сбой пода с брокером (процессом JVM) с идентификатором 0, то будет создан и запущен новый под с тем же идентификатором. Более того, это вновь запущенный под будет подключен к тому же тому хранилища, что и его предшественник. Поскольку тома содержат разделы с сообщениями Kafka, то данные сохраняются. Сохранность состояния помогает обеспечить надежную работу всей системы в целом, даже когда контейнеры имеют короткий срок службы. Каждый узел ZooKeeper также будет действовать в своем собственном поде и являться частью своей собственной группы StatefulSet.

Для тех, кто плохо знаком с Kubernetes или испытывает волнение перед переходом на такую платформу, можем посоветовать одну

из возможных стратегий миграции – запускать клиентов и приложения Kafka в кластере Kubernetes перед брокерами Kafka. Запуск клиентов без состояния поможет вам почувствовать Kubernetes в начале вашего пути. Но вообще желательно хорошо разбираться в Kubernetes, чтобы запустить Kafka поверх этой платформы.

Один из авторов этой книги имел возможность столкнуться с командой разработчиков из четырех человек, в которой половина была сосредоточена на Kubernetes, а половина – на Kafka. Конечно, такое разделение может потребоваться не в каждой команде. Время, необходимое разработчику для изучения Kubernetes, зависит от вашей команды и общего опыта.

6.7. Упражнение

Нередко бывает трудно применить новые знания на практике, и поскольку эта глава в большей степени посвящена командной строке, чем программному коду, то было бы полезно выполнить небольшое упражнение, чтобы освоить еще один способ определения количества не до конца реплицированных разделов. Существуют ли какие-нибудь средства командной строки, позволяющие сделать это, помимо графических инструментов, таких как информационные панели?

Допустим, мы решили проверить работоспособность одной из наших тем с именем `kinaction_replica_test`. Каждый раздел этой темы имеет три реплики, и мы хотим убедиться, что в списке ISR присутствуют три брокера на случай, если вдруг произойдет сбой какого-нибудь брокера. Какую команду можно использовать, чтобы просмотреть эту тему и увидеть ее текущее состояние? В листинге 6.5 показан пример, описывающий эту тему [27]. Обратите внимание, что `ReplicationFactor` равен 3, а в списке реплик также показаны три идентификатора брокеров. Однако список ISR показывает только два значения, хотя должно быть три!

Листинг 6.5. Описание реплик темы: проверка счетчика ISR

Обратите внимание на параметр
--topic и флаг --describe

```
$ bin/kafka-topics.sh --describe --bootstrap-server localhost:9094 \  
--topic kinaction_replica_test
```

```
Topic:kinaction_replica_test PartitionCount:1 ReplicationFactor:3 Configs:  
Topic: kinaction_replica_test Partition: 0
```

```
Leader: 0 Replicas: 1,0,2 Isr: 0,2
```

Информация о ведущей реплике, разделе и синхронизированных репликах

Взглянув на вывод команды, можно заметить проблему не до конца реплицированных разделов, однако ту же информацию можно получить с помощью флага `--under-replicated-partitions` [27]. В листинге 6.6 показано, как использовать этот флаг, который отфильтровывает ненужные сведения и выводит только информацию о проблеме не до конца реплицированных разделов.

Листинг 6.6. Использование флага `--under-replicated-partitions`

```
bin/kafka-topics.sh --describe --bootstrap-server localhost:9094 \
--under-replicated-partitions
Topic: kinaction_replica_test Partition: 0
→ Leader: 0 Replicas: 1,0,2 Isr: 0,2
```

Обратите внимание на флаг `--under-replicated-partitions`

В списке ISR присутствуют только два брокера!

Как показывает листинг 6.6, при использовании флага `--describe` не ограничивается проверкой не до конца реплицированных разделов в определенной теме. Мы можем использовать эту команду, чтобы быстро найти проблемы в нашем кластере. В главе 9 мы рассмотрим другие имеющиеся инструменты, входящие в состав Kafka, когда будем говорить об инструментах администрирования.

СОВЕТ. При использовании любой из команд в этой главе всегда желательно сначала выполнить команду без параметров и прочитать справку о параметрах, которые можно использовать для устранения неполадок.

По мере изучения Kafka в этой главе мы пришли к пониманию, что имеем дело со сложной системой. Однако существует множество инструментов командной строки и метрик, которые помогают следить за состоянием кластера. В следующей главе мы продолжим использовать команды для выполнения определенных задач в этой динамичной системе на протяжении всего ее жизненного цикла.

Итоги

- Брокеры являются центральным элементом Kafka и обеспечивают взаимодействие внешних клиентов с нашими приложениями. Кластеры гарантируют не только масштабирование, но и надежность.
- Для поддержки согласованности в распределенном кластере можно использовать ZooKeeper. Одним из примеров может служить выбор нового контроллера из нескольких доступных брокеров.
- Чтобы упростить управление кластером, можно задать конфигурации на уровне брокеров, которые наши клиенты могут переопределить и задать определенные параметры.

- Реплики позволяют создать несколько распределенных копий данных в кластере. Это помогает в случае, если какой-то из брокеров выйдет из строя и станет недоступен.
- Синхронизированные реплики (In-Sync Replicas, ISR) содержат все данные, имеющиеся у лидера, и могут взять на себя лидерство без потери данных.
- Мы можем использовать метрики для создания графиков и мониторинга кластера или оповещения о потенциальных проблемах.

Ссылки

- 1 «Post Kafka Deployment». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/kafka/post-deployment.html#balancing-replicas-across-racks> (доступно по состоянию на 15 сентября 2019).
- 2 «KIP-500:ReplaceZooKeeperwithaSelf-ManagedMetadataQuorum». Wiki for Apache Kafka. Apache Software Foundation (09 июля 2020). <https://cwiki.apache.org/confluence/display/KAFKA/KIP-500%3A+Replace+ZooKeeper+with+a+Self-Managed+Metadata+Quorum> (доступно по состоянию на 22 августа 2020).
- 3 F. Junqueira and N. Narkhede. «Distributed Consensus Reloaded: Apache ZooKeeper and Replication in Apache Kafka». Confluent blog (August 27, 2015). <https://www.confluent.io/blog/distributed-consensus-reloaded-apache-zookeeper-and-replication-in-kafka/> (доступно по состоянию на 15 сентября 2019).
- 4 «Kafka data structures in Zookeeper [sic]». Wiki for Apache Kafka. Apache Software Foundation (10 февраля 2017). <https://cwiki.apache.org/confluence/display/KAFKA/Kafka+data+structures+in+Zookeeper> (доступно по состоянию на 19 января 2020).
- 5 C. McCabe. «Apache Kafka Needs No Keeper: Removing the Apache ZooKeeper Dependency». Confluent blog. (May 15, 2020). <https://www.confluent.io/blog/upgrading-apache-kafka-clients-just-got-easier> (доступно по состоянию на 20 августа 2021).
- 6 Apache Kafka Binder (n.d.). https://docs.spring.io/spring-cloud-stream-binder-kafka/docs/3.1.3/reference/html/spring-cloud-stream-binder-kafka.html#_apache_kafka_binder (доступно по состоянию на 18 июля 2021).
- 7 «CLI Tools for Confluent Platform». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/installation/cli-reference.html> (доступно по состоянию на 25 августа 2021).
- 8 «ZkData.scala». Apache Kafka GitHub. <https://github.com/apache/kafka/blob/99b9b3e84f4e98c3f07714e1de6a139a004cbc5b/core/>

<src/main/scala/kafka/zk/ZkData.scala> (доступно по состоянию на 27 августа 2021).

- 9 «A Guide To The Kafka Protocol». Wiki for Apache Kafka. Apache Software Foundation (14 июня 2017). <https://cwiki.apache.org/confluence/display/KAFKA/A+Guide+To+The+Kafka+Protocol> (доступно по состоянию на 15 сентября 2019).
- 10 «Kafka Broker Configurations». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/installation/configuration/broker-configs.html> (доступно по состоянию на 21 августа 2021).
- 11 «Logging». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/kafka/post-deployment.html#logging> (доступно по состоянию на 21 августа 2021).
- 12 «Controller». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/kafka/post-deployment.html#controller> (доступно по состоянию на 21 августа 2021).
- 13 «Kafka Controller Internals». Wiki for Apache Kafka. Apache Software Foundation (26 января 2014). <https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Controller+Internals> (доступно по состоянию на 15 сентября 2019).
- 14 «Post Kafka Deployment». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/kafka/post-deployment.html#rolling-restart> (доступно по состоянию на 10 июля 2019).
- 15 «Replication». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/kafka/design.html#replication> (доступно по состоянию на 21 августа 2021).
- 16 «KIP-392: Allow consumers to fetch from closest replica». Wiki for Apache Kafka. Apache Software Foundation (5 ноября 2019). <https://cwiki.apache.org/confluence/display/KAFKA/KIP-392%3A+Allow+consumers+to+fetch+from+closest+replica> (доступно по состоянию на 10 декабря 2019).
- 17 N. Narkhede. «Hands-free Kafka Replication: A lesson in operational simplicity». Confluent blog (July 1, 2015). <https://www.confluent.io/blog/hands-free-kafka-replication-a-lesson-in-operational-simplicity/> (доступно по состоянию на 02 октября 2019).
- 18 «Observability Overview and Setup». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/tutorials/examples/ccloud-observability/docs/observability-overview.html> (доступно по состоянию на 26 августа 2021).
- 19 «Kafka Monitoring and Metrics Using JMX». Confluent documentation. (n.d.). <https://docs.confluent.io/platform/current/installation/docker/operations/monitoring.html> (доступно по состоянию на 12 июня 2020).

- 20 «Scaling the Cluster (Adding a node to a Kafka cluster)». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/kafka/post-deployment.html#scaling-the-cluster-adding-a-node-to-a-kafka-cluster> (доступно по состоянию на 21 августа 2021).
- 21 «Graceful shutdown». Apache Software Foundation (n.d.). https://kafka.apache.org/documentation/#basic_ops_restarting (доступно по состоянию на 11 мая 2018).
- 22 C. McCabe. «Upgrading Apache Kafka Clients Just Got Easier». Confluent blog. (18 июля 2017). <https://www.confluent.io/blog/upgrading-apache-kafka-clients-just-got-easier> (доступно по состоянию на 02 октября 2019).
- 23 «Backup and Restoration». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/kafka/post-deployment.html#backup-and-restoration> (доступно по состоянию на 21 августа 2021).
- 24 Release Notes, Kafka Version 2.4.0. Apache Software Foundation (n.d.). https://archive.apache.org/dist/kafka/2.4.0/RELEASE_NOTES.html (доступно по состоянию на 12 мая 2020).
- 25 «Multi-DC Solutions». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/multi-dc-deployments/index.html#multi-dc-solutions> (доступно по состоянию на 21 августа 2021).
- 26 G. Shapira. «Recommendations_for_Deploying_Apache_Kafka_on_Kubernetes». White paper (2018). <https://www.confluent.io/resources/recommendations-for-deploying-apache-kafka-on-kubernetes> (доступно по состоянию на 15 декабря 2019).
- 27 «Replication tools». Wiki for Apache Kafka. Apache Software Foundation (4 февраля 2019). <https://cwiki.apache.org/confluence/display/kafka/replication+tools> (доступно по состоянию на 19 января 2019).

7

Темы и разделы

Эта глава охватывает следующие темы:

- параметры создания и настройки;
- как организованы разделы в виде файлов журналов;
- как влияет сегментирование на размещение данных внутри разделов;
- тестирование с помощью `EmbeddedKafkaCluster`;
- сжатие тем и способы сохранения данных.

В этой главе мы подробнее рассмотрим, как хранятся данные в темах, как создавать и поддерживать темы, особенности размещения разделов в общей архитектуре и как можно просматривать наши данные в брокерах. Мы также посмотрим, как обновить данные темы, не добавляя их в конец журнала.

7.1. Темы

Чтобы освежить вашу память, напомним, что тема – это не конкретное понятие, а некоторая логическая структура. Обычно данные темы хранятся несколькими брокерами. Большинство приложений, потребляющих данные из Kafka, видят эти данные как принадлежащие одной теме; и не нужно никакой другой информации, кроме имени темы, чтобы подписаться на получение сообщений из нее. Однако за именем темы скрывается один или несколько

разделов, которые фактически хранят данные [1]. Данные, составляющие тему, Kafka записывает в журналы, находящиеся в файловых системах брокеров.

На рис. 7.1 показано деление на разделы одной темы с именем `kinaction_helloworld`. Каждый брокер хранит на своем диске полную копию раздела. На рис. 7.1 также видно, что разделы состоят из сообщений, отправляемых в тему.

Тема `kinaction_helloworld` состоит из трех разделов, которые почти наверняка будут распределены между разными брокерами

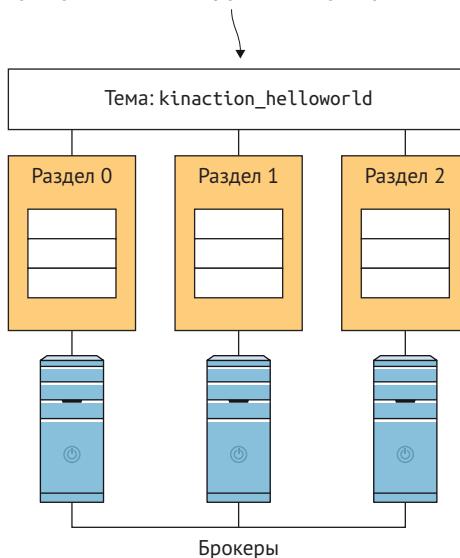


Рис. 7.1. Пример темы с разделами

Если запись данных в тему выполняется так просто, как показано в начальных примерах, то зачем нам понимать роль тем и их организацию? А затем, что это влияет на то, как потребители получают доступ к данным. Допустим, наша компания предлагает платное обучение и предоставляет веб-приложение для записи на курсы, которое отправляет события с описанием действий пользователя в наш кластер Kafka. Процесс подачи заявки может генерировать множество событий. Например, событие первоначального поиска, выбор конкретного курса обучения и подтверждение выбора. Куда приложения-производители должны отправлять все эти данные – в одну или в несколько тем? Каждое сообщение относится к определенному типу события, и должны ли они оставаться распределенными по разным темам? Каждый подход имеет свои плюсы и минусы, и в каждом случае необходимо учитывать конкретные особенности, чтобы определить наилучший подход в каждой ситуации.

Проектирование темы – это двухэтапный процесс. На первом этапе определяется перечень событий и необходимость их распределения по разным темам. На втором этапе внимание сосредотачивается на каждой конкретной теме и выборе количества разделов для каждой из них. Важно помнить, что деление на разделы – это вопрос организации каждой темы, а не кластера. Мы можем установить количество разделов по умолчанию перед созданием темы, но в большинстве случаев необходимо учитывать, как будет использоваться тема и какие данные она будет хранить.

Выбор определенного количества разделов должен быть обоснованным. Юн Рао (Jun Rao) написал фантастическую статью под названием «How to choose the number of topics/partitions in a Kafka cluster?» в блоге Confluent, посвященную этому вопросу [2]! Допустим, мы решили создать по одному разделу на каждом сервере. Однако наличие разделов по количеству серверов не означает, что производители будут писать в них равномерно. Для этого нужно гарантировать, что каждый лидер раздела распределен именно так.

Нам также необходимо поближе познакомиться с нашими данными. Давайте перечислим, о чём следует подумать как в общем случае, так и в данном примере с платным обучением:

- о корректности данных;
- количестве сообщений, приходящихся на одного потребителя;
- объем хранимых данных, которые нужно будет обрабатывать.

Корректность данных является, пожалуй, самой животрепещущей проблемой в реальных проектах. Это довольно расплывчатое понятие, поэтому наше определение в большей степени отражает наше мнение. Для данного примера это означает, что события должны быть упорядочены, попадать в один и тот же раздел и, следовательно, в одну и ту же тему. Конечно, мы можем упорядочить события по отметке времени, но, как показывает наш опыт, координация событий между темами сопряжена с большими трудностями (и подвержена ошибкам). Если мы используем сообщения с ключами и нам необходимо упорядочивать их, то следует позаботиться о разделах и любых будущих изменениях этих разделов [1].

Чтобы обеспечить корректность данных с тремя предыдущими примерами событий, может быть полезно поместить события с ключом сообщения (включая идентификатор обучаемого) в две отдельные темы, предназначенные для событий бронирования

и подтверждения/оплаты. Эти события относятся к конкретному учащемуся, и такой подход мог бы обеспечить подтверждение для этого конкретного учащегося. А вот события поиска, возможно, не имеет смысла упорядочивать по конкретным учащимся. Например, нашей команде аналитиков может быть более полезна информация о наиболее популярных поисковых запросах и городах, а не об учащихся.

Далее следует учесть *количество сообщений*, приходящихся на одного потребителя. Давайте посмотрим, сколько сообщений будет поступать в темы нашей вымышленной системы обучения. Количество событий поиска будет намного превосходить количество других событий. Допустим, место обучения находится недалеко от большого города и получает 50 000 запросов в день, но в нем может разместиться только 100 учащихся. Таким образом, в течение дня поступает 50 000 поисковых событий и менее 100 бронирований мест в классах. Будет ли приложение нашей группы, обрабатывающей бронирования, подписываться на общую тему, в которой интерес представляет менее чем 1 % сообщений от общего числа? По сути, большую часть времени потребитель будет фильтровать массу событий, чтобы обработать лишь несколько избранных.

Еще один важный момент – это *объем данных*, которые потребуется обрабатывать. Потребуется ли запускать несколько потребителей для обработки событий в установленные временные рамки? Если да, то мы должны вспомнить, что количество потребителей в группе ограничено количеством разделов в нашей теме [2]. В такой ситуации проще создать больше разделов, чем может потребоваться на первый взгляд. Наличие запаса для увеличения числа потребителей позволит нам увеличивать объем передаваемых данных без необходимости заботиться о перераспределении данных. Однако важно понимать, что разделы не являются неограниченным бесплатным ресурсом, как говорилось в статье Rao, упоминавшейся выше. Это также означает наличие большего количества брокеров для миграции на случай сбоев, что может стать потенциальной головной болью при создании.

Лучше всего найти золотую середину и постараться придерживаться ее при разработке наших систем. На рис. 7.2 показана архитектура, которая лучше всего подходит для двух тем с тремя типами событий, использованных в нашем сценарии. Как всегда, дополнительные требования или детали могут изменить фактическую реализацию.

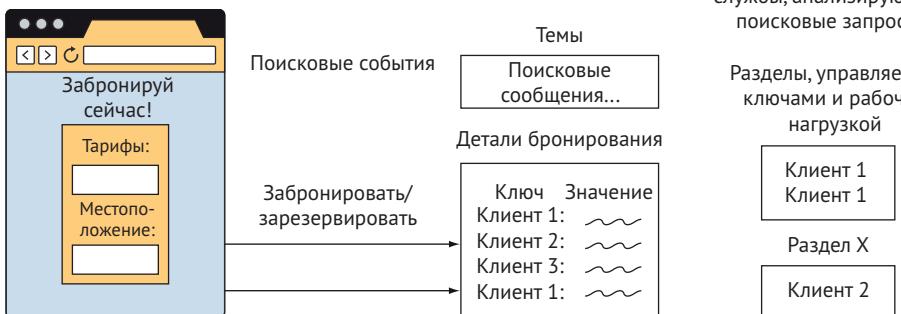


Рис. 7.2. Пример организации тем для учебного центра

Наконец, при выборе количества разделов для темы следует учесть, что в настоящее время не поддерживается возможность уменьшения их числа [3]. Есть некоторые способы сделать это, но применять их в промышленном окружении определенно не рекомендуется! Давайте задумаемся, почему.

Когда потребители подписываются на тему, они в действительности подключаются к разделу. Удаление раздела может привести к потере его текущей позиции в потоке событий, когда или если потребитель начнет чтение из переназначенного раздела. В такой ситуации нам придется побеспокоиться о том, чтобы наши сообщения с ключами и клиенты-потребители могли правильно обрабатывать любые изменения, которые могут происходить на уровне брокера. Мы влияем на потребителей своими действиями. Теперь, обсудив общую архитектуру, углубимся в параметры, которые можно настроить при создании тем. Мы уже кратко затрагивали их, когда создавали темы в главе 3, поэтому здесь мы просто копнем чуть глубже.

7.1.1. Параметры создания темы

Темы Kafka имеют несколько основных параметров, которые необходимо определить при их создании. Хотя мы уже создавали темы, начиная с главы 2 (тема `kinaction_helloworld`), нам все же нужно глубже исследовать основные параметры, которые прежде не упоминались. Настраивать эти параметры следует обдуманно и осторожно, понимая, на что они влияют [4].

Одно из важнейших решений, которое необходимо принять во время создания, – это необходимость удаления темы. Поскольку эта операция чрезвычайно важна, мы должны гарантировать, что она не будет выполнена без логического подтверждения. Для этого Kafka требует, чтобы мы включили параметр `delete.topic.enable`. Если в этом параметре установлено значение `true`, мы сможем потребовать удалить тему, и она будет удалена [5].

Приятно знать, что сценарии применения Kafka в целом хорошо и подробно описаны в документации. Мы рекомендуем сначала выполнить команду `kafka-topics.sh`, чтобы увидеть, какие действия можно предпринять. В листинге 7.1 показана неполная команда для получения справки.

Листинг 7.1. Команда для получения параметров тем

```
bin/kafka-topics.sh ←
    Запуск общей команды Kafka,
    связанной с темами
```

В выводе этой команды выделяется один очевидный параметр: `--create`. Добавив этот параметр, можно получить дополнительную информацию, связанную с действием создания, например: «Missing required argument "[topic]"» (отсутствует обязательный аргумент "[тема]"). В листинге 7.2 показана все еще неполная команда, в которую добавлен один параметр.

Листинг 7.2. Команда для получения списка параметров создания темы

```
bin/kafka-topics.sh --create ←
    Выводит сообщение об ошибке для
    конкретной команды и справочную
    документацию
```

Зачем тратить время на обсуждение этих шагов, если многие пользователи и без того знакомы со страницами справочного руководства (`man`), работая в Linux®? Несмотря на то что в документации Kafka не описывается такой способ применения ее инструментов, им можно воспользоваться, прежде чем начать поиск в Google.

Задав имя с длиной не более 249 символов (удивительно, но нам известны случаи, когда люди пытались задать более длинные имена), можно создать тему [6]. Давайте создадим тему `kinaction_topictopicandpart` с коэффициентом репликации 2 и с двумя разделами. В листинге 7.3 показан синтаксис соответствующей команды [3].

Листинг 7.3. Создание темы

```
bin/kafka-topics.sh
--create --bootstrap-server localhost:9094 \
--topic kinaction_topictopicandpart \
--partitions 2 \
--replication-factor 2 ←
    Добавить параметр
    --create в команду
    Задать имя темы
    Создать два раздела в теме
    Гарантировать создание двух копий наших данных
```

После создания темы можно попробовать получить ее описание, чтобы убедиться в правильности настроек. Обратите внимание на рис. 7.3, что количество разделов и реплик соответствуют только что выполненной команде.

```
> bin/kafka-topics.sh --bootstrap-server localhost:9092 --describe --topic kinaction_topicandpart
Topic: kinaction_topicandpart    PartitionCount: 2      ReplicationFactor: 2      Configs:
  Topic: kinaction_topicandpart    Partition: 0        Leader: 1        Replicas: 1,0   Isr: 1,0
  Topic: kinaction_topicandpart    Partition: 1        Leader: 0        Replicas: 0,2   Isr: 0,2
```

Рис. 7.3. Описание темы, в котором видно, что она содержит два раздела

Еще одна настройка на уровне брокера, о которой желательно позаботиться, – установить параметр `auto.create.topics.enable` в значение `false` [7]. Это гарантирует, что темы будут создаваться только явно, а не по желанию производителя, отправляющего сообщение на имя темы, которое было введено с ошибкой и никогда не существовало до попытки отправить сообщение. Несмотря на отсутствие тесной связи, производители и потребители обычно должны знать правильное название темы, где должны храниться их данные. Автоматическое создание тем может вызвать путаницу. Но при тестировании и изучении Kafka автоматическое создание тем может быть полезным. Рассмотрим конкретный пример. Если выполнить команду: `kafka-console-producer.sh --bootstrap-server localhost:9094 --topic notexisting` в отсутствие темы `notexisting`, то Kafka автоматически создаст ее. А выполнив команду:

```
kafka-topics.sh --bootstrap-server localhost:9094 --list
```

вы увидите, что эта тема появилась в кластере.

В промышленных окружениях обычно не принято удалять данные, но при изучении тем можно столкнуться с некоторыми ошибками. Приятно знать, что мы действительно можем удалить тему, если это необходимо [3]. Удаление темы сопровождается удалением всех данных, хранящихся в ней. Мы бы не советовали делать это, если вы не готовы навсегда расстаться с данными! В листинге 7.4 показано, как использовать все ту же команду `kafka-topic`, но на этот раз для удаления темы с именем `kinaction_topicandpart` [3].

Листинг 7.4. Удаление темы

```
bin/kafka-topics.sh --delete --bootstrap-server localhost:9094
  --topic kinaction_topicandpart
```

Удалит тему
kinaction_topicandpart

Обратите внимание на параметр `--delete`. После выполнения этой команды вы не сможете использовать эту тему для передачи своих данных, как раньше.

7.1.2. Коэффициенты репликации

Планируя количество реплик, необходимо учитывать количество брокеров – реплик не должно быть больше. На самом деле попытка создать тему с количеством реплик, превышающим общее количе-

ство брокеров, приведет к ошибке `InvalidReplicationFactorException` [8]. Причину этого легко понять. Представьте, что у нас всего два брокера и мы пытаемся создать три реплики раздела. Одна из них будет находиться в одном брокере, а две других – в другом. В этом случае, если из-за сбоя мы потеряем брокера, в котором разместились две реплики, у нас останется только одна копия данных. Одновременная потеря нескольких реплик не лучший способ обеспечить возможность восстановления в случае сбоя.

7.2. Разделы

Теперь перейдем от работы с командами Kafka на уровень тем и начнем с того, что более подробно рассмотрим разделы. С точки зрения потребителя, каждый раздел – это неизменяемый журнал с сообщениями. Он должен только расти и добавлять сообщения в наше хранилище данных. На практике объем данных не может расти вечно, однако такое представление о том, что данные только добавляются, но не изменяются на месте, является хорошей мысленной моделью, которой следует придерживаться. Кроме того, клиенты-потребители не могут напрямую удалять сообщения. Эта особенность позволяет воспроизводить сообщения из темы и может пригодиться во многих ситуациях.

7.2.1. Размещение раздела

Часто полезно знать, как данные хранятся в наших брокерах. Для начала давайте найдем каталог `log.dirs` (или `log.dir`). Его можно найти, выполнив поиск строки «`log.dirs`» в файле `server.properties`, если вы следовали инструкциям в приложении А. В этом каталоге должны находиться папки с именами, совпадающими с названиями тем и номерами разделов. Если выбрать одну из них и заглянуть внутрь, то можно увидеть несколько файлов с расширениями `.index`, `.log` и `.timeindex`. На рис. 7.4 показано, как выглядит один раздел (в данном случае раздел 1) в нашей тестовой теме (список каталогов получен командой `ls`).

```
> ls /tmp/kafkainaction/kafka-logs-0/kinaction_topicandpart-1
00000000000000000000000000000000.index      00000000000000000000000000000000.log      00000000000000000000000000000000.timeindex  leader-epoch-checkpoint
```

Рис. 7.4. Список файлов в каталоге раздела

Внимательные читатели могут увидеть в своем каталоге файл с именем `leader-epoch-checkpoint` и, возможно, даже файлы с расширением `.snapshot` (не показаны на рис. 7.4). Но мы не будем тратить время на рассмотрение этих файлов.

В файлах с расширением `.log` хранятся наши фактические данные. Другая важная информация в файле журнала включает сме-

щение сообщения, а также поле `CreateTime`. Но зачем нужны другие файлы? Поскольку Kafka создавалась с прицелом на высокую производительность, она использует файлы `.index` и `.timeindex` для хранения соответствий между смещением сообщения и физической позицией внутри индексного файла [9].

Как уже было показано выше, разделы состоят из множества файлов. По сути, это означает, что на физическом диске раздел представлен несколькими файлами, представляющими сегменты [10]. На рис. 7.5 показано, как несколько сегментов могут составлять один раздел.

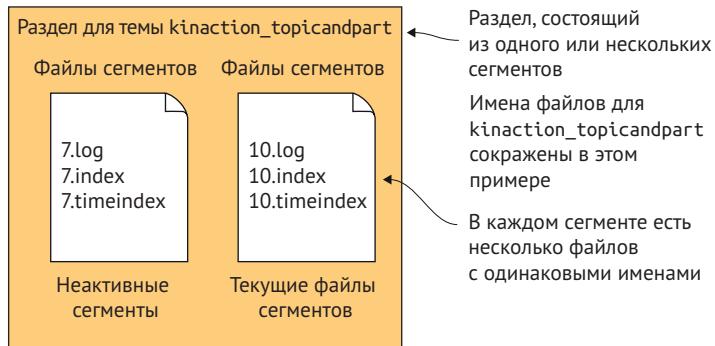


Рис. 7.5 Сегменты, составляющие раздел

Активный сегмент – это файл, куда в данный момент записываются новые сообщения [11]. На рис. 7.5 файл `10.log` – это место, куда записываются сообщения. Kafka поддерживает старые сегменты для решения разных задач, в которых активный сегмент не участвует, включая управление хранением на основе размеров сообщений или времени их поступления. Эти старые сегменты (такие как `7.log` на рис. 7.5) могут использоваться для сжатия тем, о чём мы поговорим ниже в этой главе.

Итак, теперь мы знаем, почему в каталоге раздела может быть несколько файлов с одинаковыми именами, но с разными расширениями: `.index`, `.timeindex` и `.log`. Например, для четырех сегментов у нас будет четыре набора файлов с тремя расширениями, упомянутыми выше, т. е. всего 12 файлов. Если мы видим только по одному файлу с каждым расширением, это означает, что у нас только один сегмент.

7.2.2. Просмотр журналов

Давайте попробуем заглянуть в файл журнала, чтобы увидеть сообщения, которые были отправлены в тему. Если открыть его в текстовом редакторе, то можно увидеть сообщения в удобочитаемом формате. В Confluent есть специальный сценарий для

просмотра сегментов журнала [12]. В листинге 7.5 показано, как передать команды `awk` и `grep` для просмотра файла сегмента журнала для раздела 1 темы `kinaction_topicandpart`.

Листинг 7.5. Просмотр сегмента журнала

```
bin/kafka-dump-log.sh --print-data-log \
    --files /tmp/kafkainaction/kafka-logs-0/
    ↳ kinaction_topicandpart-1/*.log \
    | awk -F: '{print $NF}' | grep kinaction
```

Вывести данные, которые имеют трудночитаемый формат в текстовом редакторе

Передать файл для чтения

Используя параметр `--files`, который является обязательным, мы указали файл сегмента. В случае успешного выполнения эта команда выведет на экран список сообщений. Без `awk` и `grep` в списке также появятся смещения и другие метаданные, такие как кодеки сжатия. Это определенно интересный способ увидеть, как Kafka размещает сообщения в брокере и какие данные хранятся вместе с этими сообщениями. Возможность увидеть фактические сообщения весьма полезна, поскольку помогает понять, как действует журнал в Kafka.

Взглянув на рис. 7.6, можно увидеть фактические данные в виде текста, который читается немногого проще, чем при использовании текстового редактора. Например, в файле сегмента можно увидеть сообщение с текстом `kinaction_helloworld`. Надеюсь, теперь вы сможете получить для себя больше ценных данных!

```
> bin/kafka-dump-log.sh --print-data-log --files /tmp/kafkainaction/kafka-logs-0/kinaction_topicandpart-1/*
log | awk -F: '{print $NF}' | grep kinaction
Dumping /tmp/kafkainaction/kafka-logs-0/kinaction_topicandpart-1/000000000000000000.log
kinaction_helloworld
```

Рис. 7.6. Просмотр сегмента журнала

Что касается большого числа в имени файла журнала, то оно не случайно. Имя сегмента должно совпадать с первым смещением в этом файле.

Узнав о возможности видеть эти данные, у многих из вас может возникнуть беспокойство, что их может увидеть кто-то еще. Поскольку безопасность данных и управление доступом являются общими проблемами для большинства данных, мы рассмотрим способы защиты Kafka и тем в главе 10. Однако знание, как просматривать журналы, может пригодиться для понимания того, как они устроены на самом деле.

Это знание помогает представить Kafka как живую и сложную систему (сложность во многом обусловлена ее распределенностью), которая время от времени нуждается в некотором уходе и подпитке. В следующем разделе мы займемся тестированием нашей темы.

7.3. Тестирование с помощью EmbeddedKafkaCluster

После знакомства с параметрами конфигурации было бы неплохо протестировать их. Есть ли возможность развернуть кластер Kafka, не имея под рукой настоящего действующего кластера? Kafka Streams предоставляет служебный класс интеграции под названием `EmbeddedKafkaCluster`, который занимает промежуточное положение между фиктивными объектами и полноценным кластером. Этот класс реализует кластер Kafka в памяти [13]. Он создан специально для Kafka Streams, но мы можем использовать его для тестирования наших клиентов Kafka.

Листинг 7.6 повторяет структуру тестов из книги «Kafka Streams in Action» Билла П. Беджека (William P. Bejeck Jr.)⁶, например его класса `KafkaStreamsYellingIntegrationTest` [14]. Эта книга, а также следующая книга этого автора, «Event Streaming with Kafka Streams and ksqlDB», содержат более подробные примеры тестирования. Мы рекомендуем ознакомиться с ними, включая его предложение использовать Testcontainers (<https://www.testcontainers.org/>). В листинге 7.6 показан пример тестирования с использованием `EmbeddedKafkaCluster` и JUnit 4.

Листинг 7.6. Тестирование с использованием EmbeddedKafkaCluster

```
@ClassRule
public static final EmbeddedKafkaCluster embeddedKafkaCluster
    = new EmbeddedKafkaCluster(BROKER_NUMBER); ← Использовать аннотацию JUnit для создания кластера с определенным количеством брокеров

private Properties kaProducerProperties;
private Properties kaConsumerProperties;

@Before
public void setUpBeforeClass() throws Exception {
    embeddedKafkaCluster.createTopic(TOPIC,
        PARTITION_NUMBER, REPLICATION_NUMBER);
    kaProducerProperties = TestUtils.producerConfig(
        embeddedKafkaCluster.bootstrapServers(),
        AlertKeySerde.class,
        StringSerializer.class); ← Настроить потребителя для работы с брокерами встроенного кластера

    kaConsumerProperties = TestUtils.consumerConfig(
        embeddedKafkaCluster.bootstrapServers(),
        AlertKeySerde.class,
        StringDeserializer.class); ←

}
```

⁶ Беджек Билл, «Kafka Streams в действии», Прогресс книга, 2019, ISBN: 978-5-4461-1201-2. – Прим. перев.

```

@Test
public void testAlertPartitioner() throws InterruptedException {
    AlertProducer alertProducer = new AlertProducer();
    try {
        alertProducer.sendMessage(kaProducerProperties);
    } catch (Exception ex) {
        fail("kinaction_error EmbeddedKafkaCluster exception"
            + ex.getMessage());
    }
    Клиент вызывается как обычно, без передачи любой информации о встроенным базовом кластере
}

AlertConsumer alertConsumer = new AlertConsumer();
ConsumerRecords<Alert, String> records =
    alertConsumer.getAlertMessages(kaConsumerProperties);
TopicPartition partition = new TopicPartition(TOPIC, 0);
List<ConsumerRecord<Alert, String>> results = records.records(partition);
assertEquals(0, results.get(0).partition()); Убедиться, что встроенный кластер обработал сообщение, получив его от производителя и передав потребителю
}

```

При тестировании с помощью `EmbeddedKafkaCluster` важно убедиться, что встроенный кластер запущен до начала фактического тестирования. Поскольку этот кластер является временным, также важно убедиться, что производители и потребители знают, как сослаться на этот кластер в памяти. Чтобы обнаружить необходимые конечные точки, можем использовать метод `bootstrapServers()`, возвращающий всю информацию, необходимую клиентам. Внедрение этой конфигурации в экземпляры клиентов зависит от вашей стратегии конфигурации, но в простейшем случае можно установить значения с помощью вызова метода. Помимо настройки конфигурационных параметров, в остальном клиенты могут тестироваться без реализации фиктивных функций Kafka!

Тест в листинге 7.6 проверяет логику работы `AlertLevelPartitioner`. Эта логика с нашим примером кода в главе 4 должна отправить критическое сообщение в раздел 0. Получение сообщения для `TopicPartition(TOPIC, 0)` и просмотр включенных сообщений подтверждает местоположение раздела сообщений. В целом этот уровень тестирования обычно считается интеграционным тестированием и выходит за пределы одного тестируемого компонента. На данный момент мы протестировали клиентскую логику вместе с кластером Kafka, интегрировав более одного модуля.

ПРИМЕЧАНИЕ. Обязательно проверьте изменения в файле `rom.xml` в примерах исходного кода для главы 7. В него добавлены дополнительные файлы JAR, которые не были нужны в предыдущих главах. Кроме того, некоторые файлы JAR включены лишь в определенные классификаторы, и они нужны только для тестирования.

7.3.1. Использование Kafka Testcontainers

Если вам понадобится создать, а затем удалить свою инфраструктуру, то вы можете использовать один из вариантов (особенно для интеграционного тестирования) – Testcontainers (<https://www.testcontainers.org/modules/kafka/>). Эта библиотека для Java использует Docker и один из множества фреймворков тестирования JVM, например JUnit. Testcontainers зависит от образов Docker, содержащих действующий кластер. Если ваш рабочий процесс основан на Docker, то вам определенно стоит изучить Testcontainers и настроить кластер Kafka для тестирования.

