

Приложения и микросервисы, управляемые событиями

# Kafka Streams в действии

ВТОРОЕ ИЗДАНИЕ

Билл Беджек

Предисловие Дэяна Рао



MANNING

# *Kafka Streams in Action*

SECOND EDITION  
EVENT-DRIVEN APPLICATIONS AND MICROSERVICES

BILL BEJECK  
FOREWORD BY JUN RAO



<https://t.me/javalib>

# *Kafka Streams* в действии

ВТОРОЕ ИЗДАНИЕ  
ПРИЛОЖЕНИЯ И МИКРОСЕРВИСЫ,  
УПРАВЛЯЕМЫЕ СОБЫТИЯМИ

Билл Беджек  
ПРЕДИСЛОВИЕ Дзюн Рао



2025

<https://t.me/javalib>

Корпоративным приложениям приходится ежедневно обрабатывать тысячи и даже миллионы событий. Благодаря интуитивно понятному API и безупречной надежности библиотека Kafka Streams по праву занимает в этих системах центральное место. Она обеспечивает именно ту мощь и простоту, которые необходимы для управления обработкой событий в реальном времени или обмена сообщениями между микросервисами.

Создавайте приложения потоковой обработки на удивительной платформе Apache Kafka. Переработанное новое издание охватывает более широкий спектр потоковых архитектур и включает интеграцию данных с Kafka Connect. Здесь собраны практические примеры, которые познакомят вас с компонентами и брокерами, а также пояснят особенности управления схемами. Попутно вы освоите приемы использования Kafka с фреймворком Spring, методы низкоуровневого управления узлами-обрабатчиками и хранилищами состояний, сохранения данных событий с помощью ksqlDB и тестирования потоковых приложений.

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



# *Краткое содержание*

---

## **Часть I**

Глава 1. Добро пожаловать в Kafka Streams . . . . .	28
Глава 2. Брокеры Kafka . . . . .	43

## **Часть II**

Глава 3. Schema Registry . . . . .	70
Глава 4. Клиенты Kafka . . . . .	113
Глава 5. Kafka Connect . . . . .	157

## **Часть III**

Глава 6. Разработка приложений Kafka Streams . . . . .	183
Глава 7. Потоки данных и состояние . . . . .	213
Глава 8. KTable API . . . . .	252
Глава 9. Оконные операции и отметки времени . . . . .	283
Глава 10. API узлов-обработчиков . . . . .	326
Глава 11. ksqlDB . . . . .	349
Глава 12. Spring Kafka . . . . .	381
Глава 13. Интерактивные запросы Kafka Streams . . . . .	403
Глава 14. Тестирование . . . . .	418

## **Приложения**

Приложение А. Практикум по совместимости схем . . . . .	442
Приложение Б. Ресурсы Confluent . . . . .	452
Приложение В. Работа с Avro, Protobuf и JSON Schema. . . . .	454
Приложение Г. Архитектура Kafka Streams . . . . .	477

# *Оглавление*

---

Предисловие . . . . .	16
Вступление . . . . .	17
Благодарности . . . . .	18
О книге . . . . .	20
Кому стоит прочитать эту книгу. . . . .	20
Структура издания . . . . .	21
О коде . . . . .	22
Другие онлайн-ресурсы . . . . .	23
Об авторе . . . . .	24
Иллюстрация на обложке . . . . .	25
От издательства . . . . .	26

## **Часть I**

<b>Глава 1.</b> Добро пожаловать в Kafka Streams . . . . .	28
1.1. Потоковая обработка событий . . . . .	28
1.2. Что такое событие . . . . .	31
1.3. Пример потока событий . . . . .	32
1.4. Знакомство с Apache Kafka, платформой потоковой обработки событий . . . . .	33
1.4.1. Брокеры Kafka . . . . .	35
1.4.2. Schema Registry . . . . .	35
1.4.3. Производители и потребители . . . . .	36
1.4.4. Kafka Connect . . . . .	36

---

1.4.5. Kafka Streams . . . . .	37
1.4.6. ksqlDB . . . . .	38
1.5. Конкретный пример применения платформы потоковой обработки событий Kafka . . . . .	39
Итоги главы . . . . .	42
<b>Глава 2. Брокеры Kafka . . . . .</b>	<b>43</b>
2.1. Введение в брокеры Kafka . . . . .	43
2.2. Запросы на производство . . . . .	44
2.3. Запросы на извлечение . . . . .	45
2.4. Топики и разделы . . . . .	46
2.4.1. Смещения . . . . .	48
2.4.2. Выбор правильного количества разделов . . . . .	50
2.5. Отправка первых сообщений . . . . .	51
2.5.1. Создание топика . . . . .	51
2.5.2. Создание записей в командной строке . . . . .	52
2.5.3. Потребление записей из командной строки . . . . .	52
2.5.4. Разделы в действии . . . . .	53
2.6. Сегменты . . . . .	54
2.6.1. Хранение данных . . . . .	55
2.6.2. Сжатые топики . . . . .	56
2.6.3. Содержимое каталога раздела топика . . . . .	58
2.7. Многоуровневое хранилище . . . . .	60
2.8. Метаданные кластера . . . . .	61
2.9. Ведущие и ведомые брокеры . . . . .	62
2.9.1. Репликация . . . . .	63
2.10. Проверка работоспособности брокера . . . . .	67
2.10.1. Процент простоя обработчика запросов . . . . .	68
2.10.2. Процент простоя сетевого обработчика . . . . .	68
2.10.3. Несинхронизированные разделы . . . . .	68
Итоги главы . . . . .	68

## Часть II

<b>Глава 3. Schema Registry . . . . .</b>	<b>70</b>
3.1. Объекты . . . . .	71
3.2. Что такое схема и зачем она нужна . . . . .	72
3.2.1. Что такое Schema Registry . . . . .	73

3.2.2. Запуск Schema Registry . . . . .	75
3.2.3. Архитектура . . . . .	75
3.2.4. Использование Schema Registry REST API . . . . .	77
3.2.5. Регистрация схемы . . . . .	77
3.2.6. Плагины и инструменты сериализации . . . . .	82
3.2.7. Выгрузка файла схемы . . . . .	83
3.2.8. Генерация кода из схем . . . . .	84
3.2.9. Полный пример . . . . .	85
3.3. Стратегии именования субъектов . . . . .	90
3.3.1. TopicNameStrategy . . . . .	91
3.3.2. RecordNameStrategy . . . . .	92
3.3.3. TopicRecordNameStrategy . . . . .	93
3.4. Совместимость схем . . . . .	96
3.4.1. Обратная совместимость . . . . .	96
3.4.2. Прямая совместимость . . . . .	97
3.4.3. Полная совместимость . . . . .	97
3.4.4. Отсутствие совместимости . . . . .	98
3.5. Ссылки на схемы . . . . .	99
3.6. Ссылки на схемы и наличие нескольких событий в топике . . . . .	104
3.7. Сериализаторы и десериализаторы Schema Registry . . . . .	107
3.7.1. Сериализаторы и десериализаторы Avro . . . . .	108
3.7.2. Protobuf . . . . .	108
3.7.3. JSON Schema . . . . .	109
3.8. Сериализация без Schema Registry . . . . .	110
Итоги главы . . . . .	112
<b>Глава 4. Клиенты Kafka . . . . .</b>	<b>113</b>
4.1. Знакомство с клиентами Kafka . . . . .	113
4.2. Производство записей с KafkaProducer . . . . .	115
4.2.1. Конфигурации производителя . . . . .	118
4.2.2. Семантика доставки в Kafka . . . . .	119
4.2.3. Назначение разделов . . . . .	120
4.2.4. Реализация своего механизма назначения разделов . . . . .	121
4.2.5. Подключение механизма назначения разделов . . . . .	123
4.2.6. Отметки времени . . . . .	123
4.3. Потребление записей с KafkaConsumer . . . . .	123
4.3.1. Интервал опроса . . . . .	126
4.3.2. Идентификатор группы . . . . .	127

---

4.3.3. Применение стратегий назначения разделов. . . . .	133
4.3.4. Статическое членство . . . . .	134
4.3.5. Фиксация смещений. . . . .	136
4.4. Семантика доставки «точно один раз» . . . . .	141
4.4.1. Идемпотентный производитель. . . . .	142
4.4.2. Транзакционный производитель . . . . .	144
4.4.3. Потребители в транзакциях . . . . .	147
4.4.4. Производители и потребители в транзакционном окружении. . . . .	148
4.5. Использование Admin API для программного управления топиками. . . . .	149
4.6. Обработка нескольких типов событий в одном топике . . . . .	151
4.6.1. Создание нескольких типов событий . . . . .	152
4.6.2. Потребление событий нескольких типов . . . . .	153
Итоги главы . . . . .	156
<b>Глава 5. Kafka Connect . . . . .</b>	<b>157</b>
5.1. Введение в Kafka Connect . . . . .	158
5.2. Интеграция внешних приложений с Kafka . . . . .	159
5.3. Начало работы с Kafka Connect . . . . .	160
5.4. Применение преобразований отдельных сообщений . . . . .	166
5.5. Добавление коннектора-приемника. . . . .	169
5.6. Создание и развертывание собственного коннектора . . . . .	171
5.6.1. Реализация коннектора . . . . .	171
5.6.2. Создание динамического коннектора с помощью потока мониторинга . . . . .	175
5.6.3. Создание пользовательского преобразования . . . . .	177
Итоги главы . . . . .	180

### Часть III

<b>Глава 6. Разработка приложений Kafka Streams . . . . .</b>	<b>183</b>
6.1. Обзор Kafka Streams. . . . .	183
6.2. Kafka Streams DSL. . . . .	184
6.3. Программа Hello World для Kafka Streams . . . . .	185
6.3.1. Создание топологии для приложения Yelling . . . . .	186
6.3.2. Настройка Kafka Streams . . . . .	191
6.3.3. Создание объектов Serde . . . . .	191
6.4. Маскировка номеров кредитных карт и отслеживание вознаграждений за покупки . . . . .	193
6.4.1. Создание узла-источника и маскирующего узла-обработчика. . . . .	194