ПРИМЕЧАНИЕ. Один из соавторов этой книги, Виктор Гамов (Viktor Gamov), поддерживает репозиторий (<https://github.com/gAmUssA/testcontainers-java-module-confluentplatform>) с инструментами для интеграционного тестирования компонентов Confluent Platform (включая Kafka, Schema Registry, ksqlDB).

7.4. Сжатые темы

Теперь, когда у нас есть достаточно полное понимание устройства тем, состоящих из разделов, и разделов, состоящих из сегментов, можно поговорить о деталях сжатия журналов. Целью сжатия является не удаление сообщений, срок хранения которых истек, а обеспечение существования самого последнего значения ключа вместо какого-либо предыдущего состояния. Как только что упоминалось, сжатие зависит от того, является ли ключ частью сообщения и имеет ли непустое значение [10].

Для создания сжатой темы мы использовали параметр конфигурации `cleanup.policy=compact` [15]. Эта настройка отличается от настройки по умолчанию `delete`. Иначе говоря, мы должны явно создать сжатую тему, иначе тема не будет существовать в таком виде. В листинге 7.7 добавлен конфигурационный параметр, необходимый для создания этой новой сжатой темы.

Листинг 7.7. Создание сжатой темы

```
bin/kafka-topics.sh --create --bootstrap-server localhost:9094 \
--topic kinaction_compact --partitions 3 --replication-factor 3 \
--config cleanup.policy=compact
```

Сжатая тема создается так же, как любая другая тема

Указывает, что тема должна быть сжатой

При использовании сжатой темы код будет обновлять существующее поле массива, а не добавлять дополнительные данные.

Давайте посмотрим, как это происходит. Допустим, мы хотим сохранить текущий статус пользователя. В каждый конкретный момент пользователь может иметь только один статус: Basic (простой) или Gold (золотой). Сначала пользователь регистрируется со статусом Basic, но со временем повышает его до Gold и получает дополнительные возможности. Хотя Kafka по-прежнему хранит это событие, но в нашем случае достаточно хранить только самый статус пользователя (наш ключ). На рис. 7.7 показан пример с тремя пользователями.

Сегмент журнала: до сжатия			Сжатая тема		
Смещение	Ключ	Значение	Смещение	Ключ	Значение
0	Customer 0	Basic	2	Customer 0	Gold
1	Customer 1	Gold	3	Customer 2	Basic
2	Customer 0	Gold	100	Customer 1	Basic
3	Customer 2	Basic			
.	.	.			
100	Customer 1	Basic			

Рис. 7.7. Работа сжатой темы в общих чертах

После сжатия для пользователя Customer 0 в теме останется только последнее обновление (в нашем примере) – значение из сообщения со смещением 2 заменит старое значение Basic (в сообщении со смещением 0) на Gold. Пользователь Customer 1 получит текущий статус Basic, потому что значение из последнего сообщения со смещением 100 для этого пользователя заменит прежнее значение Gold в сообщении со смещением 1. Поскольку для пользователя Customer 2 имеется только одно событие, то оно будет перенесено в сжатую тему без каких-либо изменений.

Еще один пример использования сжатых тем – внутренняя тема в Kafka `_consumer_offsets`. Kafka не нуждается в хранении смещений сообщений, уже полученных группой потребителей; ей нужно лишь последнее смещение. Сохраняя смещения в сжатой теме, журнал фактически хранит текущее состояние своего мира.

Когда тема помечена как сжатая, мы можем просмотреть один журнал в двух разных состояниях: сжатом и несжатом. В старых сегментах после завершения сжатия повторяющиеся значения для каждого ключа будут удалены, а активный сегмент продолжит хранить все сообщения без сжатия [11]. Для определенного ключа может существовать несколько значений, пока они не будут очищены. На рис. 7.8 показано, как с помощью указателя можно показать, какие сообщения были обработаны с помощью сжатия, а какие нет [16].

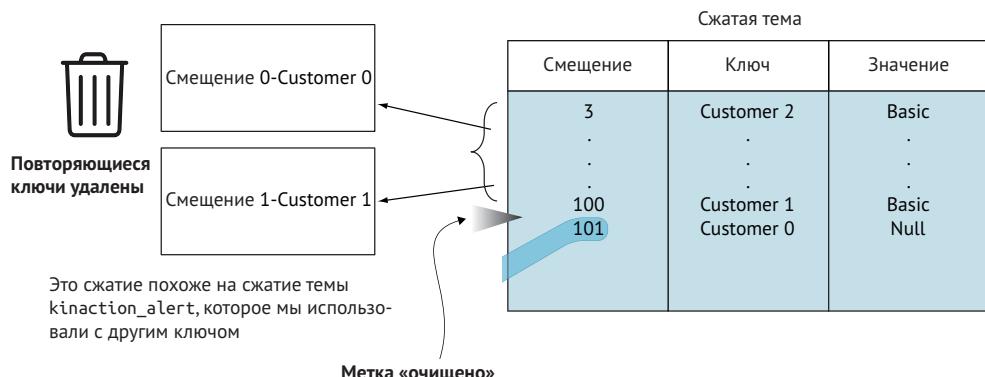


Рис. 7.8. Очистка сжатием

Присмотревшись к смещениям на рис. 7.8, можно заметить отсутствие чисел, соответствующих очищенным смещениям. Поскольку повторяющиеся сообщения удаляются и остаются только последние, мы можем удалить некоторые смещения из файла сегмента. Так, например, было удалено смещение 2. В активных разделях мы, скорее всего, увидим постоянно увеличивающиеся числа смещений, к которым мы привыкли, без случайных пропусков.

Теперь давайте посмотрим, что случится, если пользователь решит удалить свою учетную запись. При отправке события, например, с ключом Customer 0 и пустым значением (null) это событие будет рассматриваться как событие удаления учетной записи. Это сообщение считается «надгробием» [10]. Если вы использовали другие системы, такие как Apache HBase™, то принцип должен быть вам понятен. На рис. 7.9 показано, что пустое значение не удаляет сообщение, а обслуживается как любое другое сообщение [10].

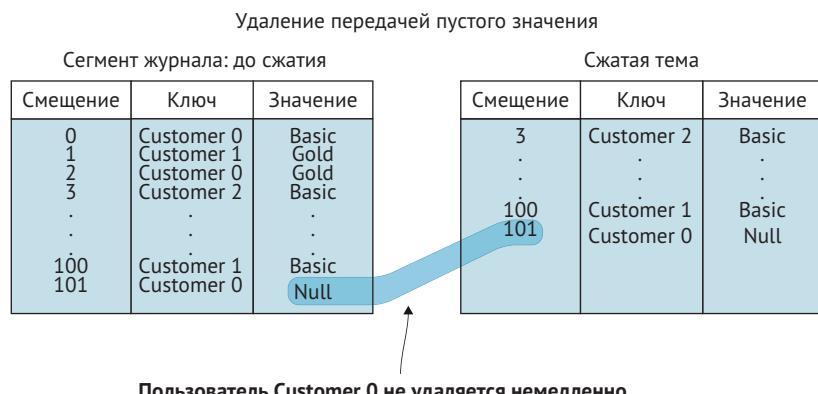


Рис. 7.9. Сжатие при получении сообщения об удалении

Приложение может использовать или не использовать правила, определяющие порядок удаления данных, но в любом случае Kafka готова помочь нам сделать это с помощью своего основного набора функций.

В этой главе мы рассмотрели различные особенности тем, разделов и сегментов. Конечно, они зависят от конкретного брокера, но могут также повлиять и на наших клиентов. Теперь мы знаем и понимаем, как Kafka хранит некоторые свои данные, поэтому в следующей главе мы подробнее обсудим, как мы можем хранить свои данные, в том числе длительное время.

Итоги

- Темы – это логические, а не физические структуры. Чтобы пользоваться темой, потребитель должен знать о количестве разделов и действующих коэффициентах репликации.
- Темы состоят из разделов, что обеспечивает возможности параллельной обработки данных внутри темы.
- Сегменты файла журнала записываются в каталоги разделов и управляются брокером.
- Для тестирования логики разделов можно использовать кластер в памяти.
- Сжатые темы дают возможность представить последнее значение определенного события.

Ссылки

- 1 «Main Concepts and Terminology». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/kafka/introduction.html#main-concepts-and-terminology> (доступно по состоянию на 28 августа 2021).
- 2 J. Rao. «How to choose the number of topics/partitions in a Kafka cluster?» (12 марта 2015). Confluent blog. <https://www.confluent.io/blog/how-choose-number-topics-partitions-kafka-cluster/> (доступно по состоянию на 19 мая 2019).
- 3 «Documentation: Modifying topics». Apache Software Foundation (n.d.). https://kafka.apache.org/documentation/#basic_ops_modify_topic (доступно по состоянию на 19 мая 2018).
- 4 «Documentation: Adding and removing topics». Apache Software Foundation (n.d.). https://kafka.apache.org/documentation/#basic_ops_add_topic (доступно по состоянию на 11 декабря 2019).
- 5 «delete.topic.enable». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/installation/configuration/bro>

- ker-configs.html#broker-configs_delete.topic.enable (доступно по состоянию на 15 января 2021).
- 6 «Topics.java». Apache Kafka GitHub. <https://github.com/apache/kafka/blob/99b9b3e84f4e98c3f07714e1de6a139a004cbc5b/clients/src/main/java/org/apache/kafka/common/internals/Topic.java> (доступно по состоянию на 27 августа 2021).
 - 7 «auto.create.topics.enable». Apache Software Foundation (n.d.). https://docs.confluent.io/platform/current/installation/configuration/broker-configs.html#brokerconfigs_auto.create.topics.enable (доступно по состоянию на 19 декабря 2019).
 - 8 «AdminUtils.scala». Apache Kafka GitHub. <https://github.com/apache/kafka/blob/d9b898b678158626bd2872bbfef-883ca60a41c43/core/src/main/scala/kafka/admin/AdminUtils.scala> (доступно по состоянию на 27 августа 2021).
 - 9 «Documentation: index.interval.bytes». Apache Kafka documentation. https://kafka.apache.org/documentation/#topicconfigs_index.interval.bytes (доступно по состоянию на 27 августа 2021).
 - 10 «Log Compaction». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/kafka/design.html#log-compaction> (доступно по состоянию на 20 августа 2021).
 - 11 «Configuring The Log Cleaner». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/kafka/design.html#configuring-the-log-cleaner> (доступно по состоянию на 27 августа 2021).
 - 12 «CLI Tools for Confluent Platform». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/installation/cli-reference.html> (доступно по состоянию на 25 августа 2021).
 - 13 «EmbeddedKafkaCluster.java». Apache Kafka GitHub. <https://github.com/apache/kafka/blob/9af81955c497b31b211b1e21d8323c875518df39/streams/src/test/java/org/apache/kafka/streams/integration/utils/EmbeddedKafkaCluster.java> (доступно по состоянию на 27 августа 2021).
 - 14 W. P. Bejeck Jr. «Kafka Streams in Action». Shelter Island, NY, USA: Manning, 2018.⁷
 - 15 «cleanup.policy». Confluent documentation (n.d.). https://docs.confluent.io/platform/current/installation/configuration/topic-configs.html#topicconfigs_cleanup.policy (доступно по состоянию на 22 ноября 2020).
 - 16 «Log Compaction Basics». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/kafka/design.html#log-compaction-basics> (доступно по состоянию на 20 августа 2021).

⁷ Беджек Билл, «Kafka Streams в действии», Прогресс книга, 2019, ISBN: 978-5-4461-1201-2. – Прим. перев.

8

Kafka как хранилище

Эта глава охватывает следующие темы:

- как долго можно хранить данные;
- перемещение данных в Kafka и из нее;
- возможности архитектуры данных Kafka;
- хранилище для облачных экземпляров и контейнеров.

До сих пор мы предполагали, что наши данные помещаются в Kafka и хранятся в течение недолгого времени. Еще одно решение, которое необходимо рассмотреть, – когда наши данные должны храниться довольно долго. Используя базы данных, такие как MySQL или MongoDB®, мы не всегда задумываемся об истечении срока хранения этих данных. Обычно мы считаем, что данные будут храниться на протяжении большей части жизненного цикла приложения. Для сравнения: хранилище Kafka логически находится где-то между решениями долгосрочного хранения, которыми являются базы данных, и временными хранилищами брокеров сообщений, особенно если мы считаем, что брокеры хранят сообщения лишь до тех пор, пока они не будут получены клиентом, как это часто бывает в других решениях. Давайте рассмотрим пару вариантов хранения и перемещения данных в нашей среде Kafka.

8.1. Как долго можно хранить данные

В настоящее время данные в темах Kafka по умолчанию хранятся семь дней, однако мы легко можем настроить этот период по времени или размеру данных [1]. Но может ли Kafka хранить данные в течение нескольких лет? Одним из реальных примеров является использование Kafka в *New York Times*. Содержимое их кластера хранится в единственном разделе, объем которого на момент написания книги составлял почти 100 Гбайт [2]. Как отмечалось при обсуждении разделов в главе 7, все данные хранятся на одном диске брокера (а любые реплики – на своих дисках), потому что разделя не могут распределяться между брокерами. Поскольку дисковые хранилища считаются относительно дешевыми, а емкость современных жестких дисков исчисляется сотнями гигабайт, у большинства компаний не возникнет проблем с объемом хранимых данных. Допустимо ли такое использование Kafka, или это злоупотребление ее предполагаемой целью и архитектурой? Если у вас достаточно места на диске для запланированного роста, то Kafka вполне может служить хорошим средством для поддержки вашей конкретной рабочей нагрузки.

Как настроить хранение данных в брокерах? Основными параметрами, доступными для настройки, являются размер журналов и продолжительность существования данных. В табл. 8.1 показаны некоторые конфигурационные параметры брокера, связанные с настройками хранения [3].

Таблица 8.1. Конфигурационные параметры брокера, связанные с настройками хранения

Ключ	Назначение
<code>log.retention.bytes</code>	Максимальный размер журнала в байтах, при превышении которого будет производиться удаление данных
<code>log.retention.ms</code>	Максимальная продолжительность хранения данных в журнале в миллисекундах
<code>log.retention.minutes</code>	Максимальная продолжительность хранения данных в журнале в минутах. Если в конфигурации установлен параметр <code>log.retention.ms</code> , то используется он, а параметр <code>log.retention.minutes</code> игнорируется
<code>log.retention.hours</code>	Максимальная продолжительность хранения данных в журнале в часах. Если в конфигурации установлен параметр <code>log.retention.ms</code> и/или <code>log.retention.minutes</code> , то значение <code>log.retention.hours</code> игнорируется и используется параметр с наибольшей точностью

Как отключить ограничение времени хранения? Для этого достаточно установить параметры `log.retention.bytes` и `log.retention.ms` в значение `-1` [4].

Еще один важный вопрос: можно ли обеспечить аналогичное долговременное хранение последних значений, используя собы-

тия с ключами и сжатую тему? Да, можно. Старые сообщения в сжатых темах по-прежнему будут удаляться при обновлении значений, но самые последние сообщения с ключами всегда будут оставаться в журнале. Это хороший способ хранения данных в случаях, когда не нужны все события (или их история) изменения состояния ключа и достаточно иметь только текущее значение.

А что, если нам понадобится, чтобы наши данные хранились как можно дольше, но у нас просто недостаточно места на дисках наших брокеров? В таком случае можно использовать другой вариант долгосрочного хранения – перенести данные за пределы Kafka, а не хранить их внутри самих брокеров. Перед удалением данных из Kafka их можно сохранить в базе данных, в распределенной файловой системе Hadoop (HDFSTM) или выгрузить в облачное хранилище. Все эти решения являются вполне допустимыми и могут обеспечить более экономичные способы хранения данных после того, как потребители обработают их.

8.2. Перемещение данных

Почти всем компаниям нужно так или иначе преобразовывать получаемые ими данные. Иногда эти преобразования обусловлены внутренними особенностями компании, иногда – необходимостью интеграции с внешним миром. Многие используют популярный термин, существующий в области преобразования данных, – *ETL (extract, transform, load – извлечение, преобразование, загрузка)*. Мы можем использовать некоторые инструменты, чтобы получить данные в их исходном формате, преобразовать эти данные, а затем поместить их в нужную таблицу или хранилище данных. Kafka может играть ключевую роль в этих конвейерах данных.

8.2.1. Сохранение исходных событий

Один из аспектов, который хотелось бы отметить, – это предпочтаемые форматы представления событий внутри Kafka. Несмотря на то что мы открыты для обсуждения требований к вариантам использования, в большинстве случаев предпочтительнее хранить сообщения в исходном формате. Зачем сохранять исходные сообщения, а не форматировать их непосредственно перед помещением в тему? Имея исходное сообщение, проще вернуться назад и начать заново, если в логике преобразования вдруг обнаружится ошибка. Вместо попыток выяснить, как исправить испорченные данные, всегда можно просто вернуться к исходным данным и начать заново. Ярким примером может служить форматирование даты. Иногда приходится сделать несколько попыток, прежде чем будет получен желаемый формат.

Еще один плюс хранения исходных сообщений заключается в том, что данные, которые не используются сегодня, могут понадобиться в будущем. Допустим, сейчас 1995 год, и вы получаете от производителя сообщение с полем `mobile`. А вы уверены, что вашему бизнесу никогда не понадобится это поле? Как только вы обнаружите, что нужно запустить свою первую маркетинговую кампанию с рассылкой на мобильные устройства, вы будете благодарить себя за то, что сохранили эти исходные «бесполезные» данные.

Поле `mobile` может показаться тривиальным примером, но подумайте об использовании сообщений для анализа. Что, если ваши модели начнут замечать тенденции в данных, которые, как считалось прежде, не имеют значения? Сохранив все поля данных, вы сможете вернуться к ним и найти важную и ценную информацию.

8.2.2. Отказ от пакетного мышления

Ведет ли общая тема ETL или конвейеров данных к таким терминам, как *пакет*, *конец дня*, *месяца* или даже *года*? Одним из изменений в процессах преобразования данных прошлого является идея возможности непрерывной передачи данных в различные системы. С помощью Kafka, например, можно поддерживать конвейер, работающий практически в реальном времени, и использовать эту платформу потоковой обработки для обработки данных как бесконечной последовательности событий.

Мы упомянули об этом, чтобы напомнить, что Kafka может помочь кардинально изменить ваше отношение к данным. Вам не нужно ждать, пока ночное задание запустится и обновит базу данных. Также не нужно ждать ночного окна с меньшим трафиком, чтобы выполнить ресурсоемкие задачи ETL; вы можете обрабатывать данные по мере их поступления в систему и поддерживать конвейеры, постоянно обслуживающие ваши приложения в режиме реального времени. Давайте рассмотрим инструменты, которые могут помочь использовать конвейеры в будущем или даже сегодня.

8.3. Инструменты

Перемещение данных является ключевой операцией во многих системах, включая Kafka. Вы можете оставаться в рамках предложений Kafka и Confluent с открытым исходным кодом, таких как Connect, которое обсуждалось в главе 3, но существуют и другие инструменты, соответствующие вашей инфраструктуре или доступные в вашем наборе инструментов. В зависимости от конкретных источников данных или приемников упомянутые в следующих разделах инструменты могут помочь достичь ваших целей.

Обратите внимание, что некоторые инструменты в этом разделе содержат примеры конфигурации и команд, но иногда может потребоваться дополнительная настройка (здесь не показана), чтобы получить возможность запускать эти команды на своих локальных компьютерах. Надеюсь, этот раздел даст вам достаточно информации, чтобы пробудить интерес и позволить вам начать изучение самостоятельно.

8.3.1. Apache Flume

Если вы познакомились с Kafka благодаря работе в области больших данных, то велика вероятность, что вы могли использовать Flume. Если вы когда-либо слышали термин *Flafka*, то определенно использовали эту комбинацию Kafka и Flume. Flume может обеспечить более простой способ передачи данных в кластер и в большей степени зависит от конфигурации, чем от пользовательского кода. Например, если вам нужно организовать прием данных в свой кластер Hadoop и вы уже заручились поддержкой поставщика по этим различным компонентам, то Flume станет для вас надежным вариантом приема данных в кластер Kafka.

На рис. 8.1 показан пример, где агент Flume работает на узле как отдельный процесс. Он следит за локальными файлами на этом сервере и использует предоставленную вами конфигурацию для отправки данных в приемник.

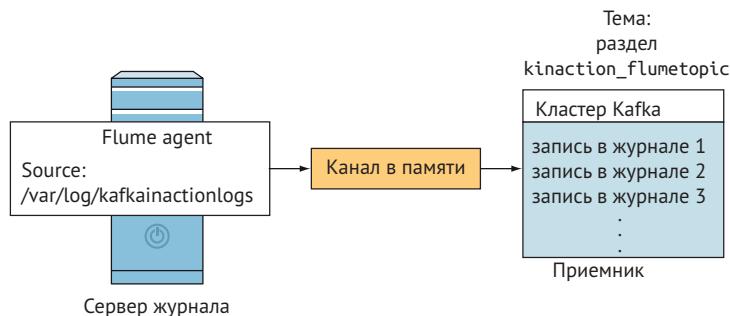


Рис. 8.1. Агент Flume

Давайте еще раз взглянем на интеграцию файлов журнала (нашего источника данных) в тему Kafka (наш приемник данных) с помощью агента Flume. В листинге 8.1 показан пример конфигурационного файла, который можно использовать для настройки локального агента Flume, следящего за изменениями в каталоге [5]. Изменения помещаются в тему Kafka с именем `kinaction_flumetopic`. Этот пример похож на применение команды `cat` к файлу в каталоге, чтобы прочитать его содержимое и отправить в определенную тему Kafka.

Листинг 8.1. Конфигурация агента Flume, наблюдающего за изменениями в каталоге

```
ag.sources = logdir
ag.sinks = kafkasink
ag.channels = c1

#Определить исходный каталог для наблюдения
ag.sources.logdir.type = spooldir
ag.sources.logdir.spoolDir = /var/log/kafka/inactionlogs
...
ag.sinks.kafkasink.channel = c1
ag.sinks.kafkasink.type = org.apache.flume.sink.kafka.KafkaSink
ag.sinks.kafkasink.kafka.topic = kinaction_flumetopic
...
# Связать источник и приемник каналом
ag.sources.logdir.channels = c1
ag.sinks.kafkasink.channel = c1
```

← Определяет имена источника, приемника и канала

Источник spooldir позволяет агенту Flume узнать, за каким каталогом он должен наблюдать

← Здесь определяются тема и информация о кластере Kafka, куда должны перемещаться данные

← Связывает источник с приемником с помощью указанного канала

В листинге 8.1 показано, как можно настроить агента Flume, работающего на сервере. Обратите внимание, что в конфигурации приемника применяются знакомые свойства, которые мы использовали ранее в коде производителя на Java.

Также интересно отметить, что Flume может использовать Kafka не только как источник или как приемник, но и как канал. Поскольку Kafka считается более надежным каналом для событий, Flume может использовать Kafka для доставки сообщений между различными источниками и приемниками.

Если при просмотре конфигурации Flume вы увидите упоминание Kafka, то обязательно обратите внимание, где и как она используется. В листинге 8.2 показана конфигурация агента Flume, которую можно задействовать для организации надежного канала между различными источниками и приемниками, которые поддерживаются Flume [5].

Листинг 8.2. Настройка канала Kafka в конфигурации Flume

```
ag.channels.channel1.type = KafkaChannel
ag.channels.channel1.kafka.bootstrap.servers = localhost:9092,localhost:9093,localhost:9094
ag.channels.channel1.kafka.topic = kinaction_channel1_ch
ag.channels.channel1.kafka.consumer.group.id = kinaction_flume
```

Определение группы потребителей, чтобы избежать конфликтов с другими потребителями

Flume использует класс KafkaChanne в качестве типа канала Kafka

Список серверов для подключения

Тема, в которой хранятся данные, перемещаемые между источником и приемником

8.3.2. Red Hat® Debezium™

На сайте проекта Debezium (<https://debezium.io>) этот инструмент позиционируется как распределенная платформа, помогающая превращать базы данных в потоки событий. Другими словами, обновления нашей базы данных можно рассматривать как события! Если у вас есть опыт работы с базами данных, то, возможно, вы слышали термин *сбор изменений в данных* (Change Data Capture, CDC). Как следует из названия, можно выявлять изменения в данных и реагировать на эти изменения. На момент написания этой главы инструмент Debezium поддерживал MySQL, MongoDB, PostgreSQL®, Microsoft SQL Server™, Oracle и IBM Db2. Готовилась к выпуску также поддержка Cassandra™ и Vitess™ [6]. Полный список доступных коннекторов можно найти по адресу <https://debezium.io/documentation/reference/connectors/>.

Debezium использует коннекторы и Kafka Connect для записи событий, которые наше приложение может получать из Kafka как обычно. На рис. 8.2 показан пример использования Debezium в качестве коннектора, взаимодействующего с Kafka Connect.

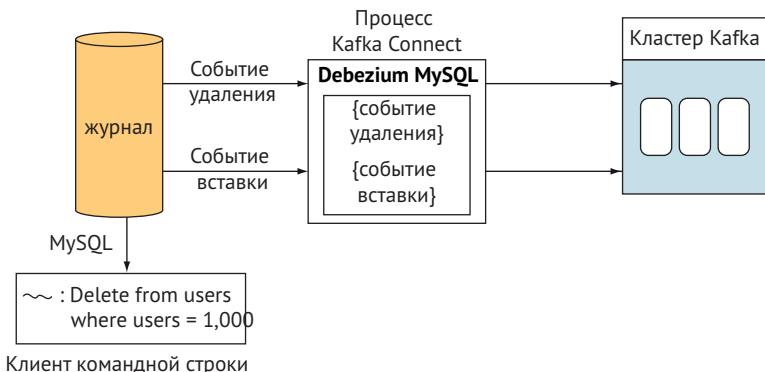


Рис. 8.2. Использование Kafka Connect и Debezium с базой данных MySQL

В этом сценарии разработчик применяет интерфейс командной строки и удаляет пользователя из экземпляра базы данных MySQL, за которым ведется наблюдение. Debezium фиксирует событие, записываемое во внутренний журнал базы данных, и через службу коннектора передает его в Kafka. С появлением нового события, например добавления нового пользователя в базу данных, это событие также фиксируется и передается в Kafka.

Отметим также, что существуют другие варианты использования таких методов, как CDC, для своевременной передачи событий или изменений в данных, не являющиеся характерными для Kafka, которые могут помочь провести параллель с тем, к чему стремится Debezium в целом.

8.3.3. Secor

Secor (<https://github.com/pinterest/secor>) – интересный проект от Pinterest, который был основан в 2014 году. Его цель – помочь сохранить данные из журнала Kafka в различных хранилищах, включая S3 и Google Cloud Storage™ [7]. Данные могут выводиться в самых разных форматах, включая последовательности, файлы Apache ORC™, Apache Parquet™, и многих других. Как всегда, одним из основных преимуществ проектов с исходным кодом в общедоступном репозитории является возможность увидеть, как другие команды реализуют требования, которые могут быть похожи на наши.

На рис. 8.3 показан пример, где Secor действует как потребитель кластера Kafka, подобно любому другому приложению. Добавление потребителя в кластер для резервного копирования данных не имеет большого значения. Он использует способ, которым Kafka всегда обрабатывал несколько потребителей событий.

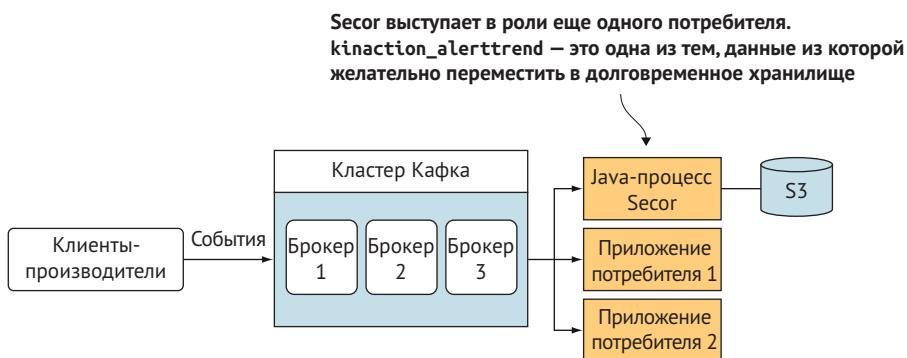


Рис. 8.3. Secor действует как потребитель и помещает данные в хранилище

Secor работает как Java-процесс и может настраиваться с помощью конкретных конфигураций. По сути, он действует как еще один потребитель существующих тем, выбирая из них данные и передавая в указанное место назначения, например в корзину S3. Secor не мешает другим потребителям и позволяет получить копии событий, чтобы не потерять их после того, как механизм сжатия в Kafka удалит данные из своих журналов.

Порядок вызова Secor должен быть знаком тем, кто привык работать с файлами JAR. Приложению Secor, например, можно передавать аргументы в стандартных параметрах -D. В этом случае наиболее важным файлом становится файл свойств с параметрами конфигурации. Этот файл позволяет определить информацию, например, о нашей конкретной корзине в облачном хранилище.

8.3.4. Пример сохранения данных

Рассмотрим пример перемещения данных из Kafka во внешнее хранилище для возможного последующего использования. Во-первых, разделим использование одних и тех же данных между двумя разными областями. Одна область – обработка операционных данных по мере их поступления в Kafka.

Операционные данные – это события, возникающие в результате повседневных операций. В качестве примера можно представить событие заказа товара на веб-сайте. Событие заказа запускает наше приложение с малой задержкой. Учитывая ценность этих данных, было бы желательно гарантировать их сохранность в течение нескольких дней, пока заказ не будет выполнен и отправлен по почте. После этого событие теряет операционную ценность, приобретает ценность для аналитических систем.

Аналитические данные, хотя и основаны на тех же операционных данных, используются в основном для принятия бизнес-решений. В традиционных системах именно здесь на первое место выходят такие процессы, как хранилища данных, системы оперативной аналитической обработки (Online Analytical Processing, OLAP) и Hadoop. Эти данные о событиях можно анализировать, используя различные комбинации полей в событиях, чтобы, например, получить представление о продажах. Если мы заметим, что продажи чистящих средств всегда резко увеличиваются перед праздниками, то мы сможем использовать эту информацию, чтобы увеличить продажи наших товаров в будущем.

8.4. Возврат данных в Kafka

Важно отметить, что если данные покинули Kafka, то это не означает, что их нельзя вернуть обратно. На рис. 8.4 показан пример, когда данные, закончив свой обычный жизненный цикл в Kafka, были заархивированы в облачном хранилище, таком как S3. Если новое изменение логики приложения потребует повторной обработки старых данных, то нет необходимости создавать клиента для чтения как из S3, так и из Kafka, потому что с помощью такого инструмента, как Kafka Connect, данные можно загрузить из S3 обратно в Kafka! С точки зрения приложений интерфейс остается прежним. На первый взгляд неочевидно, зачем это нужно, но давайте рассмотрим ситуацию, когда по истечении некоторого времени мы посчитали необходимым переместить наши данные обратно в Kafka.

Представьте себе команду, пытающуюся найти закономерности в данных, собиравшихся в течение многих лет. Это могут быть телеработы данных. После обработки клиентами-потребителями эти

данные были перемещены для хранения из Kafka в HDFS. Должно ли теперь приложение извлекать данные из HDFS? Почему бы просто не вернуть их обратно в Kafka, чтобы приложение могло обработать данные так же, как раньше? Возврат данных в Kafka – это допустимый способ повторной обработки данных. На рис. 8.4 показан еще один пример, как можно вернуть данные обратно в Kafka.

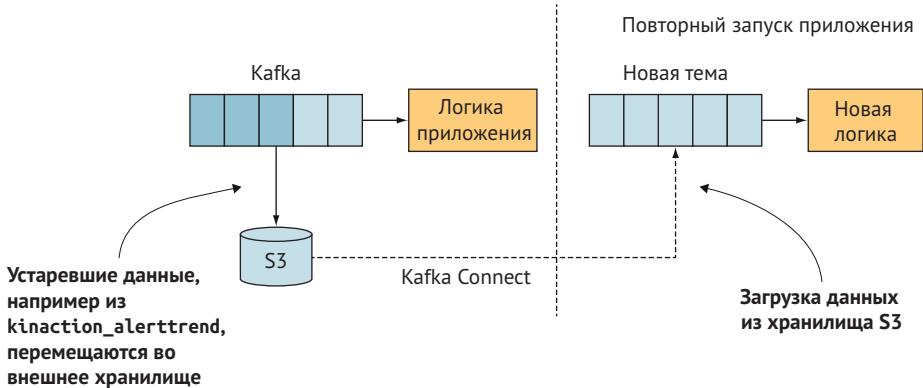


Рис. 8.4. Перемещение данных обратно в Kafka

Через некоторое время события становятся недоступными для приложений из-за ограничений, заданных настройками хранения данных в Kafka. Однако есть копия всех предыдущих событий в корзине S3. Допустим, что мы создали новую версию приложения и хотели бы получить все события, которые в прошлом получило предыдущее приложение. Однако, поскольку этих событий уже нет в Kafka, их придется извлекать из S3, или нет? Нужно ли нам предусматривать в логике приложения возможность извлечения данных из разных источников? Или можно обойтись единственным интерфейсом (с Kafka)? Мы можем создать новую тему в существующем кластере Kafka и загрузить в нее данные из S3 с помощью Kafka Connect. После этого новое приложение сможет работать с Kafka, обрабатывая события без изменения логики извлечения событий.

Идея заключается в том, чтобы сохранить Kafka в качестве интерфейса нашего приложения и не предусматривать множество способов извлечения данных для обработки. Зачем создавать и поддерживать код для извлечения данных из разных мест, если можно использовать существующий инструмент, такой как Connect, и с его помощью перемещать данные в Kafka или из нее? Имея данные в этом интерфейсе, мы можем извлекать их так же, как прежде.

ПРИМЕЧАНИЕ. Имейте в виду, что этот метод применим только к данным, которые были удалены из Kafka. Если данные продолжают храниться в Kafka, вы всегда можете обратиться к более ранним смещениям.

8.4.1. Многоуровневое хранилище

Более современный вариант, поддерживаемый начиная с версии Confluent Platform 6.0.0, называется многоуровневым хранилищем (Tiered Storage). В этой модели локальное хранилище по-прежнему управляет самим брокером, а для старых данных, управление которыми контролируется конфигурацией `confluent.tier.local.hot-set.ms`, вводится удаленное хранилище [8].

8.5. Архитектуры с использованием Kafka

Существует множество различных архитектурных шаблонов, рассматривающих данные как события, таких как модель-представление-контроллер (Model-View-Controller, MVC), одноранговая сеть (peer-to-peer, P2P) или сервис-ориентированная архитектура (Service-Oriented Architecture, SOA). Kafka может изменить ваши представления об архитектуре. Давайте взглянем на пару архитектур, которые могут работать на Kafka (и на других потоковых платформах). Это поможет вам по-новому взглянуть на то, как можно проектировать системы для наших клиентов.

В некоторых обсуждениях используется термин *большие данные* (big data). Поэтому важно отметить, что большой объем данных и необходимость своевременной их обработки были движущими силами, которые привели к появлению некоторых из этих систем. Однако эти архитектуры не ограничиваются только приложениями для быстрой обработки данных или обработки больших данных. Преодолев ограничения конкретных традиционных технологий баз данных, появились новые взгляды на данные. Давайте рассмотрим два из них.

8.5.1. Лямбда-архитектура

Если вам приходилось видеть или использовать приложения для работы с данными, поддерживающие как пакетную, так и оперативную обработку, то вы могли встречать ссылки на лямбда-архитектуру. Реализация этой архитектуры может также начаться с Kafka, но она немного сложнее.

Взгляд на обработку данных в режиме реального времени сочетается с историческим взглядом на обслуживание конечных пользователей. Не следует игнорировать сложность слияния этих двух взглядов на данные. В свое время авторы столкнулись с проблемой перестройки службы таблиц. Кроме того, вам, вероятно, придется поддерживать разные интерфейсы для ваших данных при работе с результатами из обеих систем.

В книге «Big Data»⁸, написанной Натаном Марцем (Nathan Marz) совместно с Джеймсом Уорреном (James Warren), лямбда-архитектура обсуждается более подробно и рассказывается об уровнях пакетной обработки, обслуживания и скорости [9]. На рис. 8.5 показан пример, как можно рассматривать прием заказов от клиентов в пакетном режиме и в режиме реального времени. Итоговые данные за предыдущие дни могут объединяться с заказами, выполненными в течение дня, в комбинированное представление данных для передачи конечным пользователям.

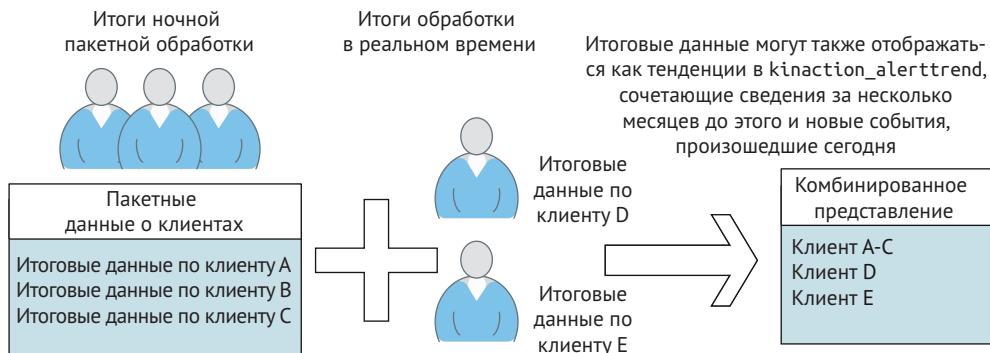


Рис. 8.5. Лямбда-архитектура

Опираясь на представление архитектуры, показанное на рис. 8.5, рассмотрим в общих чертах каждый ее уровень. В книге Марца «Big Data» обсуждаются следующие уровни:

- *пакетная обработка* – этот уровень осуществляет пакетную обработку по аналогии с MapReduce в таких системах, как Hadoop. По мере добавления новых данных хранилища уровень пакетной обработки продолжает обновлять представление данных, которые уже имеются в системе;
- *ускорение* – концептуально этот уровень похож на уровень пакетной обработки, но создает представления из последних данных;
- *обслуживание* – этот уровень обновляет представления, отправляемые потребителям, после каждого обновления пакетных представлений.

С точки зрения конечного пользователя, лямбда-архитектура объединяет данные из уровней обслуживания и ускорения, чтобы возвращать в ответ на запросы максимально полное пред-

⁸ Натан Марц и Джеймс Уоррен, «Большие данные. Принципы и практика построения масштабируемых систем обработки данных в реальном времени», Вильямс, 2017, ISBN: 978-5-8459-2075-1, 978-1-617-29034-3. – Прим. перев.

ставление, составленное из последних и прошлых данных. Этот уровень потоковой передачи в реальном времени является наиболее очевидным местом для внедрения Kafka, но ее также можно использовать для передачи данных на уровень пакетной обработки.

8.5.2. Каппа-архитектура

Еще один архитектурный шаблон, который может выиграть от использования Kafka, – это каппа-архитектура. Этот шаблон был предложен соавтором Kafka – Джейм Крепсом (Jay Kreps) [10]. Представьте, что вам нужно обеспечить бесперебойное обслуживание пользователей. Один из способов добиться этого – сначала создать обновленные представления, а затем переключиться на них, как в лямбда-архитектуре. Другой способ – продолжать поддерживать текущую систему параллельно с новой и отключить ее, как только новая версия будет готова обслуживать весь трафик. Частью этого перехода, конечно же, является обеспечение в новой версии правильного отражения данных, обслуживаемых старой версией.

Вы регенерируете данные для конечных пользователей только тогда, когда это нужно. Нет необходимости в объединении старых и новых данных, которое выполняется постоянно в некоторых реализациях лямбда-архитектуры. Объединение не обязательно должно производиться непрерывно и может происходить, только когда нужно изменить логику приложения. Кроме того, нет необходимости менять интерфейс доступа к данным. Kafka может использоваться как новой, так и старой версией приложения одновременно. На рис. 8.6 показано, как события клиентов используются для создания представления без применения уровня пакетной обработки.

На рис. 8.6 показаны события клиентов из прошлого и настоящего, используемые непосредственно для создания представления. Вообразите, что события поступают из Kafka, а затем с помощью Kafka Streams или ksqlDB извлекаются практически в реальном времени для создания представления для конечных пользователей. Если когда-либо потребуется изменить способ обработки событий клиентов, можно создать второе приложение с другой логикой (например, новый запрос ksqlDB), используя тот же источник данных (Kafka), что и раньше. В этом случае отпадает необходимость в слое пакетной обработке (и управлении им), потому что для создания конечных представлений используется только потоковая логика.

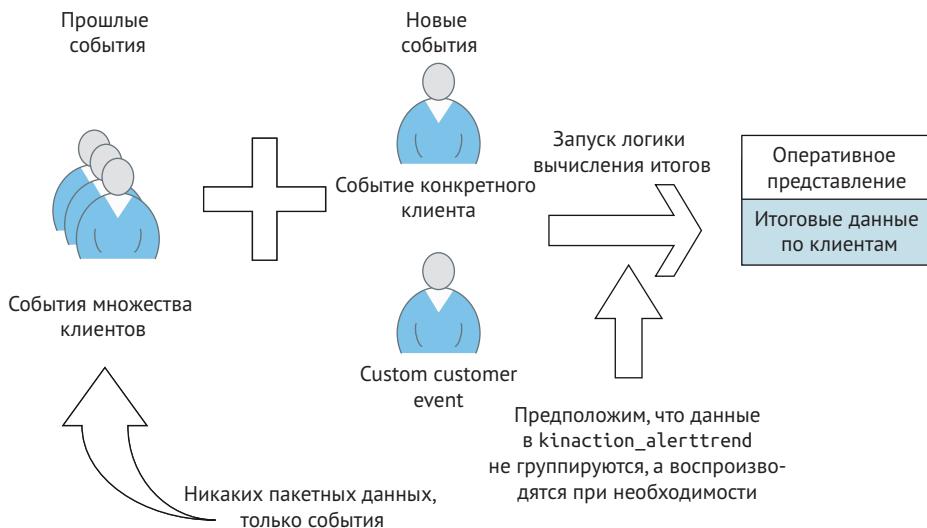


Рис. 8.6. Каппа-архитектура

8.6. Окружения с несколькими кластерами

До сих пор все наши дискуссии велись с точки зрения обработки данных в одном кластере. Kafka хорошо масштабируется и вполне может охватить сотни брокеров, действующих в одном кластере. Однако кластеры гигантского размера подходят не для всех инфраструктур. Одна из проблем, с которой мы сталкиваемся, говоря о кластерном хранилище, заключается в том, где находятся данные по отношению к конечным пользователям. В этом разделе мы поговорим о масштабировании за счет добавления кластеров, а не только брокеров.

8.6.1. Масштабирование путем добавления кластеров

Обычно в первую очередь масштабируются ресурсы внутри существующего кластера. Количество брокеров – это первый вариант, открывающий прямой путь к росту. МультиклUSTERная стратегия Netflix® – это захватывающий пример масштабирования Kafka путем увеличения числа кластеров [11]. В этой компании обнаружили, что масштабировать можно не только количество брокеров, но и сами кластеры!

Этот подход напоминает идею разделения ответственности команд и запросов (Command Query Responsibility Segregation, CQRS). За более подробной информацией о CQRS обращайтесь к статье Мартина Фаулера (Martin Fowler), доступной по адресу <https://martinfowler.com/bliki/CQRS.html>, где подробно обсуждается идея разделения нагрузки чтения и записи данных [12]. Каждое действие может масштабироваться независимо, не ограничивая другие действия. CQRS – это шаблон, который может усложнить ваши си-

стемы, однако интересно отметить, как этот конкретный пример помогает управлять производительностью большого кластера, отделяя нагрузку производителей, отправляющих данные в Kafka, от гораздо большей нагрузки потребителей, читающих данные.

8.7. Варианты хранения в облаке и в контейнерах

Мы уже говорили о каталогах журналов Kafka в главе 6, но не рассматривали типы экземпляров, которые можно использовать в окружениях, обеспечивающих краткосрочное хранение. Для справки: проект Confluent опубликовал результаты исследований развертываний в AWS, где они рассмотрели достоинства и недостатки хранилищ разных типов [13].

Другой вариант – Confluent Cloud (<https://www.confluent.io/confluent-cloud/>). Этот вариант позволяет меньше беспокоиться о базовом хранилище, используемом облачными провайдерами, и о том, как оно управляется. Помните, что Kafka тоже продолжает развиваться и реагировать на потребности, с которыми сталкиваются пользователи. В KIP-392 показано решение, принятое к моменту написания этой книги и призванное помочь в решении проблем с кластерами Kafka, охватывающими несколько центров обработки данных. Это предложение по улучшению Kafka (Kafka Improvement Proposals, KIP) озаглавлено «Allow consumers to fetch from the closest replica» (Поддержка получения данных потребителями из ближайшей реплики) [14]. Обязательно проверяйте время от времени последние KIP, чтобы быть в курсе развития Kafka.

8.7.1. Кластеры Kubernetes

При использовании контейнерного окружения можно столкнуться с проблемами, характерными для облака. При неправильной настройке лимита памяти в нашем брокере мы можем оказаться на совершенно новом узле без наших данных, если только данные не сохраняются правильно. При использовании изолированных окружений, которые могут терять данные, нашим брокерам могут потребоваться постоянные хранилища, чтобы гарантировать, что данные сохранятся при любых перезапусках, сбоях или перемещениях. Контейнер экземпляра брокера может измениться, и мы должны иметь возможность получить предыдущий том хранилища.

Приложения Kafka, вероятнее всего, будут использовать StatefulSet API для поддержки идентичности каждого брокера при сбоях или перемещениях подов. Эта статическая идентификация также помогает запрашивать те же тома хранилищ, которые использовались до выхода пода из строя. Уже существуют диаграммы Helm® (<https://github.com/confluentinc/cp-helm-charts>), которые

могут помочь запустить тестовое окружение при изучении Kubernetes [15]. Также в управлении Kubernetes может помочь Confluent for Kubernetes [16].

Тема Kubernetes настолько обширна, что не представляется возможным охватить ее в нашем обсуждении, но основные проблемы не зависят от окружения. Наши брокеры имеют идентичность в кластере и привязаны к своим данным. Чтобы кластер оставался работоспособным, брокерам нужна возможность идентифицировать управляемые ими журналы при сбоях, перезапусках или обновлениях.

Итоги

- Особенности хранения данных должны определяться потребностями бизнеса. При выборе решения следует взвесить стоимость хранения и темпы роста объемов данных с течением времени.
- Размер и время являются основными параметрами, определяющими, как долго данные хранятся на диске.
- Долгосрочное хранение данных за пределами Kafka – это вполне приемлемый вариант для данных, которые, возможно, потребуется хранить в течение долгого времени. Данные могут повторно вводиться по мере необходимости путем их передачи в кластер позднее.
- Способность Kafka быстро обрабатывать данные, а также воспроизводить их позволяет использовать такие архитектуры, как лямбда и каппа.
- Облачные и контейнерные рабочие нагрузки часто включают недолговечные экземпляры брокеров. Для данных, которые необходимо охранить, требуется план, гарантирующий возможность использования этих данных вновь созданными или восстановленными экземплярами.

Ссылки

- 1 «Kafka Broker Configurations». Confluent documentation (n.d.). https://docs.confluent.io/platform/current/installation/configuration/broker-configs.html#brokerconfigs_log.retention.hours (доступно по состоянию на 14 декабря 2020).
- 2 B. Svingen. «Publishing with Apache Kafka at The New York Times». Confluent blog (September 6, 2017). <https://www.confluent.io/blog/publishing-apache-kafka-new-york-times/> (доступно по состоянию на 25 сентября 2018).

- 3 «Kafka Broker Configurations». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/installation/configuration/broker-configs.html> (доступно по состоянию на 14 декабря 2020).
- 4 «Kafka Broker Configurations: log.retention.ms». Confluent documentation (n.d.). https://docs.confluent.io/platform/current/installation/configuration/broker-configs.html#brokerconfigs_log.retention.ms (доступно по состоянию на 14 декабря 2020).
- 5 «Flume 1.9.0 User Guide: Kafka Sink». Apache Software Foundation (n.d.). <https://flume.apache.org/releases/content/1.9.0/FlumeUser-Guide.html#kafka-sink> (доступно по состоянию на 10 октября 2019).
- 6 «Connectors». Debezium documentation (n.d.). <https://debezium.io/documentation/reference/connectors/> (доступно по состоянию на 20 июля 2021).
- 7 «Pinterest Secor». Pinterest. GitHub. <https://github.com/pinterest/secor/blob/master/README.md> (доступно по состоянию на 1 июня 2020).
- 8 «Tiered Storage». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/kafka/tiered-storage.html> (доступно по состоянию на 2 июня 2021).
- 9 N. Marz and J. Warren. «Big Data: Principles and best practices of scalable real-time data systems». Shelter Island, NY, USA: Manning, 2015.⁹
- 10 J. Kreps. «Questioning the Lambda Architecture». O'Reilly Radar (2 июля 2014). <https://www.oreilly.com/radar/questioning-the-lambda-architecture/> (доступно по состоянию на 11 октября 2019).
- 11 A. Wang. «Multi-Tenant, Multi-Cluster and Hierarchical Kafka Messaging Service». Презентация на конференции Kafka Summit, San Francisco, USA, 2017 Presentation [online]. <https://www.confluent.io/kafka-summit-sf17/multitenant-multicluster-and-hieracrchical-kafka-messaging-service/>.
- 12 M. Fowler. «CQRS» (14 июля 2011). <https://martinfowler.com/bliki/CQRS.html> (доступно по состоянию на 11 декабря 2017).
- 13 A. Loddengaard. «Design and Deployment Considerations for Deploying Apache Kafka on AWS». Confluent blog (28 июля 2016). <https://www.confluent.io/blog/design-and-deployment-considerations-for-deploying-apache-kafka-on-aws/> (доступно по состоянию на 11 июня 2021).
- 14 KIP-392: «Allow consumers to fetch from closest replica». Wiki for Apache Kafka. Apache Software Foundation (05 ноября 2019). <https://cwiki.apache.org/confluence/display/KAFKA/KIP-392%3A+>

⁹ Натан Марц и Джеймс Уоррен, «Большие данные. Принципы и практика построения масштабируемых систем обработки данных в реальном времени», Вильямс, 2017, ISBN: 978-5-8459-2075-1, 978-1-617-29034-3. – Прим. перев.

[Allow+consumers+to+fetch+from+closest+replica](#) (доступно по состоянию на 10 декабря 2019).

- [15 cp-helm-charts](#). Confluent Inc. GitHub (n.d.). <https://github.com/confluentinc/cp-helm-charts> (доступно по состоянию на 10 июня 2020).
- [16 «Confluent for Kubernetes»](#). Confluent documentation (n.d.). <https://docs.confluent.io/operator/2.0.2/overview.html> (доступно по состоянию на 16 августа 2021).

Управление: инструменты и журналы

Эта глава охватывает следующие темы:

- клиентов администрирования;
- знакомство с REST API, инструментами и утилитами;
- управление журналами Kafka и ZooKeeper;
- поиск метрик JMX;
- публикуемые слушатели и клиенты;
- трассировку с использованием перехватчиков с заголовками.

Выше в книге мы более или менее подробно обсудили проблемы брокеров и клиентов и рассмотрели некоторые приемы разработки, подходящие для большинства ситуаций. Но жизнь непредсказуема, и всегда есть вероятность столкнуться с ситуациями, требующими особого подхода. Лучший способ поддерживать кластер в рабочем состоянии – понимать, какие данные передаются через него, и следить за их обработкой. Работа с Apache Kafka может отличаться от разработки и эксплуатации Java-приложений как таких, но она также требует мониторинга файлов журналов и знания происходящего в наших рабочих нагрузках.

9.1. Клиенты администрирования

До сих пор большую часть действий по управлению кластером мы выполняли с помощью инструментов командной строки, поставляемых вместе с Kafka. В конце концов, мы должны владеть приемами работы в командной строке, чтобы установить и настроить Kafka. Однако есть более удобные решения, которые можно использовать, чтобы не ограничивать себя применением только предоставляемых инструментов.

9.1.1. Решение задач администрирования в коде с помощью AdminClient

Одним из полезных инструментов является класс `AdminClient` [1]. Конечно, удобно иметь под рукой сценарии командной оболочки для управления Kafka для быстрого выполнения разовых действий, однако в некоторых ситуациях, например для реализации автоматизации, можно с успехом использовать Java-класс `AdminClient`. Определение этого класса находится в файле `kafka-clients.jar`, который мы использовали при реализации клиентов производителей и потребителей. Его можно включить в проект Maven (см. `pom.xml` из главы 2) или найти в подкаталоге `share/` или `libs/` в каталоге установки Kafka.

Давайте посмотрим, как можно выполнить команду, которую мы использовали для создания новой темы, но на этот раз с помощью `AdminClient`. В листинге 9.1 показано, как мы запускали эту команду из командной строки в главе 2.

Листинг 9.1. Создание темы `kinaction_selfserviceTopic` из командной строки

```
bin/kafka-topics.sh
--create --topic kinaction_selfserviceTopic \
--bootstrap-server localhost:9094 \
--partitions 2 \
--replication-factor 2
```

Для создания новой темы используется сценарий `kafka-topic.sh`

Здесь задаются иные значения, определяющие количество разделов и реплик для темы

Этот способ создания темы в командной строке работает normally, но нам не хотелось бы прибегать к нему каждый раз, когда кому-то понадобится новая тема. Вместо этого можно создать портал самообслуживания, который другие разработчики смогут использовать для создания новых тем в нашем кластере для разработки. Форма приложения будет принимать название темы и количество разделов и реплик. На рис. 9.1 показан пример по-

доброго приложения для конечных пользователей. Пользователь открывает приложение, заполняет и отправляет веб-форму, после чего запускается код Java-класса `AdminClient` и создает новую тему.

В этом примере мы могли бы добавить логику, проверяющую соответствие соглашениям об именовании новых тем (если бы у нас было такое требование). Это способ сохранить больший контроль над нашим кластером, а не над пользователями, работающими с инструментами командной строки. Для начала нужно создать класс `NewTopic`. Конструктор этого класса принимает три аргумента:

- название темы;
- количество разделов;
- количество реплик.

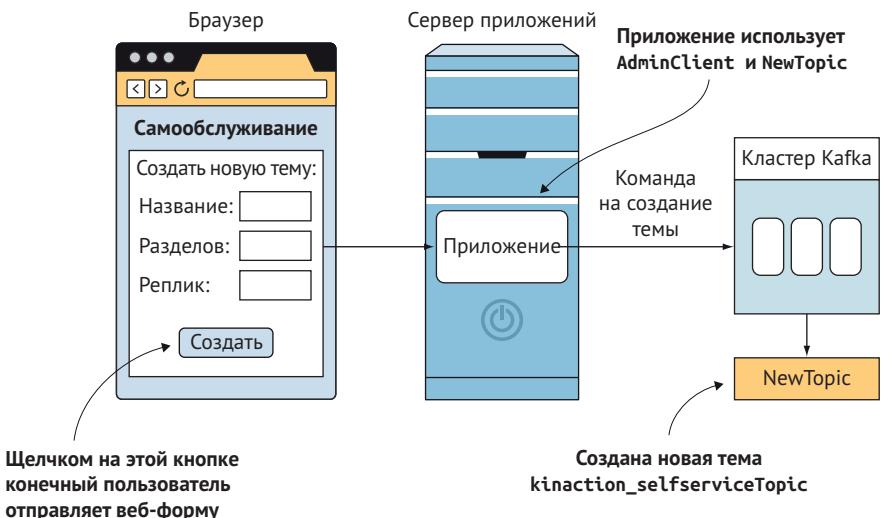


Рис. 9.1. Веб-приложение для самообслуживания

После этого можно использовать объект `AdminClient`, чтобы завершить начатое. `AdminClient` принимает объект `Properties` с теми же свойствами, которые мы использовали с другими клиентами, например `bootstrap.servers` и `client.id`. Обратите внимание, что константы для значений конфигурации, таких как `BOOTSTRAP_SERVERS_CONFIG`, можно найти во вспомогательном классе `AdminClientConfig` (<http://mng.bz/8065>). Затем нужно вызвать метод `createTopics` на стороне клиента. Обратите внимание, что возвращаемый результат `topicResult` является объектом `Future`. В листинге 9.2 показано, как создать новую тему `kinaction_selfserviceTopic` с помощью класса `AdminClient`.

Листинг 9.2. Использование AdminClient для создания новой темы

```

NewTopic requestedTopic =
    new NewTopic("kinaction_selfserviceTopic", 2,(short) 2); ←

AdminClient client = ←
    AdminClient.create(kaProperties); ← Создать AdminClient, клиент-
CreateTopicsResult topicResult = ←
client.createTopics( ←
    List.of(requestedTopic)); ← Вызвать createTopics на стороне клиента,
topicResult.values(). ←
    get("kinaction_selfserviceTopic").get(); ← чтобы получить объект Future
    ← Страна показывает, как
    ← получить конкретное
    ← значение объекта Future
    ← для темы kinaction_
    ← selfserviceTopic

```

В настоящее время не существует синхронного API, но мы можем сделать синхронный вызов с помощью функции `get()`. В нашем случае это означает попытку, начав с переменной `topicResult`, получить значение объекта `Future` для конкретной темы.

Этот API еще продолжает развиваться, поэтому в следующем списке административных задач, которые можно выполнить с помощью `AdminClient`, перечислены лишь те, что были доступны на момент написания книги [1]:

- изменение конфигурации;
- создание/удаление/перечисление списков управления доступом (ACL);
- создание разделов;
- создание/удаление/перечисление тем;
- получение описания/перечисление групп потребителей;
- получение описания кластеров.

`AdminClient` – отличный инструмент создания приложений для пользователей, не желающих или не имеющих возможности использовать сценарии командной оболочки Kafka. Он также дает возможность контроля и мониторинга происходящего в кластере.

9.1.2. `kcat`

`kcat` (<https://github.com/edenhill/kcat>) – это удобный инструмент для рабочей станции, особенно при удаленном подключении к кластерам. В настоящее время его главное предназначение – играть роль производителя и потребителя, но он также может предоставлять метаданные о кластере. Если вам понадобится выполнить пару операций с темой, не загружая весь набор инструментов Kaf-

ка на текущий компьютер, то этот выполняемый файл поможет вам обойтись без них.

В листинге 9.3 показано, как быстро передать данные в тему с помощью `kcat` [2]. Сравните этот пример с примером использования сценария `kafka-console-producer` в главе 2.

Листинг 9.3. Использование `kcat` в роли производителя

```
kcat -P -b localhost:9094 \
-t kinaction_selfserviceTopic
```

Передать адрес брокера и имя темы из нашего кластера для записи сообщений в эту тему


```
// для сравнения та же операция с помощью сценария kafka-console-producer
bin/kafka-console-producer.sh --bootstrap-server localhost:9094 \
--topic kinaction_selfserviceTopic
```

Для сравнения та же операция с помощью сценария командной строки

Обратите внимание, что в листинге 9.3 флаг `-P` передается программе `kcat` для включения режима производителя, позволяющий отправлять сообщения в кластер. С помощью флага `-b` передается список брокеров, и с помощью флага `-t` – имя целевой темы. Поскольку нам также может понадобиться протестировать потребление сообщений, давайте посмотрим, как использовать `kcat` в роли потребителя (листинг 9.4). Здесь так же, как в листинге 9.3, для сравнения показано выполнение той же операции с помощью сценария `kafka-console-consumer`. Также обратите внимание, что для включения режима потребителя используется флаг `-C`, но информация о брокере передается в том же параметре `-b`, что и в режиме производителя [2].

Листинг 9.4. Использование `kcat` в роли потребителя

```
kcat -C -b localhost:9094 \
-t kinaction_selfserviceTopic
```

Передать адрес брокера и имя темы из нашего кластера для чтения сообщений из этой темы


```
// для сравнения та же операция с помощью сценария kafka-console-consumer
bin/kafka-console-consumer.sh --bootstrap-server localhost:9094 \
--topic kinaction_selfserviceTopic
```

Для сравнения та же операция с помощью сценария командной строки

Давая возможность быстро проверить наши темы и собрать метаданные в нашем кластере, эта маленькая утилита займет достойное место в вашем наборе инструментов. Но давайте посмотрим, есть ли еще какие-нибудь инструменты, которые мы могли

бы использовать и которые не являются инструментами командной строки. Такие инструменты действительно есть! Для тех, кто любит REST, есть Confluent REST Proxy.

9.1.3. Confluent REST Proxy API

Иногда пользователи нашего кластера могут предпочесть RESTful API как наиболее распространенный способ взаимодействий между приложениями либо из-за личных предпочтений или из-за простоты использования. Кроме того, некоторые компании со строгими правилами в брандмауэрах могут весьма неохотно открывать дополнительные порты, подобные тем, что мы использовали до сих пор для соединения с брокерами (например, 9094) [3]. Одним из хороших вариантов в таких ситуациях является использование Confluent REST Proxy API (рис. 9.2). Этот прокси реализован как отдельное приложение, которое, скорее всего, будет размещено на собственном сервере для использования в промышленном окружении, и его функциональность аналогична утилите `kcat`, которую мы только что обсудили.

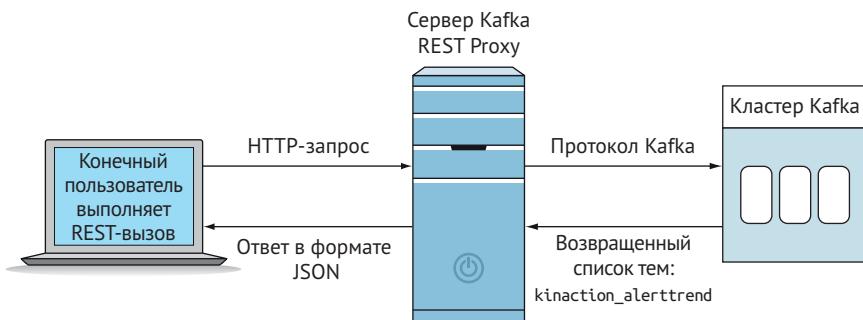


Рис. 9.2. Поиск темы с помощью Confluent REST Proxy

На момент написания этой книги из функций администрирования были доступны только запросы состояния кластера. Однако в документации Confluent можно найти список параметров администрирования, запланированных для реализации в будущем [4]. Чтобы проверить работу REST Proxy, давайте запустим его, как показано в листинге 9.5. При этом у вас уже должны быть запущены экземпляры ZooKeeper и Kafka.

Листинг 9.5. Запуск REST Proxy

```
bin/kafka-rest-start.sh \
etc/kafka-rest/kafka-rest.properties
```

Выполните эту команду в каталоге установки Kafka, чтобы запустить конечную точку REST

Мы уже знаем, как получить список тем в командной строке, теперь посмотрим, как это можно сделать с помощью REST

Proxy, обратившись к конечной точке HTTP с помощью команды `curl` (листинг 9.6) [5]. Поскольку это запрос GET, адрес `http://localhost:8082/topics` можно также скопировать в адресную строку браузера и посмотреть результат.

Листинг 9.6. Вызов REST Proxy с помощью команды `cURL` для получения списка тем

```
curl -X GET \
  -H "Accept: application/vnd.kafka.v2+json" \
  localhost:8082/topics
// Вывод:
[ "__confluent.support.metrics", "__confluent-metrics",
  → "__schemas", "kinaction_alert" ]
```

Annotations for Listing 9.6:

- An annotation from the right side points to the header `-H "Accept: application/vnd.kafka.v2+json"` with the text "Задает формат и версию".
- An annotation from the right side points to the URL `localhost:8082/topics` with the text "Наша цель, конечная точка /topics, содержит список тем, включая созданные нами и внутренние темы Kafka".
- An annotation from the right side points to the output `["__confluent.support.metrics", "__confluent-metrics", → "__schemas", "kinaction_alert"]` with the text "Пример вывода команды curl".

Использование такого инструмента, как `curl`, позволяет определять заголовки отправляемых запросов. Установливая заголовок `Accept`, как показано в листинге 9.6, можно сообщить кластеру Kafka формат ожидаемого ответа и версию используемого API. В данном случае мы указали версию API v2 и формат JSON.

ПРИМЕЧАНИЕ. Поскольку API постоянно развивается, следите за обновлениями в справочнике «Confluent REST Proxy API Reference», доступном по адресу <http://mng.bz/q5Nw>, чтобы быть в курсе последних возможностей.

9.2. Запуск Kafka как службы `systemd`

Одно из решений, которое на определенном этапе придется принять в отношении Kafka, – как запускать и перезапускать брокеры. Те, кто привык управлять серверами в Linux как службами с помощью такого инструмента, как Puppet (<https://puppet.com/>), почти наверняка знакомы с установкой сервисов и, вероятно, могут использовать эти знания для запуска экземпляров с помощью `systemd`. Для тех, кто не знаком с `systemd`, отмечу, что этот демон инициализирует и поддерживает компоненты всей системы [6]. Один из распространенных способов – определить службы ZooKeeper и Kafka как файлы модулей для `systemd`.

В листинге 9.7 показана часть примера файла модуля, запускающего ZooKeeper при загрузке сервера. Он также перезапускает ZooKeeper в случае аварийного сбоя. На практике это предполагает выполнение чего-то похожего на команду `kill -9` с идентификатором процесса (Process ID, PID), чтобы перезапустить процесс. Если вы установили Confluent tar (см. приложение A), то

найдете пример оформления модуля службы в файле *lib/systemd/system/confluent-zookeeper.service*. Подробная информация об использовании этих файлов содержится в документации «Using Confluent Platform systemd Service Unit Files» (<http://mng.bz/7IG9>). Файл модуля в листинге 9.7 просто содержит команды запуска ZooKeeper, которые мы использовали до сих пор в наших примерах.

Листинг 9.7. Модуль службы ZooKeeper

```
...
[Service]
...
ExecStart=/opt/kafkaaction/bin/zookeeper-server-start.sh
  ➔ /opt/kafkaaction/config/zookeeper.properties

ExecStop=
  /opt/kafkaaction/bin/zookeeper-server-stop.sh
Restart=on-abnormal
...

```

Команда запуска ZooKeeper (аналогична той, что мы использовали для запуска вручную)

Завершает работу экземпляра ZooKeeper

ExecStart выполняется, если обнаружится сбой

Также при установке Confluent tar устанавливается пример модуля для службы Kafka в файле *lib/systemd/system/confluent-kafka.service*. В листинге 9.8 показано, как запустить наши службы с помощью *systemctl* после того, как модули будут определены [6].

Листинг 9.8. Запуск Kafka с помощью systemctl

```
sudo systemctl start zookeeper
sudo systemctl start kafka
```

Запуск службы ZooKeeper

Запуск службы Kafka

Если вы используете файлы примеров, полученные в составе загруженного пакета Confluent, то после распаковки папки проверьте корневую папку *./lib/systemd/system*, где можно найти примеры файлов с определениями некоторых других служб, таких как Connect, Schema Registry и REST API.

9.3. Журналы

Помимо журналов событий, где хранятся наши данные о событиях, имеются также другие журналы, о существовании которых необходимо знать, – это журналы приложений, которые Kafka создает как типичное приложение. Журналы, рассматриваемые в этом разделе, не содержат наших событий и сообщений, а только данные, характеризующие работу самой Kafka. И не забывайте о ZooKeeper!

9.3.1. Журналы приложений Kafka

Вы, наверное, уже привыкли к тому, что мы обсуждаем один файл журнала, но, вообще, в Kafka есть несколько файлов журнала, которые могут вас заинтересовать и помочь при устранении неполадок. Из-за наличия нескольких файлов вам, возможно, придется подумать об изменении различных параметров механизма журналирования Log4j, чтобы обеспечить сохранение необходимого представления об операциях.

Какой механизм используется в Kafka?

kafkaAppender – это не то же самое, что KafkaAppender (<http://mng.bz/5ZpB>). Чтобы использовать KafkaLog4jAppender в качестве механизма журналирования, нужно изменить следующую строку, а также включить зависимости для клиентов, указав один и тот же файл JAR вместо класса org.apache.log4j.ConsoleAppender:

```
log4j.appenders.kafkaAppender=
    org.apache.kafka.log4jappender.KafkaLog4jAppender
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-log4j-appender</artifactId>
    <version>2.7.1</version>
</dependency>
```

Это интересный подход к размещению файлов журналов непосредственно в Kafka. Некоторые решения сами анализируют файлы журналов, а затем отправляют их в Kafka.

По умолчанию журналы сервера постоянно добавляются в каталог по мере их создания. Однако никакие журналы не удаляются, и для кого-то это может быть предпочтительным поведением, особенно если эти файлы необходимы для аудита или отладки. Если вам потребуется контролировать их количество и размеры, то самый простой способ добиться этого – обновить файл *log4j.properties* до запуска сервера брокера. В листинге 9.9 определяются два важных свойства для kafkaAppender: *MaxFileSize* и *MaxBackupIndex* [7].

Листинг 9.9. Настройки хранения журналов в Kafka

```
log4j.appenders.kafkaAppender.MaxFileSize=500KB ← Определяет размер файла, по достижении которого должен создаваться новый файл журнала
log4j.appenders.kafkaAppender.MaxBackupIndex=10 ← Устанавливает количество хранимых старых файлов на случай, если потребуются более старые журналы, кроме текущего
```

Обратите внимание, что изменение `kafkaAppender` меняет порядок обработки только файла `server.log`. Если понадобится задать разные размеры файлов и номера файлов резервных копий для других журналов Kafka, можно использовать таблицу имен файлов, чтобы определить, какие механизмы следует обновить. В табл. 9.1 имя механизма приводится в левом столбце – это ключ, определяющий файлы журналов (справа) в брокерах [8].

Таблица 9.1. Механизмы журналирования и журналы

Имя механизма журналирования	Имя файла журнала
<code>kafkaAppender</code>	<code>server.log</code>
<code>stateChangeAppender</code>	<code>state-change.log</code>
<code>requestAppender</code>	<code>kafka-request.log</code>
<code>cleanerAppender</code>	<code>log-cleaner.log</code>
<code>controllerAppender</code>	<code>controller.log</code>
<code>authorizerAppender</code>	<code>kafka-authorizer.log</code>

Изменения в файле `log4j.properties` требуют перезапуска брокера, поэтому требования к журналированию лучше всего определить перед первым запуском брокеров, если это возможно. Конфигурационные значения можно также изменить с помощью JMX, но такие изменения не сохраняются при перезапуске брокера.

В этом разделе мы сосредоточились на журналах Kafka, но журналы ZooKeeper тоже требуют внимания. ZooKeeper работает и регистрирует данные точно так же, как брокеры, поэтому необходимо помнить и о журналах для этих серверов.

9.3.2. Журналы ZooKeeper

В зависимости от особенностей установки и использования ZooKeeper нам также может потребоваться изменить конфигурацию журналирования в ZooKeeper. По умолчанию ZooKeeper не удаляет файлы журналов, но эта функция может быть добавлена при настройке Kafka. Конфигурационные параметры, описанные в инструкции по настройке локального узла ZooKeeper в приложении А, можно установить в файле `config/zookeeper.properties`. В любом случае желательно убедиться, что в конфигурации определены следующие параметры, управляющие сохранением журналов ZooKeeper и имеющие верные значения:

`autopurge.purgeInterval` – интервал в часах для запуска процедуры очистки. Этот параметр должен иметь значение больше 0, чтобы гарантировать запуск очистки [9];

`autopurge.snapRetainCount` – количество последних моментальных снимков и соответствующих журналов транзакций в `dataDir`

и `dataLogDir` [9]. При превышении этого числа самые старые журналы будут удалены. В зависимости от конкретных потребностей может понадобиться хранить больше или меньше журналов. Например, если журналы используются только для устранения неполадок, то время хранения можно уменьшить, а если еще и для аудита, то увеличить;

`snapCount` – ZooKeeper регистрирует свои транзакции в журнале транзакций. Этот параметр определяет количество транзакций, записываемых в один файл. Если со значением по умолчанию (100 000) в этом параметре файлы получаются слишком большими, то его можно уменьшить [10].

Существуют и другие решения для ротации и очистки журналов помимо Log4j. Например, `logrotate` – удобный инструмент, поддерживающий ротацию журналов и сжатие файлов журналов.

Обслуживание журналов является важной, но не единственной обязанностью администратора. Есть также ряд других задач, о которых необходимо помнить, начиная разворачивать новый кластер Kafka. Одна из них – убедиться, что клиенты могут подключаться к нашим брокерам.

9.4. Брандмауэры

В зависимости от конфигурации сети может потребоваться обслуживать клиентов, находящихся внутри или вне сети, где действуют брокеры Kafka [3]. Брокеры Kafka могут прослушивать несколько портов. Например, по умолчанию используется порт 9092. На том же хосте можно настроить порт SSL с номером 9093. Оба этих порта могут быть открыты в зависимости от настроек подключения клиентов к брокерам.

Кроме того, клиенты могут подключаться к ZooKeeper через порт 2181. Порт 2888 используется ведомыми узлами ZooKeeper для подключения к ведущему узлу ZooKeeper, а для взаимодействий между узлами ZooKeeper используется порт 3888 [11]. Настраивая удаленное подключение JMX или других служб Kafka (например, REST Proxy), не забудьте учсть влияние этого порта на другие окружения или пользователей. Проще говоря, если вы используете какие-либо инструменты командной строки, требующие указывать номер порта после имени хоста с сервером ZooKeeper или Kafka, убедитесь, что эти порты доступны, особенно если используется брандмауэр.

9.4.1. Публикуемые слушатели

Одна из ошибок подключения, которая выглядит как проблема с брандмауэром, связана с использованием свойств `listeners` и `advertised.listeners`. Клиенты должны использовать правильное

имя хоста для подключения, если оно задано, и соответствующий хост должен быть достижимым. Например, давайте рассмотрим случаи, когда значения параметров `listeners` и `advertised.listeners` могут не совпадать.

Представим, что на запуске клиент может подключиться к брокеру, но терпит неудачу при попытке получить сообщения. Почему возможно такое поведение, которое кажется непоследовательным? Напомним, что в момент запуска клиент подключается к любому брокеру, чтобы получить метаданные и выяснить, к какому конкретному брокеру он должен подключиться для получения сообщений. Первоначальное соединение клиент устанавливает, используя информацию из конфигурационного параметра `listeners`, а информацию для последующего подключения клиент получает из параметра `advertised.listeners` [12]. Соответственно, есть вероятность, что для выполнения своей работы он должен подключиться к другому хосту.

На рис. 9.3 показано, как клиент использует одно имя хоста для начального подключения, а затем другое для создания рабочего соединения. Второе имя хоста клиент получает в ответ на начальный вызов.

Важным параметром, на который следует обратить внимание, является параметр `inter.broker.listener.name`, определяющий, как брокеры связываются друг с другом в кластере [12]. Если брокеры не могут связаться друг с другом, репликация становится невозможной, и кластер окажется, мягко говоря, не в лучшем состоянии! Замечательное описание роли публикуемых слушателей можно найти в статье Робина Моффатта (Robin Moffatt) «Kafka Listeners – Explained» [12]. Основой для схемы на рис. 9.3, кстати, послужили диаграммы Робина Моффата из его статьи [12].