6.4.2. Добавление узла, извлекающего закономерности при выполнении покупок . . . . .	196
6.4.3. Создание узла, определяющего величину вознаграждения . . . . .	197
6.4.4. Использование объектов Serde с сериализаторами и десериализаторами в Kafka Streams . . . . .	199
6.4.5. Kafka Streams и Schema Registry . . . . .	200
6.5. Интерактивная разработка . . . . .	201
6.6. Выбор событий для обработки . . . . .	203
6.6.1. Фильтрация покупок . . . . .	203
6.6.2. Разделение/ветвление потока данных . . . . .	204
6.6.3. Именование узлов топологии . . . . .	208
6.6.4. Динамическая маршрутизация сообщений . . . . .	209
Итоги главы . . . . .	211
<b>Глава 7. Потоки данных и состояние . . . . .</b>	<b>213</b>
7.1. С состоянием и без состояния . . . . .	214
7.2. Добавление операций с состоянием в Kafka Streams . . . . .	215
7.2.1. Подробности о группировке . . . . .	216
7.2.2. Агрегирование и свертка . . . . .	218
7.2.3. Перераспределение данных . . . . .	221
7.2.4. Проактивное перераспределение . . . . .	226
7.2.5. Перераспределение для увеличения количества задач . . . . .	228
7.2.6. Использование оптимизаций Kafka Streams . . . . .	229
7.3. Соединение потоков . . . . .	231
7.3.1. Реализация соединения потоков . . . . .	232
7.3.2. Особенности работы соединений . . . . .	233
7.3.3. ValueJoiner . . . . .	235
7.3.4. JoinWindows . . . . .	236
7.3.5. Совместимое секционирование . . . . .	237
7.3.6. StreamJoined . . . . .	238
7.3.7. Другие варианты соединений . . . . .	238
7.3.8. Внешние соединения . . . . .	238
7.3.9. Внешнее левое соединение . . . . .	239
7.4. Хранилища состояний в Kafka Streams . . . . .	240
7.4.1. Восстановление хранилища состояний с помощью топика журнализирования изменений . . . . .	241
7.4.2. Резервные задачи . . . . .	243

---

7.4.3. Назначение хранилищ состояний в Kafka Streams . . . . .	244
7.4.4. Местоположение хранилищ состояний в файловой системе. . . . .	245
7.4.5. Именование операций с состоянием . . . . .	246
7.4.6. Настройка типа хранилища . . . . .	249
7.4.7. Настройка топиков журнализирования изменений. . . . .	250
Итоги главы . . . . .	251
<b>Глава 8. KTable API. . . . .</b>	<b>252</b>
8.1. KTable: поток обновлений . . . . .	253
8.1.1. Обновления записей (журнал изменений) . . . . .	255
8.1.2. KStream и KTable в действии . . . . .	256
8.2. Объекты KTable имеют хранимое состояние . . . . .	258
8.3. KTable . . . . .	259
8.4. Агрегирование с помощью KTable. . . . .	259
8.5. GlobalKTable . . . . .	265
8.6. Соединение с таблицами . . . . .	267
8.6.1. Соединение потоков и таблиц . . . . .	269
8.6.2. Версионированные таблицы KTable . . . . .	271
8.6.3. Соединение потоков и глобальных таблиц . . . . .	273
8.6.4. Соединение двух таблиц . . . . .	276
Итоги главы . . . . .	282
<b>Глава 9. Оконные операции и отметки времени . . . . .</b>	<b>283</b>
9.1. Роль и типы окон . . . . .	287
9.1.1. Прыгающие окна . . . . .	289
9.1.2. Кувыркающиеся окна . . . . .	292
9.1.3. Сеансовые окна . . . . .	294
9.1.4. Скользящие окна . . . . .	297
9.1.5. Выравнивание окон по времени. . . . .	301
9.1.6. Получение результатов в окне для анализа. . . . .	303
9.2. Обработка неупорядоченных данных с отсрочкой. . . . .	309
9.3. Окончательные результаты оконного агрегирования . . . . .	312
9.3.1. Строгая буферизация . . . . .	317
9.3.2. Жадная буферизация . . . . .	318
9.4. Отметки времени в Kafka Streams . . . . .	319
9.5. TimestampExtractor . . . . .	321
9.5.1. Метод WallclockTimestampExtractorSystem.currentTimeMillis() . . . . .	322

9.5.2. Пользовательский экстрактор отметок времени . . . . .	322
9.5.3. Выбор TimestampExtractor для использования . . . . .	323
9.6. Время потока . . . . .	323
Итоги главы . . . . .	325
<b>Глава 10.</b> API узлов-обработчиков . . . . .	326
10.1. Создание топологии с использованием источников, узлов-обработчиков и стоков . . . . .	327
10.1.1. Добавление узла-источника . . . . .	328
10.1.2. Добавление узла-обработчика . . . . .	329
10.1.3. Добавление узла-приемника . . . . .	332
10.2. Углубляемся в API узлов-обработчиков на примере узла биржевой аналитики . . . . .	334
10.2.1. Узел-обработчик показателей акций . . . . .	335
10.2.2. Семантика пунктуации . . . . .	337
10.2.3. Метод process() . . . . .	339
10.2.4. Выполнение пунктуатора . . . . .	341
10.3. Агрегирование, управляемое данными . . . . .	342
10.4. Интеграция API узлов-обработчиков и Kafka Streams API . . . . .	346
Итоги главы . . . . .	348
<b>Глава 11.</b> ksqlDB . . . . .	349
11.1. Знакомство с ksqlDB . . . . .	350
11.2. Подробности о потоковых запросах . . . . .	353
11.3. Постоянные запросы, а также запросы push и pull . . . . .	361
11.4. Создание потоков и таблиц . . . . .	367
11.5. Интеграция с Schema Registry . . . . .	370
11.6. Расширенные возможности ksqlDB . . . . .	374
Итоги главы . . . . .	380
<b>Глава 12.</b> Spring Kafka . . . . .	381
12.1. Введение в Spring . . . . .	381
12.2. Использование Spring для создания приложений с поддержкой Kafka . . . . .	384
12.2.1. Компоненты приложения Spring Kafka . . . . .	387
12.2.2. Расширенные требования к приложению . . . . .	392
12.3. Spring Kafka Streams . . . . .	396
Итоги главы . . . . .	402

---

<b>Глава 13.</b> Интерактивные запросы Kafka Streams . . . . .	403
13.1. Kafka Streams и обмен информацией . . . . .	404
13.2. Интерактивные запросы . . . . .	405
13.2.1. Создание приложения интерактивных запросов с помощью Spring Boot . . . . .	408
Итоги главы . . . . .	416
<b>Глава 14.</b> Тестирование . . . . .	418
14.1. Разница между модульным и интеграционным тестированием . . . . .	419
14.1.1. Тестирование производителей и потребителей Kafka . . . . .	420
14.1.2. Тестирование операторов Kafka Streams . . . . .	424
14.1.3. Тестирование топологии . . . . .	426
14.1.4. Тестирование сложных приложений Kafka Streams . . . . .	430
14.1.5. Эффективное интеграционное тестирование . . . . .	434
Итоги главы . . . . .	439

## Приложения

<b>Приложение А.</b> Практикум по совместимости схем . . . . .	442
A.1. Обратная совместимость . . . . .	443
A.2. Прямая совместимость . . . . .	446
A.3. Полная совместимость . . . . .	449
<b>Приложение Б.</b> Ресурсы Confluent . . . . .	452
B.1. Confluent Cloud . . . . .	452
B.2. Интерфейс командной строки Confluent . . . . .	452
B.3. Запуск Confluent локально . . . . .	453
<b>Приложение В.</b> Работа с Avro, Protobuf и JSON Schema . . . . .	454
B.1. Apache Avro . . . . .	454
B.1.1. Значения по умолчанию и псевдонимы . . . . .	455
B.1.2. Объединения . . . . .	456
B.2. Protocol Buffers . . . . .	457
B.2.1. Сложные сообщения . . . . .	459
B.2.2. Импортирование . . . . .	460
B.2.3. Тип oneof . . . . .	461
B.2.4. Генерация кода . . . . .	464
B.2.5. Конкретные и динамические типы . . . . .	465

B.3. JSON Schema . . . . .	465
B.3.1. Вложенные объекты . . . . .	466
B.3.2. Ссылки JSON . . . . .	467
B.3.3. Ссылки на схему JSON Schema в Schema Registry . . . . .	469
B.3.4. Генерация кода на основе схемы JSON Schema . . . . .	470
B.3.5. Конкретные и обобщенные типы . . . . .	473
<b>Приложение Г. Архитектура Kafka Streams . . . . .</b>	<b>477</b>
Г.1. Общий вид архитектуры . . . . .	477
Г.2. Потребители и производители в Kafka Streams . . . . .	478
Г.3. Назначение, распределение и обработка событий . . . . .	480
Г.4. Потоки выполнения в Kafka Streams: StreamThread . . . . .	484
Г.5. Обработка записей . . . . .	488

## Отзывы к первому изданию

Книга отлично подойдет для знакомства с Kafka Streams — ключевым инструментом для создания событийно-ориентированных приложений.

*Ния Нархид (Neha Narkhede),  
одна из создателей Apache Kafka*

Исчерпывающее руководство по Kafka Streams — от знакомства до эксплуатации!

*Боян Джуркович (Bojan Djurkovic), Cvent*

Устраниет разрыв между обработкой сообщений с помощью брокеров и потоковой аналитикой в масштабе реального времени.

*Джим Мантий — младший (Jim Mantheiy,Jr.),  
Next Century*

Ценное руководство и для изучения потоковой обработки, и для постоянного использования в качестве справочника.

*Робин Коу (Robin Coe), TD Bank*

Потоковая обработка может быть сложной, но эта книга научит вас использовать данную технологию с умом.

*Хосе Сан Леандро (Jose San Leandro),  
программист и инженер DevOps, OSOCO.es*

Отличный источник информации о Kafka Streams с практическими примерами.

*Ласло Хегедус (László Hegedüs), PhD,  
разработчик программного обеспечения,  
Erlang Solutions*

# *Предисловие*

---

Когда происходит бизнес-событие, традиционные системы хранения данных регистрируют его, но откладывают обработку данных на более позднее время. Напротив, Apache Kafka — это платформа потоковой передачи данных, спроектированная так, чтобы реагировать на бизнес-события в режиме реального времени. За последнее десятилетие Apache Kafka превратилась в стандарт потоковой передачи данных. Сотни тысяч организаций, включая многие крупнейшие предприятия с мировым именем, используют Kafka для быстрого реагирования на происходящее в их бизнесе, что помогает им улучшать качество обслуживания клиентов, повышать эффективность, снижать риски и т. д., и все это за несколько коротких секунд.

Приложения, основанные на Kafka, управляются событиями. Обычно они принимают один или несколько потоков данных и непрерывно преобразуют их в новый поток. Преобразование часто включает потоковые операции, такие как фильтрация, проекция, объединение и агрегирование. Выражение этих операций в низкоуровневом коде на Java неэффективно и подвержено ошибкам, поэтому Kafka Streams предлагает разработчикам ряд высокуюровневых абстракций для краткого выражения многих общих потоковых операций и является очень мощным инструментом для создания событийно-управляемых приложений на Java.