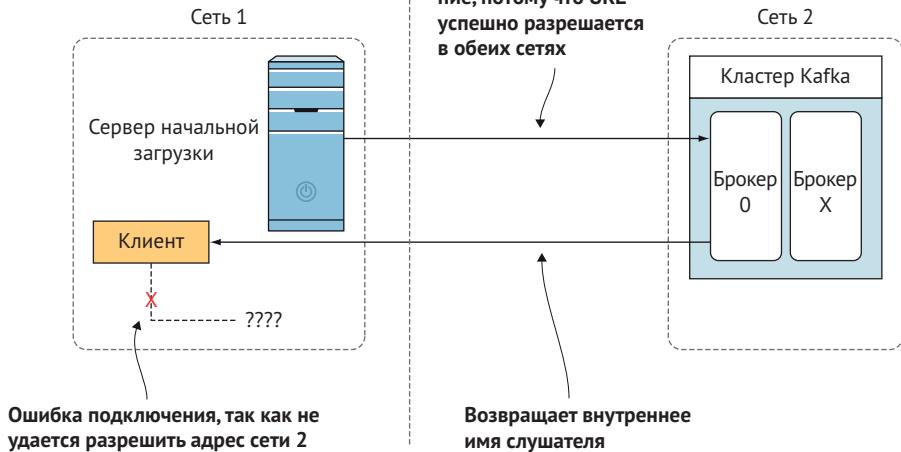
9.5. Метрики

В главе 6 мы рассмотрели пример настройки для получения некоторых метрик JMX из нашего приложения. Возможность видеть эти метрики – первый шаг. Теперь давайте рассмотрим некоторые из них, способные помочь выявить проблемные области.

9.5.1. Консоль JMX

Для исследования открытых метрик и получения представления о доступных возможностях можно использовать графический интерфейс. Один из таких интерфейсов – VisualVM (<https://visualvm.github.io/>). Просмотр доступных метрик JMX может помочь обнаружить объекты инфраструктуры, в которые можно добавить оповещения. При установке VisualVM обязательно выполните дополнительный шаг и установите браузера MBeans.

Сценарий 1: нет публикуемых слушателей.
 Клиент-производитель запускается и
 запрашивает метаданные с сервера
 начальной загрузки



Сценарий 2: публикуемые слушатели имеют URL,
 которые успешно разрешается в обеих сетях.
 Клиент-производитель запрашивает метаданные

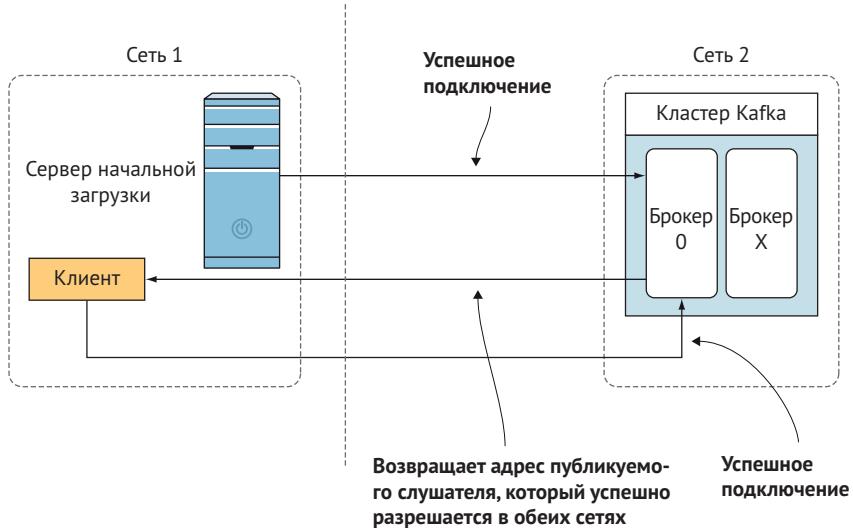


Рис. 9.3. Сравнение публикуемых и обычных слушателей в Kafka

Как отмечалось в главе 6, для каждого брокера, к которому предполагается подключаться, должен быть определен параметр `JMX_PORT`. Это можно сделать с помощью переменной окружения в терминале, например: `export JMX_PORT=49999` [13]. Убедитесь, что

правильно определили область видимости для каждого брокера, а также для каждого узла ZooKeeper.

`KAFKA_JMX_OPTS` – еще один вариант удаленного подключения к брокерам Kafka. Обязательно укажите правильные имя хоста и номер порта. В листинге 9.10 показан пример настройки параметра `KAFKA_JMX_OPTS` с различными аргументами [13]. Здесь используется порт 49999 и `localhost` в качестве имени хоста. Другие параметры в листинге 9.10 обеспечивают возможность подключения без SSL и без аутентификации.

Листинг 9.10. Параметры настройки JMX в Kafka

```
KAFKA_JMX_OPTS="-Djava.rmi.server.hostname=127.0.0.1
-Dcom.sun.management.jmxremote.local.only=false
-Dcom.sun.management.jmxremote.rmi.port=49999
-Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false"
```

Открывает этот порт для JMX

Давайте взглянем на ключевую метрику брокера и посмотрим, как найти нужное нам значение. На рис. 9.4 показано, как с помощью небольшого приложения MBeans найти значение параметра `UnderReplicatedPartitions`. Используя имя

`kafka.server:type=ReplicaManager, name=UnderReplicatedPartitions`

мы можем развернуть иерархию, выглядящую как структура папок, начиная с `kafka.server`.

Продолжая, можно найти тип `ReplicaManager` с атрибутом `UnderReplicatedPartitions`. Также на рис. 9.4 показано, как найти значение `RequestQueueSize` [14]. Теперь, когда вы знаете, как найти определенное значение, давайте подробнее рассмотрим некоторые из наиболее важных параметров, на которые следует обращать особое внимание.

Если вы используете Confluent Control Center или Confluent Cloud, большинство этих метрик доступно для мониторинга по умолчанию. Confluent Platform предлагает для начала настроить оповещения для следующих трех основных параметров: `UnderMinIsrPartitionCount`, `UnderReplicatedPartitions`, `UnderMinIsr` [14].

В следующем разделе мы рассмотрим другой вариант мониторинга и узнаем, как использовать специальные перехватчики.

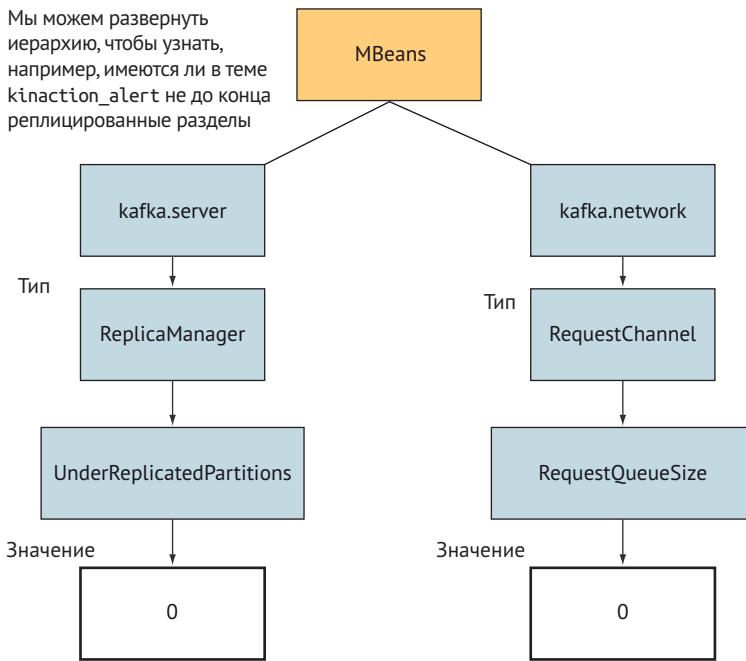


Рис. 9.4. Местоположение значений `UnderReplicatedPartitions` и `RequestQueueSize`

9.6. Способы трассировки

Встроенные метрики, которые мы видели до сих пор, позволяют получить достаточно полное представление о текущем состоянии, но как быть, если понадобится проследить движение одного конкретного сообщения через систему? Какие средства можно использовать, чтобы увидеть созданное сообщение и статус его потребления? Давайте поговорим о простой, но понятной модели, которая может пригодиться вам.

Допустим, у нас есть производитель, присваивающий каждому событию уникальный идентификатор. Поскольку каждое сообщение важно, ни одно из них не должно быть пропущено. При использовании одного клиента бизнес-логика работает как обычно и потребляет сообщения из темы. В этом случае имеет смысл регистрировать идентификатор обработанного события в базе данных или в простом файле журнала. Отдельный потребитель, назовем его проверяющим потребителем, извлекает данные из той же темы и следит за тем, чтобы в списке событий, обработанных первым приложением, не было отсутствующих идентификаторов. Этот процесс может неплохо справляться со своей задачей, но требует добавления логики в приложение и поэтому может оказаться не лучшим выбором.

На рис. 9.5 показан другой подход с использованием перехватчиков Kafka. Перехватчики, определяемые нами, дают возможность добавить логику в производителя, потребителя или в оба клиента, которая включается в рабочий процесс, перехватывает события и добавляет дополнительные данные до того, как события продолжат перемещение по своему обычному пути. Изменения в клиентах зависят от конфигурации и помогают отделить нашу конкретную логику от клиентов.

Давайте еще раз вернемся к перехватчикам, с которыми мы кратко познакомились в главе 4, когда обсуждали возможность применения перехватчиков-производителей для дополнительной обработки сообщений. Добавив перехватчика в производителя и в потребителя, можно отделить логику мониторинга от логики приложения. Мы надеемся, что такой подход может лучше соответствовать сквозному характеру задачи мониторинга.

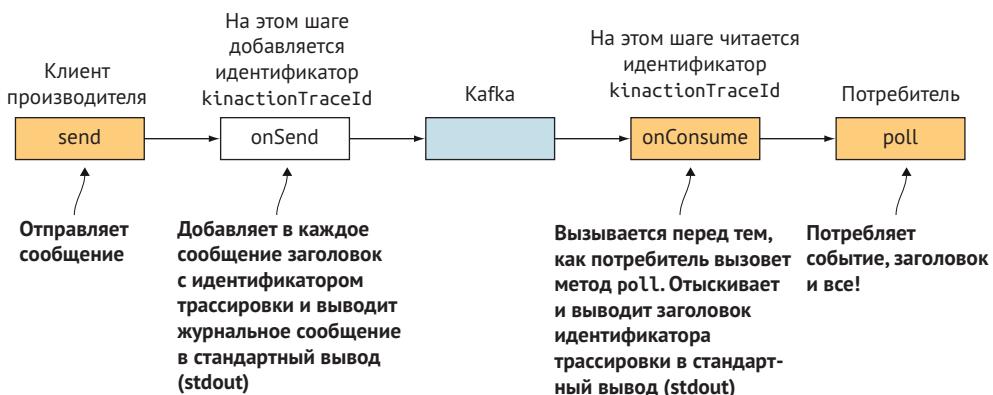


Рис. 9.5. Трассировка событий с помощью перехватчиков

9.6.1. Логика на стороне производителя

Следует отметить, что перехватчиков может быть несколько, поэтому нет необходимости включать всю логику в один класс; ее можно добавлять и удалять по мере необходимости. Порядок перечисления классов важен, потому что именно в таком порядке они будут выполняться. Первый перехватчик получает событие от клиента-производителя. Если перехватчик изменит событие, то другие перехватчики в цепочке после изменения увидят уже измененное событие [15].

Начнем с обзора Java-интерфейса `ProducerInterceptor`. Давайте добавим новый перехватчик в нашего производителя `Alert`, созданного в главе 4. Для этого определим новый класс с именем `AlertProducerMetricsInterceptor`, в который добавим логику, связанную с предупреждениями, как показано в листинге 9.11. Реализация интерфейса `ProducerInterceptor` позволяет подключиться к жизненному циклу перехватчика-производителя. Метод `onSend` вызывает-

ся после вызова метода `send()` клиентом-производителем, который мы использовали до сих пор [15]. Также в листинг 9.11 добавим создание заголовка `kinactionTraceId`. Использование уникального идентификатора помогает подтвердить на стороне потребителя, что в конце жизненного цикла получено то же самое сообщение, которое было создано в начале.

Листинг 9.11. Пример AlertProducerMetricsInterceptor

```

public class AlertProducerMetricsInterceptor
    implements ProducerInterceptor<Alert, String> {

    final static Logger log =
        LoggerFactory.getLogger(AlertProducerMetricsInterceptor.class);

    public ProducerRecord<Alert, String>
        onSend(ProducerRecord<Alert, String> record) {
            Headers headers = record.headers();
            String kinactionTraceId = UUID.randomUUID().toString();
            headers.add("kinactionTraceId",
                kinactionTraceId.getBytes());
            log.info("kinaction_info Created kinactionTraceId: {}", kinactionTraceId);
            return record;
        }

    public void onAcknowledgement(
        RecordMetadata metadata, Exception exception)
    {
        if (exception != null) {
            log.info("kinaction_error " + exception.getMessage());
        } else {
            log.info("kinaction_info topic = {}, offset = {}",
                metadata.topic(), metadata.offset());
        }
    }

    // остальной код опущен
}

```

Мы также должны изменить существующий класс `AlertProducer`, чтобы зарегистрировать новый перехватчик. Для этого нужно добавить свойство `interceptor.classes` в конфигурацию производителя, указав в значении полное имя нового класса `AlertProducerMetricsInterceptor`. Здесь мы использовали имя свойства для большей

ясности, но вообще можно взять константу, определяемую классом `ProducerConfig`. В данном случае можно было бы использовать `ProducerConfig.INTERCEPTOR_CLASSES_CONFIG` [15]. Необходимое изменение показано в листинге 9.12.

Листинг 9.12. Настройка перехватчика в AlertProducer

```
Properties kaProperties = new Properties();
...
kaProperties.put("interceptor.classes",
    AlertProducerMetricsInterceptor.class.getName()); ←
Producer<Alert, String> producer =
new KafkaProducer<Alert, String>(kaProperties);
```

Настройка перехватчиков (значением может быть одно или несколько имен классов, перечисленных через запятую)

В этом примере мы определили один перехватчик, который добавляет уникальный идентификатор в каждое созданное сообщение. Этот идентификатор добавляется в заголовок, чтобы при извлечении сообщения потребителем соответствующий перехватчик смог извлечь и зафиксировать обработанный идентификатор. Цель состоит в том, чтобы обеспечить сквозной мониторинг за пределами Kafka. В журналах приложений мы сможем увидеть сообщения, подобные показанным в листинге 9.13, добавленные классом `AlertProducerMetricsInterceptor`.

Листинг 9.13. Вывод перехватчика AlertProducerMetricsInterceptor

```
kinaction_info Created kinactionTraceId:
603a8922-9fb5-442a-a1fa-403f2a6a875d ←
kinaction_info topic = kinaction_alert, offset = 1
```

Перехватчик производителя регистрирует в журнале значение идентификатора

9.6.2. Логика на стороне потребителя

Теперь, реализовав перехватчика, обрабатывающего отправляемые сообщения, посмотрим, как реализовать аналогичную логику на стороне потребителя. Наша цель – проверить заголовок, добавленный перехватчиком на стороне производителя. В листинге 9.14 показана реализация интерфейса `ConsumerInterceptor`, помогающая получить этот заголовок [16].

Листинг 9.14. Пример AlertConsumerMetricsInterceptor

```
public class AlertConsumerMetricsInterceptor
    implements ConsumerInterceptor<Alert, String> { ←
    public ConsumerRecords<Alert, String>
onConsume(ConsumerRecords<Alert, String> records) {
    if (records.isEmpty()) {
        return records;
}
```

Реализует интерфейс Consumer-Interceptor, чтобы Kafka распознала наш перехватчик

```

} else {
    for (ConsumerRecord<Alert, String> record : records) {
        Headers headers = record.headers(); ← Цикл по заголовкам
        for (Header header : headers) {
            if ("kinactionTraceId".equals(
                header.key())))
                log.info("KinactionTraceId is: " + new String(header.value()));
        }
    }
    return records; ← Возвращает сообщение для передачи коду, вызвавшему перехватчик
}
}

```

Вывод найденного заголовка в стандартный вывод

По аналогии с перехватчиком на стороне производителя для определения перехватчика на стороне потребителя мы использовали специализированный интерфейс `ConsumerInterceptor`. Наш перехватчик просматривает в цикле все сообщения и их заголовки, чтобы отыскать идентификатор с ключом `kinactionTraceId` и отправить его на стандартный вывод. Мы также изменили существующий класс `AlertConsumer`, чтобы зарегистрировать новый перехватчик. В конфигурацию потребителя нужно добавить свойство `interceptor.classes` с полным именем нашего нового класса `AlertConsumerMetricsInterceptor`. Этот обязательный шаг показан в листинге 9.15.

Листинг 9.15. Настройка перехватчика в AlertConsumer

```

public class AlertConsumer {
    Properties kaProperties = new Properties();
    ...
    kaProperties.put("group.id",
                     "kinaction_alertinterceptor"); ← Использует новый group.id, чтобы гарантировать начало потребления с текущего смещения
    kaProperties.put("interceptor.classes",
                     AlertConsumerMetricsInterceptor.class.getName()); ← Имя свойства для добавления перехватчика и значение с именем класса
    ...
}

```

В значении свойства `interceptor.classes` можно указать несколько классов, перечислим их через запятую [16]. Здесь мы использовали имя свойства для большей ясности, но вообще можно использовать константу, определяемую классом `ConsumerConfig`. В данном случае можно было бы использовать `ConsumerConfig.INTERCEPTOR_CLASSES_CONFIG` [16]. Необходимое изменение показано в листинге 9.12.

Итак, мы рассмотрели подход, основанный на использовании перехватчиков с обеих сторон, однако есть и другой способ расширения функциональности клиентов – их переопределение.

9.6.3. Переопределение клиентов

Если исходный код клиентов доступен, то можно определить подкласс существующего клиента или создать свой класс, реализующий интерфейсы производителя/потребителя Kafka. На момент написания этой книги в проекте Brave (<https://github.com/openzipkin/brave>) имелся пример реализации такого клиента, обслуживающего данные трассировки.

Для тех, кто не знаком с Brave, отметим, что это библиотека, предназначенная для добавления поддержки распределенной трассировки. Она поддерживает возможность отправлять данные трассировки, например на сервер Zipkin (<https://zipkin.io/>), который может заниматься сбором и поиском этих данных. Если вам будет интересно, загляните в пример реализации класса `TracingConsumer` (<http://mng.bz/6mAo>), добавляющий функциональные возможности в клиентов с помощью Kafka.

Для включения трассировки можно переопределить оба класса, производителя и потребителя, но в следующем примере мы сосредоточимся только на клиенте-потребителе. Код, представленный в листинге 9.16, в действительности является псевдокодом, иллюстрирующим добавление дополнительной логики на сторону потребителя Kafka. Разработчики, желающие расширить логику обработки сообщений, могут использовать экземпляр `KInActionCustomConsumer`, включающий ссылку на обычного клиента-потребителя с именем `normalKafkaConsumer`. Дополнительная логика обеспечивает необходимое поведение при взаимодействии с традиционным клиентом. Ваши разработчики будут работать с потребителем, который за кулисами действует подобно обычному клиенту.

Листинг 9.16. Расширение логики клиента-потребителя

```
final class KInActionCustomConsumer<K, V> implements Consumer<K, V> {
    ...
    final Consumer<K, V> normalKafkaConsumer; ← Дополнительная логика
                                                использует обычного клиента-потребителя
    @Override
    public ConsumerRecords<K, V> poll( ← Потребители все должны вызывать привычные методы интерфейса
        final Duration timeout)
    {
        // Здесь находится дополнительная логика ← Необходимая дополнительная логика
        ...
        // После нее вызывается обычный клиент-потребитель Kafka
        return normalKafkaConsumer.poll(timeout); ← Вызов обычного клиента-потребителя для выполнения обычных обязанностей
    }
    ...
}
```

В этом листинге показан только комментарий, отмечающий дополнительную логику, но ваши пользователи могут все так же использовать привычные методы клиентов, запуская при этом дополнительную логику, такую как проверка отправки дубликатов данных или регистрация данных трассировки из заголовков. Дополнительная логика не мешает работе обычного клиента.

9.7. Общие инструменты мониторинга

Kafka – это приложение Scala™, поэтому она может использовать JMX и библиотеку Yammer Metrics [17]. Эта библиотека используется для предоставления метрик JMX в различных частях приложения, и мы уже видели некоторые из них. Но с развитием Kafka появилось несколько дополнительных инструментов, которые позволяют не только получать метрики JMX, но и выполнять команды администрирования, а также поддерживают некоторые другие средства, упрощающие управление кластерами. Конечно, мы не можем привести в одном разделе полный список таких инструментов и их возможностей, которые к тому же могут меняться со временем. И все же давайте рассмотрим несколько примеров.

Cluster Manager for Apache Kafka, или CMAK (<https://github.com/yahoo/CMAK>), когда-то известный как Kafka Manager, – интересный проект, разработанный в Yahoo™, который фокусируется на управлении Kafka и является пользовательским интерфейсом для различных административных действий. Одна из ключевых особенностей проекта – возможность управления несколькими кластерами. В числе других его функций можно назвать проверку общего состояния кластера, а также поддержку создания и переназначения разделов. Этот инструмент способен выполнять аутентификацию пользователей с помощью LDAP, что может пригодиться в некоторых продуктах [18].

Cruise Control (<https://github.com/linkedin/cruise-control>) – создан разработчиками LinkedIn. В кластерах LinkedIn работают тысячи брокеров, поэтому разработчики накопили богатый опыт работы с кластерами Kafka, который помог им систематизировать и автоматизировать работу с некоторыми из болевых точек Kafka. Этот инструмент предлагает REST API, а также пользовательский интерфейс и, соответственно, несколько способов взаимодействия с ним. В числе интересных особенностей Cruise Control можно назвать возможность наблюдения за кластером и генерирование предложений по балансировке в зависимости от рабочих нагрузок [19].

Confluent Control Center (<https://docs.confluent.io/current/control-center/index.html>) – еще один веб-инструмент, помогающий контролировать кластеры и управлять ими. Но следует отметить, что в настоящее время этот инструмент распространяется на ком-

мерческой основе, и для его использования в промышленном окружении необходимо приобретать корпоративную лицензию. Если у вас уже есть подписка на платформу Confluent, то стоит проверить ее. Этот инструмент использует информационные панели и может помочь определить сбои, задержку в сети и работу внешних соединений.

В целом Kafka предоставляет массу интересных возможностей не только для управления, но и для мониторинга кластера. Распределенные системы сложны, и чем больше опыта вы приобретете, тем лучше будут ваши навыки мониторинга.

Итоги

- Помимо сценариев командной оболочки, поставляемых вместе с Kafka, существует также клиент администрирования, предоставляющий API для выполнения важных задач, таких как создание тем.
- Такие инструменты, как `kcat` и Confluent REST Proxy API, позволяют разработчикам взаимодействовать с кластером.
- Kafka сохраняет клиентские данные в журнале, однако, помимо журнала для данных, существуют те, куда записывается информация, касающаяся работы брокеров. Вы должны помнить об этих журналах (и журналах ZooKeeper) и управлять ими, чтобы при необходимости иметь под рукой подробную информацию для устранения неполадок.
- Понимание публикуемых слушателей может помочь объяснить поведение, которое на первый взгляд выглядит как сбои подключения клиентов.
- Для доступа к метрикам Kafka использует JMX. Благодаря этому есть возможность получать метрики клиентов (производителей и потребителей), а также брокеров.
- Для реализации сквозных задач можно использовать перехватчики на стороне производителей и потребителей. Один из примеров таких задач – добавление идентификаторов в сообщения для трассировки их доставки.

Ссылки

- 1 «Class AdminClient». Confluent documentation (n.d.). <https://docs.confluent.io/5.3.1/clients/javadocs/index.html?org/apache/kafka/clients/admin/AdminClient.html> (доступно по состоянию на 17 ноября 2020).
- 2 «`kcat`». GitHub. <https://github.com/edenhill/kcat/#readme> (доступно по состоянию на 25 августа 2021).

- 3 «Kafka Security & the Confluent Platform». Confluent documentation (n.d.). <https://docs.confluent.io/2.0.1/kafka/platform-security.html#kafka-security-the-confluent-platform> (доступно по состоянию на 25 августа 2021).
- 4 «Confluent REST APIs: Overview: Features». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/kafka-rest/index.html#features> (доступно по состоянию на 20 февраля 2019).
- 5 «REST Proxy Quick Start». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/kafka-rest/quickstart.html> (доступно по состоянию на 22 февраля 2019).
- 6 «Using Confluent Platform systemd Service Unit Files». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/installation/scripted-install.html#overview> (доступно по состоянию на 15 января 2021).
- 7 «Class RollingFileAppender». Apache Software Foundation (n.d.). <https://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/RollingFileAppender.html> (доступно по состоянию на 22 апреля 2020).
- 8 log4j.properties. Apache Kafka GitHub (26 марта 2020). <https://github.com/apache/kafka/blob/99b9b3e84f4e98c3f07714e1de6a139a004cbc5b/config/log4j.properties> (доступно по состоянию на 17 июня 2020).
- 9 «Running ZooKeeper in Production». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/zookeeper/deployment.html#running-zk-in-production> (доступно по состоянию на 23 июля 2021).
- 10 «ZooKeeper Administrator’s Guide». Apache Software Foundation (n.d.). <https://zookeeper.apache.org/doc/r3.4.5/zookeeperAdmin.html> (доступно по состоянию на 10 июня 2020).
- 11 «ZooKeeper Getting Started Guide». Apache Software Foundation (n.d.). <https://zookeeper.apache.org/doc/r3.1.2/zookeeperStarted.html> (доступно по состоянию на 19 августа 2020).
- 12 R. Moffatt. «Kafka Listeners – Explained». Confluent blog (1 июля 2019). <https://www.confluent.io/blog/kafka-listeners-explained/> (доступно по состоянию на 11 июня 2020).
- 13 «Kafka Monitoring and Metrics Using JMX». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/installation/docker/operations/monitoring.html> (доступно по состоянию на 12 июня 2020).
- 14 «Monitoring Kafka: Broker Metrics». Confluent documentation (n.d.). <https://docs.confluent.io/5.4.0/kafka/monitoring.html#broker-metrics> (доступно по состоянию на 1 мая 2020).
- 15 «Interface ProducerInterceptor». Apache Software Foundation (n.d.). <https://kafka.apache.org/27/javadoc/org/apache/kafka/>

<clients/producer/ProducerInterceptor.html> (доступно по состоянию на 1 июня 2020).

- 16 «Interface ConsumerInterceptor». Apache Software Foundation (n.d.). <https://kafka.apache.org/27/javadoc/org/apache/kafka/clients/consumer/ConsumerInterceptor.html> (доступно по состоянию на 1 июня 2020).
- 17 «Monitoring». Apache Software Foundation (n.d.). <https://kafka.apache.org/documentation/#monitoring> (доступно по состоянию на 1 мая 2020).
- 18 Yahoo CMAK README.md. GitHub (5 марта 2020). <https://github.com/yahoo/CMAK/blob/master/README.md> (доступно по состоянию на 20 июля 2021).
- 19 README.md. LinkedIn Cruise Control for Apache Kafka GitHub (30 июня 2021). https://github.com/linkedin/cruise-control/blob/migrate_to_kafka_2_4/README.md (доступно по состоянию на 21 июля 2021).

Часть III

Дополнительные возможности

Ч

асть III фокусируется на дополнительных возможностях Kafka, помимо тех, что были рассмотрены в части II. В этой части мы выйдем за границы простого кластера Kafka, позволяющего читать и записывать данные. Позаботимся о безопасности, добавим схемы данных и познакомимся с другими продуктами Kafka:

- в главе 10 мы рассмотрим приемы повышения защищенности кластера Kafka с помощью SSL, ACL и квот;
- в главе 11 обсудим реестры схем и как они используются для улучшения совместимости данных;
- в главе 12 мы познакомимся с Kafka Streams и ksqlDB.

Все эти компоненты являются частью экосистемы Kafka и представляют более высокие уровни абстракции, основанные на базовых компонентах, которые вы исследовали в части II. К концу этой части вы будете готовы продолжить изучение более сложных тем Kafka самостоятельно и, что особенно важно, использовать Kafka в своей повседневной работе.

10

Защита Kafka

Эта глава охватывает следующие темы:

- основы безопасности и соответствующую терминологию;
- SSL-соединения между кластером и клиентами;
- списки управления доступом (Access Control List, ACL);
- установку квот пропускной способности сети и частоты запросов для ограничения требований к ресурсам.

В этой главе основное внимание уделяется защите наших данных, чтобы доступ к ним имели только те, кто имеет право их читать или записывать. Поскольку безопасность – слишком широкая область, в этой главе мы поговорим лишь о некоторых основных понятиях, чтобы получить общее представление о возможностях, имеющихся в Kafka. Наша цель здесь не в том, чтобы настроить защиту, а в том, чтобы изучить некоторые варианты, которые вы сможете обсудить с вашей службой безопасности в будущем и ознакомиться с основными концепциями. Эта глава не является руководством по безопасности в целом, но она закладывает необходимые основы. Мы обсудим практические действия, которые можно предпринять при настройке, и рассмотрим особенности клиентов, брокеров и ZooKeeper, влияющих на безопасность кластера.

Возможно, ваши данные не нуждаются в тех средствах защиты, которые мы обсудим далее, но знание особенностей данных помо-

жет вам принять обоснованные решения по управлению доступом. Если вы имеете дело с личной или финансовой информацией, такой как даты рождения или номера кредитных карт, то вам, вероятно, нужно познакомиться с большинством параметров безопасности, обсуждаемых в этой главе. Однако если вы работаете только с общей информацией, не являющейся критической для безопасности, то вам может не понадобиться эта защита. В последнем случае не нужно будет внедрять в работу такие функции, как SSL. Начнем с примера вымышленных данных, которые требуется защитить.

Представим, что перед нами стоит цель найти клад, приняв участие в состязании по поиску сокровищ. В этом состязании участвуют две команды, и мы не хотим, чтобы противник имел доступ к данным нашей команды. Вначале каждая команда выбирает свое название темы и сообщает это название только своим участникам. (Не зная названия темы, другая команда не сможет прочитать ваши данные.) Каждая команда начинает с отправки подсказок в тему, которую они считают своей личной. Со временем члены команд могут начать задаваться вопросом о прогрессе противника и о том, есть ли у них какие-либо подсказки, которых нет у оппонента. Вот тут и начинаются проблемы. На рис. 10.1 показана организация темы для команд Team Clueful и Team Clueless¹⁰.

**На данный момент ничто, кроме незнания, не мешает командам читать сообщения из обеих тем.
Каждая команда может читать из любой темы и писать в любую тему**

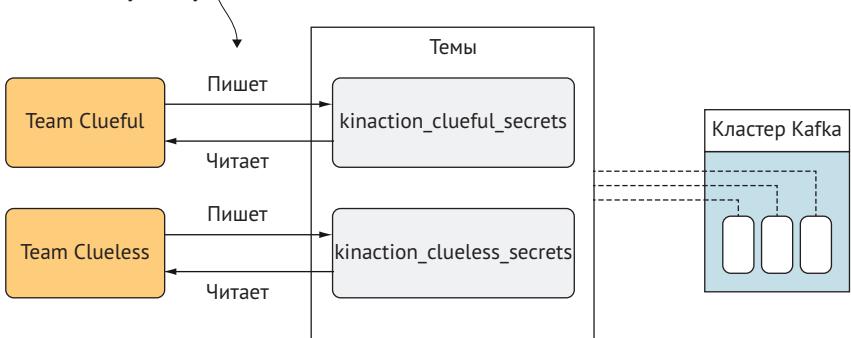


Рис. 10.1. Темы для состязания в поиске сокровищ

Один технически подкованный участник, который, как оказалось, имел опыт использования Kafka, обратился к инструментам командной строки, чтобы найти темы (как своей, так и другой команды). Получив список тем, он теперь знает название темы команды-соперника. Допустим, что это член команды Team Clueless обнаружил тему `--topic kinaction_clueful_secrets` команды

¹⁰ Team Clueful – команда умников; Team Clueless – команда дилетантов. – Прим. перев.

Team Clueful. Ему понадобился лишь консольный потребитель, чтобы получить все данные команды Team Clueful! Но злоумышленник решил не останавливаться на достигнутом.

Чтобы сбить Team Clueful со следа, он начал писать в тему соперника ложную информацию. Теперь Team Clueful регулярно получает ложные данные, что очень мешает им! Поскольку члены Team Clueful не уверены в авторстве сообщений в их теме, они теперь должны определить, какие сообщения являются ложными, и при этом потерять драгоценное время, которое можно было бы использовать для поиска клада.

Как избежать ситуации, в которой оказалась команда Team Clueful? Есть ли способ закрыть доступ к теме для тех, у кого нет соответствующих полномочий? Да, такой способ есть! Решение состоит из двух частей. Первая часть – шифрование данных. Вторая часть – аутентификация и авторизация, позволяющие не только узнать, кто обращается к теме, но и убедиться, что это действительный пользователь. После проверки пользователя нужно определить, какие операции ему разрешено выполнять. Мы подробнее обсудим все это, когда рассмотрим несколько решений, предложенных Kafka.

10.1. Основы безопасности

Занимаясь безопасностью компьютерных приложений, вы, скорее всего, столкнетесь с шифрованием, аутентификацией и авторизацией. Давайте подробнее рассмотрим эти термины (за более подробной информацией обращайтесь по адресу <http://mng.bz/o802>).

Шифрование не означает, что другие не смогут увидеть ваши сообщения, шифрование лишь препятствует получению их содержимого в исходном виде. Многие из вас наверняка знакомы с рекомендацией поощрять использование защищенные сайты (HTTPS) для онлайн-покупок в сети Wi-Fi®. Позже мы используем SSL (*Secure Sockets Layer* – слой защищенных сокетов) для защиты наших соединений, но не между веб-сайтом и нашим компьютером, а между нашими клиентами и брокерами! Кстати, в этой главе SSL используется как обобщенное название, хотя в настоящее время существует более новая версия протокола – TLS [1].

Теперь поговорим об *аутентификации*. Чтобы проверить личность пользователя или приложения, нужен способ его аутентификации: аутентификация – это процесс доказательства, что пользователь или приложение действительно является тем, за кого себя выдает. Например, подписываясь на читательский билет, вы наверняка хотели бы, чтобы библиотекарь не выдал карточку никому, не убедившись, что стоящий перед ним человек тот, за кого себя выдает. В большинстве случаев библиотекарь

должен проверить имя и адрес пользователя, потребовав от него удостоверение личности государственного образца. Цель этого процесса – гарантировать, что никто другой не сможет просто так выдать себя за вас, чтобы воспользоваться вашими привилегиями в своих целях. Если кто-то заявит, что он – это вы, возьмет книги и не вернет их, то штрафы будут отправлены вам. Этот маленький пример наглядно показывает, насколько важно подтвердить личность пользователя.

Авторизация, с другой стороны, определяет круг привилегий пользователя – какие действия ему разрешено выполнять. Продолжая наш пример с библиотекой, карточки, выданные взрослому читателю или ребенку, могут предоставлять разные разрешения. А доступ к электронным публикациям может быть ограничен только терминалами внутри библиотеки.

10.1.1. Шифрование с помощью SSL

До сих пор все наши брокеры в этой книге поддерживали незащищенные соединения [1]. Они не требовали ни аутентификации, ни шифрования. Если заглянуть в любой из ваших текущих файлов *server.properties* (например, *config/server0.properties*, который демонстрируется в приложении A), вы найдете параметр вида *listeners = PLAINTEXT:localhost://:9092*. Он связывает протокол с конкретным портом брокера. Поскольку брокеры поддерживают несколько портов, этот параметр позволяет открыть порт PLAINTEXT, чтобы протестировать добавление SSL или других протоколов на другом порту. Наличие двух портов помогает упростить переход, когда мы решим отказаться от использования незащищенных соединений [2]. На рис. 10.2 показан пример использования незащищенного порта в сравнении с SSL.

На данный момент у нас имеется кластер без какой-либо встроенной защиты. (К счастью, мы можем совершенствовать наш кластер и защитить его от несанкционированного доступа.) Настройка SSL-соединений между брокерами в кластере и клиентами – это один из первых шагов [1]. Не требуется никаких дополнительных серверов или каталогов и никаких изменений в клиентском коде, потому что все зависит только от конфигурации.

Мы не знаем, насколько технически подкованы другие пользователи и смогут ли они перехватить наш трафик в сети с помощью каких-либо инструментов, поэтому для нас желательно шифровать сообщения, пересылаемые между нашими брокерами и клиентами. Настройка, описанная в следующем разделе, необходима для обеспечения безопасности Kafka, но читатели, настраивавшие SSL или HTTPS в прошлом (и особенно в Java), найдут этот подход похожим на другие клиент/серверные механизмы.



Рис. 10.2. Незащищенный порт и порт SSL

10.1.2. Настройка соединений SSL между брокерами и клиентами

В наших предыдущих примерах клиентов Kafka мы не использовали SSL. Однако теперь мы собираемся задействовать этот протокол для шифрования сетевого трафика, передаваемого между клиентами и кластером. Давайте обсудим этот процесс и посмотрим, что нужно сделать, чтобы добавить эту функцию в наш кластер.

ПРИМЕЧАНИЕ. Команды, представленные в этой главе, весьма специфичны и будут работать по-разному в разных операционных системах (или даже при использовании разных доменных имен серверов, перечисленных в настройках брокеров). Поэтому важно иметь представление об общих понятиях. Кроме того, использование других инструментов (например, OpenSSL®) может повлиять на ваши настройки и команды. Разобравшись с базовыми концепциями, обязательно зайдите на сайт Confluent по адресу <http://mng.bz/nrza>, где вы найдете ссылки на другие ресурсы и руководства. Документация на сайте проекта Confluent содержит указания для любых примеров, представленных в этой главе, и они помогут вам реализовать идеи, которые мы освещаем здесь весьма поверхностно.

ВНИМАНИЕ! Чтобы правильно настроить окружение, проконсультируйтесь со специалистом по безопасности. Наши команды предназначены лишь для ознакомления и обучения, а не как руководство для настройки безопасности промышленного уровня. Это не полное руководство. Все, что здесь написано, вы используете на свой страх и риск!

Прежде всего нужно создать ключ и сертификат для наших брокеров [3]. Поскольку на вашем компьютере уже должен быть установлен язык программирования Java, для этой цели можно использовать утилиту keytool, входящую в состав Java. Приложение keytool управляет хранилищем ключей и доверенных сертификатов [4]. Важным моментом, на который следует обратить внимание, является *хранение*. В этой главе в имена некоторых файлов включено слово *broker0*, обозначающее одного конкретного брокера, чтобы показать, что эти файлы не предназначен для всех брокеров. Хранилище ключей лучше всего рассматривать как базу данных, откуда наши программы JVM могут извлекать информацию, необходимую нашим процессам [4]. На данном этапе также генерируем ключ для наших брокеров, как показано в листинге 10.1 [3]. Обратите внимание, что в следующих далее листингах упоминается адрес *manning.com*. Он используется только в качестве примера и не предназначен для использования читателями, которые будут опробовать примеры кода.

Листинг 10.1. Генерирование ключа SSL для брокера

```
keytool -genkey -noprompt \
    -alias localhost \
    -dname "CN=ka.manning.com,OU=TEST,O=TREASURE,L=Bend,S=0г,C=US" \
    -keystore kafka.broker0.keystore.jks \
    -keyalg RSA \
    -storepass changeTreasure \
    -keypass changeTreasure \
    -validity 999
```

Имя хранилища ключей, куда
должен быть помещен вновь
сгенерированный ключ

Использовать пароль, чтобы хранилище
не смог изменить посторонни

Эта команда создаст новый ключ и сохранит его в файле хранилища ключей *kafka.broker0.keystore.jks*. Теперь у нас есть ключ, который (в некотором роде) идентифицирует нашего брокера, нам нужно что-то, что сигнализировало бы, что у нас есть не просто сертификат, выпущенный случайным пользователем. Один из способов удостоверить сертификаты – подписать их с помощью центра сертификации (Certificate Authority, CA). Возможно, вы слышали о центрах сертификации, таких как Let's Encrypt® (<https://letsencrypt.org/>) или GoDaddy® (<https://www.godaddy.com/>). Центры сертификации играют роль доверенного органа,

который удостоверяет право собственности и подлинность открытого ключа [3]. Однако в наших примерах мы организуем свой центр сертификации, чтобы избежать необходимости проверки нашей личности третьей стороной. Итак, наш следующий шаг – создание собственного центра сертификации, как показано в листинге 10.2 [3].

Листинг 10.2. Создание собственного центра сертификации

```
openssl req -new -x509 \
    -keyout cakey.crt -out ca.crt \ <-- Создает новый центр сертификации, а затем генерирует файлы
    -days 999 \
    -subj '/CN=localhost/OU=TEST/O=TREASURE/L=Bend/S=Or/C=US' \
    -passin pass:changeTreasure -passout pass:changeTreasure
```

Теперь нам нужно сообщить клиентам, что они могут доверять этому сгенерированному центру сертификации. По аналогии с термином *хранилище ключей* мы будем использовать термин *хранилище доверенных сертификатов* для обозначения хранилища этой новой информации [3].

После создания своего центра сертификации командой в листинге 10.2 мы можем использовать его для подписи уже созданных сертификатов для брокеров. Сначала экспортируем сертификат, сгенерированный в листинге 10.2, каждому брокеру, взяв его из хранилища ключей, подписав с помощью нового центра сертификации, а затем импортировав сертификат центра сертификации и подписанный сертификат брокера обратно в хранилище ключей [3]. Для автоматизации подобных действий Confluent предоставляет сценарий командной оболочки (<http://mng.bz/v497>) [3]. Остальные команды вы найдете в примерах исходного кода для этой книги.

ПРИМЕЧАНИЕ. При выполнении этих команд ваша операционная система или инструменты могут выводить другое приглашение к вводу, чем у нас. Обычно после выполнения команды появится приглашение к вводу, настроенное в вашей командной оболочке. Чтобы не было разнотечений мы стараемся избегать показывать эти приглашения в наших примерах.

Нам также необходимо обновить файл конфигурации *server.properties* для каждого брокера, как показано в листинге 10.3 [3]. Обратите внимание, что здесь показан лишь файл, соответствующий брокеру *broker0*, и только его часть.

Листинг 10.3. Изменения в конфигурационном файле брокеров

```
...
listeners=PLAINTEXT://localhost:9092,
→ SSL://localhost:9093
ssl.truststore.location=
→ /opt/kafkainaction/private/kafka
→ .broker0.truststore.jks
ssl.truststore.password=changeTreasure
ssl.keystore.location=
→ /opt/kafkainaction/kafka.broker0.keystore.jks
ssl.keystore.password=changeTreasure
ssl.key.password=changeTreasure
...

```

Добавить порт SSL, оставив старый порт PLAINTEXT

Указать местоположение хранилища доверенных сертификатов и пароль доступа к нему

3. Указать местоположение хранилища ключей и пароль доступа к нему

Также необходимо внести изменения в конфигурацию наших клиентов. Например, нужно установить параметр `security.protocol=SSL`, а также указать местоположение хранилища доверенных сертификатов и пароль в файле с именем `custom-ssl.properties`. Это поможет установить протокол, используемый для SSL, а также указать местоположение нашего хранилища доверенных сертификатов [3].

При тестировании этих изменений можно настроить несколько слушателей для брокера. Это поможет клиентам мигрировать постепенно, поскольку оба порта смогут обслуживать трафик до того, как мы удалим старый порт PLAINTEXT [3]. Файл `kinaction-ssl.properties` помогает клиентам получить информацию, необходимую для взаимодействия с брокером, который становится все более безопасным!

Листинг 10.4. Настройка конфигурации для клиента командной строки

```
bin/kafka-console-producer.sh --bootstrap-server localhost:9093 \
--topic kinaction_test_ssl \
--producer.config kinaction-ssl.properties
bin/kafka-console-consumer.sh --bootstrap-server localhost:9093 \
--topic kinaction_test_ssl \
--consumer.config kinaction-ssl.properties

```

Сообщить клиенту-производителю настройки SSL

Передать конфигурацию SSL клиенту-потребителю

Одна из самых приятных особенностей – возможность использовать одну и ту же конфигурацию как для производителей, так и для потребителей. Однако, взглянув на содержимое этого файла конфигурации, многие из вас заметят одну проблему – использование паролей в этих файлах. Самый простой вариант обезопасить пароли – настроить разрешения доступа для этого файла. Прежде чем размещать конфигурацию в вашей файловой системе, важно

проанализировать возможность ограничения на доступ к файлу, а также его принадлежность. Как всегда, проконсультируйтесь со своими экспертами по безопасности, чтобы выбрать лучший из вариантов, доступных в вашем окружении.

10.1.3. Настройка соединений SSL между брокерами

Еще одна деталь, на которую стоит обратить внимание: поскольку наши брокеры взаимодействуют не только с клиентами, но и между собой, мы можем использовать SSL и для этих взаимодействий. В таком случае можно установить параметр `security.inter.broker.protocol = SSL` в файле `server.properties`, а также подумать о возможности изменения номера порта. Более подробную информацию можно найти по адресу <http://mng.bz/4KBw> [5].

10.2. Kerberos и Simple Authentication and Security Layer (SASL)

Если в вашей организации есть группа безопасности со своим сервером Kerberos, то, скорее всего, у вас работают эксперты по безопасности, к которым можно обратиться за помощью. Когда мы впервые начали работать с Kafka, у нас уже были инструменты для работы с большими данными, использующие Kerberos. Kerberos часто применяются в организациях для сквозной авторизации (Single Sign-On, SSO).

Если у вас уже настроен сервер Kerberos, то вам понадобится учетная запись для доступа к Kerberos, чтобы создать принципала (principal – уникальное имя клиента) для каждого брокера, а также для каждого пользователя (или приложения), который будет иметь доступ к кластеру. Поскольку эта настройка слишком сложна для локального тестирования, внимательно прочитайте следующее обсуждение, чтобы познакомиться с форматом файлов службы аутентификации и авторизации Java (Java Authentication and Authorization Service, JAAS) – распространенным типом файлов для брокеров и клиентов. По адресу <http://mng.bz/QqxG> вы найдете список ресурсов, к которым можно обратиться за более подробной информацией [6].

Файлы JAAS с информацией о файле `keytab` помогают передать в Kafka принципала и учетные данные. Как правило, `keytab` – это отдельный файл, содержащий принципала и зашифрованные ключи. Этот файл можно использовать для аутентификации брокеров Kafka без запроса пароля [7]. Однако важно отметить, что с файлом `keytab` нужно обращаться с такой же осторожностью, как и с любыми другими учетными данными.

Рассмотрим некоторые изменения в свойствах сервера, которые нужно внести, а также пример конфигурации JAAS. Прежде

всего каждому брокеру понадобится свой файл *keytab*. Наш файл JAAS поможет брокерам найти этот файл на сервере, а также объявить принципала для использования [7]. В листинге 10.5 показан пример файла JAAS, который будет использоваться при запуске.

Листинг 10.5. Файл SASL JAAS для брокера

```
KafkaServer { ← Настройка файла JAAS  
для брокера Kafka  
...  
keyTab="/opt/kafkainaction/kafka_server0.keytab"  
principal="kafka/kafka0.ka.manning.com@MANNING.COM";  
};
```

Добавим еще один порт *SASL_SSL* для тестирования, прежде чем удалить старые порты [7]. Необходимое для этого изменение показано в листинге 10.6. В зависимости от того, какой порт вы использовали для подключения к брокерам, в этом примере используется протокол *PLAINTEXT*, *SSL* или *SASL_SSL*.

Листинг 10.6. Изменения в файле SASL для брокера

```
listeners=PLAINTEXT://localhost:9092,SSL://localhost:9093,  
→ SASL_SSL://localhost:9094 ← Добавить порт SASL_SSL для брокера,  
оставив старые порты
```

Настройки для клиентов выглядят аналогично [7]. В листинге 10.7 показаны необходимые изменения в файле JAAS.

Листинг 10.7. Изменения в файле SASL для клиента

```
KafkaClient { ← Запись в файле JAAS для поддержки  
SASL на стороне клиента  
...  
keyTab="/opt/kafkainaction/kafkaclient.keytab"  
principal="kafkaclient@MANNING.COM";  
};
```

Нам также нужно обновить конфигурацию клиента, добавив в нее значения SASL [3]. Файл клиента похож на файл *kinaction-ssl.properties*, использовавшийся выше, но определяет протокол *SASL_SSL*. После проверки возможности соединения с портами 9092 и 9093 можно использовать нашу новую конфигурацию и убедиться, что мы получаем те же результаты с использованием нового протокола *SASL_SSL*.

10.3. Авторизация в Kafka

Теперь, увидев в общих чертах, как использовать аутентификацию в Kafka, давайте посмотрим, как применять эту информацию для организации доступа пользователей, и начнем мы со списков управления доступом.

10.3.1. Списки управления доступом

Авторизация – это процесс, определяющий действия, которые может выполнять пользователь. Одним из способов поддержки авторизации являются списки управления доступом (Access Control List, ACL). Большинство пользователей Linux хорошо знакомо с правами доступа к файлам, которыми можно управлять с помощью команды `chmod` (такими как право на чтение, на запись и на выполнение). Одним из недостатков такой системы прав является ее недостаточная гибкость. Списки управления доступом позволяют задавать разрешения для нескольких лиц и групп, а также другие типы разрешений и часто используются, когда нужны разные уровни доступа к общей папке [8]. Один из примеров – разрешение, позволяющее пользователю редактировать файл, но запрещающее тому же пользователю удалять его (для этого требуется отдельное разрешение). На рис. 10.3 показано управление привилегиями пользователя Франц на доступ к ресурсам нашей гипотетической команды, принимающей участие в состязаниях по поиску сокровищ.

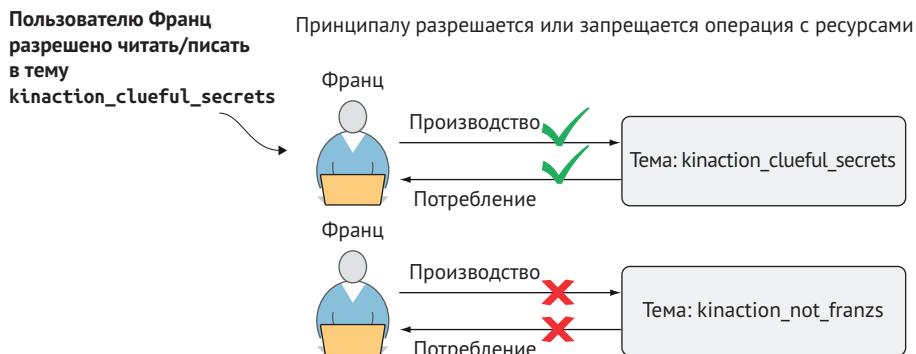


Рис. 10.3. Списки управления доступом (ACL)

Kafka поддерживает возможность подключения смешанных механизмов авторизации, что позволяет пользователям реализовать свою логику [8]. В Kafka есть класс `SimpleAclAuthorizer`, который мы будем использовать в нашем примере.

В листинге 10.8 показано добавление класса, реализующего авторизацию, и суперпользователя Франца (Franz) в файл `server.properties` брокера для использования ACL. Важно отметить, что после настройки класса авторизации нам нужно настроить списки управления доступом, иначе доступ к ресурсам будут иметь только суперпользователи [8].

Листинг 10.8. Настройка класса авторизации и суперпользователя

```
authorizer.class.name= kafka.security.auth.SimpleAclAuthorizer
super.users=User:Franz
```

Конфигурация брокера должна включать SimpleAclAuthorizer
Определение суперпользователя, имеющего доступ ко всем ресурсам, независимо от настроек ACL

Давайте посмотрим, как определить привилегии для команды Team Clueful, чтобы только она могла производить и потреблять данные из своей собственной темы `kinaction_clueful_secrets`. Для простоты предположим, что в команде есть только два пользователя, Франц и Хемингуэй. Поскольку мы уже создали файлы `keytab` для пользователей, мы имеем основную информацию, которая нам нужна. Как можно заметить в листинге 10.9, операция `Read` (чтение) позволяет потребителям получать данные из темы [8]. Вторая операция, `Write` (запись), позволяет тем же членам команды посыпать данные в тему.

Листинг 10.9. Списки управления доступом в Kafka, управляющие разрешениями для чтения и записи в тему

```
bin/kafka-acls.sh --authorizer-properties \
--bootstrap-server localhost:9094 --add \
--allow-principal User:Franz \
--allow-principal User:Hemingway \
--operation Read --operation Write \
--topic kinaction_clueful_secrets
```

Определение двух пользователей для предоставления разрешений
Позволяет именованным принципалам читать и писать в указанную тему

Инструмент командной строки `kafka-acls.sh` поставляется вместе с Kafka и позволяет добавлять, удалять или перечислять текущие списки управления доступом [8].

10.3.2. Управление доступом на основе ролей

Управление доступом на основе ролей (Role-Based Access Control, RBAC) – это вариант, который поддерживает Confluent Platform. RBAC обеспечивает возможность управления доступом на основе ролей [9]. Пользователям назначаются роли в соответствии с их потребностями (например, служебными обязанностями). В этом случае разрешения определяются не для конкретных пользователей, а для ролей [9]. На рис. 10.4 показано, как назначение роли пользователю дает ему новые разрешения.

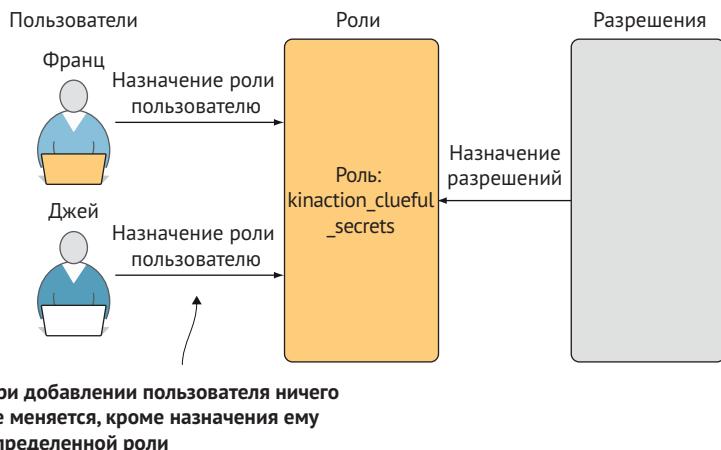


Рис. 10.4. Управление доступом на основе ролей

Нашим командам, участвующим в состязании по поиску сокровищ, имеет смысл назначить определенные роли. Подобное назначение может отражать, например, роль команды из отдела маркетинга, отличную от роли команды из отдела бухгалтерского учета. Если какой-либо пользователь перейдет в другой отдел, ему будет назначена другая роль, а не конкретные привилегии. Поскольку это более новый способ разграничения привилегий, который может измениться в процессе развития и предназначен для среды Confluent Platform, мы упоминаем о нем только в информативных целях и не будем углубляться в его описание.

10.4. ZooKeeper

Занимаясь безопасностью Kafka, необходимо принять во внимание безопасность всех составляющих нашего кластера, включая ZooKeeper. Если защитить только брокеры, но не систему, где хранятся данные, связанные с безопасностью, то злоумышленник, обладающий знаниями, сможет без особых усилий изменить эти данные. Для защиты метаданных нужно установить параметр `zookeeper.set.acl` в значение `true` для каждого брокера, как показано в листинге 10.10 [10].

Листинг 10.10. Списки управления доступом в ZooKeeper

```
zookeeper.set.acl=true
```

← Конфигурация любого брокера включает
этот параметр настройки ZooKeeper

10.4.1. Настройка Kerberos

Чтобы убедиться, что ZooKeeper работает с Kerberos, необходимо внести множество изменений в конфигурацию. Прежде всего

в файл конфигурации `zookeeper.properties` нужно добавить значения, сообщающие ZooKeeper, что для взаимодействий с клиентами следует использовать SASL, и указать, какого провайдера использовать. За более подробной информацией обращайтесь по адресу <http://mng.bz/XrOv> [10]. Пока мы были заняты изучением других вариантов настройки, некоторые пользователи нашей системы поиска сокровищ продолжали злоупотреблять своими полномочиями. Давайте посмотрим, сможем ли мы воспрепятствовать злоупотреблениям с помощью квот.

10.5. Квоты

Допустим, некоторые пользователи нашего веб-приложения не испытывают проблем с повторным запросом данных. Обычно нет ничего плохого в том, что конечные пользователи используют службу без всяких ограничений ради решения поставленной перед ними задачи, но нам может потребоваться защитить кластер от пользователей, которые могут использовать его ресурсы в своих личных интересах. В нашем примере мы защищили свои данные, и теперь они доступны только членам нашей команды, но некоторые пользователи из противоположной команды придумали новый способ помешать нам успешно двигаться вперед, фактически предприняв распределенную атаку типа «отказ в обслуживании» (DDoS) против нашей системы [11]!

Целенаправленная атака на кластер может вызвать сбои в наших брокерах и окружающей их инфраструктуре. Другая команда раз за разом запрашивает данные из наших тем, пытаясь извлечь их с самого начала темы. Чтобы воспрепятствовать такому поведению, можно использовать квоты. Здесь важно уточнить, что квоты определяются для каждого брокера отдельно [11]. Кластер не оценивает нагрузку на каждого брокера, чтобы подсчитать итоговую сумму, поэтому квоты должны определяться для каждого брокера отдельно. На рис. 10.5 показан пример использования квот на выполнение запросов, задаваемых в процентах.

Чтобы установить собственные квоты для пользователей, нужно знать, *кого* ограничивать и какое *ограничение* установить. Настойки защиты влияют на доступные варианты, с помощью которых можно определить, кого ограничивать. В отсутствие защиты можно использовать свойство `client.id`. С включенной защитой можно также использовать свойство `user` и использовать любые комбинации `user` и `client.id` [11]. Есть несколько типов квот, которые можно установить для наших клиентов: пропускная способность сети и частота запросов. Рассмотрим сначала ограничение пропускной способности сети.

Все запросы от клиентов с идентификаторами из `kinaction_clueless_secrets` будут обрабатываться с задержкой задержки после слишком большого количества попыток получить данные

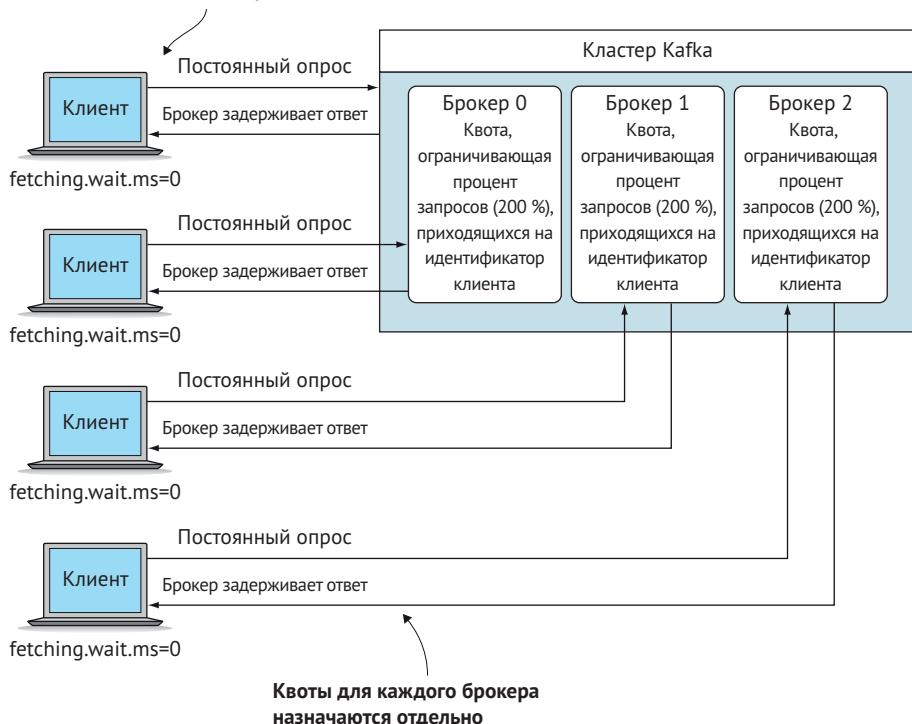


Рис. 10.5. Квоты

10.5.1. Ограничение пропускной способности сети

Пропускная способность сети измеряется количеством байтов в секунду [12]. В этом примере наша цель заставить каждого клиента бережно относиться к сетевому ресурсу и не переполнять сеть своими запросами, чтобы не мешать работе других пользователей. Каждый пользователь в нашем состязании использует идентификатор клиента, присвоенный его команде, который передается во всех запросах на запись или чтение данных. В листинге 10.11 показано, как ограничить количество клиентов, использующих идентификатор `kinaction_clueful`, настроив параметры `producer_byte_rate` и `consumer_byte_rate` [13].

Листинг 10.11. Определение квоты, ограничивающей пропускную способность сети для клиента `kinaction_clueful`

```
bin/kafka-configs.sh --bootstrap-server localhost:9094 --alter \
--add-config 'producer_byte_rate=1048576,
→ consumer_byte_rate=5242880' \
--entity-type clients --entity-name kinaction_clueful
```

Квота распространяется на клиентов с идентификатором `kinaction_clueful`

Производителям разрешено передавать до 1 Мбайт/с, а потребителям читать до 5 Мбайт/с

С помощью параметра `--add-config` мы задали скорость записи и чтения, а параметр `--entity-name` применяет правило к конкретным клиентам с идентификатором `kinaction_clueful`. Как это часто бывает, нам может понадобиться узнать текущие квоты, а также удалить их, если они больше не нужны. Все эти действия можно выполнить, передавая различные аргументы сценарию `kafka-configs.sh`, как показано в листинге 10.12 [13].

Листинг 10.12. Вывод списка и удаление квот, назначенных клиенту `kinaction_clueful`

```
bin/kafka-configs.sh --bootstrap-server localhost:9094 \
--describe \
--entity-type clients --entity-name kinaction_clueful
```

Перечисляет существующие настройки для указанного клиента

```
bin/kafka-configs.sh --bootstrap-server localhost:9094 --alter \
--delete-config
→ 'producer_byte_rate,consumer_byte_rate' \
--entity-type clients --entity-name kinaction_clueful
```

Команда `delete-config` удалит только что добавленную квоту

Команда `--describe` помогает увидеть существующую конфигурацию. На основе полученной с ее помощью информации можно решить, нужно ли изменить или даже удалить конфигурацию с помощью команды `--delete-config`.

В какой-то момент, начав добавлять квоты, может случиться так, что клиенту будет назначено более одной. В таких случаях важно знать, в каком порядке будут применяться различные квоты. Может показаться, что будет действовать самая ограничительная квота, но это не всегда верно. В следующем списке перечислены квоты в порядке убывания их приоритетов [14]:

- назначаемые пользователю и клиенту;
- назначаемые пользователю;
- назначаемые клиенту.

Например, если пользователю с именем Франц назначено ограничение пропускной способности 10 Мбайт/с, а для клиента, которого он использует, – 1 Мбайт/с, то ему будет разрешено потреблять данные со скоростью 10 Мбайт/с согласно квоте, назначенной пользователю, имеющей более высокий приоритет.

10.5.2. Ограничение частоты запросов

Другая квота, о которой следует помнить, ограничивает *частоту запросов*. Зачем нужна вторая квота? Обычно DDoS-атаки рассматриваются как действие, направленное на перегрузку сети, однако клиенты, создающие множество подключений, могут также вызвать перегрузку брокера, выполняя запросы, требующие интенсивных вычислений. Клиенты-потребители, постоянно посылающие запросы с настройкой `fetch.max.wait.ms=0`, тоже способны вызвать проблемы, которые можно решить с помощью ограничения частоты запросов, как показано на рис. 10.5 [15].

Чтобы установить эту квоту, мы используем те же типы сущностей и параметры `--add-config`, что и при установке других квот [13]. Единственная разница заключается в параметре `--request_percentage`. Формулу для вычисления значения на основе количества потоков ввода/вывода и количества сетевых потоков вы найдете на <http://mng.bz/J6Yz> [16]. В листинге 10.13 мы установили ограничение 100 % [13].

Листинг 10.13. Определение квоты, ограничивающей частоту запросов для клиента `kinaction_clueful`

```
bin/kafka-configs.sh --bootstrap-server localhost:9094 --alter \
    --add-config 'request_percentage=100' \
    --entity-type clients --entity-name kinaction_clueful
```

Квота распространяется на клиентов с идентификатором `kinaction_clueful`
Частота запросов на запись ограничивается величиной 100 %

Использование квот – хороший способ защитить кластер. Кроме того, они позволяют утихомирить клиентов, которые внезапно могут начать создавать большую нагрузку на наших брокеров.

10.6. Данные в состоянии покоя

Еще одна вещь, которую следует учесть, – необходимость шифрования данных, которые Kafka записывает на диск. По умолчанию Kafka не шифрует события, которые добавляет в свои журналы. В свое время было подано несколько предложений по улучшению Kafka (Kafka Improvement Proposal, KIP), в которых рассматривалась эта возможность, но на момент публикации книги ни одно из них не было реализовано, поэтому вам придется предусмотреть

свою стратегию, отвечающую вашим требованиям. В зависимости от потребностей вашего бизнеса может понадобиться шифровать только определенные темы или только определенные темы с уникальными ключами.

10.6.1. Управляемые варианты

Если вы используете управляемое окружение для организации своего кластера, то, возможно, стоит проверить, какие возможности предлагает поставщик услуг. Управляемая потоковая передача Amazon для Apache Kafka (<https://aws.amazon.com/msk/>) – один из примеров облачного провайдера, который берет на себя основные хлопоты по управлению вашим кластером, включая некоторые функции безопасности. Обновление ваших брокеров и узлов ZooKeeper с помощью автоматически развертываемых исправлений является одним из основных способов предотвращения проблем. Другое преимущество этих обновлений заключается в отсутствии необходимости предоставлять доступ к кластеру еще большему количеству разработчиков. Amazon MSK также обеспечивает шифрование ваших данных и поддержку протокола TLS для взаимодействий между различными компонентами Kafka [17].

Дополнительные средства управления, рассмотренные в примерах в этой главе, включали возможность использования протокола SSL для взаимодействий между клиентами и кластером и списков управления доступом. Confluent Cloud (<https://www.confluent.io/confluent-cloud/>) – еще один вариант, который можно развернуть в различных облачных окружениях. Также не следует забывать о поддержке шифрования данных, находящихся в состоянии покоя или в движении, и списков управления доступом при сопоставлении ваших требований к безопасности с фактическим предложением провайдера.

Если вы предполагаете придерживаться стека Confluent, то отметим, что в Confluent Platform 5.3 имеется коммерческая функция *secret protection* (защита секретных сведений; <http://mng.bz/yJYB>). Как вы видели выше, настраивая поддержку SSL, мы хранили пароли в открытом виде в определенных файлах. Функция защиты секретных сведений как раз предназначена для решения этой проблемы путем шифрования секретных данных в файле и их получения [18]. Поскольку это коммерческое предложение, мы не будем обсуждать особенности его работы, но просто имейте в виду, что такая возможность доступна.

Итоги

- Передача данных в открытом виде вполне подходит для этапа создания прототипа, но перед внедрением в производство необходимо оценить ее допустимость.

- Протокол SSL (Secure Sockets Layer – слой защищенных сокетов) может помочь защитить данные, передаваемые между клиентами и брокерами и даже между брокерами.
- Для идентификации принципалов можно использовать серверы Kerberos, уже существующие в инфраструктуре.
- Списки управления доступом (Access Control List, ACL) помогают определить, каким пользователям какие операции разрешены. Управление доступом на основе ролей (Role-Based Access Control, RBAC) – еще один вариант, который поддерживается в Confluent Platform. RBAC обеспечивает возможность управления доступом на основе ролей.
- Для ограничения пропускной способности сети и частоты запросов с целью защиты доступных ресурсов кластера можно использовать квоты. Их можно изменять и настраивать, чтобы обеспечить обработку нормальных и пиковых рабочих нагрузок.

Ссылки

- 1 «Encryption and Authentication with SSL». Confluent documentation (n.d.). https://docs.confluent.io/platform/current/kafka/authentication_ssl.html (доступно по состоянию на 10 июня 2020).
- 2 «Adding security to a running cluster». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/kafka/incremental-security-upgrade.html#adding-security-to-a-running-cluster> (доступно по состоянию на 20 августа 2021).
- 3 «Security Tutorial». Confluent documentation (n.d.). https://docs.confluent.io/platform/current/security/security_tutorial.html (доступно по состоянию на 10 июня 2020).
- 4 keytool. Oracle Java documentation (n.d.). <https://docs.oracle.com/javase/8/docs/technotes/tools/unix/keytool.html> (доступно по состоянию на 20 августа 2021).
- 5 «Documentation: Incorporating Security Features in a Running Cluster». Apache Software Foundation (n.d.). http://kafka.apache.org/24/documentation.html#security_rolling_upgrade (доступно по состоянию на 1 июня 2020).
- 6 V. A. Brennen. «An Overview of a Kerberos Infrastructure». Kerberos Infrastructure HOWTO. <https://tldp.org/HOWTO/Kerberos-Infrastructure-HOWTO/overview.html> (доступно по состоянию на 22 июля 2021).
- 7 «Configuring GSSAP». Confluent documentation (n.d.). https://docs.confluent.io/platform/current/kafka/authentication_sasl/authentication_sasl_gssapi.html (доступно по состоянию на 10 июня 2020).

- 8 «Authorization using ACLs». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/kafka/authorization.html> (доступно по состоянию на 10 июня 2020).
- 9 «Authorization using Role-Based Access». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/security/rbac/index.html> (доступно по состоянию на 10 июня 2020).
- 10 «ZooKeeper Security». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/security/zk-security.html> (доступно по состоянию на 10 июня 2020).
- 11 «Quotas». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/kafka/design.html#quotas> (доступно по состоянию на 21 августа 2021).
- 12 «Network Bandwidth Quotas». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/kafka/design.html#network-bandwidth-quotas> (доступно по состоянию на 21 августа 2021).
- 13 «Setting quotas». Apache Software Foundation (n.d.). <https://kafka.apache.org/documentation/#quotas> (доступно по состоянию на 15 июня 2020).
- 14 «Quota Configuration». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/kafka/design.html#quota-configuration> (доступно по состоянию на 21 августа 2021).
- 15 KIP-124 «Request rate quotas». Wiki for Apache Kafka. Apache Software Foundation (30 марта 2017). <https://cwiki.apache.org/confluence/display/KAFKA/KIP-124--Request+rate+quotas> (доступно по состоянию на 1 июня 2020).
- 16 «Request Rate Quotas». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/kafka/design.html#request-rate-quotas> (доступно по состоянию на 21 августа 2021).
- 17 «Amazon MSK features». Amazon Managed Streaming for Apache Kafka (n.d.). <https://aws.amazon.com/msk/features/> (доступно по состоянию на 23 июля 2021).
- 18 «Secrets Management». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/security/secrets.html> (доступно по состоянию на 21 августа 2021).

11

Регистр схем

Эта глава охватывает следующие темы:

- разработку предлагаемой модели зрелости Kafka;
- схемы значений могут предоставлять данные по мере их изменения;
- обзор Avro и сериализацию данных;
- правила совместимости для изменений в схемах с течением времени.

По мере знакомства со все новыми способами использования Apache Kafka может быть интересным экспериментом вспомнить, как менялись ваши взгляды на Kafka. С развитием предприятия (или даже инструментов) их иногда можно моделировать с использованием *уровней зрелости*. Мартин Фаулер (Martin Fowler) дает отличное объяснение этой методологии в своей статье <https://martinfowler.com/bliki/MaturityModel.html> [1]. У Фаулера также есть хороший пример, объясняющий модель зрелости Ричардсона, в которой рассматривается REST [2]. Дополнительную информацию также можно найти в докладе Леонарда Ричардсона (Leonard Richardson) «Justice Will Take Us Millions Of Intricate Moves: Act Three: The Maturity Heuristic», доступном по адресу <https://www.crummy.com/writing/speaking/2008-QCon/act3.html>¹¹.

¹¹ Текст доклада act3.html распространяется на условиях лицензии Creative Commons License, доступной по адресу <https://creativecommons.org/licenses/by-sa/2.0/legalcode>.

11.1. Предлагаемая модель зрелости Kafka

В следующих разделах мы сосредоточимся на обсуждении уровней зрелости, характерных для Kafka. Для сравнения ознакомьтесь с официальным документом Confluent под названием «Five Stages to Streaming Platform Adoption», в котором представлена другая точка зрения, описывающая пять этапов модели зрелости платформы потоковой передачи с отдельными критериями для каждого этапа [3]. Рассмотрим первый уровень (конечно, как программисты, мы начинаем с уровня 0).

Это упражнение с моделью зрелости поможет нам задуматься о том, как Kafka может стать мощным инструментом для одного приложения или даже основой для всех приложений предприятия, а не простым брокером сообщений. Описываемые далее уровни не являются обязательным руководством к действию, скорее это способ представить, с чего можно было бы начать, а затем расширять использование Kafka. Кому-то эти этапы могут показаться спорными, но мы не претендуем на истину в последней инстанции, а лишь предлагаем примерный путь.

11.1.1. Уровень 0

На этом уровне Kafka используется как сервисная шина предприятия (Enterprise Service Bus, ESB) или как система публикации/подписки (pub/sub). События обеспечивают асинхронную связь между приложениями независимо от того, используем ли мы брокера сообщений, такого как RabbitMQ, или только начинаем воплощать этот шаблон.

Одним из примеров использования является отправка пользователем текстового документа для преобразования в PDF. Приложение сохраняет документ, отправленный пользователем, а затем пересыпает сообщение в тему Kafka. Далее потребитель Kafka читает сообщение, чтобы определить, какие документы необходимо преобразовать в PDF. В этом примере задание может быть передано для обработки серверной системе, которая, как известно пользователю, не возвращает ответ немедленно. На рис. 11.1 показана эта шина сообщений в действии.

Один только этот уровень дает нам преимущество, позволяя отделить систему, чтобы сбой во внешней системе, отправляющей текстовые документы, не влиял на внутреннюю систему. Кроме того, отпадает необходимость обеспечивать одновременную работоспособность обеих систем, чтобы гарантировать успешное получение желаемого результата.