Билл много лет входил в состав команды разработчиков Kafka и по праву считается экспертом в Kafka Streams. Он не только понимает технологию, лежащую в основе Kafka Streams, но также знает, как использовать Kafka Streams для решения практических задач. В этой книге Билл познакомит вас с ключевыми понятиями Kafka Streams и покажет множество практических примеров создания приложений, управляемых событиями, с использованием Kafka Streams и других Kafka API и Schema Registry. Если вы разработчик и желаете узнать, как на основе Kafka создавать событийно-ориентированные приложения следующего поколения, то эта книга станет для вас бесценным источником знаний. Наслаждайтесь книгой и широтой возможностей потоковой передачи данных!

*Дзюн Рао (Jun Rao), сооснователь  
Confluent и Apache Kafka*

<https://t.me/javalib>

# *Вступление*

---

Закончив работу над первым изданием «Kafka Streams», я думал, что сделал все, что планировал. Но мое понимание экосистемы Kafka и мое восхищение Kafka Streamsросли. Я увидел, что Kafka Streams обладает гораздо более широкими возможностями, чем я думал изначально. Кроме того, я заметил другие важные особенности создания приложений с поддержкой потоковой передачи событий; Kafka Streams по-прежнему является ключевым игроком на этом поле, но не единственным. Я понял, что Apache Kafka можно считать центральной нервной системой для данных организации. Если Kafka — это центральная нервная система, то Kafka Streams — это жизненно важный орган, выполняющий ряд необходимых операций.

Но в своей работе Kafka Streams опирается на другие компоненты, когда требуется перенести события в Kafka или экспорттировать их во внешний мир, где они могут быть использованы. Я говорю о клиентах — производителях и потребителях и Kafka Connect. Когда я собрал все воедино, то понял, что эти другие компоненты совершенно необходимы, чтобы картина потоковой передачи событий получилась полной. Добавив ко всему этому существенные улучшения, внесенные в Kafka Streams после 2018 года, я понял, что хочу написать второе издание.

Но я хотел не просто добавить косметические штрихи к предыдущему изданию, а выразить свое улучшенное понимание и представить более полный охват всей экосистемы Kafka. Это означало расширение некоторых тем, преобразование разделов в целые главы (например, описание производителей и потребителей) и добавление совершенно новых глав (например, о Connect и Schema Registry). Существующие главы тоже не остались без внимания и во втором издании были значительно обновлены и дополнены с целью передать мое более глубокое понимание.

Взяться за второе издание во время пандемии Covid-19 было нелегко, и работа над ним не обошлась без серьезных личных проблем. Но, в конце концов, эта работа стоила каждой минуты, потраченной на нее, и если бы я вернулся назад во времени, то принял бы то же самое решение.

Я надеюсь, что новые читатели «Kafka Streams» сочтут эту книгу ценным источником информации, а читатели первого издания оценят внесенные улучшения.

# Благодарности

---

Прежде всего мне хотелось бы поблагодарить мою жену Бет и выразить ей признательность за всю поддержку, которую она мне оказывала в процессе работы над вторым изданием. Написание первого издания занимает немало времени, и кому-то из вас могло бы показаться, что написать второе издание проще, потому что нужно всего лишь внести корректизы в описание изменившихся API. Но в данном случае я хотел расширить свою предыдущую работу и решил полностью переписать ее. Бет никогда не подвергала сомнению мое решение и полностью поддержала меня, и, как и прежде, без ее содействия у меня ничего бы не получилось. Бет, ты потрясающая, и я очень рад, что ты моя жена. Я хотел бы также поблагодарить моих детей, которые тоже поддерживали и подбадривали меня, пока я работал над вторым изданием.

Я благодарю моего редактора из издательства Manning, Фрэнсиса Лефковица (Frances Lefkowitz), чьи советы эксперта и бесконечное терпение превратили написание данной книги *почти* в развлечение. Я также благодарен Джону Гатри (John Guthrie) за меткие технические замечания и Карстену Стробеку (Karsten Strobaek), техническому корректору, за великолепную работу по анализу исходного кода. Кроме того, свой вклад в книгу внесли многие сотрудники издательства Manning, и я очень благодарен им всем.

Я также хотел бы поблагодарить всех разработчиков и сообщество Kafka Streams за создание Kafka Streams — лучшей библиотеки потоковой обработки из существующих ныне. Я хочу выразить признательность всем разработчикам Kafka за создание столь превосходного программного продукта, особенно Джою Крепсу (Jay Kreps), Нии Нархид (Neha Narkhede) и Дзюну Рао (Jun Rao), не только за саму идею Kafka в первую очередь, но и за основание компании Confluent — потрясающего и вдохновляющего места для работы. И особое спасибо Дзюну Рао за предисловие ко второму изданию.

Наконец, я благодарю рецензентов за их упорный труд и бесценные отзывы, которые помогли сделать эту книгу лучше. Спасибо вам: Ален Кунио (Alain Couniot), Аллен Гуч (Allen Gooch), Andres Sacco, Балакришнан

Баласубраманиан (Balakrishnan Balasubramanian), Кристиан Тудаль (Christian Thoudahl), Даниэла Запата (Daniela Zapata), Дэвид Онг (David Ong), Эзра Симелофф (Ezra Simeloff), Джампьеро Гранателла (Giampiero Granatella), Джон Реслер (John Roesler), Хоце Сан Леандро Армендарис (Jose San Leandro Armendáriz), Джозеф Паход (Joseph Pachod), Кент Спилнер (Kent Spillner), Мансур Мухитдинов (Manzur Mukhitdinov), Майкл Хейл (Michael Heil), Милорад Имбра (Milorad Imbra), Милош Миливоевич (Miloš Milivojević), Наджиб Ариф (Najeeb Arif), Натан Б. Крокер (Nathan B. Crocker), Робин Коу (Robin Coe), Руй Лю (Rui Liu), Самбасива Андалури (Sambasiva Andaluri), Скотт Хуанг (Scott Huang), Саймон Хьюитт (Simon Hewitt), Саймон Чок (Simon Tschöke), Стэнфорд С. Гиллори (Stanford S. Guillory), Тан Ви (Tan Wee), Томас Пеклак (Thomas Peklak) и Зородзай Мукуя (Zorodzayi Mukuya).

# *O книге*

---

Я написал второе издание «Kafka Streams», чтобы научить вас создавать приложения потоковой передачи событий, использующие Kafka Streams и другие компоненты экосистемы Kafka, клиенты Producer и Consumer, Connect и Schema Registry. Такой подход выбран по той простой причине, что для максимальной эффективности вашему приложению понадобится не только Kafka Streams, но и другие инструменты. Я писал эту книгу с точки зрения парного программирования, представляя, что сижу рядом с вами, пока вы пишете код и изучаете API. Я познакомлю вас с брокером Kafka и клиентами Producer и Consumer. Затем расскажу о роли схем и о том, как управлять ими с помощью Schema Registry, а также о том, как Kafka Connect связывает внешние компоненты с Kafka. Мы начнем с создания простого приложения и будем добавлять в него новый функционал по мере погружения в Kafka Streams. Вы также познакомитесь с ksqlDB, узнаете, как выполнять тестирование и мониторинг, и, наконец, разберетесь с особенностями интегрирования Kafka с популярной средой Spring.

## **КОМУ СТОИТ ПРОЧИТАТЬ ЭТУ КНИГУ**

Эта книга подойдет для любого разработчика, который хочет разобраться в потоковой обработке. Понимание распределенного программирования поможет лучше изучить Kafka и Kafka Streams. Было бы неплохо знать и сам фреймворк Kafka, но это не обязательно: я расскажу вам все, что нужно. Опытные разработчики Kafka, как и новички, благодаря этой книге освоят разработку интересных приложений для потоковой обработки с помощью библиотеки Kafka Streams. Java-разработчики среднего и высокого уровня, уже привычные к таким понятиям, как сериализация, научатся применять свои навыки для создания приложений Kafka Streams. Исходный код книги написан на Java 17, и в нем существенно используется синтаксис лямбда-выражений Java, так что умение работать с лямбда-функциями (даже на другом языке программирования) вам пригодится.

## СТРУКТУРА ИЗДАНИЯ

Книга состоит из трех частей, разбитых на 14 глав. Несмотря на название «Kafka Streams», эта книга охватывает всю платформу потоковой передачи событий Kafka. Соответственно, первые пять глав охватывают различные компоненты: брокеры Kafka, потребители и производители, Schema Registry и Kafka Connect. Такой подход не лишен смысла, особенно если учесть, что Kafka Streams — это абстракция над потребителями и производителями. Поэтому если вы уже знакомы с Kafka, Connect и Schema Registry или с нетерпением ждете возможности начать работу с Kafka Streams, то можете смело переходить к части III.

Часть I знакомит с потоковой передачей событий и описывает различные части экосистемы Kafka, чтобы показать вам общую картину, как все это работает и сочетается друг с другом. В этих главах также приводятся основы Kafka для тех, кто ничего о ней не знает или хотел бы освежить свои знания.

- Глава 1 описывает историю вопроса: как и почему потоковая обработка стала необходимым элементом широкомасштабной обработки данных в режиме реального времени. В ней также приводится ментальная модель различных компонентов: брокеров, клиентов, Kafka Connect, Schema Registry и, конечно же, Kafka Streams. Здесь я не буду демонстрировать программный код, а просто опишу, как работает Kafka Streams.
  - Глава 2 — руководство для разработчиков, еще не имевших дела с Kafka. Здесь я расскажу о роли брокеров, топиков и разделов и о некоторых аспектах мониторинга. Те, у кого уже есть опыт работы с Kafka, могут смело пропустить эту главу.
- Часть II продолжает вводное знакомство и охватывает вопросы передачи данных в Kafka и из нее, а также управление схемами.
- Глава 3 посвящена использованию Schema Registry — компоненту, помогающему управлять эволюцией схем данных. Приоткрою один секрет: в Kafka схемы используются всегда — даже если вы не определяете схему явно, она все равно присутствует неявно.
  - В главе 4 обсуждаются клиенты Kafka: производители и потребители. Клиенты — это компоненты Kafka, позволяющие получать и отправлять данные. Они служат строительными блоками для Kafka Connect и Kafka Streams.
  - Глава 5 посвящена Kafka Connect. Kafka Connect дает возможность получать данные в Kafka через коннекторы-источники и экспорттировать их во внешние системы с помощью коннекторов-приемников.

Часть III составляет суть книги и охватывает разработку приложений Kafka Streams. Здесь, кроме всего прочего, вы познакомитесь с ksqlDB и тестированием приложения потоковой передачи событий, а в заключение узнаете, как осуществляется интеграция Kafka с Spring Framework.

- Глава 6 — это введение в Kafka Streams. Здесь вы найдете приложение Hello World, а далее и более реалистичный пример: приложение для вымышленного розничного торговца. Попутно вы познакомитесь с Kafka Streams DSL.
- В главе 7 мы продолжим изучение Kafka Streams и обсудим понятие состояния и выясним, почему оно необходимо для потоковых приложений. В этой главе вы узнаете о реализации таких операций, как агрегирование и соединения.

- Глава 8 познакомит вас с новым понятием: интерфейсом `KTable`. Если `KStream` представляет собой поток событий, то `KTable` – это поток взаимосвязанных событий или поток, предназначенный для обновления записей.
- Глава 9 посвящена оконным операциям и отметкам времени. Оконное агрегирование позволяет группировать результаты по времени, а отметки времени в записях управляют действиями.
- В главе 10 мы углубимся в Kafka Streams Processor API. До сих пор вы имели дело с высокоуровневым предметно-ориентированным языком (Domain Specific Language, DSL), а здесь научитесь использовать Processor API на случай, если вам понадобится осуществлять более точное управление.
- Глава 11 познакомит вас с процессом разработки. Здесь вы познакомитесь с `ksqlDB` – инструментом, позволяющим писать приложения потоковой передачи событий без программного кода, а исключительно на SQL.
- Глава 12 обсуждает использование Spring Framework с клиентами Kafka и Kafka Streams. Spring позволяет писать модульный и легко тестируемый код, представляя механизм внедрения зависимостей для подключения ваших приложений.
- Глава 13 знакомит вас с Kafka Streams Interactive Queries, или просто IQ. IQ – это возможность напрямую извлекать состояние операций в Kafka Streams из хранилища состояний. Здесь вы примените знания, полученные в главе 12, для создания веб-приложения IQ с поддержкой Spring.
- Глава 14 рассказывает о тестировании приложений Kafka Streams. Вы научитесь тестировать клиентские приложения с топологией Kafka Streams, узнаете разницу между модульным и интеграционным тестированием и о том, когда их применять.

Наконец, в книге имеется четыре приложения с дополнительными пояснениями.

- Приложение А поможет получить практический опыт использования Schema Registry в различных режимах совместимости схем.
- В приложении Б представлена информация об использовании Confluent Cloud, знание которой поможет вам в разработке приложений потоковой передачи событий.
- Приложение В содержит обзор использования различных типов схем Avro, Protobuf и JSON Schema.
- Приложение Г описывает архитектуру и внутреннее устройство Kafka Streams.

## О КОДЕ

Эта книга содержит множество примеров исходного кода как в пронумерованных листингах, так и внутри обычного текста. В обоих случаях исходный код отформатирован с помощью такого монотипного шрифта, чтобы можно было отличить его от обычного текста.

Во многих случаях исходный код был переформатирован: добавлены разрывы строк и изменены отступы, чтобы оптимально использовать место на странице. В редких случаях даже этого было недостаточно и листинги содержат символы продолжения строки (→).

Кроме того, из исходного кода часто убраны комментарии, если он описывается в тексте. Многие листинги сопровождаются пояснениями к коду, подчеркивающими важные понятия.

Наконец, важно отметить, что многие примеры кода не самодостаточны и содержат лишь фрагменты кода, наиболее важные для обсуждаемого вопроса. Все примеры из книги в их изначальном виде можно найти в прилагаемом к данной книге исходном коде.

Исходный код примеров находится на GitHub по адресу <https://github.com/bbejeck/KafkaStreamsInAction2ndEdition>. Исходный код для книги представляет собой комплексный проект, использующий утилиту сборки Gradle (<https://gradle.org>). Этот проект можно импортировать в среды разработки IntelliJ или Eclipse с помощью соответствующих команд. Полные инструкции по применению исходного кода и навигации по нему можно найти в прилагаемом к нему файле README.md.

## ДРУГИЕ ОНЛАЙН-РЕСУРСЫ

- Документация по фреймворку Apache Kafka: <https://kafka.apache.org/>.
- Документация по платформе Confluent: <https://docs.confluent.io/current>.
- Документация по библиотеке Kafka Streams: <https://kafka.apache.org/documentation/streams/>.
- Документация по движку ksqlDB: <https://ksqldb.io/>.
- Документация по фреймворку Spring Framework: <https://spring.io/>.

## *Об авторе*

---

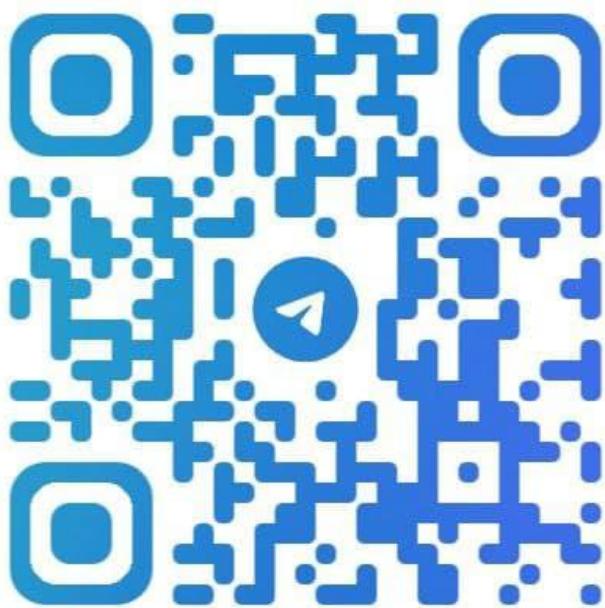
**Билл Беджек** (Bill Bejeck) — участник проекта и член консультационного совета Apache Kafka®. Более 20 лет проработал инженером-программистом. В настоящее время работает в компании Confluent как инженер DevX, а до этого более трех лет проработал инженером в команде Kafka Streams. До Confluent он работал над созданием различных приложений для правительства США и использовал распределенное программное обеспечение, такое как Apache Kafka, Spark и Hadoop. Кроме того, он регулярно ведет блог «Беспорядочные размышления о написании кода» (<http://codingjunkie.net/>).

## *Иллюстрация на обложке*

---

На обложке второго издания книги «Kafka Streams» представлена иллюстрация под названием «Привычки турецкого джентльмена в 1700 году» из книги Томаса Джеффериса, изданной между 1757 и 1772 годами.

В те дни по одежде было легко понять, где живут люди, определить их профессию или социальное положение. Издательство Manning прославляется изобретательность и инициативу компьютерного бизнеса, помещая на обложки своих книг изображения из подобных этой коллекций, иллюстрирующих богатое разнообразие региональной культуры многовековой давности.



**@JAVALIB**

<https://t.me/javalib>

# *Часть I*

В первой части вы познакомитесь с событиями и потоковой обработкой событий в целом. Потоковая обработка событий — это подход к разработке программного обеспечения, который рассматривает события как основные входные и выходные данные приложения. Но для разработки эффективного приложения, осуществляющего потоковую обработку событий, сначала нужно узнать, что такое событие (скажу по секрету: события — это все!). Затем вы прочтете о том, какие варианты использования считаются хорошими кандидатами для приложений потоковой обработки событий, а какие — нет.

Сначала вы узнаете, что такое брокер Kafka, который находится в центре экосистемы Kafka, и какие задачи он решает. Затем вы узнаете, что такое Schema Registry, производители и потребители, Connect и Kafka Streams, и познакомитесь с их ролями. Я также представлю вам платформу потоковой обработки событий Apache Kafka. Хотя эта книга посвящена клиентской библиотеке Kafka Streams, она является частью большого инструментария, позволяющего разрабатывать приложения потоковой обработки событий. Если первая часть оставит у вас больше вопросов, чем ответов, не волнуйтесь; я отвечу на все ваши вопросы в последующих главах.

# 1

## Добро пожаловать в Kafka Streams

### В этой главе

- ✓ Определение событий и потоковой обработки событий.
- ✓ Введение в Kafka — платформу потоковой обработки событий.
- ✓ Конкретный пример применения платформы.

Постоянный приток данных порождает все больше возможностей для потребителя, и все чаще пользователями этих данных становятся программные системы, обращающиеся к другим программным системам. Давайте посмотрим, например, как работает ваше любимое приложение потокового вещания, когда вы смотрите фильмы. Вы входите в приложение, выбираете фильм, смотрите его, а затем можете поставить свою оценку и оставить свой отзыв о фильме. Эта простая череда взаимодействий генерирует несколько событий, фиксируемых службой потокового вещания. Но эта информация нуждается в анализе, чтобы быть полезной для бизнеса. И здесь в игру вступает все остальное программное обеспечение.

### 1.1. ПОТОКОВАЯ ОБРАБОТКА СОБЫТИЙ

Программные системы потребляют и хранят всю информацию, полученную в процессе взаимодействий с вами и другими подписчиками. Затем другие программные системы используют эту информацию, чтобы предложить вам рекомендации и передать потоковому сервису информацию о том, что от него может потребоваться в будущем. Теперь представьте, что этот процесс происходит сотни тысяч или даже миллионы раз в день, и вы поймете, какой огромный объем информации должно получить

и обработать программное обеспечение, чтобы предприятие могло соответствовать требованиям и ожиданиям клиентов и оставаться конкурентоспособным.

С другой стороны, любые действия современных потребителей, от просмотра фильма онлайн до покупки пары обуви в обычном магазине, генерируют *события*. Чтобы организация выжила и преуспела в нашей цифровой экономике, у нее должен иметься эффективный способ получения этих событий и реагирования на них. Другими словами, компании должны искать способы идти в ногу с этим бесконечным потоком событий, если хотят удовлетворить клиентов и обеспечить устойчивый рост прибыли. Этот бесконечный поток разработчики так и называют — *потоком событий*. И все чаще для удовлетворения спроса на эту бесконечную цифровую деятельность привлекают *платформы потоковой обработки событий*, которые используют последовательность приложений потоковой обработки событий.

Платформа потоковой обработки событий подобна нашей центральной нервной системе, которая обрабатывает миллионы событий (нервных сигналов) и в ответ посыпает сообщения соответствующим частям тела. Наши сознательные мысли и действия являются ответами на некоторые из этих сообщений. Когда мы голодны и открываем холодильник, центральная нервная система получает сообщение и посыпает руке приказ потянуться за красивым красным яблоком на первой полке. Другие действия, такие как учащение пульса в ожидании захватывающих новостей, обрабатываются бессознательно.

Платформа потоковой обработки фиксирует события, генерируемые мобильными устройствами, взаимодействием клиентов с веб-сайтами, онлайн-активностью, отслеживанием поставок и другими бизнес-транзакциями. Но платформа, как и нервная система, делает больше, чем просто фиксирует события. Ей также нужен механизм для надежной передачи и хранения информации из этих событий в том порядке, в котором они произошли. Затем другие приложения могут обрабатывать или анализировать события, чтобы извлекать различные биты этой информации.

Обработка потока событий в реальном времени имеет большое значение для принятия решений, чувствительных к времени. Например, не выглядит ли подозрительной эта покупка у клиента X? Указывают ли сигналы от этого датчика температуры, что что-то в производственном процессе пошло не так? Была ли отправлена информация о маршрутизации в соответствующий отдел компании?