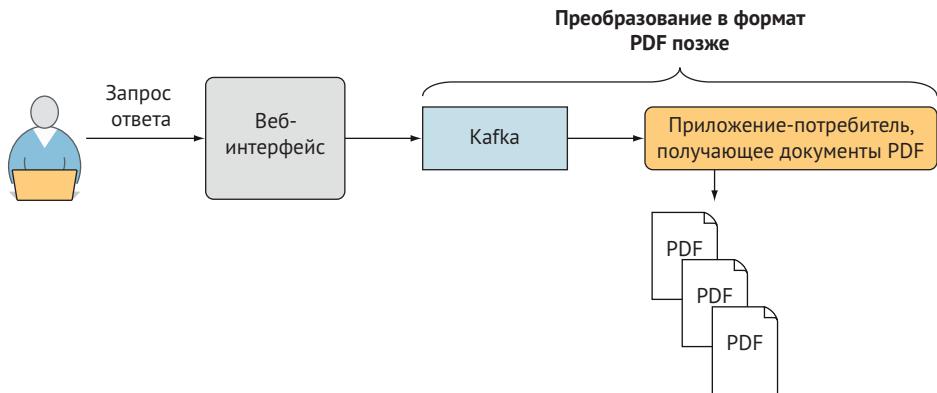


Рис. 11.1. Пример уровня 0

11.1.2. Уровень 1

В некоторых подразделениях предприятия все еще может использоваться пакетная обработка, но большая часть производимых данных теперь переносится в Kafka. Используя процессы извлечения, преобразования, загрузки (ETL) или сбора измененных данных (CDC), Kafka начинает собирать события из все большего числа систем предприятия. Уровень 1 позволяет иметь оперативный поток данных, обрабатываемый в режиме реального времени, и дает возможность быстро передавать данные в аналитические системы.

Примером может служить база данных с информацией о клиентах. Для нас нежелательно, чтобы маркетологи выполняли сложные запросы, способные замедлить производственный трафик. С этой целью мы можем использовать Kafka Connect для записи данных из таблиц базы данных в темы Kafka, которые можно использовать на наших условиях. На рис. 11.2 показано, как Kafka Connect выбирает данные из реляционной базы данных и перемещает эти данные в тему Kafka.

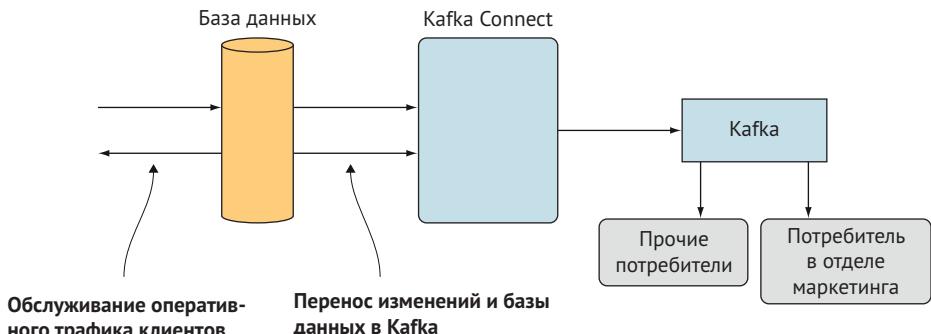


Рис. 11.2. Пример уровня 1

11.1.3. Уровень 2

Мы понимаем, что данные со временем будут меняться, и поэтому необходимы схемы. Наши производители и потребители могут быть отделены друг от друга, но им все равно нужен способ анализа самих данных. Именно с этой целью внедряются схемы и реестры схем. В идеале схемы должны внедряться на самых первых этапах, но реальность такова, что эта необходимость часто возникает лишь спустя несколько итераций развития приложения после первоначального развертывания.

Одним из примеров этого уровня является изменение структуры данных события для получения заказов от системы обработки. Новые данные добавляются, но новые поля необязательны, и такой подход работает нормально, потому что наш реестр схем настроен на поддержку обратной совместимости. На рис. 11.3 показана потребность потребителя в схемах. Мы подробнее рассмотрим эти детали далее в этой главе.

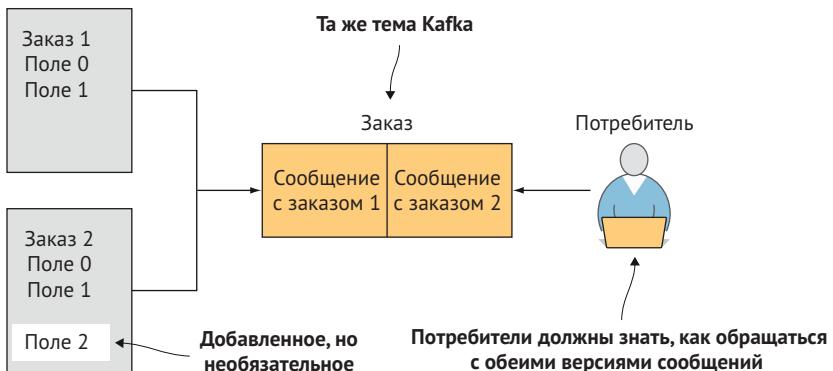


Рис. 11.3. Пример уровня 2

11.1.4. Уровень 3

Все сущее является бесконечным потоком событий. Kafka – это система нашего предприятия, обслуживающая приложения, основанные на событиях. Другими словами, у большинства нас нет клиентов, ожидающих рекомендаций или отчетов о состоянии, которые раньше создавались в ходе ночной пакетной обработки. Клиенты уведомляются об изменении за доли секунд, а не за минуты. Теперь данные извлекаются не из других источников, а непосредственно из кластера. Пользовательские приложения могут получать состояние и материализованные представления в зависимости от потребностей основной инфраструктуры Kafka.

11.2. Реестр схем

В рамках этой главы мы сосредоточимся на уровне 2 и посмотрим, как можно планировать изменение данных с течением времени. Мы научились отправлять данные в Kafka и извлекать их из нее, но, несмотря на краткое упоминание схем в главе 3, мы пока не затронули некоторые важные детали. Поэтому теперь углубимся в исследование возможностей, предоставляемых Confluent Schema Registry.

Реестр Confluent Schema Registry будет хранить наши именованные схемы и позволит поддерживать несколько версий [4]. Он чем-то похож на реестр Docker Registry, в котором хранятся и откуда распространяются образы Docker. Зачем нужно это хранилище? Производители и потребители не связаны друг с другом, но им все равно нужен способ обнаружения схемы, связанной с данными. Кроме того, благодаря удаленному размещению реестра пользователям не придется запускать свою копию реестра локально или пытаться создать свой реестр на основе списка схем.

Схемы могут предоставлять своего рода интерфейс для приложений, но мы также можем использовать их для предотвращения критических изменений [4]. Почему мы должны заботиться о данных, которые быстро перемещаются по нашей системе? Возможности хранения в Kafka позволяют потребителям вернуться к обработке ранних сообщений. Эти сообщения могут быть отправлены несколько месяцев назад (или больше), и нашим потребителям необходимо иметь возможность обрабатывать эти различные версии данных.

В роли реестра схем для Kafka можно использовать Confluent Schema Registry – отличный инструмент для работы со схемами. Если вы установили Kafka через Confluent Platform до того, как приступили к чтению этой главы, то у вас должны иметься все необходимые инструменты для дальнейшего изучения. Если нет, то мы обсудим установку и настройку этого реестра в следующих разделах.

11.2.1. Установка Confluent Schema Registry

Реестр схем Confluent Schema Registry – это программное обеспечение, предлагаемое сообществом как составная часть Confluent Platform [5]. Реестр схем находится за пределами Kafka Brokers, но сам использует тему `_schemas` Kafka в качестве уровня хранения [6]. Не удалите эту тему случайно! При использовании в промышленном окружении реестр схем должен размещаться на сервере, отдельном от брокеров, как показано на рис. 11.4 [6]. Поскольку мы имеем дело с распределенной системой и научились предвидеть сбои, мы можем предоставить несколько экземпляров реестра.

А поскольку все узлы могут обрабатывать поисковые запросы клиентов и пересыпать запросы на запись первичному узлу, клиентам реестра не нужно вести список конкретных узлов.

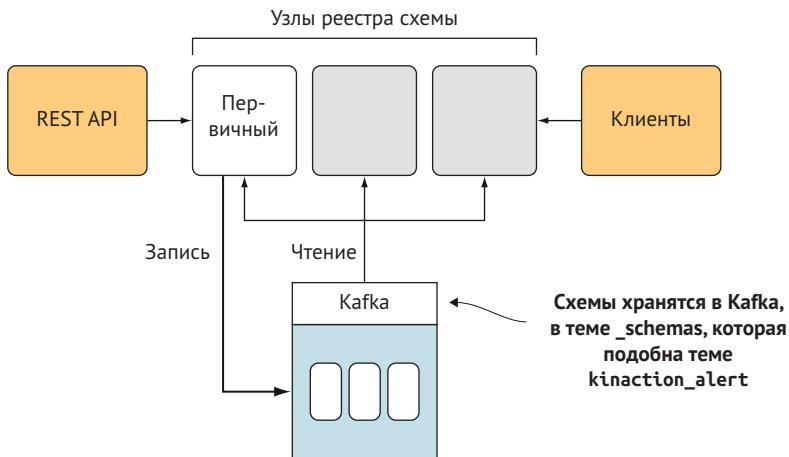


Рис. 11.4. Организация реестра схем

11.2.2. Конфигурация реестра

По аналогии с другими компонентами Kafka настройки реестра можно разместить в конфигурационном файле. Значения по умолчанию можно увидеть в файле `etc/schema-registry/schema-registry.properties`. Для успешной работы реестр должен знать, в какой теме хранить схемы и как работать с данным конкретным кластером Kafka.

В листинге 11.1 мы используем ZooKeeper для выбора основного узла. Важно отметить это, потому что только первый узел записывает сообщения в тему Kafka. Если ваша команда пытается отказаться от использования ZooKeeper, то можно использовать механизм выбора ведущей реплики в Kafka (как показано в конфигурации `kafkastore.bootstrap.servers`) [7].

Листинг 11.1. Конфигурация реестра схем

```
listeners=http://localhost:8081           ← Обслуживает наш реестр на порту
                                         с номером 8081
→ kafkastore.connection.url=localhost:2181
                                         ← Ссылка на наш сервер ZooKeeper
kafkastore.topic=_schemas               ← Использует для хранения схемы тему по умолчанию,
                                         но при необходимости ее можно изменить
debug=true                                ← Это отладочный флаг. Ему можно присвоить
                                         значение false, чтобы избавиться от дополнительной
                                         информации об ошибках
```

Давайте продолжим и запустим реестр схем. При этом не забудьте убедиться, что брокеры ZooKeeper и Kafka уже запущены для

опробования наших примеров. После этого можно использовать командную строку для запуска сценария запуска реестра, как показано в листинге 11.2 [8].

Листинг 11.2. Запуск Schema Registry

```
bin/schema-registry-start.sh \ ↪ Вызов сценария запуска в каталоге bin  
    .etc/schema-registry/schema-registry.properties ↪ установки  
                                                с файлом свойств,  
                                                которые можно из-  
                                                менять
```

Для проверки запуска процесса можно использовать `jps`, потому что это приложение Java, точно так же как брокеры и ZooKeeper. Теперь, запустив реестр, посмотрим, как использовать компоненты системы. Поскольку теперь у нас появилось место для хранения формата данных, вернемся к схеме, которую мы использовали в главе 3.

11.3. Компоненты реестра схем

Реестр Confluent Schema Registry содержит несколько важных компонентов, один из которых – REST API (и реализующее его приложение), предназначенный для хранения и выборки схем. Второй компонент – клиентские библиотеки для управления локальными схемами. В следующих разделах мы подробнее рассмотрим каждый из этих двух компонентов и начнем с REST API.

11.3.1. REST API

REST API помогает управлять следующими ресурсами: *схемами, субъектами, совместимостью и конфигурацией* [9]. Из всех этих ресурсов субъекты больше всего, пожалуй, нуждаются в дополнительных пояснениях. Мы можем создавать, извлекать и удалять версии и сами субъекты. Давайте рассмотрим тему с именем `kinaction_schematest` и связанного с нею субъекта.

В реестре схем у нас будет субъект с именем `kinaction_schema-test-value`, потому что мы используем поведение по умолчанию, основанное на имени текущей темы. Если бы мы дополнительно использовали схему для ключа сообщения, то у нас также был бы субъект с именем `kinaction_schematest-key`. Обратите внимание, что ключ и значение рассматриваются как разные субъекты [10]. Но зачем они нужны? Субъекты гарантируют возможность управления версиями и изменения схем независимо друг от друга, потому что ключ и значение сериализуются отдельно.

Чтобы убедиться, что реестр запущен и работает, отправим запрос GET с помощью `curl` [9]. В листинге 11.3 мы пытаемся получить текущую конфигурацию, такую как уровень совместимости.

Листинг 11.3. Получение конфигурации Schema Registry

```
curl -X GET http://localhost:8081/config
```

Получение конфигурации
Registry с помощью REST API

Также, взаимодействуя с Schema Registry REST API, нужно добавить заголовок Content-Type. Во всех следующих примерах, таких как листинг 11.7, мы будем использовать значение application/vnd.schema.registry.v1+json в этом заголовке [9]. Так же как в случае с самими схемами, мы планируем изменять API, объявляя используемую версию. Это поможет гарантировать использование нашими клиентами определенной версии.

REST API отлично подходит для администраторов субъектов и схем, а клиентская библиотека – это то, с чем чаще всего будут иметь дело разработчики, взаимодействуя с реестром.

11.3.2. Клиентская библиотека

Рассмотрим поближе взаимодействие клиента-производителя с реестром схем. Вспомним пример из главы 3 с производителем, настроенным на использование механизма сериализации Avro для обработки наших сообщений. У нас уже запущен локальный реестр, поэтому теперь нужно настроить клиента-производителя для его использования (листинг 11.4). Для примера из главы 3 мы создали схему для объекта Alert, который является значением сообщения. В нашем случае в свойстве value.serializer должен быть указан класс KafkaAvroSerializer. Он сериализует пользовательский объект с помощью реестра.

Листинг 11.4. Производитель, использующий механизм сериализации Avro

```
...
kaProperties.put("key.serializer",
  "org.apache.kafka.common.serialization.LongSerializer");
kaProperties.put("value.serializer",           ←
  "io.confluent.kafka.serializers.KafkaAvroSerializer");
kaProperties.put("schema.registry.url",
  "http://localhost:8081");           ←
Producer<Long, Alert> producer =           ←
  new KafkaProducer<Long, Alert>(kaProperties);
Alert alert = new Alert();
alert.setSensorId(12345L);
alert.setTime(Calendar.getInstance().getTimeInMillis());
alert.setStatus(alert_status.Critical);
log.info("kinaction_info = {}", alert.toString());
```

Отправляет Alert
как значение и
использует Kaf-
kaAvroSerializer

Настраивает URL реестра, содержащего
историю версий наших схем, чтобы помочь
с проверкой и развитием схемы

```
ProducerRecord<Long, Alert> producerRecord =
    ➔ new ProducerRecord<Long, Alert>(
        "kinaction_schematest", alert.getSensorId(), alert
    );
producer.send(producerRecord);
```

ПРИМЕЧАНИЕ. Поскольку мы используем TopicNameStrategy по умолчанию, Schema Registry регистрирует субъекта kinaction_schematest-value с нашей схемой для Alert. Чтобы использовать другую стратегию, клиент-производитель может установить один из следующих конфигурационных параметров, управляющих стратегиями значений и ключей: value.subject.name.strategy и key.subject.name.strategy [10]. В нашем случае мы могли бы потребовать использовать символы подчеркивания вместо дефисов, чтобы имя нашей темы не содержало сочетаний дефисов и подчеркиваний.

На стороне потребителя, успешно найдя схему, клиент сможет извлекать данные из сообщений, которые он читает. Давайте посмотрим, как используется та же схема, которую мы создали для темы, и получим ее на стороне потребителя, чтобы увидеть, сможем ли мы получить значение из сообщения без ошибок (листинг 11.5) [11].

Листинг 11.5. Потребитель, использующий механизм десериализации Avro

```
kaProperties.put("key.deserializer",
    ➔ "org.apache.kafka.common.serialization.LongDeserializer");
kaProperties.put("value.deserializer",
    ➔ "io.confluent.kafka.serializers.KafkaAvroDeserializer");           ↗
kaProperties.put("schema.registry.url",
    ➔ "http://localhost:8081");   ↗ URL нашего реестра
...
KafkaConsumer<Long, Alert> consumer =
    ➔ new KafkaConsumer<Long, Alert>(kaProperties);

consumer.subscribe(List.of("kinaction_schematest"));           ↗

while (keepConsuming) {
    ConsumerRecords<Long, Alert> records =
    ➔ consumer.poll(Duration.ofMillis(250));
    for (ConsumerRecord<Long, Alert> record : records) {
        log.info("kinaction_info Alert Content = {},",
        ➔         record.value().toString());
    }
}
```

Настраивает использование KafkaAvroDeserializer на стороне потребителя

Подписывается на ту же тему, в которую мы опубликовали наши тестовые сообщения для проверки использования схемы

До сих пор наши производители и потребители работали только с одной версией схемы. Однако планирование изменений данных может избавить вас от многих неприятностей. Далее мы рассмотрим правила, которые помогут нам вносить изменения в схемы в будущем и уменьшить их отрицательное влияние на наших клиентов.

11.4. Правила совместимости

Выбор поддерживаемой стратегии совместимости – важный шаг. Правила совместимости, описываемые в этом разделе, призваны помочь в управлении схемами по мере их изменения с течением времени. Может показаться, что существует большое количество типов совместимости, однако многие из тех, что отмечены как *переходные* (transitive), подчиняются тем же правилам, что и типы без этого суффикса. Для непереходных типов проверяется только последняя версия схемы, тогда как для переходных проверяются все предыдущие версии [12]. Вот список типов, определяемых в Confluent: BACKWARD (тип по умолчанию), BACKWARD_TRANSITIVE, FORWARD, FORWARD_TRANSITIVE, FULL, FULL_TRANSITIVE и NONE [12].

Давайте посмотрим, что означает тип BACKWARD для наших приложений. Изменения с обратной (backward) совместимостью могут включать добавление необязательных полей или удаление полей [12]. Другим важным аспектом, который следует учитывать при выборе типа совместимости, является порядок, в котором клиенты должны изменяться. Например, часто желательно, чтобы клиенты-потребители сначала обновились для типа BACKWARD [12]. Потребители должны знать, как читать сообщения, прежде чем будут созданы новые варианты сообщений.

На другом конце линейки типов находятся прямо (forward) совместимые изменения. Тип FORWARD позволяет добавлять новые поля, и, в отличие от обновления типа BACKWARD, часто желательно сначала обновить клиентов-производителей [12].

Давайте посмотрим, как можно изменить нашу схему Alert, чтобы сохранить обратную совместимость. В листинге 11.6 в схему добавляется новое поле recovery_details со значением по умолчанию "Analyst recovery needed", которое необходимо для учета сообщений, не содержащих значения для нового поля.

Листинг 11.6. Изменение схемы Alert

```
{"name": "Alert",
...
"fields": [
    {"name": "sensor_id", "type": "long",
     "doc": "The unique id that identifies the sensor"},
```

```
...
  {"name": "recovery_details", "type": "string", <-- В этот экземпляр
   "default": "Analyst recovery needed"} добавлено новое
} поле (recovery_
details)
```

Любые более старые сообщения с версией 1 схемы получат значение по умолчанию для поля, добавленного позже. Оно будет прочитано потребителем, использующим схему с версией 2 [12].

11.4.1. Проверка изменений схемы

Если у вас имеются тесты, проверяющие конечные точки API или даже такой инструмент, как Swagger (<https://swagger.io/>), то важно подумать о том, как автоматизировать тестирование изменений в схемах. Проверить и подтвердить изменения в схеме можно несколькими способами:

- использовать конечные точки REST API ресурсов совместимости;
- использовать плагин Maven для приложений на основе JVM.

Рассмотрим пример REST-вызова, который поможет проверить совместимость изменения схемы. В листинге 11.7 показано, как это делается [13]. В качестве примечания: перед проверкой совместимости необходимо иметь в реестре копию старой схемы. Если ее нет и вызов не удался, загляните в примеры с исходным кодом для этой книги.

Листинг 11.7. Проверка совместимости с использованием Schema Registry REST API

```
curl -X POST -H "Content-Type: application/vnd.schemaregistry.v1+json" \
      --data '{ "schema": "{ \"type\": \"record\", \"name\": \"Alert\", \
      \"fields\": [{ \"name\": \"notafield\", \"type\": \"long\" } ]}" }' \
      http://localhost:8081/compatibility/subjects/kinaction_schematest-value/ \
      versions/latest
{"is_compatible":false} <-- Результат проверки совместимости в виде логического значения
```

Передача
содержи-
мого схе-
мы в ко-
мандной
строке

Результат проверки совместимости в виде логического значения

Также можно использовать плагин Maven, если вы уже используете Maven и работаете на платформе JVM [14]. В листинге 11.8 показана часть содержимого файла *pom.xml*, необходимая для этого подхода, а полный файл можно найти в примерах исходного кода для этой главы.

Листинг 11.8. Проверка совместимости с использованием плагина Maven

```

<plugin>
    <groupId>io.confluent</groupId>
    <artifactId>
        kafka-schema-registry-maven-plugin
    </artifactId>
    <configuration>
        <schemaRegistryUrls>
            <param>http://localhost:8081</param>
        </schemaRegistryUrls>
        <subjects>
            <kinaction_schematest-value>
                src/main/avro/alert_v2.avsc
            </kinaction_schematest-value>
        </subjects>
        <goals>
            <goal>test-compatibility</goal>
        </goals>
    </configuration>
    ...
</plugin>

```

Подключение плагина на Maven

URL реестра схем Schema Registry

Список субъектов для проверки схем в указанном файле

Цель Maven можно вызвать с помощью mvn schema-registry:test-compatibility

Плагин извлекает схемы, находящиеся в указанном файле, и подключается к реестру для проверки схем, уже хранящихся там.

11.5. Альтернатива реестру схем

Поскольку не все проекты начинаются со схем или изменений данных, есть несколько простых решений, позволяющих обойти проблему изменения формата данных. Одним из таких решений является передача данных с изменениями, нарушающими совместимость, в другую тему. В такой ситуации потребители могут обновляться по мере необходимости и начинать извлекать данные из другой темы. Это решение хорошо подходит для случаев, когда не планируется повторно обрабатывать данные. На рис. 11.5 показано переключение на новую тему после извлечения всех старых сообщений из первой темы. На диаграмме текст *o1* означает «обновление 1», а *o2* – «обновление 2» и отмечает измененную логику.

Предположим, мы планируем повторно обрабатывать данные в разных форматах. В этом случае можно создать новую тему, куда переместить преобразованные старые сообщения, существовавшие в исходной теме, и, конечно же, добавлять любые новые сообщения. В преобразовании данных из одной темы в другую может помочь компонент Kafka Streams, который мы обсудим в главе 12.

Схема 1: kinaction_alert



Схема 2: обновленная тема kinaction_alert

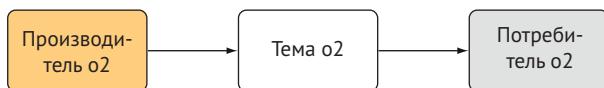


Рис. 11.5. Альтернативный поток данных

Итоги

- Kafka обладает широкими возможностями, подходящими и для простых случаев использования, и для формирования основной системы предприятия.
- Схемы помогают проверять изменения в представлении данных.
- Реестр схем Schema Registry, предлагаемый проектом Confluent, предлагает средства управления схемами, связанными с Kafka.
- При изменении схем правила совместимости помогают узнать, являются ли эти изменения обратно, прямо или полностью совместимыми.
- Если решение на основе схем по какой-то причине вам не подходит, то для обработки разных версий данных можно использовать разные темы.

Ссылки

- 1 M. Fowler. «Maturity Model». (26 августа 2014). <https://martinfowler.com/bliki/MaturityModel.html> (доступно по состоянию на 15 июня 2021).
- 2 M. Fowler. «Richardson Maturity Model». (18 марта 2010). <https://martinfowler.com/articles/richardsonMaturityModel.html> (доступно по состоянию на 15 июня 2021).
- 3 L. Hedderly. «Five Stages to Streaming Platform Adoption». Confluent white paper (2018). <https://www.confluent.io/resources/5-stages-streaming-platform-adoption/> (доступно по состоянию на 15 января 2020).

- 4 «Schema Registry Overview». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/schema-registry/index.html> (доступно по состоянию на 15 июля 2020).
- 5 «Confluent Platform Licenses: Community License». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/installation/license.html#community-license> (доступно по состоянию на 21 августа 2021).
- 6 «Running Schema Registry in Production». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/schema-registry/installation/deployment.html#schema-registry-prod> (доступно по состоянию на 25 апреля 2019).
- 7 «Schema Registry Configuration Options». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/schema-registry/installation/config.html#schemaregistry-config> (доступно по состоянию на 22 августа 2021).
- 8 «Schema Registry and Confluent Cloud». Confluent documentation (n.d.). <https://docs.confluent.io/cloud/current/cp-component/schema-reg-cloudconfig.html> (доступно по состоянию на 22 августа 2021).
- 9 «Schema Registry API Reference». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/schema-registry/develop/api.html> (доступно по состоянию на 15 июля 2020).
- 10 «Formats, Serializers, and Deserializers». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/schema-registry/serdes-develop/index.html> (доступно по состоянию на 25 апреля 2019).
- 11 «On-Premises Schema Registry Tutorial». Confluent documentation (n.d.). https://docs.confluent.io/platform/current/schema-registry/schema_registry_onprem_tutorial.html (доступно по состоянию на 25 апреля 2019).
- 12 «Schema Evolution and Compatibility». Confluent Platform. <https://docs.confluent.io/current/schema-registry/avro.html#compatibility-types> (доступно по состоянию на 1 июня 2020).
- 13 «Schema Registry API Usage Examples». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/schema-registry/develop/using.html> (доступно по состоянию на 22 августа 2021).
- 14 «Schema Registry Maven Plugin». Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/schema-registry/develop/maven-plugin.html> (доступно по состоянию на 16 июля 2020).

12

Потоковая обработка с помощью Kafka Streams и ksqlDB

Эта глава охватывает следующие темы:

- введение в Kafka Streams;
- использование основных API-интерфейсов Kafka Streams;
- использование хранилищ состояний для постоянного хранения;
- добавление дополнительной информации в поток транзакций.

До сих пор основное внимание мы уделяли компонентам Kafka, помогающим создать полноценную платформу потоковой передачи событий, включая брокеров Kafka, клиентов-производителей и клиентов-потребителей. Опираясь на этот фундамент, мы можем расширить наш набор инструментов и подняться на следующий уровень экосистемы Kafka – потоковую обработку с использованием технологий Kafka Streams и ksqlDB, предлагающих свои абстракции, API и DSL (Domain-Specific Languages – предметно-ориентированные языки).

В этой главе представлено простое банковское приложение, которое обрабатывает операции с денежными средствами на

счетах. В нашем приложении мы реализуем топологию Kafka Streams для атомарной обработки запросов, отправленных в тему `transaction-request`.

ПРИМЕЧАНИЕ. В соответствии с нашим бизнес-требованием мы должны проверять, достаточно ли средств на счете для обработки полученного запроса, прежде чем обновить баланс. Также в соответствии с требованиями наше приложение не может одновременно обрабатывать две транзакции для одной и той же учетной записи, потому что может возникнуть состояние гонки из-за невозможности гарантированно проверить баланс перед снятием средств.

Для упорядоченной обработки операций с конкретным счетом мы будем использовать гарантии *упорядочения между разделами* (inter-partition ordering), поддерживаемые в Kafka. У нас также есть программа-генератор данных, которая записывает смоделированные запросы в тему Kafka с номером счета в ключе. С учетом всего этого мы сможем гарантировать, что все транзакции будут обрабатываться одним экземпляром нашей службы независимо от того, сколько экземпляров будет запущено одновременно. Kafka Streams не станет фиксировать смещение сообщения, пока не завершится выполнение нашей бизнес-логики обработки запроса.

Мы введем Processor API, реализовав компонент преобразования из Kafka Streams. Эта утилита позволит обрабатывать события одно за другим, взаимодействуя с хранилищем состояний, еще одним элементом Kafka Streams, помогающим сохранять баланс счета в локальном экземпляре встроенной базы данных RocksDB. Наконец, мы напишем второй потоковый процессор, генерирующий подробный отчет о транзакции, дополненного данными о счете. Вместо создания еще одного приложения Kafka Streams мы объявим потоковый процессор с помощью ksqlDB. Этот процессор будет в режиме реального времени дополнять транзакционные данные сведениями, поступающими из темы `account`.

Цель этого раздела состоит в том, чтобы показать, как можно использовать SQL-подобный язык запросов для создания потоковых процессоров (с функциональностью, аналогичной Kafka Streams) без компиляции и запуска какого-либо кода. А после знакомства с идеей приложений потоковой обработки мы углубимся в детали Kafka Streams API.

12.1. Kafka Streams

Под *потоковой обработкой* подразумевается процесс или приложение, которое обрабатывает непрерывный поток данных и выполняет свою работу по мере поступления этих данных, как обсуж-

далось в главе 2. Это приложение не запускается по расписанию и не запрашивает информацию из базы данных. На основе данных могут быть созданы определенные представления, но мы не ограничены представлением на определенный момент времени. Добро пожаловать в Kafka Streams!

Kafka Streams – это библиотека, а не автономный кластер [1]. Обратите внимание на слово *библиотека*. Это обстоятельство может помочь реализовать потоковую обработку в наших приложениях. Никакой другой инфраструктуры не требуется, кроме необходимости использовать существующий кластер Kafka [2]. Библиотека Kafka Streams является частью приложения на основе JVM.

Отсутствие дополнительных компонентов делает этот API легко тестируемым при запуске нового приложения. Для других окружений может потребоваться больше компонентов управления кластером, но приложения Kafka Streams могут создаваться и развертываться с помощью любых инструментов или платформ, поддерживающих запуск приложений на основе JVM.

ПРИМЕЧАНИЕ. Наше приложение не обязано работать на хостах, где действуют брокеры нашего кластера. Поэтому будем запускать его вне кластера Kafka. Такой подход гарантирует возможность рационального разделения ресурсов между брокерами Kafka и потоковыми процессорами.

Streams API выполняет обработку данных на уровне записей или сообщений [3]. Вам не нужно ждать формирования пакета или откладывать эту работу, если вы заинтересованы в том, чтобы ваша система реагировала на события по мере их появления.

Одним из первых решений, которые необходимо принять на первых этапах разработки приложений, является выбор клиента-производителя/потребителя для библиотеки Kafka Streams. Несмотря на то что Producer API отлично подходит для точного управления передачей данных в Kafka, a Consumer API – для обработки событий, иногда может оказаться излишним реализовать каждый аспект фреймворка потоковой обработки. Поэтому нередко для потоковой обработки данных вместо низкоуровневых API предпочтительнее использовать абстракции, позволяющие эффективно обслуживать наши темы.

Kafka Streams может быть идеальным вариантом, когда требования включают преобразование данных с использованием потенциально сложной логики, извлекающей и возвращающей данные обратно в Kafka. Технология Streams предлагает выбор между функциональным DSL и более императивным Processor API [2]. Давайте сначала взглянем на Kafka Streams DSL.

Предметно-ориентированные языки (Domain-Specific Language, DSL)

Предметно-ориентированные языки (DSL) призваны упростить работу с конкретным предметом. SQL (обычно используемый в базах данных) и HTML (используемый для создания веб-страниц) – хорошие примеры языков, которые можно использовать с DSL (<https://martinfowler.com/dsl.html>). В официальной документации Kafka Streams высоконивневый Kafka Streams API называется предметно-ориентированным языком, однако мы предпочитаем называть его текущим API или, как описывает его Мартин Фаулер (Martin Fowler), текущим интерфейсом (<https://martinfowler.com/.bliki/FluentInterface.html>).

12.1.1. KStreams API DSL

Первым мы рассмотрим KStreams API. Kafka Streams – это система обработки данных, основанная на идее графа, в котором отсутствуют циклы [2]. У такого графа есть начальный и конечный узлы, и данные передаются в направлении от начального к конечному. Попутно узлы (или процессоры) обрабатывают и преобразуют данные. Рассмотрим сценарий, который можно смоделировать как процесс обработки данных в форме графа.

У нас есть приложение, получающее транзакции от платежной системы. В начале графа находится источник этих данных. Поскольку в качестве источника данных мы используем Kafka, нашей отправной точкой будет тема Kafka. Эта исходная точка часто называется *исходным процессором* (или *исходным узлом*). Он запускает обработку, перед ним нет никаких других процессоров. Таким образом, нашим первым примером станет служба, которая принимает транзакции из внешней платежной системы и помещает события в тему.

ПРИМЕЧАНИЕ. Мы смоделируем это поведение с помощью простого приложения-генератора данных.

Событие требует обновить баланс конкретного счета. Результаты процессора транзакций передаются в две темы Kafka: успешные транзакции попадают в тему `transaction-success`, а неудачные – в тему `transaction-failure`. Поскольку это конец пути для нашего небольшого приложения, мы создадим пару процессоров-приемников (или узлов-приемников) для записи в темы `transaction-success` и `transaction-failure`.

ПРИМЕЧАНИЕ. Некоторые узлы-процессоры могут не иметь соединения с узлами-приемниками. В таком случае эти узлы создают побочные эффекты в другом месте (например, выводят информацию в консоль или записывают данные в хранилища состояний) и не требуют отправки данных обратно в Kafka.

На рис. 12.1 показано представление потоков данных в виде ориентированного ациклического графа (Directed Acyclic Graph, DAG), а на рис. 12.2 – как этот график отображается в топологию Kafka Streams.

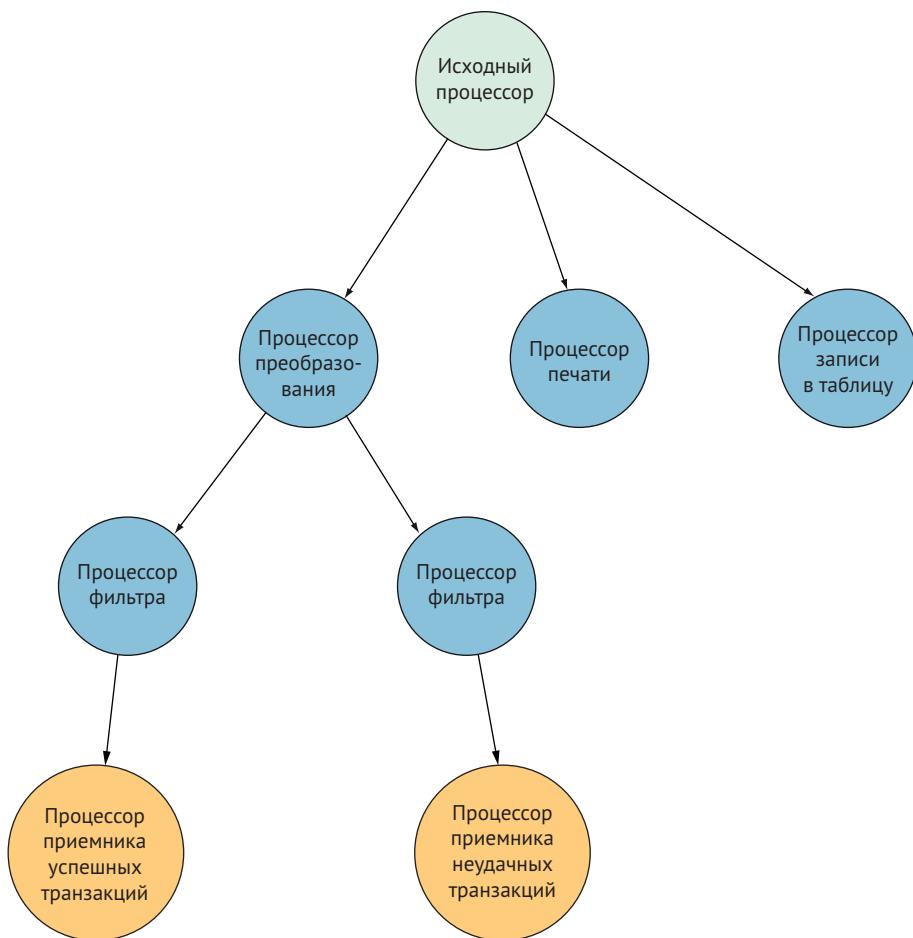


Рис. 12.1. Ориентированный ациклический график, представляющий потоки данных в приложении потоковой обработки

Теперь, получив представление о том, как действует приложение, посмотрим, как выглядит его реализация с кодом DSL. В отличие от предыдущих примеров использование этого API не требует обращаться к потребителю напрямую, чтобы прочитать сообщения, но мы можем использовать построитель, чтобы начать создавать поток. В листинге 12.1 показано создание исходного процессора.

ВАЖНО. На этом этапе мы определяем нашу топологию, но не вызываем ее, так как обработка еще не началась.