Но ценность платформы потоковой обработки событий не связана с немедленным получением информации. Наличие надежного хранилища позволяет нам вернуться и исследовать поток событий в первоначальном необработанном виде, выполнить некоторые манипуляции с данными для их более глубокого исследования или воспроизвести последовательность событий, чтобы попытаться понять, что привело к определенному результату. Например, сайт электронной коммерции предлагает фантастическую скидку на некоторые товары в выходные после большого праздника. Реакция на распродажу настолько сильная, что приводит к сбою нескольких серверов и останавливает работу бизнеса на несколько минут. Воспроизведя произошедшие события, инженеры могут лучше понять, что вызвало сбой и как исправить систему, чтобы она могла справиться с большим внезапным наплывом покупателей.

Итак, где нужны приложения потоковой обработки событий? Поскольку все в жизни можно считать событием, любая предметная область выигрывает от потоковой

обработки событий. Но есть области, где это особенно важно. Вот несколько типичных примеров.

- *Мошенничество с платежными картами* — владелец платежной карты может не заметить ее кражи, но путем анализа покупок и сравнения их с устоявшимися паттернами (местоположение, общий характер потребительских расходов) можно заметить кражу платежной карты и оповестить ее владельца.
- *Обнаружение вторжений* — возможность выявлять аномальное поведение в режиме реального времени имеет решающее значение для защиты конфиденциальных данных и благополучия организации.
- *Интернет вещей* — при использовании технологии Интернета вещей датчики размещаются в самых разных местах и часто отправляют данные. Возможность быстро собирать и осмысленно обрабатывать эти данные имеет большое значение, а отсутствие такой возможности снижает эффект от развертывания этих датчиков.
- *Финансовый сектор* — для принятия удачных решений о покупке или продаже брокерам и потребителям необходимо иметь возможность отслеживать рыночные цены и тенденции в режиме реального времени.
- *Обмен данными в режиме реального времени* — крупным организациям, таким как корпорации или холдинги, имеющим множество приложений, необходимо обмениваться данными стандартным способом и в режиме реального времени.

Итог: если поток событий несет важную и полезную информацию, то предприятиям и организациям необходимы событийно-ориентированные приложения, чтобы извлечь выгоду из полученной информации.

Но потоковые приложения подходят не для всех ситуаций. Они становятся необходимыми, когда данные находятся в разных местах или генерируется большой объем событий, требующих распределенных хранилищ данных. Поэтому, если есть возможность обойтись одним экземпляром базы данных, то потоковая обработка не нужна. Например, небольшой бизнес электронной коммерции или сайт городской администрации с преимущественно статическими данными не лучшие кандидаты для использования потоковой обработки событий.

В этой книге вы познакомитесь с особенностями потоков событий, поймете, когда и почему они нужны и как использовать платформу потоковой обработки событий Kafka для создания надежных и отзывчивых приложений. Узнаете, как использовать различные компоненты платформы Kafka для захвата событий и передачи их другим приложениям. Мы вместе рассмотрим использование компонентов платформы для выполнения простых действий, таких как запись (производство) или чтение (потребление) событий, в расширенных приложениях с состоянием, требующих сложных преобразований, чтобы вы знали, как можно решать бизнес-задачи с помощью подходов, основанных на потоковой обработке событий. Эта книга подойдет любому разработчику, который хочет заняться созданием приложений потоковой обработки событий.

Название книги «Kafka Streams» дает понять, что основное внимание в ней уделяется Kafka Streams, но вообще эта книга описывает всю платформу Kafka от начала до конца. Эта платформа включает ряд важнейших компонентов, таких как производители, потребители и схемы, с которыми вам придется поработать перед созданием своих потоковых приложений, и о них я расскажу в первой части. Таким

образом, мы не будем углубляться конкретно в Kafka Streams до конца второй части, то есть до конца главы 6.

Но такое расширенное знакомство стоит потраченных времени и сил. Kafka Streams – это абстракция, выстроенная поверх компонентов платформы потоковой обработки событий Kafka, поэтому понимание этих компонентов поможет вам лучше осознать, как можно использовать Kafka Streams.

## 1.2. ЧТО ТАКОЕ СОБЫТИЕ

Итак, мы определили поток событий, но что такое само событие? Мы определим событие просто как «что-то, что произошло» (<https://www.merriam-webster.com/dictionary/event>). Термин «событие» часто вызывает у многих ассоциации с чем-то *примечательным*, например с рождением ребенка, свадьбой или спортивным событием, однако мы сосредоточимся на более мелких, более обыденных событиях, таких как совершение покупки клиентом (онлайн или очно), щелчок кнопкой мыши на ссылке или передача данных датчиком. Генерировать события могут как люди, так и машины. Именно последовательность событий и их постоянное течение составляют поток событий.

Концептуально события содержат три основных компонента:

- *ключ* – идентификатор события;
- *значение* – само событие;
- *отметка времени* – когда произошло событие.

Обсудим каждый из них более подробно. Ключ может служить идентификатором события и, как мы узнаем в последующих главах, используется для маршрутизации и группировки событий. Вообразите онлайн-покупку: идентификатор клиента – отличный пример ключа события покупки. Значение – это информация о событии. Значение может быть, например, сигналом от датчика, сообщающим, что кто-то открыл дверь, или результатом какого-либо действия, например покупки товара в онлайн-магазине. Наконец, отметка времени – это дата и время, определяющие, когда произошло событие. Далее практически во всех главах мы неизменно будем сталкиваться со всеми тремя компонентами событий.

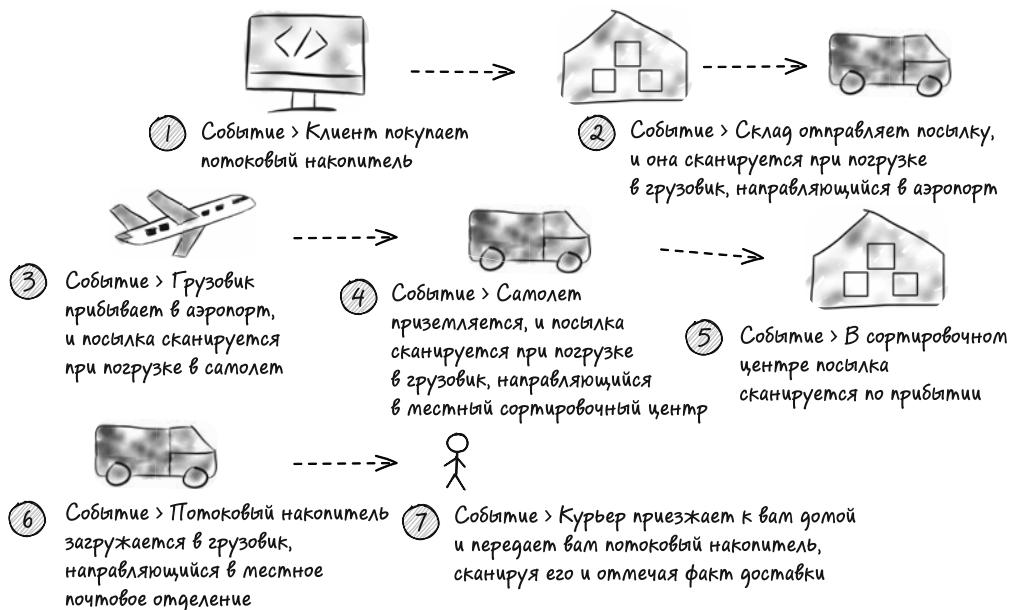
В этом введении я использовал много разных терминов, поэтому завершу раздел таблицей определений (табл. 1.1).

**Таблица 1.1.** Определения

Событие	Что-то, что произошло, и сопутствующие характеристики произошедшего
Поток событий	Серия событий, зафиксированных в режиме реального времени из таких источников, как мобильные устройства или устройства Интернета вещей
Платформа потоковой обработки событий	Программное обеспечение для обработки потоков событий, способное создавать, потреблять, обрабатывать и хранить потоки событий
Apache Kafka	Ведущая платформа потоковой обработки событий, предоставляющая все компоненты платформы потоковой обработки событий в одном проверенном решении
Kafka Streams	Библиотека потоковой обработки событий для Kafka

## 1.3. ПРИМЕР ПОТОКА СОБЫТИЙ

Допустим, вы купили потоковый накопитель и с нетерпением ждете, когда покупка будет вам доставлена. Рассмотрим события, которые привели к моменту, когда вы получите свой новый накопитель, используя иллюстрацию на рис. 1.1 в качестве примера.



**Рис. 1.1.** Последовательность, составляющая поток событий, начинающийся с онлайн-покупки потокового накопителя

Рассмотрим шаги, ведущие к получению потокового накопителя.

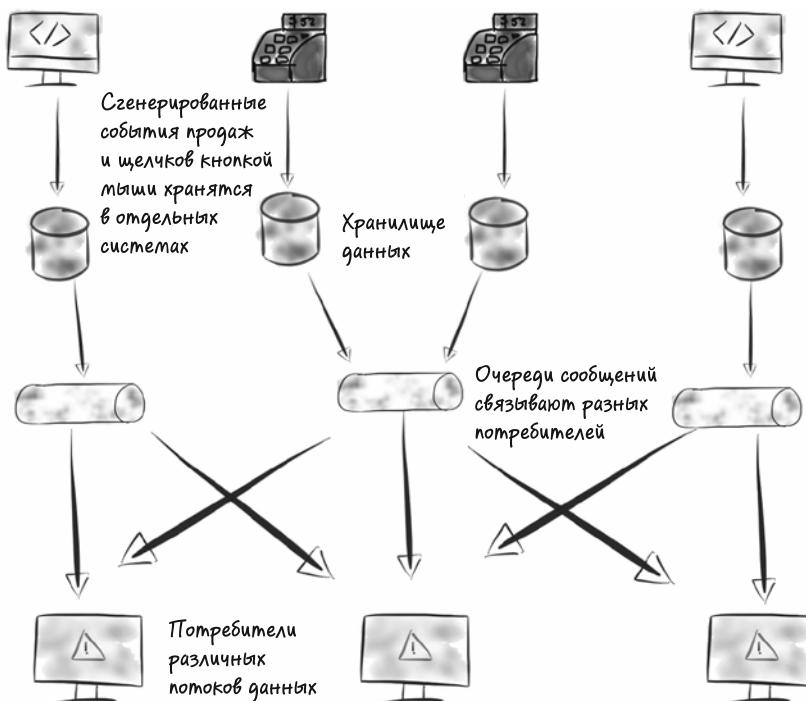
- Вы совершаете покупку на сайте, и сайт сообщает вам трек-номер.
- Склад продавца получает информацию о событии покупки, упаковывает потоковый накопитель и грузит его в грузовик, фиксируя дату и время, когда ваша покупка покинула склад.
- Грузовик прибывает в аэропорт, водитель выгружает потоковый накопитель в самолет и сканирует штрихкод, фиксируя дату и время.
- Самолет приземляется, и посылка снова загружается в грузовик, направляющийся в местный сортировочный центр. Служба доставки регистрирует дату и время погрузки посылки в грузовик.
- Грузовик приезжает в местный сортировочный центр. Сотрудник службы доставки выгружает посылку и, сканируя ее, фиксирует дату и время прибытия в сортировочный центр.
- Другой сотрудник забирает посылку, сканирует, фиксирует дату и время и передает ее курьеру для доставки вам.
- Курьер приезжает к вам домой, сканирует посылку в последний раз и передает ее вам. Теперь вы можете начать строить свою машину для путешествий во времени!