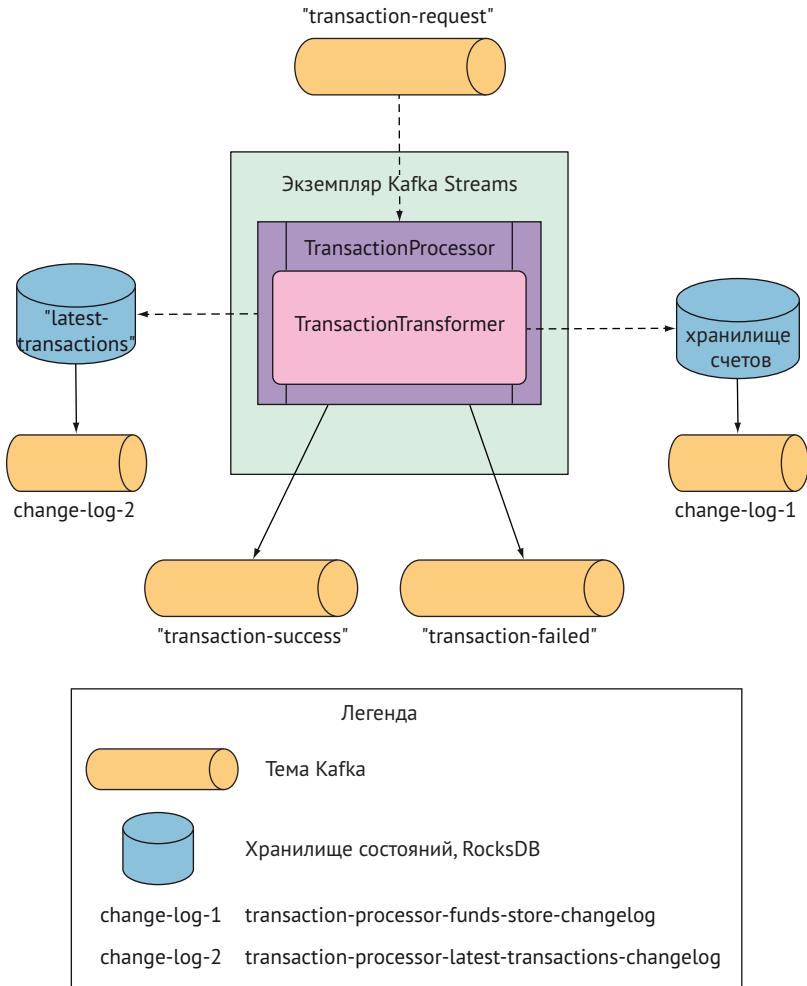


Рис. 12.2. Топология приложения обработки транзакций

В листинге 12.1 мы используем объект `StreamsBuilder` для создания потока из темы Kafka `transaction-request`. Эта тема является нашим источником данных и логической отправной точкой для обработки.

Листинг 12.1. Определение исходной темы на DSL

```
StreamsBuilder builder = new StreamsBuilder() <-- Отправная точка построения нашей топологии
    ➤ KStream<String, Transaction> transactionStream =
        builder.stream("transaction-request",
            Consumed.with(stringSerde, transactionRequestAvroSerde));
```

Создает объект KStream для transaction-request, чтобы начать обработку с этой темы

Следующим шагом будет расширение нашей топологии с помощью экземпляра KStream, созданного из исходного процессора. Этот процесс показан в листинге 12.2.

Листинг 12.2. Определение процессора и темы-приемника

```

final KStream<String, TransactionResult> resultStream =
    transactionStream.transformValues(
        () -> new TransactionTransformer()
    );
resultStream
    .filter(TransactionProcessor::success)
    .to(this.transactionSuccessTopicName,
        Produced.with(Serdes.String(), transactionResultAvroSerde));

resultStream
    .filterNot(TransactionProcessor::success)
    .to(this.transactionFailedTopicName,
        Produced.with(Serdes.String(), transactionResultAvroSerde));

KafkaStreams kafkaStreams = 
    new KafkaStreams(builder.build(), kaProperties);
kafkaStreams.start();
...
kafkaStreams.close();
```

Продолжение конструирования топологии с использованием потока, созданного предыдущим исходным процессором

В зависимости от критериев успеха транзакции наш процессор-приемник выполняет запись данных в одну из двух тем: transaction-success или transaction-failed

Наша топология и конфигурация передаются для создания объекта KafkaStreams

Запускает потоковое приложение, которое действует подобно потребителю, извлекающему события в бесконечном цикле

Закрывает поток, чтобы остановить обработку

Сейчас у нас только один обрабатывающий узел, не выполняющий ни чтения, ни записи данных, однако легко увидеть, как можно связать несколько узлов в один поток. Просматривая код в листинге 12.2, вы могли заметить отсутствие следующих компонентов:

- клиента-потребителя, читающего сообщения из исходной темы, как было показано в главе 5;
- клиента-производителя, отправляющего сообщения в конце потока, как было показано в главе 4.

Этот уровень абстракции позволяет выстраивать логику, а не детали ее функционирования. Рассмотрим еще один практический пример. Представьте, что мы решили просто выводить запросы на выполнение транзакций в консоль, не обрабатывая их. В листинге 12.3 показано, как можно организовать чтение событий из темы transaction-request.

Листинг 12.3. Трассировщик транзакций на основе KStream

```
KStream<String, Transaction> transactionStream = builder.stream("transaction-request", Consumed.with(stringSerde, transactionRequestAvroSerde));

transactionStream.print(Printed.<String, Transaction>toSysOut()
    .withLabel("transactions logger"));

KafkaStreams kafkaStreams = new KafkaStreams(builder.build(), kaProperties);
kafkaStreams.cleanUp();
kafkaStreams.start();
...
```

Получает данные из темы transaction-request и использует объект Transaction для их хранения

Очищает локальное хранилище данных, гарантируя, что работа начнется без прошлого состояния

Выводит запросы на обработку транзакций в консоль по мере их получения, чтобы нам было проще следовать за примером

Этот поток необычайно прост – мы просто выводим транзакции в консоль, но точно так же мы могли бы использовать вызов API для отправки SMS или электронного письма. Обратите внимание на дополнительный вызов `cleanup()` перед запуском приложения. Этот метод позволяет очистить локальные хранилища состояний нашего приложения. Только запомните, что это должно делаться или перед запуском, или после закрытия приложения.

Несмотря на простоту использования KStreams, это не единственный способ обработки данных. KTable API предоставляет другую альтернативу, позволяющую всегда добавлять события, представляя данные как обновления.

12.1.2. KTable API

В отличие от KStream, который всегда добавляет данные о событиях в конец журнала, KTable позволяет представить тему как сжатый журнал [2]. На самом деле мы можем также провести параллель с таблицей базы данных, которая хранит последние обновления. Как рассказывалось в главе 7, где мы обсуждали сжатые темы, чтобы этот прием сработал, наши данные должны иметь ключ. Без ключа обновление значения не имеет практического смысла. Запустив код из листинга 12.4, можно обнаружить, что отображается не каждое событие.

Листинг 12.4. Транзакции в KTable

```
StreamsBuilder builder = new StreamsBuilder();

KTable<String, Transaction> transactionStream = builder.stream("transaction-request",
    Consumed.with(stringSerde, transactionRequestAvroSerde),
    Materialized.as("latest-transactions"));

Записи в KTable материализуются локально в хранилище состояний latest-transactions

KafkaStreams kafkaStreams = new KafkaStreams(builder.build(), kaProperties);
```

StreamsBuilder.table() создает KTable из события в теме transaction-request

Записи в KTable материализуются локально в хранилище состояний latest-transactions

Процесс конструирования потока в этом листинге уже знаком нам. Мы используем построитель для определения шагов, а по окончании вызываем `start`. До этого момента наше приложение ничего не обрабатывает.

12.1.3. GlobalKTable API

`GlobalKTable` похож на `KTable`, но заполняется данными из всех разделов темы [2]. Здесь вам пригодятся базовые знания о темах и разделах, они помогут понять, как экземпляры `KafkaStreams` используют каждый раздел темы. В листинге 12.5 показан пример использования соединения с `GlobalKTable`. Представьте поток, в котором обновляются сведения об отправленном клиенту пакете. События содержат идентификатор клиента, благодаря чему мы можем выполнить соединение с таблицей клиентов, чтобы найти соответствующий адрес электронной почты и отправить сообщение.

Листинг 12.5. Отправка уведомления по электронной почте с помощью `GlobalKTable`

```
...
StreamsBuilder builder = new StreamsBuilder();

final KStream<String, MailingNotif> notifiers =
    builder.stream("kinaction_mailingNotif"); ←
final GlobalKTable<String, Customer> customers =
    builder.globalTable("kinaction_custinfo"); ←

lists.join(customers,
    (mailingNotifID, mailing) -> mailing.getCustomerId(),
    (mailing, customer) -> new Email(mailing, customer))
    .peek((key, email) ->
        emailService.sendMessage(email)); ←

KafkaStreams kafkaStreams = new KafkaStreams(builder.build(), kaProperties);
kafkaStreams.cleanUp();
kafkaStreams.start();
...
```

Поток уведомлений извлекает новые сообщения для отправки клиенту

GlobalKTable содержит список с информацией о клиентах, включая адрес электронной почты

Метод `join` отыскивает соответствие клиенту, которого необходимо уведомить по электронной почте

Как показано в листинге 12.5, создать новый экземпляр `GlobalKTable` можно с помощью метода `globalTable`. Локальная таблица может потреблять не все данные входной темы из-за наличия нескольких разделов, однако глобальная таблица будет потреблять их все [2].

ПРИМЕЧАНИЕ. Идея глобальной таблицы состоит в том, чтобы сделать данные доступными для приложения независимо от того, в какой раздел они отображаются.

Streams DSL отлично подходит для быстрой разработки простых приложений, но иногда может потребоваться более полный контроль над отправкой данных по нашим логическим путям. В таких случаях для получения дополнительных возможностей разработчики могут использовать Processor API отдельно или вместе с Streams DSL.

12.1.4. Processor API

Важно отметить, что при просмотре кода других потоковых приложений или даже при изучении более низких уровней абстракции в нашей собственной логике можно столкнуться с примерами из Processor API. Этот API считается более сложным в использовании, чем DSL, рассмотренный в предыдущих разделах, но он дает больше возможностей [2]. Рассмотрим пример в листинге 12.6, где создается топология, и отметим отличия от наших предыдущих приложений Streams.

Листинг 12.6. Processor API как источник

```
import static org.apache.kafka.streams.Topology.AutoOffsetReset.LATEST;

public static void main(String[] args) throws Exception {
//...
final Serde<String> stringSerde = Serdes.String();
Deserializer<String> stringDeserializer = stringSerde.deserializer();
Serializer<String> stringSerializer = stringSerde.serializer();

Topology topology = new Topology(); <-- Класс для десериализации ключа
topology = topology.addSource(LATEST, <-- Создание потока с помощью объекта Topology
    "kinaction_source", <-- Устанавливает смещение LATEST
    stringDeserializer, <-- Присваивает узлу имя, на которое мы сможем ссылаться в последующих этапах
    stringDeserializer,
    "kinaction_source_topic"); <-- Тема Kafka, из которой извлекаются сообщения
//...
```

Класс для десериализации ключа

Создание потока с помощью объекта Topology
Устанавливает смещение LATEST
Присваивает узлу имя, на которое мы сможем ссылаться в последующих этапах
Класс для десериализации значения

Здесь сначала мы конструируем наш граф, используя объект `Topology` [4]. Установка смещения `LATEST` и выбранные механизмы десериализации ключей и значений уже знакомы вам по настройке конфигурационных свойств наших клиентов-потребителей в главе 5. В листинге 12.6 мы присвоили узлу имя `"kinaction_source"`, который будет читать данные из темы `kinaction_source_topic`. Наш следующий шаг – добавить узел обработки. Он показан в листинге 12.7.

Листинг 12.7. Узел обработки в Processor API

```
topology = topology.addProcessor(
    "kinactionTestProcessor", ← Имя для нашего нового узла обработки
    () -> new TestProcessor(),
    "kinaction_source"); ← Создает экземпляр процессора из ProcessorSupplier
    ↘ Один узел или их список, посылающих данные в этот узел
```

Как показано в листинге 12.7, когда определяется узел обработки, ему присваивается имя (в данном случае "kinactionTestProcessor"), определяется логика его работы и задаются узлы, которые будут поставлять данные.

В завершение нашего простого примера рассмотрим листинг 12.8. В нем показано определение двух простых приемников, завершающих нашу топологию. Приемник – это место, куда помещаются данные после обработки. Название темы и механизмы сериализации ключей и значений должны быть знакомы вам по предыдущим примерам клиентов-производителей. Так же как в случае с другими частями топологии, здесь kinactionTestProcessor определяется как один из узлов, откуда будут извлекаться данные.

Листинг 12.8. Процессор-приемник в Processor API

```
topology = topology.addSink(
    "Kinaction-Destination1-Topic", ← Имя для узла-приемника
    "kinaction_destination1_topic", ← Имя для выходной темы, которую планируется использовать
    stringSerializer, ← Класс для сериализации значения
    stringSerializer, ← узел, служащий источником данных для записи в приемник
    "kinactionTestProcessor"); ← добавляет второй приемник в топологию

topology = topology.addSink(
    "Kinaction-Destination2-Topic", ←
    "kinaction_destination2_topic",
    stringSerializer,
    stringSerializer,
    "kinactionTestProcessor");
```

Класс для сериализации ключа

...

Далее мы хотим показать код процессора, чтобы вы могли увидеть, как можно управлять потоком данных, используя собственную логику. Наш kinactionTestProcessor позволяет перенаправить поток, включая ключ и значение, в приемник с именем Kinaction-Destination2-Topic. В листинге 12.9 имя узла-приемника жестко запрограммировано в коде, однако нам ничто не мешает использовать дополнительную логику, определяющую, когда следует отправлять данные во второй приемник.

Листинг 12.9. Собственная логика управления выбором узла-приемника

```
public class KinactionTestProcessor
    extends AbstractProcessor<String, String> { ←
        ↑
    @Override
    public void process(String key, String value) {
        context().forward(key, value,
            To.child("Kinaction-Destination2-Topic"));
    }
}
```

Наследует `AbstractProcessor` для реализации метода `process` с нашей собственной логикой

Жестко запрограммированное значение, но точно так же можно было бы реализовать свою логику выбора узла-приемника

Для реализации этого примера потребовалось написать больше кода, чем для примера с DSL, однако важно отметить, что Processor API дает более широкие возможности управления потоком, чем DSL API. Если понадобится управлять расписанием обработки или временем фиксации результатов, то вам определенно потребуется изучить более сложные методы Processor API.

12.1.5. Настройка Kafka Streams

В нашем примере приложения используется только один экземпляр, однако потоковые приложения можно масштабировать за счет увеличения количества потоков и развертывания нескольких экземпляров. По аналогии с количеством экземпляров потребителя в одной группе степень параллелизма нашего приложения прямо связана с количеством разделов в исходной теме [5]. Например, если исходная тема имеет восемь разделов, то можно смело масштабировать приложение до восьми экземпляров. Если только вы не собираетесь иметь резервные экземпляры на случай сбоя, то нет нужды запускать экземпляры сверх этого числа, потому что они не будут потреблять трафик.

Продумывая архитектуру приложения, очень важно учесть гарантии обработки для конкретного варианта использования. Kafka Streams поддерживает семантики обработки «не менее одного раза» и «точно один раз».

ПРИМЕЧАНИЕ. Семантика «точно один раз» появилась в версии 2.6.0. Эта версия обеспечивает более высокую пропускную способность и масштабируемость, одновременно стремясь снизить потребление ресурсов [6].

Если логика приложения зависит от семантики «точно один раз», то экосистема Kafka поможет обеспечить ее поддержку. Но в отношении отправки данных во внешние системы обязательно нужно посмотреть, как они обеспечивают гарантии вариантов доставки. Streams API может осуществлять получение дан-

ных из темы, обновление хранилищ и запись результатов в другую тему как одну атомарную операцию, однако взаимодействия с внешними системами не могут производиться атомарно. Границы системы становятся особенно важными, если они влияют на ваши гарантии.

Важно отметить, что семантика «не менее одного раза» гарантирует невозможность потери данных, поэтому вам, возможно, придется подготовиться к ситуации, когда сообщения будут обрабатываться более одного раза. На момент написания этой книги семантика доставки «не менее одного раза» использовалась по умолчанию, поэтому убедитесь, что вы готовы столкнуться с повторяющимися данными в логике вашего приложения.

Технология Kafka Streams разрабатывалась с учетом отказоустойчивости, которая обеспечивается теми же механизмами, которые мы видели, когда исследовали кластер Kafka. Хранилища состояний поддерживаются реплицируемыми темами Kafka, распределенными по нескольким разделам. Благодаря способности Kafka сохранять сообщения и воспроизводить то, что произошло до сбоя, пользователи могут успешно продолжать работу, не заботясь о воссоздании своего состояния вручную. Желающим заняться более глубоким изучением возможностей Kafka Streams мы рекомендуем книгу «*Kafka Streams in Action*» (<https://www.manning.com/books/kafka-streams-in-action>) Уильяма П. Беджека (William P. Bejeck Jr.), вышедшую в издательстве Manning в 2018 году¹².

12.2. *ksqlDB*: база данных потоковой передачи событий

ksqlDB (<https://ksqldb.io>) – это база данных потоковой передачи событий. Этот продукт был впервые представлен как KSQL, но в ноябре 2019 года название проекта было изменено. В рамках проекта Apache Kafka было разработано несколько клиентов, упрощающих работу с данными.

ksqlDB раскрывает возможности Kafka для всех, кто когда-либо использовал SQL. Как ни странно, но для использования тем и данных в наших кластерах не требуется писать код на Java или Scala. Также некоторым приложениям не нужна полная архитектура Kafka и достаточно только части ее возможностей. На рис. 12.3 показан пример одного из подходов к использованию Kafka.

¹² Билл Беджек, «*Kafka Streams в действии. Приложения и микросервисы для работы в реальном времени*». Питер, 2019, ISBN: 978-5-4461-1201-2. – Прим. перев.

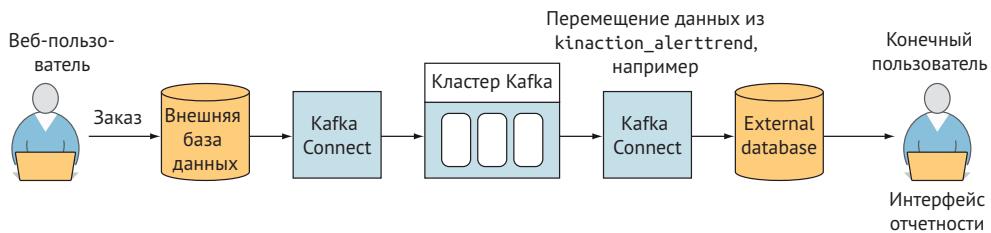


Рис. 12.3. Пример приложения Kafka

Обратите внимание, что для целей обслуживания пользователей данные из Kafka перемещаются во внешнее хранилище. Например, представьте приложение, добавляющее заказ в систему электронной коммерции. Для каждого этапа обработки заказа запускается отдельное событие, которое служит признаком состояния обработки заказа для покупателя.

До появления ksqlDB события обработки заказов часто сохранялись в Kafka (и обрабатывались с помощью Kafka Streams или Apache Spark), а затем передавались во внешнюю систему с помощью Kafka Connect API. Приложение могло читать информацию из представления в этой базе данных, сгенерированного на основе потока событий, чтобы показать пользователю состояние его заказа на определенный момент времени. После добавления в ksqlDB функций pull-запроса (извлечение информации по инициативе клиента) и управления коннекторами разработчики получили возможность оставаться в рамках экосистемы и предоставлять пользователям материализованные представления. На рис. 12.4 показана общая схема, как экосистема Kafka позволяет создавать более консолидированные приложения без использования внешних систем. Далее мы рассмотрим типы запросов, которые поддерживает ksqlDB, и начнем с только что представленных pull-запросов.

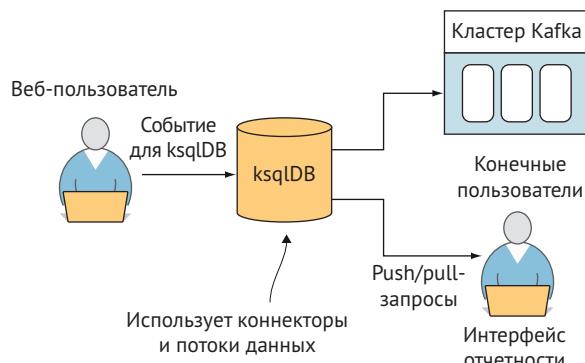


Рис. 12.4. Пример приложения ksqlDB

12.2.1. Запросы

Pull- и push-запросы могут помочь нам в создании приложений. Pull-запросы хорошо подходят для работы в синхронном режиме, например с использованием шаблона запрос/ответ [7]. Мы можем запросить текущее состояние представления, материализованного наступившими событиями. Запрос возвращает ответ и считается завершенным. Большинство разработчиков знакомы с этим шаблоном и должны знать, что возвращаемые данные представляют моментальный снимок событий на момент получения запроса.

Push-запросы (передача информации по инициативе сервера), напротив, особенно хорошо подходят для работы в асинхронном режиме [7]. По сути, в этом случае мы подписываемся на получение информации, как мы делали это в клиентах-потребителях. По мере поступления новых событий наш код может реагировать, выполняя необходимые действия.

12.2.2. Локальная разработка

Мы старались не использовать дополнительные технологии, кроме Kafka, но для опробования ksqlDB намного проще использовать локальную версию в образах Docker от Confluent. Образы, доступные по адресу <https://ksqldb.io/quickstart.html>, включают полный набор компонентов Kafka или только файлы *ksqldb-server* и *ksqldb-cli*.

Образы Docker можно запустить командой `docker-compose up`. После запуска у вас появится возможность использовать *ksqldb-cli* для создания интерактивного сеанса KSQL в окне терминала. Как известно, после установки, настройки и запуска сервера базы данных необходимо определить данные. Дополнительные сведения о запуске Kafka и инструментах для работы с Docker вы найдете в приложении А. В листинге 12.10 показана команда, использующая Docker для запуска интерактивного сеанса ksqlDB [8].

Листинг 12.10. Запуск интерактивного сеанса ksqlDB

```
docker exec -it ksqldb-cli \
  ksql http://ksqldb-server:8088 <-- Подключение к серверу ksqlDB для
> SET 'auto.offset.reset'='earliest'; <-- создания интерактивного сеанса
                                            Установка политики сброса
                                            смешения в самую раннюю по-
                                            зицию, что позволяет ksqlDB
                                            обрабатывать данные, уже до-
                                            ступные в темах Kafka
```

Рассмотрим далее пример использования ksqlDB для расширения нашего обработчика транзакций. Используя существующую информацию об обработанных транзакциях, мы можем сгенерировать *отчет с выпиской из банковского счета*. Такой отчет включает дополнительные сведения о счете, с которым выполнялась

транзакция. Для этого мы объединим успешные транзакции с данными о счете. Для начала создадим поток успешных транзакций из темы Kafka.

ПРИМЕЧАНИЕ. Поскольку ранее данные, доступные в теме Kafka, использовались нашим приложением Kafka Streams, может потребоваться сбросить смещение командой SET ‘auto.offset.reset’ = ‘earliest’; чтобы дать возможность ksqlDB обработать существующие данные. Эту команду также нужно будет запустить перед выполнением оператора CREATE.

В листинге 12.11 показан наш следующий шаг в этом процессе – создание потока для успешных транзакций, источником для которого служит тема `transaction-success`.

Листинг 12.11. Создание потока для успешных транзакций

```
CREATE STREAM TRANSACTION_SUCCESS ( numkey string KEY, transaction STRUCT<guid STRING, account STRING, amount DECIMAL(9, 2), type STRING, currency STRING, country STRING>, funds STRUCT<account STRING, balance DECIMAL(9, 2)>, success boolean, errorType STRING ) WITH ( KAFKA_TOPIC='transaction-success', VALUE_FORMAT='avro');
```

Сообщить ksqlDB ключ записи

ksqlDB поддерживает работу с вложенными данными

Атрибут KAFKA_TOPIC в предложении WITH задает тему для чтения

Интеграция ksqlDB со схемами в Avro

Поскольку ksqlDB поддерживает работу с вложенными данными, мы использовали в нашем примере с Kafka Streams вложенный тип `Transaction` в классе `TransactionResult`. С помощью ключевого слова `STRUCT` мы определили структуру вложенного типа. Кроме того, ksqlDB интегрируется с реестром Confluent Schema Registry и изначально поддерживает схемы в форматах Avro, Protobuf, JSON и JSON-schema. Используя эту интеграцию с реестром схем, ksqlDB во многих случаях может использовать схемы для определения структуры потоков или таблиц. Это, в частности, здорово помогает обеспечить эффективное взаимодействие между микросервисами.

Как уже упоминалось, нам нужна исчерпывающая информация о счетах. В отличие от истории успешных транзакций нас не интересует полная история изменения состояния счета. Нам просто нужна возможность отыскивать счета по их идентификаторам. Для этой цели можно использовать TABLE в ksqlDB. В листинге 12.12 показано, как это сделать.

Листинг 12.12. Создание таблицы в *ksqldb*

```
CREATE TABLE ACCOUNT (number INT PRIMARY KEY) ←
WITH (KAFKA_TOPIC = 'account', VALUE_FORMAT='avro'); ←
```

Поле номера счета
назначается на роль
первичного ключа
нашей таблицы

Используя схему Avro, *ksqldb*
может получить информацию
о полях в таблице account

Следующий шаг – заполнение нашей таблицы. Оператор SQL в листинге 12.13 похож на операторы SQL обычных баз данных, которые вы могли запускать в прошлом, однако обратите внимание на небольшое, но существенное отличие. `EMIT CHANGES` создает то, что выше мы называли push-запросом. Вместо возврата в командную строку этот поток продолжит работать в фоновом режиме!

Листинг 12.13. Поток транзакции с информацией о счетах

```
CREATE STREAM TRANSACTION_STATEMENT AS
    SELECT *
    FROM TRANSACTION_SUCCESS
    LEFT JOIN ACCOUNT
        ON TRANSACTION_SUCCESS.numkey = ACCOUNT.numkey
    EMIT CHANGES;
```

Чтобы протестировать этот запрос, понадобится запустить новый экземпляр *ksqldb-cli*, который будет добавлять в наш поток тестовые транзакции. Они будут обрабатываться приложением Kafka Streams. В случае успеха процессор Kafka Streams запишет результат в тему `transaction-success`, откуда он будет получен *ksqldb* и использован в потоках `TRANSACTION_SUCCESS` и `TRANSACTION_STATEMENT`.

12.2.3. Архитектура *ksqldb*

Использовав образы Docker, мы замаскировали архитектуру, являющуюся частью *ksqldb*. Однако важно знать, что, в отличие от Streams API, для работы *ksqldb* требуются дополнительные компоненты. Основной компонент – *сервер ksqldb* [9]. Он отвечает за выполнение отправленных ему SQL-запросов и передачу данных в кластер Kafka и из него. В дополнение к механизму запросов также предоставляется REST API. Этот API используется командой *ksqldb-cli*, которую мы демонстрировали в примерах [9].

Еще один момент, который следует рассмотреть, – это один из режимов развертывания, называемый *автономным режимом* (*headless mode*). Он не позволяет разработчикам выполнять запросы через интерфейс командной строки [10]. Чтобы включить этот режим, можно запустить сервер *ksqldb* с аргументом командной строки `--queries-file` или изменить настройки в файле *ksql-server*.

properties [10]. Конечно, это также означает необходимость создать файл запроса. В листинге 12.14 показано, как запустить ksqlDB в таком автономном режиме [10].

Листинг 12.14. Запуск ksqlDB в автономном режиме

```
bin/ksql-server-start.sh \
etc/ksql/ksql-server.properties --queries-file kinaction.sql
```

Запуск ksqlDB в неинтерактивном режиме
без поддержки интерфейса командной строки

Теперь, попробовав Kafka Streams и ksqlDB, как узнать, какой из этих компонентов лучше подходит для решения тех или иных задач? Конечно, команда *ksqldb-cli* не является интерактивной оболочкой REPL (read-eval-print loop – прочитать-выполнить-напечатать-повторить), однако с ее помощью можно быстро опробовать прототипы, что может стать отличным предварительным этапом в разработке новых приложений. Еще одно важное преимущество ksqlDB заключается в том, что пользователи, не применяющие Java или Scala (языки JVM), могут получить доступ к возможностям Kafka Streams с помощью этого диалекта SQL. Однако те, кто занимается созданием микросервисов, скорее всего, предпочтут Streams API.

12.3. Куда пойти дальше

Мы только что познакомились с Kafka Streams и ksqlDB, однако есть масса других ресурсов, которые помогут вам продолжить изучение Kafka. В следующих разделах мы расскажем о некоторых из них.

12.3.1. Предложения по улучшению Kafka (KIP)

Изучение предложений по улучшению Kafka (Kafka Improvement Proposals, KIP) может показаться не самым захватывающим занятием, однако это один из лучших способов оставаться в курсе событий Kafka. Конечно, реализуются далеко не все предложения, тем не менее всегда интересно посмотреть, что другие пользователи Kafka считают достойным изучения, потому что варианты использования меняются со временем.

Как мы видели в главе 5, предложение KIP 392 (<http://mng.bz/n2aV>) было обусловлено необходимостью получения данных пользователями, когда лидер раздела находится в удаленном центре обработки данных. Если бы Kafka действовала только в локальных центрах обработки данных, без использования отдельных центров обработки данных для аварийного восстановления, предложение могло бы не получить одобрения. Просмотр подобных новых предложений KIP позволяет каждому понять проблемы или особенности, с которыми другие пользователи Kafka сталкиваются в своей повседневной практике.

Предложения KIP достаточно важны, чтобы их рассматривать и обсуждать на основных мероприятиях, таких как Kafka Summit 2019, где было представлено предложение KIP 500 (<http://mng.bz/8WvD>) с обоснованием замены ZooKeeper.

12.3.2. Проекты *Kafka*, которые вы можете исследовать

Помимо исходного кода Kafka исследование проектов в общедоступных репозиториях GitHub или GitLab может помочь вам извлечь дополнительные уроки. Конечно, не все проекты отличаются высококачественным кодом, но мы надеемся, что сумели в предыдущих главах дать вам достаточно информации, чтобы вы могли самостоятельно исследовать его и понять, что к чему. В этой книге упоминается несколько проектов, в которых Kafka частично используется для улучшения программного обеспечения, и их исходный код доступен для просмотра на GitHub. Одним из примеров был Apache Flume (<https://github.com/apache/flume>).

12.3.3. Каналы сообщества Slack

Если вам нравится более интерактивный способ сбора информации, то отличным местом для поиска ответов на вопросы вам послужит страница сообщества Confluent Community (<https://www.confluent.io/community/>). Здесь вы найдете группу Slack с каналами, посвященными конкретным компонентам Kafka, таким как клиенты, Connect и многим другим темам, связанным с Kafka. Количество подробных вопросов и ответов, опубликованных другими (и которые вы тоже можете опубликовать), показывает широту опыта, приобретенного пользователями, готовыми им поделиться. Существует также форум сообщества, где можно зарегистрироваться и познакомиться с другими активными участниками.

В этой главе вы приобрели дополнительные знания об абстракциях KStreams и ksqlDB и смогли соотнести их с вашими базовыми знаниями о Kafka. Экосистема Kafka продолжает развиваться, изменяться и расширяться, поэтому мы уверены, что представленные в этой книге основы Kafka помогут вам понять, что происходит внутри. Успехов вам в дальнейшем изучении Kafka!

Итоги

- Kafka Streams обеспечивает потоковую обработку каждой записи (или для каждого сообщения). Это уровень абстракции над клиентами-производителями и потребителями.
- Kafka Streams предлагает выбор между функциональным предметно-ориентированным языком (Domain-Specific Language, DSL) и Processor API.
- Потоки можно моделировать как топологию с помощью Kafka Streams DSL.

- ksqlDB – это база данных, предлагающая возможности Kafka тем, кто уже знаком с SQL. ksqlDB выполняет запросы непрерывно и может помочь быстро создавать прототипы потоковых приложений.
- Предложения по улучшению Kafka (Kafka Improvement Proposals, KIP) – отличный способ увидеть, какие изменения запрашиваются и, возможно, появятся в будущих версиях Kafka.

Ссылки

- 1 «Documentation: Kafka Streams». Apache Software Foundation (n.d.). <https://kafka.apache.org/documentationstreams/> (доступно по состоянию на 30 мая 2021).
- 2 «Streams Concepts». Confluent documentation (n.d.). <https://docs.confluent.io/platform/currentstreams/concepts.html> (доступно по состоянию на 17 июня 2020).
- 3 «Documentation: Kafka Streams: Core Concepts». Apache Software Foundation (n.d.). <https://kafka.apache.org/26/documentationstreams/core-concepts> (доступно по состоянию на 25 июня 2021).
- 4 «Kafka Streams Processor API». Confluent documentation (n.d.). <https://docs.confluent.io/platform/currentstreams/developer-guide/processor-api.html#streams-developer-guide-processor-api> (доступно по состоянию на 22 августа 2021).
- 5 «Streams Architecture». Confluent documentation (n.d.). <https://docs.confluent.io/platform/currentstreams/architecture.html> (доступно по состоянию на 17 июня 2020).
- 6 «Documentation: Streams API changes in 2.6.0». Apache Software Foundation. https://kafka.apache.org/26/documentationstreams/upgrade-guide#streams_api_changes_260 (доступно по состоянию на 22 августа 2021).
- 7 «ksqlDB Documentation: Queries». Confluent documentation (n.d.). <https://docs.ksqldb.io/en/latest/concepts/queries/> (доступно по состоянию на 5 мая 2021).
- 8 «ksqlDB: Configure ksqlDB CLI». Confluent documentation (n.d.). <https://docs.ksqldb.io/en/0.7.1-ksqldb/operate-and-deploy/installation/cli-config/> (доступно по состоянию на 23 августа 2021).
- 9 «Installing ksqlDB». Confluent documentation (n.d.). <https://docs.confluent.io/platform/currentksqldb/installing.html> (доступно по состоянию на 20 июня 2020).
- 10 «Configure ksqlDB Server». Confluent documentation (n.d.). <https://docs.ksqldb.io/en/latest/operate-and-deploy/installation/server-config/> (доступно по состоянию на 23 августа 2021).

Приложение A.

Установка

Несмотря на широкий набор функций, процесс установки Apache Kafka прост. Сначала рассмотрим требования к окружению.

A.1. Требования к операционной системе (ОС)

Обычно в качестве дома для Kafka выбирается Linux, и, похоже, именно на нем сосредотачивают основное внимание многие форумы поддержки. Мы использовали macOS с командной оболочкой Bash (использовалась по умолчанию до версии macOS Catalina) или zsh (начала использоваться по умолчанию, начиная с версии macOS Catalina). Kafka можно также запустить под управлением Microsoft® Windows®, но использовать эту ОС в промышленном окружении нежелательно [1].

ПРИМЕЧАНИЕ. В следующем разделе мы дополнитель но опишем установку с помощью Docker (<http://docker.com>).

A.2. Версии Kafka

Apache Kafka – это активно развивающийся проект Apache Software Foundation, поэтому версии Kafka постоянно обновляются. Разработчики Kafka, как правило, серьезно относятся к обратной совместимости. Если у вас появится желание перейти на новую версию, то после ее установки просто обновите все вызовы, помеченные как устаревшие.

СОВЕТ. Обычно в промышленном окружении Apache ZooKeeper и Kafka устанавливаются на разные физические серверы, что обусловлено желанием добиться максимальной отказоустойчивости. В этой книге основное внимание уделялось изучению возможностей Kafka, а не управлению несколькими серверами.

A.3. Установка Kafka на локальный компьютер

Начиная использовать Kafka, некоторые авторы этой книги сочли, что одним из наиболее простых вариантов было создание кла-

стера на одном узле вручную. Майкл Нолл (Michael Noll) в статье «Running a Multi-Broker Apache Kafka 0.8 Cluster on a Single Node» четко изложил шаги, перечисленные далее в этом разделе [2].

Несмотря на то что статья была написана в далеком 2013 году, этот вариант установки по-прежнему является отличным способом увидеть все детали, которые легко упустить при автоматизированной установке. Установка с помощью Docker – еще один вариант локальной установки, описанный ниже в этом приложении, который с успехом могут использовать те из вас, кто уверенно владеет технологией контейнеров.

Исходя из личного опыта, вы можете установить Kafka на рабочую станцию со следующими минимальными требованиями (имейте в виду, что ваши выводы могут отличаться от наших). Ознакомившись с этими требованиями, используйте инструкции в следующих разделах, описывающие порядок установки Java и Apache Kafka (включая ZooKeeper) на свою рабочую станцию:

- минимальное количество процессоров (физических или логических): 2;
- минимальный объем оперативной памяти: 4 Гбайт;
- минимальное свободное место на жестком диске: 10 Гбайт.

A.3.1. Предварительный этап: установка Java

Первым обязательным условием является наличие Java на компьютере. Разрабатывая примеры для этой книги, мы использовали Java Development Kit (JDK) версии 11. Этот пакет можно получить по адресу <https://jdk.dev/download/>. Для установки и управления версиями Java на вашем компьютере мы рекомендуем использовать SDKMAN CLI, доступный по адресу <http://sdkman.io>.

A.3.2. Предварительный этап: установка ZooKeeper

На момент написания этой книги для нормальной работы Kafka требовался также ZooKeeper, поставляющийся в составе Kafka. Даже при всех усилиях, направленных на уменьшение зависимости от ZooKeeper, предпринятых в последних версиях, Kafka все еще требует установки ZooKeeper. Совместимая версия ZooKeeper уже входит в состав дистрибутива Apache Kafka, поэтому вам не придется загружать и устанавливать ее отдельно. Сценарии, необходимые для запуска и остановки ZooKeeper, также включены в дистрибутив Kafka.

A.3.3. Предварительный этап: загрузка Kafka

На момент публикации этой книги наиболее свежей была версия Kafka 2.7.1 (именно она использовалась при создании наших примеров). Проект Apache® имеет зеркала, и вы можете использовать

их для загрузки дистрибутива. Для автоматического перехода на ближайшее зеркало используйте этот URL: <http://mng.bz/aZo7>.

После загрузки файла проверьте его фактическое имя. На первый взгляд оно выглядит немного запутанным. Например, `kafka_2.13-2.7.1` означает, что этот дистрибутив содержит версию Kafka 2.7.1 (указана после дефиса).

Чтобы опробовать примеры из этой книги и при этом избежать ненужных сложностей, мы рекомендуем настроить кластер из трех узлов на одном компьютере. Эта рекомендация не относится к настройке промышленного окружения, но такая конфигурация позволит вам понять важные идеи, не тратя много времени на настройку.

ПРИМЕЧАНИЕ. Зачем использовать кластер из трех узлов? Разные части Kafka как компоненты распределенной системы предполагают работу на нескольких узлах. Наши примеры имитируют работу кластера, не требуя использовать разных физические машин, чтобы вам было проще понять происходящее.