Наш пример показывает, как повседневные действия создают события, образуя их поток. Отдельные события — покупка, погрузка/выгрузка, прием на ответственное хранение и окончательная доставка. Этот сценарий представляет события, порожденные всего одной покупкой. А теперь вообразите потоки событий, созданных покупками на Amazon и различными отправителями товаров, — количество событий в них может исчисляться миллиардами или триллионами.

## 1.4. ЗНАКОМСТВО С АРАСНЕ КАФКА, ПЛАТФОРМОЙ ПОТОКОВОЙ ОБРАБОТКИ СОБЫТИЙ

Платформа Kafka предоставляет основные возможности для реализации приложений потоковой обработки событий от начала до конца. Мы можем разбить эти возможности на три основные области: публикация/потребление, долговременное хранение и обработка. Эти три этапа — перемещение, хранение и обработка — позволяют Kafka работать подобно центральной нервной системе для данных.

Прежде чем продолжить, нелишним будет проиллюстрировать, что значит быть центральной нервной системой для данных. Для этого будут показаны схемы до и после. Но сначала рассмотрим решение для потоковой обработки событий, в котором каждый источник входных данных требует отдельной инфраструктуры (рис. 1.2).



**Рис. 1.2.** Первоначальная архитектура потоковой обработки событий порождает массу сложностей, потому что различные отделы и источники потоков данных должны знать о других источниках событий

На иллюстрации видно, что каждый отдел создает свою инфраструктуру для удовлетворения своих потребностей. Однако другие отделы тоже могут быть заинтересованы в потреблении тех же данных, и добавление дополнительных связей для передачи потоков усложняет архитектуру. Взгляните на рис. 1.3, где показано, как платформа потоковой обработки событий Kafka может изменить ситуацию.

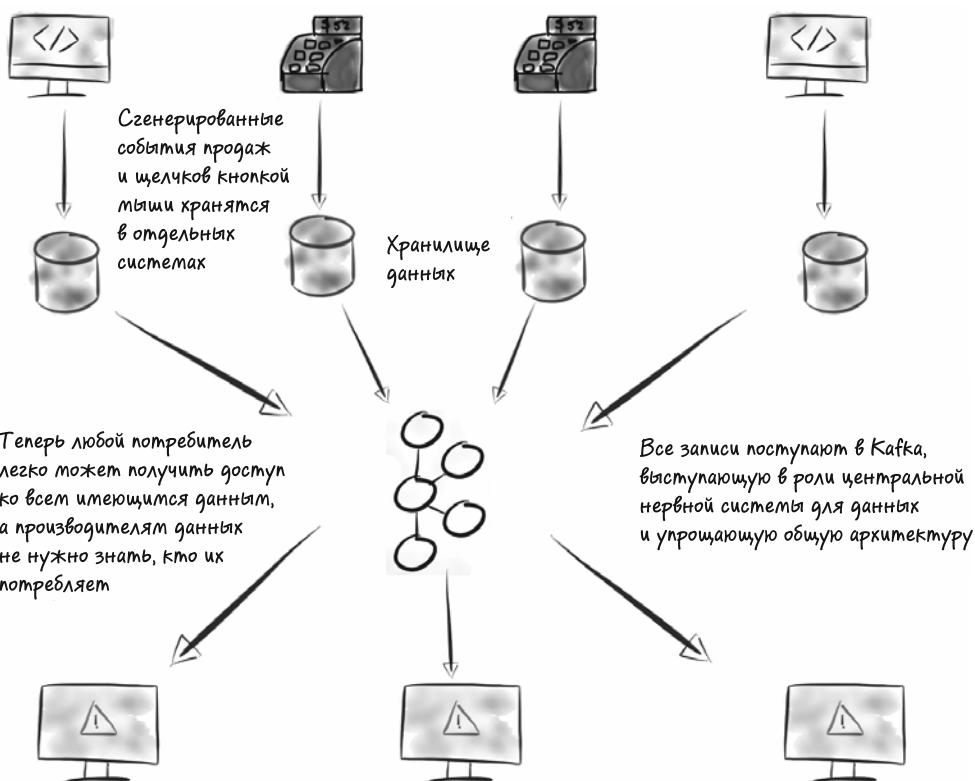


Рис. 1.3. Использование платформы Kafka помогает упростить архитектуру

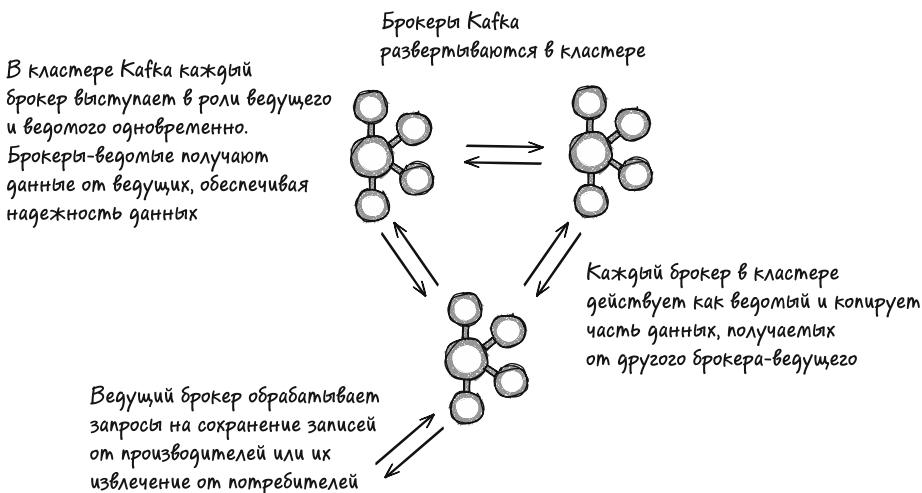
Как можно видеть на этой обновленной иллюстрации, добавление платформы Kafka значительно упрощает общую архитектуру. Теперь все компоненты отправляют свои записи в Kafka, а потребители читают данные из Kafka, ничего не зная об их производителях.

Если говорить в общих чертах, то Kafka представляет собой распределенную систему серверов и клиентов. Серверы называются брокерами, а клиенты могут быть производителями событий, отправляющими записи брокерам, и потребителями, читающими записи для обработки событий.

### 1.4.1. Брокеры Kafka

Брокеры Kafka надежно хранят ваши записи в отличие от традиционных систем обмена сообщениями (RabbitMQ или ActiveMQ), где сообщения являются эфемерными. Брокеры хранят данные в виде пар «ключ — значение» (дополненных некоторыми другими полями метаданных) в байтовом представлении и являются своего рода черными ящиками.

Сохранение событий имеет глубокие последствия и определяет разницу между сообщениями и событиями. Сообщения можно рассматривать как «оперативную» информацию, которой обмениваются две машины, тогда как события представляют критически важные для бизнеса данные, потерять которые было бы очень нежелательно (рис. 1.4).



**Рис. 1.4.** Брокеры развертываются в кластере и реплицируют данные для долговременного хранения

Эта иллюстрация показывает, что брокеры Kafka являются уровнем хранения в трилогии потоковой обработки событий. Но, помимо хранения, брокеры представляют другие важные функции, такие как обслуживание клиентских запросов и координация с потребителями. Мы подробно рассмотрим работу брокеров в главе 2.

### 1.4.2. Schema Registry

Управление данными жизненно важно, и его важность только возрастает с ростом организации. Schema Registry хранит схемы записей с событиями (рис. 1.5). Схемы обеспечивают соблюдение контракта о формате данных между производителями

и потребителями. Schema Registry также предоставляет средства для сериализации и десериализации. Предоставление средств (де)сериализации означает, что вам не нужно писать свой код, осуществляющий сериализацию. Подробнее Schema Registry будет рассматриваться в главе 3. В приложении А также есть упражнение по миграции схем Schema Registry.

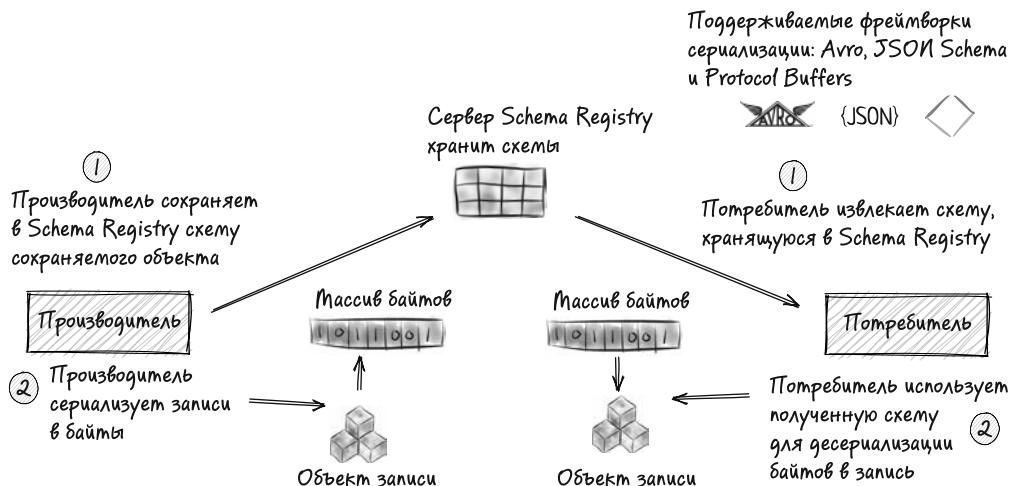


Рис. 1.5. Schema Registry обеспечивает единство моделей данных на всей платформе

### 1.4.3. Производители и потребители

Клиент-производитель отвечает за отправку записей в Kafka, а клиент-потребитель — за чтение записей из Kafka (рис. 1.6). Эти два клиента образуют основные строительные блоки для создания приложения, управляемого событиями, и не зависят друг от друга, что обеспечивает большую масштабируемость. Клиент-производитель и клиент-потребитель также образуют основу для любой абстракции более высокого уровня, работающей с Apache Kafka. Более подробно мы рассмотрим клиенты в главе 4.

### 1.4.4. Kafka Connect

Kafka Connect — это абстракция поверх производителей и потребителей, предназначенная для импорта данных в Apache Kafka и экспорта данных из Apache Kafka (рис. 1.7). Компонент Kafka Connect необходим для подключения внешних хранилищ к Apache Kafka. Он также дает возможность выполнять простые преобразования данных с помощью Simple Message Transform (SMT) при экспорте или импорте данных. Мы поближе рассмотрим Kafka Connect в главе 5.