После установки Kafka следует настроить кластер из трех узлов. Для этого нужно распаковать двоичный файл и найти каталог `bin`.

В листинге А.1 показана команда `tar`, используемая для распаковки JAR-файла, но, в зависимости от формата загруженного вами файла, вам может понадобиться использовать `unzip` или другой инструмент [3]. Путь к каталогу `bin` со сценариями Kafka рекомендуется включить в переменную окружения `$PATH`. В этом случае вы сможете запускать эти сценарии, не указывая полный путь к ним.

Листинг А.1. Распаковка файла дистрибутива Kafka

```
$ tar -xzf kafka_2.13-2.7.1.tgz
$ mv kafka_2.13-2.7.1 ~/
$ cd ~/kafka_2.13-2.7.1
$ export PATH=$PATH:~/kafka_2.13-2.7.1/bin
```

Добавление пути к каталогу `bin` в переменную окружения `$PATH`

ПРИМЕЧАНИЕ. Пользователи Windows могут найти те же сценарии, что используются в следующих примерах, с теми же именами, но с расширением `.bat` в папке `bin/windows`. Также можно использовать подсистему Windows for Linux 2 (WSL2) и запускать те же команды, что и в Linux [1].

A.3.4. Запуск сервера ZooKeeper

В примерах в этой книге используется один локальный сервер ZooKeeper. Команда в листинге А.2 запускает один сервер ZooKeeper [2]. Обратите внимание, что сервер ZooKeeper должен запускаться до начала работы с брокерами Kafka.

Листинг А.2. Запуск ZooKeeper

```
$ cd ~/kafka_2.13-2.7.1
$ bin/zookeeper-server-start.sh config/zookeeper.properties
```

A.3.5. Создание и настройка кластера вручную

Следующий шаг – создание и настройка кластера из трех узлов. Чтобы создать кластер Kafka, нужно настроить три сервера (брокера): `server0`, `server1` и `server2`. Для этого следует подготовить файлы свойств для каждого сервера [2].

В дистрибутиве Kafka имеется набор предопределенных файлов свойств с настройками по умолчанию. Выполните команды из листинга А.3, чтобы создать файлы конфигурации для каждого сервера в вашем кластере [2]. В качестве отправной точки мы будем использовать файл `server.properties` с настройками по умолчанию. Затем выполните команду из листинга А.4, чтобы открыть каждый файл конфигурации и изменить настройки в нем [2].

Листинг А.3. Создание нескольких брокеров Kafka

```
$ cd ~/kafka_2.13-2.7.1
$ cp config/server.properties config/server0.properties
$ cp config/server.properties config/server1.properties
$ cp config/server.properties config/server2.properties
```

После перехода в каталог Kafka создайте три копии файла `server.properties` с настройками по умолчанию.

ПРИМЕЧАНИЕ. В наших примерах мы используем текстовый редактор `vi`, но вы можете редактировать эти файлы в любом текстовом редакторе по вашему выбору.

Листинг А.4. Настройки в файлах serverN

```
$ vi config/server0.properties
broker.id=0
listeners=PLAINTEXT://localhost:9092
log.dirs= /tmp/kafkainaction/kafka-logs-0
```

Настроить идентификатор, номер порта и каталог журнала для брокера с идентификатором 0

```
$ vi config/server1.properties
broker.id=1
listeners=PLAINTEXT://localhost:9093
log.dirs= /tmp/kafkainaction/kafka-logs-1
```

Настроить идентификатор, номер порта и каталог журнала для брокера с идентификатором 1

```
$ vi config/server2.properties
broker.id=2
listeners=PLAINTEXT://localhost:9094
log.dirs= /tmp/kafkainaction/kafka-logs-2
```

Настроить идентификатор, номер порта и каталог журнала для брокера с идентификатором 2.

ПРИМЕЧАНИЕ. Каждый брокер Kafka использует свой порт и каталог журналов. Также каждый файл конфигурации должен задавать уникальный идентификатор для каждого брокера, потому что каждый брокер использует свой идентификатор для регистрации в качестве члена кластера. Обычно нумерация идентификаторов брокеров начинается с 0, следуя схеме индексации массивов.

После этого можно запустить брокеры, используя сценарии, входящие в состав дистрибутива (вместе с файлами конфигурации, которые вы изменили в листинге А.4). Если вам интересно понаблюдать за выводом брокеров Kafka в терминале, то мы рекомендуем запустить каждый процесс в отдельной вкладке или окне терминала и оставить их открытыми. Команды в листинге А.5 запускают Kafka в окне терминала [2].

Листинг А.5. Запуск Kafka в окне терминала

```
$ cd ~/kafka_2.13-2.7.1
$ bin/kafka-server-start.sh config/server0.properties
$ bin/kafka-server-start.sh config/server1.properties
$ bin/kafka-server-start.sh config/server2.properties
```

← После перехода в каталог Kafka запустите процессы брокеров (всего 3)

СОВЕТ. Если вы закроете окно терминала или ваш процесс зависнет, то не забудьте выполнить команду `jps` [4]. Она поможет найти процессы Java, которые, возможно, придется остановить принудительно.

В листинге А.6 показан пример, демонстрирующий, как можно получить идентификаторы процессов (PID) брокеров и метку процесса JVM ZooKeeper (`QuorumPeerMain`). Идентификаторы процессов выводятся слева и могут изменяться при каждом вызове сценариев запуска.

Листинг А.6. Вывод команды `jps`, содержащий идентификаторы трех процессов брокеров и сервера ZooKeeper

2532 Kafka		Метка процесса Kafka в JVM и идентификаторы процессов брокеров
2745 Kafka		
2318 Kafka		

← Метка в JVM и идентификатор процесса ZooKeeper

Теперь, закончив настройку локального окружения вручную, посмотрим, как можно использовать Confluent Platform, основанную на Apache Kafka и предлагаемую Confluent Inc. (<https://www.confluent.io/>).

A.4. Confluent Platform

Платформа Confluent Platform (дополнительную информацию можно найти по адресу <https://www.confluent.io/>) – это готовый к использованию вариант дистрибутива Apache Kafka для предприятий, который включает все необходимые средства разработки, в том числе пакеты для Docker, Kubernetes, Ansible и др. В рамках проекта Confluent активно разрабатываются и поддерживаются клиенты Kafka для C++, C#/.NET, Python и Go. В пакет также включен реестр схем Schema Registry, о котором мы рассказываем в главах 3 и 11. Кроме того, в состав Confluent Platform Community Edition входит ksqlDB. О потоковой обработке данных с помощью ksqlDB рассказывается в главе 12.

Дополнительно компания Confluent Inc. предоставляет управляемый облачный сервис Kafka, который может пригодиться вам для разработки будущих проектов. Управляемый сервис предоставляет возможности Apache Kafka, не требуя знаний о том, как ее запустить. Эта особенность позволяет разработчикам сосредоточиться на самом важном, то есть на программировании. Загружаемый пакет Confluent версии 6.1.1 включает версию Apache Kafka 2.7.1, которая используется в этой книге. Инструкции по установке Confluent вы найдете в официальной документации, доступной по адресу <http://mng.bz/g1oV>.

A.4.1. Интерфейс командной строки Confluent

Confluent распространяет дополнительные инструменты, позволяющие быстро запускать Confluent Platform и управлять ею из командной строки. Более подробную информацию о сценарии вы найдете в файле *README.md* на <https://github.com/confluentinc/confluent-cli>, а инструкции по установке – на <http://mng.bz/RqNR>. Интерфейс командной строки удобен тем, что позволяет запускать несколько частей вашего продукта по мере необходимости.

A.4.2. Docker

В настоящее время Apache Kafka не предлагает официальных образов Docker, но Confluent предоставляет свои образы. Эти образы тестируются, поддерживаются и используются многими разработчиками в промышленных окружениях. В репозитории с примерами для этой книги вы найдете файл *docker-compose.yaml* с предварительно настроенными компонентами Kafka, ZooKeeper и др. Чтобы запустить все эти компоненты, выполните команду *docker-compose up -d* в каталоге с файлом YAML, как показано в листинге A.7.

ПРИМЕЧАНИЕ. Если вы не знакомы с Docker или он у вас не установлен, то обращайтесь к официальной документации по адресу <https://www.docker.com/get-started>. Там же вы найдете инструкции по установке.

Листинг А.7. Запуск образа Docker с компонентами Apache Kafka

```
$ git clone \https://github.com/Kafka-In-Action-Book/Kafka-In-Action-Source-Code.git ← Получение копии репозитория GitHub с примерами для книги
$ cd ./Kafka-In-Action-Source-Code ← Запуск docker-compose в каталоге с примерами
$ docker-compose up -d

Creating network "kafka-in-action-code_default" with the default driver
Creating Zookeeper... done ← Обратите внимание на следующий вывод
Creating broker2 ... done
Creating broker1 ... done
Creating broker3 ... done
Creating schema-registry ... done
Creating ksqldb-server ... done
Creating ksqldb-cli ... done

$ docker ps --format "{{.Names}}: {{.State}}" ← Проверка запуска всех компонентов
ksqldb-cli: running
ksqldb-server: running
schema-registry: running
broker1: running
broker2: running
broker3: running
zookeeper: running
```

A.5. Как работать с примерами для книги

Для просмотра и запуска примеров, сопровождающих книгу, можно использовать любую среду разработки. Вот некоторые из тех, что мы можем порекомендовать:

- IntelliJ IDEA Community Edition (<https://www.jetbrains.com/idea/download/>);
- Apache Netbeans (<https://netbeans.org>);
- VS Code for Java (<https://code.visualstudio.com/docs/languages/java>);
- Eclipse STS (<https://spring.io/tools>).

A.5.1. Сборка из командной строки

Если вы предпочтете выполнять сборку из командной строки, то вам придется сделать еще несколько шагов. Примеры Java 11 в этой книге собираются с помощью Maven 3.6.3. Файлы JAR для каждой главы можно получить, выполнив команду `./mvnw verify` в корневом каталоге главы, содержащем файл `pom.xml`, или, например, команду `./mvnw -Dprojects KafkaInAction_Chapter2 verify` в корневом каталоге проекта.

Мы сами используем инструмент Maven Wrapper (<http://mng.bz/20yo>), поэтому если у вас не установлена система сборки Maven, то любая из предыдущих команд загрузит и запустит Maven автоматически. Чтобы запустить определенный класс, содержащий метод `main`, вы должны указать его имя в командной строке после пути к файлу JAR. В листинге A.8 показано, как запустить класс из главы 2.

ПРИМЕЧАНИЕ. Для успешного выполнения команды файл JAR должен быть собран со всеми зависимостями.

Листинг A.8. Запуск класса производителя из главы 2

```
java -cp target/chapter2-jar-with-dependencies.jar \
    replace.with.full.package.name.HelloWorldProducer
```

A.6. Устранение проблем

Все примеры исходного кода для этой книги доступны по адресу <https://github.com/Kafka-In-Action-Book/Kafka-In-Action-Source-Code>. На случай, если у вас возникнут проблемы с их запуском, мы предлагаем несколько общих рекомендаций по их устранению:

- прежде чем запускать примеры из командной строки, убедитесь, что кластер запущен и работает;
- если вы не остановили свой кластер надлежащим образом, то старый процесс может продолжать удерживать порт, который вы пытаетесь использовать в своей следующей попытке. Чтобы определить, какие процессы запущены, а какие следует остановить, можно использовать такие инструменты, как `jps` или `lsof`;
- команды должны запускаться из каталога установки, если явно не указано иное. Если вам удобнее работать с командной строкой, то выполните дополнительные настройки, например добавьте переменные окружения и псевдонимы;
- если командная оболочка сообщила вам, что не может найти указанные вами команды, – проверьте содержимое вашего каталога установки. Убедитесь, что искомые файлы присутствуют и имеют разрешение на выполнение. Возможно, проблему поможет решить такая команда, как `chmod -R 755`. Проверьте, добавлен ли путь к каталогу `bin` в переменную окружения `PATH`. Если перечисленные настройки выполнены правильно, но команды по-прежнему не запускаются, попробуйте указать абсолютный путь к команде;
- загляните в файл `Commands.md`, который присутствует в каталоге с примерами для каждой главы. Этот файл включает в себя большинство команд, используемых в конкретной главе. Попробуйте найти дополнительные подсказки в файлах `README.md`.

Ссылки

- 1 J. Galasyn. «How to Run Confluent on Windows in Minutes». Confluent blog (26 марта 2021). <https://www.confluent.io/blog/set-up-and-run-kafka-on-windows-and-wsl-2/> (доступно по состоянию на 11 июня 2021).
- 2 M. G. Noll. «Running a Multi-Broker Apache Kafka 0.8 Cluster on a Single Node». (13 марта 2013). <https://www.michael-noll.com/blog/2013/03/13/running-a-multi-broker-apache-kafka-cluster-on-a-single-node/> (доступно по состоянию на 20 июля 2021).
- 3 «Apache Kafka Quickstart». Apache Software Foundation (n.d.). <https://kafka.apache.org/quickstart> (доступно по состоянию на 22 августа 2021).
- 4 README.md. Confluent Inc. GitHub (n.d.). <https://github.com/confluentinc/customer-utilities> (доступно по состоянию на 21 августа 2021).

Приложение B.

Пример клиента

Все примеры для этой книги иллюстрируют создание клиентов Kafka на Java, однако объяснить основные идеи новым пользователям часто проще и быстрее, если продемонстрировать примеры на хорошо знакомых им языках программирования. Проект Confluent Platform предлагает множество поддерживаемых им клиентов [1]. В этом приложении мы рассмотрим примеры клиентов Kafka на Python, а затем дадим несколько замечаний по тестированию клиентов на Java.

B.1. Клиенты Kafka на Python

В этом примере мы рассмотрим Confluent Python Client [2]. Использование клиентов Confluent дает уверенность в совместимости не только с Apache Kafka, но и со всеми предложениями Confluent Platform. Далее рассмотрим примеры двух клиентов на Python (производителя и потребителя), но прежде кратко опишем установку Python.

B.1.1. Установка Python

Если вы являетесь пользователем Python, то, вероятно, уже перешли на Python 3. В противном случае нужно установить `librdkafka`. Если вы используете диспетчер пакетов Homebrew, то выполните следующую команду: `brew install librdkafka` [2].

Далее понадобится клиентский пакет, который ваш код будет использовать как зависимость. Пакет wheels с поддержкой Confluent Kafka можно установить с помощью Pip, выполнив команду `pip install confluent-kafka` [2]. Теперь, сделав это, рассмотрим создание простого клиента-производителя на Python.

B.1.2. Пример производителя на Python

В листинге B.1 показан простой клиент-производитель на Python, использующий `confluent-kafka-python` [2]. Он отправляет два сообщения в тему `kinaction-python-topic`.

Листинг В.1. Пример производителя на Python

```

from confluent_kafka import Producer      ← Сначала нужно импортировать
producer = Producer(                     пакет Confluent
    {'bootstrap.servers': 'localhost:9092'}) ← Настройка клиента-про-
                                                производителя для подклю-
                                                чения к определенному
def result(err, message):               ← брокеру Kafka
    if err:
        print('kinaction_error %s\n' % err) ← Действует как обратный вы-
    else:                                зов для обработки успеха
        print('kinaction_info : topic=%s, and kinaction_offset=%d\n' %
              (message.topic(), message.offset()))

```

messages = ["hello python", "hello again"] ← Массив с сообще-
ниями для отправки

```

for msg in messages:
    producer.poll(0)
    producer.produce("kinaction-python-topic",
                      value=msg.encode('utf-8'), callback=result) ← Отправка всех со-  
общений в Kafka

```

```

producer.flush() ← Гарантирует отправку всех
# Вывод:           сообщений из буфера
#kinaction_info: topic=kinaction-python-topic, and kinaction_offset=8 ← Пример вывода показы-  
вает метаданные о двух от-  
правленных сообщениях
#kinaction_info: topic=kinaction-python-topic, and kinaction_offset=9

```

Чтобы использовать пакет Confluent, сначала нужно импортировать зависимость `confluent_kafka`. Затем следует настроить параметры клиента `Producer`, включая адрес брокера для подключения. Обратный вызов `result` в листинге выполняет некоторую логику после каждого вызова метода `produce` независимо от того, был ли вызов успешным или нет. Затем код в примере выполняет обход элементов массива `messages` и отправляет каждое сообщение по очереди. Затем вызывается метод `flush()`, чтобы гарантировать фактическую отправку сообщений брокеру из внутреннего буфера. В конце листинга В.1 приводится пример вывода в консоль. Давайте теперь посмотрим, как реализовать на Python клиента-потребителя.

B.1.3. Пример потребителя на Python

В листинге В.2 показан пример клиента-потребителя Kafka, использующего `confluent-kafka-python` [3]. Мы используем его для чтения сообщений, созданных нашим производителем на Python в листинге В.1.

Листинг В.2. Пример потребителя на Python

```

from confluent_kafka import Consumer ← Сначала нужно импортировать
    пакет Confluent
consumer = Consumer({ ← Настройка клиента-потреби-
    'bootstrap.servers': 'localhost:9094', ← теля для подключения к опре-
    'group.id': 'kinaction_team0group', ← деленному брокеру Kafka
    'auto.offset.reset': 'earliest'
})

consumer.subscribe(['kinaction-python-topic']) ← Подписка потребите-
try: ← ля на список тем
    while True:
        message = consumer.poll(2.5) ← Проверка поступления новых сооб-
            if message is None: ← щений внутри бесконечного цикла
                continue
            if message.error():
                print('kinaction_error: %s' % message.error())
                continue
            else:
                print('kinaction_info: %s for topic: %s\n' %
                    (message.value().decode('utf-8'),
                     message.topic()))

except KeyboardInterrupt:
    print('kinaction_info: stopping\n')
finally:
    consumer.close() ← Освобождение
# Output ← ресурсов
# kinaction_info: hello python for topic: kinaction-python-topic ← Пример вывода инфор-
# мации о сообщении

```

Так же как в примере с производителем (листинг В.1), сначала нужно импортировать зависимость `confluent_kafka`. Затем настроим параметры клиента `Consumer`, включая адрес брокера для подключения. После этого клиент-потребитель подписывается на список тем, из которых предполагается получать сообщения; в данном случае в списке указана единственная тема `kinaction-python-topic`. Так же как в клиентах-потребителях на Java, мы используем бесконечный цикл, в котором регулярно опрашиваем Kafka на наличие новых сообщений. Пример вывода показывает полученное сообщение, а также его смещение. Когда клиент-потребитель завершается пользователем, блок `finally` пытается освободить занятые ресурсы и покинуть группу потребителей после фиксации всех использованных смещений.

Примеры на Python, представленные в этом разделе, просты, однако они вполне справляются со своей главной целью – показать разработчикам, что взаимодействовать с Kafka можно не только из Java, но и из многих других языков программирования. Просто помните, что не все клиенты поддерживают то же разнообразие возможностей, что и клиенты Java.

B.2. Тестирование клиентов

В главе 7 мы кратко рассмотрели тестирование с помощью `EmbeddedKafkaCluster`. Теперь познакомимся с некоторыми другими средствами тестирования кода, взаимодействующего с Kafka, перед его развертыванием в рабочем окружении.

B.2.1. Модульное тестирование в Java

Модульное тестирование фокусируется на проверке одной единицы программного обеспечения. В идеале модульные тесты не должны зависеть ни от каких других компонентов. Но как протестировать клиентский класс Kafka без подключения к реальному кластеру Kafka?

Знакомые с фреймворками для тестирования, такими как Mockito (<https://site.mockito.org/>), могут создать фиктивный объект производителя, который заменит настоящий. К счастью, официальная клиентская библиотека Kafka уже предоставляет такой объект с именем `MockProducer`, который реализует интерфейс `Producer` [4]. Он избавляет от необходимости использовать реальный кластер Kafka для проверки логики производителя! У фиктивного производителя есть метод `clear`, который можно вызвать для очистки записанных сообщений, чтобы можно было запустить другие тесты [4]. Для проверки потребителя тоже можно использовать фиктивную реализацию [4].

B.2.2. Kafka Testcontainers

В главе 7 упоминался еще один вариант тестирования – `Testcontainers` (<https://www.testcontainers.org/modules/kafka/>). В отличие от `EmbeddedKafkaCluster`, который запускает брокеры Kafka и узлы ZooKeeper в памяти, `Testcontainers` запускает образы Docker.

Ссылки

- 1 «Kafka Clients». Confluent documentation (n.d.). <https://docs.confluent.io/current/clients/index.html> (доступно по состоянию на 15 июня 2020).
- 2 `confluent-kafka-python`. Confluent Inc. GitHub (n.d.). <https://github.com/confluentinc/confluent-kafka-python> (доступно по состоянию на 12 июня 2020).

- 3 consumer.py. Confluent Inc. GitHub (n.d.). <https://github.com/confluentinc/confluent-kafka-python/blob/master/examples/consumer.py> (доступно по состоянию на 21 августа 2021).
- 4 MockProducer<K,V>. Kafka 2.7.0 API. Apache Software Foundation (n.d.). <https://kafka.apache.org/27/javadoc/org/apache/kafka/clients/producer/MockProducer.html> (доступно по состоянию на 30 мая 2021).

Предметный указатель

A

acks свойство 123
add-config параметр 251
AdminClient пакет 63
advertised.listeners параметр 222
AlertConsumer класс 229
AlertConsumerMetricsInterceptor
 класс 228
AlertLevelPartitioner логика 187
AlertProducer класс 227
AlertProducerMetricsInterceptor класс 226
Alert класс 116
Alert объект 53
Alert объект данных 117
Apache Flume 197
assign метод 150
auto.offset.reset свойство 134, 146
autopurge.purgeInterval свойство 220
autopurge.snapRetainCount свойство 220

B

bootstrap.servers параметр 47, 64, 109,
 123, 129, 213

C

Callback интерфейс 122
cleanup.policy параметр 188
client.id свойство 213, 249
confluent-kafka-python зависимость 299
Confluent Platform 295
 интерфейс командной строки 295
Confluent REST Proxy API 216
Confluent Schema Registry 262
connect-standalone.sh команда 78
console-consumer команда 79
consumer_offsets тема 145, 189
consumer_byte_rate параметр 250
ConsumerConfig класс 129, 229
ConsumerInterceptor интерфейс 228
ConsumerRecord 54
controlled.shutdown.enable 168
CQRS (Command Query Responsibility
 Segregation разделение
 ответственности команд и
 запросов) 206

CreateTime поле 184
curl команда 217

D

DAG (Directed Acyclic Graph
 ориентированный ациклический
 граф) 274
DDoS, распределенная атака типа отказ в
 обслуживании 249
delete-config параметр 251
delete параметр 188
delete.topic.enable параметр 180
describe параметр 251
describe флаг 172
Docker 295
DSL (Domain-Specific Languages
 предметно-ориентированные
 языки) 270

E

EmbeddedKafkaCluster тестирование
 разделов 186
enable.auto.commit свойство 142
entity-name параметр 251
ESB (Enterprise Service Bus сервисная
 шина предприятия) 257
ETL (extract, transform, load извлечение,
 преобразование, загрузка) 195

F

FileStreamSource класс 78
from-beginning флаг 133

G

GlobalKTable API 278
group.id ключ 129

H

HDFS (Hadoop Distributed Filesystem
 распределенная файловая система
 Hadoop) 35
Hlk-список брокеров 109

I

index расширение файлов 184
in-sync replicas синхронизированные
 реплики 48

- inter.broker.listener.name параметр 222
 interceptor.classes свойство 227
 io.confluent.kafka.serializers.
 KafkaAvroSerializer класс 116
 Isr поле 48
- J**
- JAAS (Java Authentication and Authorization Service служба аутентификации и авторизации Java) 244
- Java
 клиенты 63
 JMX консоль 222
- K**
- Kafka
 Java-клиенты 63
 авторизация в 245
 брокеры 46
 журнал коммитов 59
 запуск как службы systemd 217
 использование ZooKeeper 56
 мифы 35
 обзор 26
 онлайн-ресурсы 41
 пакеты исходного кода 60
 AdminClient 63
 Kafka Connect 62
 Kafka Streams 60
 ksqlDB 63
 приложений журналы 219
 производители 51
 потребители 51
 потоковая обработка 67
 семантика точно один раз 69
 темы 55
 экскурсия 51
- kafka-acls.sh инструмент 247
 kafkaAppender::MaxFileSize свойство 219
 KafkaConfig значения 160
 kafka-configs.sh инструмент 251
 Kafka Connect 62
 kafka-console-consumer команда 215
 kafka-console-producer команда 215
 KafkaSink исходный код 123
 Kafka Streams 60, 272
 GlobalKTable API 278
 настройка 281
 KStreams API DSL 273
 KTable API 277
 Processor API 279
 Kafka Testcontainers 188
- kafka-topics.sh команда 46, 181
 kcat инструмент 214
 Kerberos 244
 key.deserializer свойство 149
 key.serializer параметр 64
 key.serializer свойство 118
 keytab файл 244
 keytool, утилита 241
 kinaction_alert тема 53
 kinaction_audit тема 54
 kinaction_clueful_secrets тема 237
 kinaction_helloworld тема 47, 49, 55, 157, 180
 kinaction_one_replica тема 159
 kinaction_replica_test тема 171
 KIP (Kafka Improvement Proposals
 предложения по улучшению Kafka) 287
 ksqlDB 282
 автономный режим 287
 архитектура 286
 запросы 284
 локальная разработка 284
 ksqlDB пакет 63
 KStreams API DSL 273
 KTable API 277
- L**
- Leader поле 48
 librdkafka библиотека 299
 listeners параметр 222
 log4j.properties файл 219
 log расширение файлов 183
 logrotate инструмент 221
- M**
- max.in.flight.requests.per.connection
 значение 108
 message.timestamp.type параметр
 настройки темы 113
- N**
- NONE тип 265
- O**
- OffsetCommitCallback интерфейс 144
 offsetsForTimes метод 147
- P**
- partition.assignment.strategy свойство 141
 partitioner.class значение 121
 partitions параметр 47
 Partitioner интерфейс 121
 Processor API 279
 producer_byte_rate параметр 250
 ProducerConfig класс 228

Producer интерфейс 302
 ProducerInterceptor интерфейс 226
 ProducerRecord 53
 ProducerRecord объект 113
 Properties объект 213
 pull-запросы 284
 push-запросы 284

Q
 queries-file аргумент командной строки 286

R
 RBAC (Role-Based Access Control) управление доступом на основе ролей 247
 RecordMetadata объект 124
 Red Hat Debezium 199
 Replicas поле 48
 replication-factor параметр 181
 request_percentage параметр 252

S
 Schema Registry, реестр схем 260
 альтернативные решения 267
 клиентская библиотека 263
 конфигурация 261
 правила совместимости 265
 REST API 262
 schemas имя темы 260
 Secor 200
 security.inter.broker.protocol параметр 244
 serde термин 118
 Serializer интерфейс 118
 server.log файл 220
 SinkCallback класс 124
 Slack группа каналов 288
 snapCount свойство 221
 StreamsBuilder.j_!trn 275
 STRUCT ключевое слово 285
 systemctl команды 218
 systemd, запуск Kafka как службы 217

T
 tar команда 292
 Testcontainers Kafka
 тестирование клиентов 302
 timeindex расширение файлов 184
 Topology объект 279
 transaction-failure тема 273
 Transaction тип 285
 transaction-request тема 271, 275
 TransactionResult класс 285
 transaction-success тема 273

U
 UnderReplicatedPartitions атрибут 224
 under-replicated-partitions флаг 172
 user свойство 249

V
 value.deserializer ключ 129
 value.serializer параметр 64, 263

W
 WorkerSinkTask класс 53

Z
 ZkData.scala класс 157
 zkNodes свойство 156
 ZooKeeper
 брокеры 156
 ансамбли 57
 запуск сервера 292
 использование в Kafka 56
 приложений журналы 220

A
 авторизация 239
 авторизация в Kafka 245
 автономный режим, ksqlDB 287
 администрирования, клиенты 212
 AdminClient 212
 Confluent REST Proxy API 216
 kcat 215
 аналитические данные 201
 ансамбли 57
 архитектура
 ksqlDB 286
 архитектура проектов Kafka
 общий план 88
 события от датчиков 82
 исходные данные 82
 требования к пользовательским данным 88
 архитектуры 203
 каппа-архитектура 205
 лямбда-архитектура 203
 аутентификация 238

Б
 безопасность
 авторизация 245
 данные в состоянии покоя 253
 квоты 249
 ограничение частоты запросов 252
 ограничение пропускной способности сети 250
 основы 238
 списки управления доступом 246

- шифрование с помощью SSL 239
управление доступом на основе
ролей 247
управляемые варианты 253
ZooKeeper 248
брандмауэры 221
брокеры 46
версии клиентов и брокеров 124
добавление в кластер 167
конфигурационные параметры 158
журнал сервера 160
журналы приложений 160
контроллеры 160
обзор 155
обновление кластера 167
обновление клиентов 168
обслуживание кластера 167
резервные копии 169
системы с сохранением состояния 169
быстрые данные 34
- В**
ведущая реплика 49, 56
ведущие реплики разделов 162
- Д**
данные аудита 92
данные в состоянии покоя 253
- Ж**
журнал коммитов 59
журналы 218
приложений 219
Kafka 218
ZooKeeper 220
журналы приложений
брокеры 160
- З**
записи 45
запросы, ksqlDB 284
- И**
инструменты
logrotate 221
systemd, запуск Kafka как службы 217
брандмауэры 221
клиенты администрирования 212
AdminClient 212
Confluent REST Proxy API 216
kcat 215
консоль JMX 222
общие инструменты мониторинга 231
интерфейс командной строки
Confluent Platform 295
- сборка 296
исходные данные 82
исходный процессор 273
исходный узел 273
- К**
каппа-архитектура 205
квоты 249
ограничение пропускной способности
сети 250
ограничение частоты запросов 252
кеш страниц 58
кластеры
Kubernetes 207
добавление брокеров 167
обновление 167
обслуживание 167
окружения с несколькими
кластерами 206
создание и настройка вручную 293
клиенты
Testcontainers Kafka 302
обновление 168
тестирование 302
модульное тестирование в Java 302
клиенты администрирования 212
AdminClient 212
Confluent REST Proxy API 216
kcat 215
клиенты Kafka на Python 299
производитель 299
потребитель 300
коэффициенты репликации 183
координатор группы 135
трассировка данных 139
координаты 133
консоль JMX 222
контроллеры 160
конфигурация
производителей 108
отметки времени 113
список брокеров 109
потребителей 129
Schema Registry, реестр схем 261
- Л**
локальная разработка, ksqlDB 284
лямбда-архитектура 203
- М**
мифы о Kafka 35
Kafka ничем не отличается от других
брокеров сообщений 36
Kafka работает только с Hadoop 35

мониторинг
общие инструменты мониторинга 231

Н
назначение диапазонов, стратегия 141
не более одного раза, семантика 29

О
ограничение пропускной способности
сети 250
ограничение частоты запросов 252
операционные данные 201
ориентированный ациклический граф
(Directed Acyclic Graph, DAG) 274
основы безопасности 238
отметки времени 113

П
пакеты исходного кода
AdminClient 63
Kafka Connect 62
Kafka Streams 60
ksqlDB 63
переопределение клиентов 230
потеря данных, и ведущие реплики
разделов 164
потоковая обработка 67
Kafka Streams 272
 GlobalKTable API 278
 настройка 281
 KStreams API DSL 273
 KTable API 277
 Processor API 279
KIP (Kafka Improvement Proposals
 предложения по улучшению
 Kafka) 287
ksqlDB 282
 автономный режим 287
 архитектура 286
 запросы 284
 локальная разработка 284
Slack группа каналов 288
потребители 51
 взаимодействия потребителей 137
 координаты 133
 параметры 129
 пример 128
 стратегия назначения разделов 141
 трассировка данных 138
 координатор группы 139
 чтение из сжатой темы 145
предложения по улучшению Kafka (Kafka Improvement Proposals, KIP) 287

предметно-ориентированные языки
(Domain-Specific Languages, DSL) 270
приложений журналы 219
 журналы Kafka 219
 журналы ZooKeeper 220
производители 51
 конфигурация 108
 отметки времени 113
 список брокеров 109
 обзор 104

Р
разделение ответственности команд
и запросов (Command Query
Responsibility Segregation, CQRS) 206
разделы 56, 183
 ведущие реплики разделов 162
размещение 183
просмотр журналов 185
тестирование с
 EmbeddedKafkaCluster 186
тестирование с Kafka Testcontainers 188
распределенная атака типа отказ в
обслуживании (DDoS) 249
распределенная файловая система
 Hadoop (Hadoop Distributed
 Filesystem, HDFS) 35
реестр схем Schema Registry 260
 альтернативные решения 267
 клиентская библиотека 263
 конфигурация 261
 правила совместимости 265
 REST API 262

С
сбор изменений в данных (Change Data
Capture, CDC) 199
семантика точно один раз 69
сервисная шина предприятия (Enterprise
Service Bus, ESB) 257
сжатые темы 188
синхронизированные реплики (In-Sync
Replicas, ISR) 163
синхронизированные реплики (ISR) 162
системы с сохранением состояния, и
 брокеры 169
служба аутентификации и авторизации
 Java (Java Authentication and
 Authorization Service, JAAS) 244
смещения 56, 133
события от датчиков 82
 исходные данные 82
совместимость 262

списки управления доступом 246
субъекты 262

Т

темы 177
коэффициенты репликации 183
обзор 55
сжатые 188
чтение из сжатой темы 145
точно один раз, семантика 29
трассировка 225
логика на стороне потребителя 228
логика на стороне производителя 226
переопределение клиентов 230
трассировка данных 138
координатор группы 139
стратегия назначения разделов 141
требования к пользовательским
данным 88

У

управление доступом на основе ролей
(Role-Based Access Control, RBAC) 247
уровни зрелости
модель зрелости Kafka 257
уровень 0 257
уровень 1 258
уровень 2 259
уровень 3 259
установка Kafka
Confluent Platform 295
Java 291
ZooKeeper 291
версии 290
загрузка 292

на локальный компьютер 291
создание и настройка кластера
вручную 293
примеры для книги 296
требования к ОС 291
устранение проблем 297

Ф

формат представления данных 93

Х

хранилища данных 194
Apache Flume 197
Red Hat Debezium 199
Secor 200
архитектуры 203
возврат данных в Kafka 201
инструменты 197
масштабирование 206
многоуровневые хранилища 203
окружения с несколькими
кластерами 206
перемещение данных 195
отказ от пакетного мышления 196
сохранение исходных событий 195
хранилище доверенных
сертификатов 242
хранилище ключей 241

Ц

центр сертификации (Certificate
Authority, CA) 241
циклический перебор, стратегия 142

Ш

шифрование с помощью SSL 239

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «КТК Галактика» наложенным платежом,
выслав открытку или письмо по почтовому адресу:

115487, г. Москва, пр. Андропова д. 38 оф. 10.

При оформлении заказа следует указать адрес (полностью),

по которому должны быть высланы книги;

фамилию, имя и отчество получателя.

Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине:

www.galaktika-dmk.com.

Оптовые закупки: тел. (499) 782-38-89.

Электронный адрес: books@aliens-kniga.ru.

Дилан Скотт, Виктор Гамов, Дейв Клейн

Предисловие Юна Рао

Kafka в действии

Главный редактор *Мовчан Д. А.*

dmkpress@gmail.com

Зам. главного редактора *Сенченкова Е. А.*

Перевод Киселев А. Н.

Корректор Абросимова Л. А.

Верстка Луценко С. В.

Дизайн обложки Мовчан А. Г.

Формат 70×100 1/16.

Гарнитура «NewBaskervilleC». Печать цифровая.

Усл. печ. л. 25,19. Тираж 200 экз.

Веб-сайт издательства: www.dmkpress.com

Kafka в действии

Apache Kafka можно рассматривать как высокопроизводительную программную шину, которая упрощает потоковую передачу событий, журнализирование, анализ и другие задачи, решаемые в рамках конвейеров данных. С помощью Kafka вы легко встроите такие функции, как оперативный мониторинг данных и масштабная обработка событий, и в крупные, и в небольшие приложения.

«Kafka в действии» знакомит с основными возможностями платформы Kafka и демонстрирует примеры ее использования в реальных приложениях. Прочитав книгу до конца, вы будете готовы решать основные задачи разработчиков и администраторов в команде, ориентированной на Kafka.

Рассматриваемые темы:

- Kafka как платформа для потоковой передачи событий;
- производители и потребители Kafka в приложениях на Java;
- Kafka как часть проекта обработки больших данных.

Для разработчиков на Java и специалистов по обработке данных. Никаких предварительных знаний о Kafka не требуется.

Дилан Скотт — разработчик программного обеспечения в сфере страхования.

Виктор Гамов — пропагандист передовых практик разработки с использованием Kafka.

Дэйв Кляйн, работающий в Confluent, помогает разработчикам, командам и предприятиям максимально использовать возможности потоковой передачи событий с помощью Apache Kafka.

Интернет-магазин:
www.dmkpress.com

Оптовая продажа:
КТК «Галактика»
books@aliens-kniga.ru



«Авторы имеют богатый опыт работы с Kafka в реальном мире, что выделяет эту книгу среди других».

Юн Рао,
кооснователь Confluent

«Удивительно доступное введение в очень сложную технологию. Эта книга займет достойное место на рабочем столе многих разработчиков».

Конор Редмонд,
InComm Payments

«Обширное практическое руководство по экосистеме Kafka».

Сумант Тамбе, LinkedIn

«Эта книга помогла мне быстро получить представление об особенностях работы Kafka, а также о разработке и защите распределенных приложений».

Грегор Рейман, Cloudfarms

ISBN 978-5-93700-118-4



9 785937 001184 >