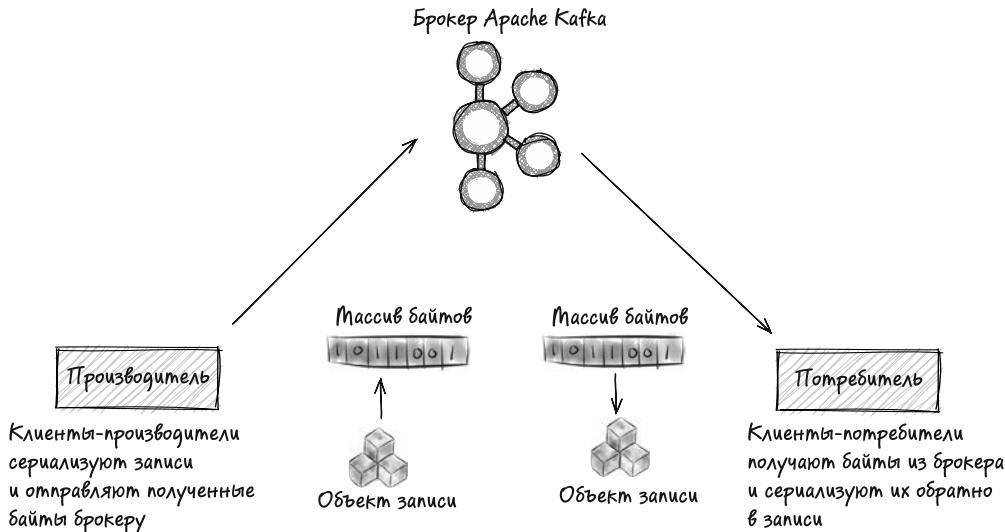
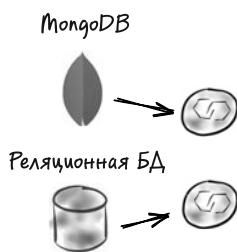


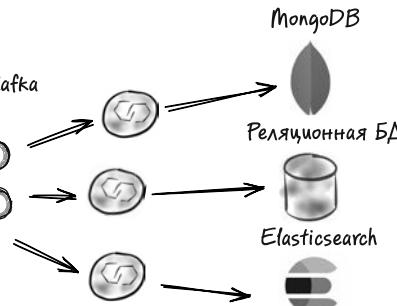
Рис. 1.6. Производители записывают записи в Kafka, а потребители читают их

Коннекторы-источники Kafka Connect



Коннекторы-источники импортируют данные из внешних систем в кластер Kafka

Коннекторы-приемники Kafka Connect



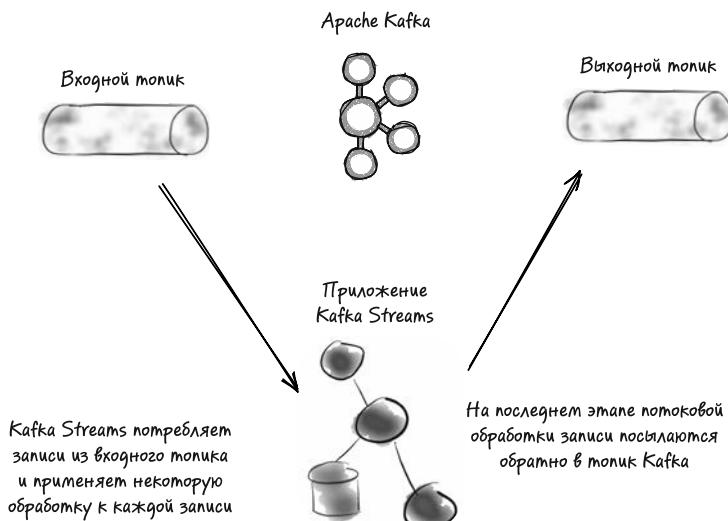
Коннекторы-приемники экспортят данные из кластера Kafka во внешние системы

Рис. 1.7. Kafka Connect связывает внешние системы с Apache Kafka

#### 1.4.5. Kafka Streams

Kafka Streams — это библиотека потоковой обработки для Kafka (рис. 1.8). Kafka Streams написана на языке Java и используется внешними клиентскими приложениями, взаимодействующими с кластером Kafka. Она поддерживает выполнение операций с данными событий, включая преобразования и операции с сохранением состояния,

такие как соединения и агрегирование. Kafka Streams — это то место, где вы будете работать с событиями. Подробнее о Kafka Streams рассказывается в главах 6–10.



**Рис. 1.8.** Kafka Streams — это API потоковой обработки для Kafka

#### 1.4.6. ksqlDB

ksqlDB — это база данных для хранения потоков событий (рис. 1.9). Она поддерживает интерфейс SQL для обработки потоков. Внутри для решения своих задач потоковой обработки событий ksqlDB использует Kafka Streams. Ключевое преимущество ksqlDB состоит в том, что она позволяет указывать операции потоковой обработки событий в виде SQL-кода; код на каком-либо языке программирования не требуется. Подробнее о ksqlDB рассказывается в главе 11.

```
CREATE TABLE activePromotions AS
  SELECT rideld
    qualifyPromotion(kmToDst) AS promotion
   FROM locations
  GROUP BY rideld
  EMIT CHANGES;
```

```
SELECT rideld, promotion
  FROM activePromotions
 WHERE ROWKEY = '6fd0fdb'
```

**Рис. 1.9.** ksqlDB предоставляет возможности потоковой базы данных

Теперь, после знакомства с основными принципами работы платформы потоковой обработки событий Kafka и ее компонентами, предлагаю рассмотреть работу Kafka на конкретном примере розничной торговли.

## 1.5. КОНКРЕТНЫЙ ПРИМЕР ПРИМЕНЕНИЯ ПЛАТФОРМЫ ПОТОКОВОЙ ОБРАБОТКИ СОБЫТИЙ KAFKA

Представьте, что некая Джейн Доу проверяет свою электронную почту и обнаруживает письмо от ZMart со ссылкой на страницу на веб-сайте ZMart, содержащую купон на 15%-ную скидку. Оказавшись на веб-странице, Джейн щелкает на другой ссылке, чтобы активировать и распечатать купон. Для Джейн вся эта последовательность действий является всего лишь еще одной онлайн-покупкой, но для ZMart – это поток событий щелчков.

Давайте приостановимся и обсудим взаимосвязь между этими простыми событиями, а также поговорим о том, как они взаимодействуют с платформой потоковой обработки событий Kafka. Данные, сгенерированные начальными щелчками кнопкой мыши для перехода к купону и его печати, создают информацию о потоке событий щелчков, которая захватывается и отправляется непосредственно в Kafka с помощью микросервиса-производителя. Отдел маркетинга запустил новую рекламную кампанию и хочет оценить ее эффективность, поэтому события в данном потоке событий щелчков представляют определенную ценность.

Первым признаком успешности начинания является факт щелчка пользователем на ссылке в электронной почте с целью получить купон. Команда по изучению данных тоже заинтересована в получении данных, предваряющих покупки. Эта команда может отслеживать действия клиентов и связывать покупки с первоначальными щелчками и маркетинговыми кампаниями. Объем данных от этого отдельного действия может показаться незначительным. Но в случае большой клиентской базы и проведения нескольких маркетинговых кампаний одновременно объем данных может получиться весьма значительным.

Теперь продолжим наш пример с покупками. На дворе конец лета, и Джейн собралась пойти по магазинам, чтобы купить своим детям школьные принадлежности. Поскольку в последнее время редкий вечер обходится без семейных мероприятий, по пути домой она заехала в ZMart.

Прогуливаясь по магазину и купив все необходимое, Джейн, проходя мимо обувного отдела, замечает новые дизайнерские туфли, которые отлично подошли бы к ее новому костюму. Она понимает, что пришла не за этим, но, черт побери, жизнь так коротка (ZMart процветает за счет импульсивных покупок!), поэтому Джейн покупает туфли.

Дойдя до кассы самообслуживания, Джейн сканирует свою карту лояльности ZMart, затем все товары и, наконец, купон, дающий право на скидку 15 %. Затем Джейн оплачивает покупки своей банковской картой, берет чек и выходит из магазина. Чуть позже тем же вечером Джейн проверила свою электронную почту и обнаружила письмо от ZMart с благодарностью за ее лояльность и с купонами со скидками на новую линию дизайнерской одежды.

Давайте разберем событие покупки и посмотрим, запускает ли оно последовательность операций, выполняемых платформой Kafka. Итак, данные о продажах ZMart поступают в Kafka. В данном случае ZMart использует Kafka Connect для создания исходного коннектора, собирающего информацию о продажах по мере их совершения и отправляющего их в Kafka. Транзакция продажи подводит нас

к первому требованию: защита данных клиентов. В этом случае ZMart использует простые преобразования данных с помощью Simple Message Transform (SMT) для маскировки данных кредитной карты при их передаче в Kafka (рис. 1.10).



**Рис. 1.10.** Все данные о продажах отправляются непосредственно в Kafka с помощью Connect, маскирующего номера кредитных карт

Когда Kafka Connect записывает данные в Kafka, различные подразделения в ZMart немедленно потребляют их. В отделе маркетинга имеется приложение, изучающее данные о продажах и подсчитывающее общую сумму покупок, совершенных клиентом, если тот предъявил карту лояльности. Если сумма достигла некоторого порогового значения, то клиенту отправляется электронное письмо с купоном (рис. 1.11).

Важно отметить, что ZMart обрабатывает данные сразу после продажи, поэтому клиенты получают письма с благодарностью и вознаграждением в течение нескольких минут. Реагирование на события покупки по мере их наступления позволяет ZMart быстро предлагать бонусы клиентам.

Аналитический отдел в ZMart тоже использует данные из топика, через который распространяется информация о продажах. В этом отделе имеется свое приложение Kafka Streams для обработки данных о продажах, выявляющее закономерности покупок, совершаемых клиентами в разных местах. Приложение Kafka Streams обрабатывает данные в режиме реального времени и отправляет результаты в топик, через который распространяются сведения о тенденциях продаж (рис. 1.12).

Еще один коннектор Kafka используется в ZMart для экспорта тенденций продаж во внешнее приложение, которое публикует результаты на информационной панели. Еще один отдел потребляет данные из топика с информацией о продажах, отслеживая запасы и заказывая новые партии товаров, если объем запасов опускается ниже заданного порога.

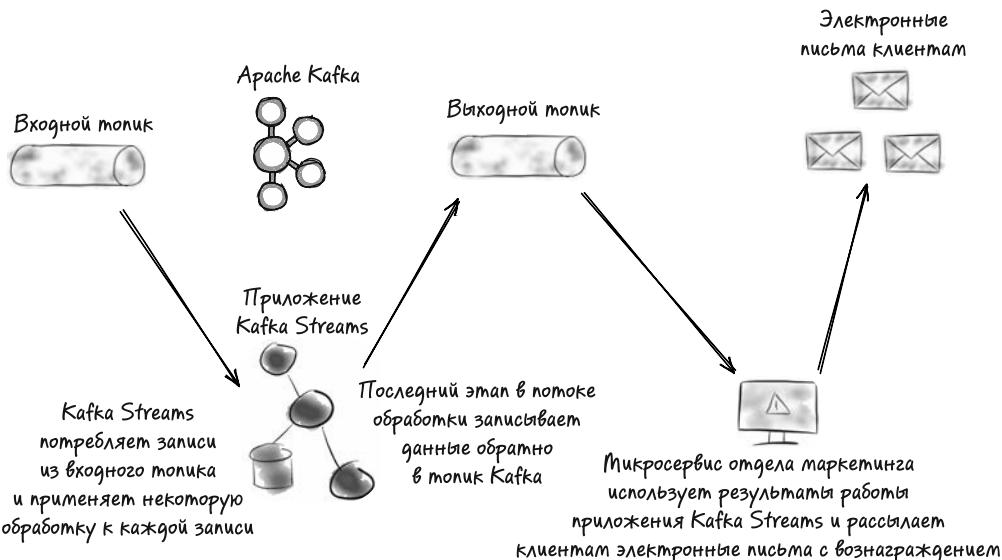


Рис. 1.11. Приложение в отделе маркетинга суммирует баллы, полученные клиентом, и отправляет купоны на скидку

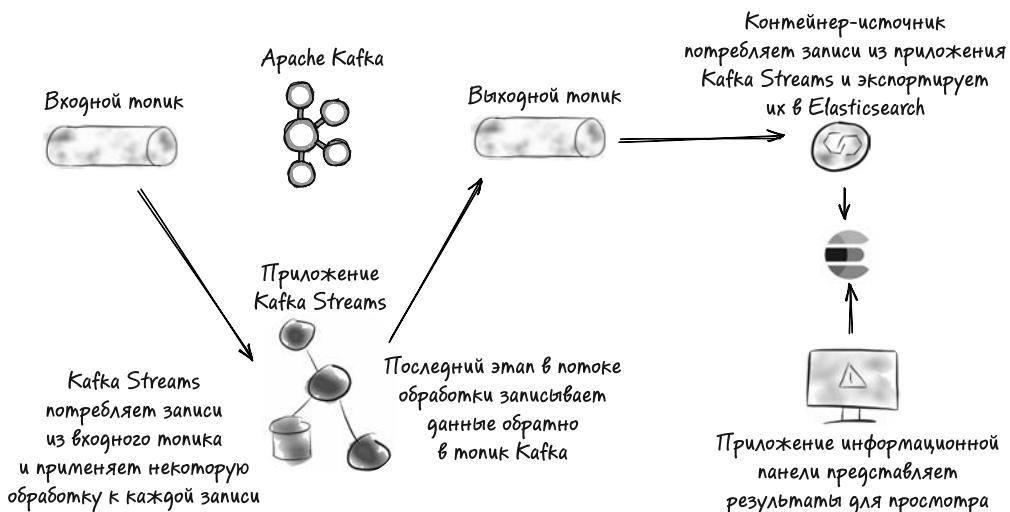


Рис. 1.12. Приложение Kafka Streams обрабатывает данные о продажах, а Kafka Connect экспортирует их для использования приложением информационной панели

Сейчас вы уже имеете представление, как ZMart использует платформу Kafka. Важно помнить, что, применяя потоковую обработку событий, ZMart имеет возможность реагировать на данные по мере их поступления и быстро принимать эффективные решения. Обратите внимание также, что запись данных в Kafka происходит один раз, но разные отделы потребляют их в разное время и независимо, не мешая друг другу.

## ИТОГИ ГЛАВЫ

- Потоковая обработка событий фиксирует события, генерируемые разными источниками, такими как мобильные устройства, веб-сайты, онлайн-сервисы, программы отслеживания поставок и бизнес-системы. Потоковая обработка событий подобна нашей нервной системе.
- Событие — это «что-то, что произошло», и возможность немедленно реагировать и просматривать информацию позже — важнейшая концепция платформы потоковой обработки событий.
- Kafka действует подобно центральной нервной системе в отношении данных и упрощает архитектуру обработки потоков событий.
- Платформа Kafka предоставляет три основных компонента для использования в приложениях потоковой обработки событий, обеспечивающих публикацию/потребление, долговременное хранение и обработку.
- Брокеры Kafka — это слой хранения. Они обслуживают запросы клиентов на запись и чтение данных. Брокеры хранят записи как байты и никак не изменяют их содержимое.
- Реестр схем Schema Registry помогает обеспечить совместимость записей между производителями и потребителями.
- Клиенты-производители записывают (производят) данные в брокер. Клиенты-потребители извлекают (потребляют) записи из брокера. Клиенты-производители и клиенты-потребители не зависят друг от друга. Кроме того, брокер Kafka не знает, кто является его клиентами, он просто обрабатывает запросы.
- Kafka Connect предоставляет механизм для интеграции с существующими системами, такими как внешнее хранилище, для загрузки данных в Kafka и выгрузки их из нее.
- Kafka Streams — это библиотека потоковой обработки для Kafka. Она работает за пределами кластера Kafka и обеспечивает поддержку преобразований данных, включая соединения и преобразования с сохранением состояния.
- ksqlDB — это потоковая база данных для Kafka. Она позволяет создавать надежные системы реального времени, написав всего несколько строк на SQL.

# Брокеры Kafka

## В этой главе

- ✓ Представление брокера как слоя хранения для платформы потоковой обработки событий Kafka.
- ✓ Описание особенностей обработки запросов клиентов брокерами Kafka.
- ✓ Топики и разделы.
- ✓ Использование метрик JMX для проверки работоспособности брокера.

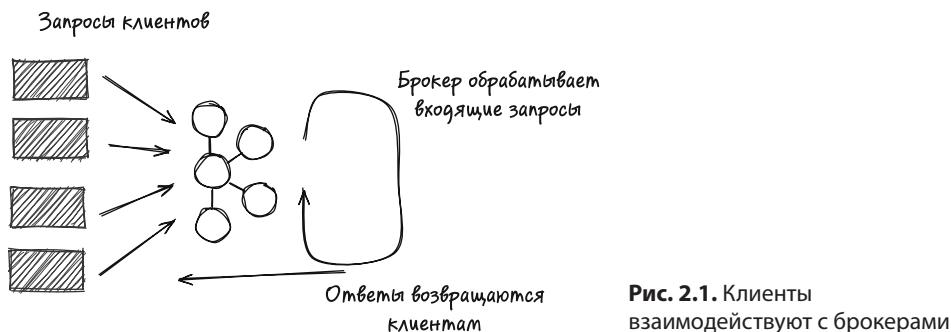
В главе 1 я представил общий обзор платформы потоковой обработки событий Kafka и различных ее компонентов. В этой главе мы сосредоточимся на сердце системы, брокерах Kafka. Брокер – это сервер в архитектуре Kafka, который играет роль слоя хранения данных.

Здесь мы обсудим некоторые низкоуровневые детали поведения брокеров, чтобы вы понимали, как они работают, потому что это очень важно. Кроме того, мы рассмотрим такие важные понятия, как топики и разделы, понимание которых вам пригодится, когда мы перейдем к главе о клиентах. Но как разработчику вам не придется иметь дело с этими деталями в своей повседневной работе.

## 2.1. ВВЕДЕНИЕ В БРОКЕРЫ КАФКА

Будучи слоем хранения, брокер управляет данными, в том числе отвечая за сохранение и репликацию. Под сохранением понимается долговременность хранения записей, а под репликацией – создание копий данных для надежного долговременного хранения, чтобы вы не потеряли данные, даже если потеряете машину.

Но брокер также обрабатывает запросы клиентов. На рис. 2.1 показаны клиентские приложения и брокеры.



Чтобы помочь вам быстро составить представление о роли брокера, обобщим иллюстрацию на рис. 2.1: клиенты отправляют запросы брокеру, а тот обрабатывает эти запросы и возвращает ответы. Я опустил некоторые детали взаимодействий, но сути это не меняет.

#### ПРИМЕЧАНИЕ

Kafka — это очень глубокая тема, поэтому я не буду описывать все тонкости этой платформы, но представлю достаточно информации, чтобы вы могли начать работать с ней. Более глубокое освещение вы найдете в книге *Kafka in Action* Дилана Скотта (Dylan Scott), Виктора Гамова (Viktor Gamov) и Дэйва Клейна (Dave Klein), опубликованной издательством Manning в 2022 году<sup>1</sup>.

Брокеры Kafka можно развернуть на стандартном оборудовании, в контейнерах, на виртуальных машинах или в облачных средах. В этой книге мы будем использовать Kafka в контейнере Docker, поэтому вам не придется устанавливать ее напрямую.

Рассказывая о брокерах Kafka, я не могу обойти стороной производители и потребители. Однако, так как эта глава посвящена брокерам, основное внимание я уделю обязанностям брокера и опущу некоторые детали, касающиеся клиентов. Но не волнуйтесь, все эти детали мы обсудим в главе 4.

Итак, начнем с пошагового руководства по обработке запросов клиентов брокером, начав с производства.

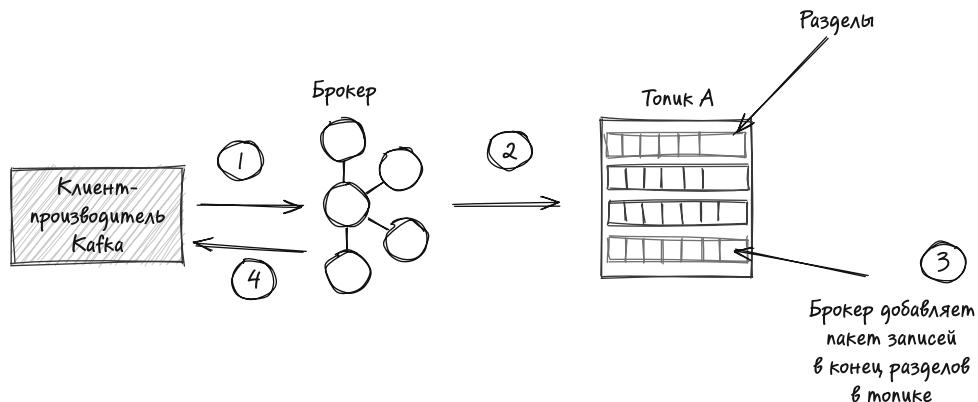
## 2.2. ЗАПРОСЫ НА ПРОИЗВОДСТВО

Чтобы отправить записи брокеру, клиент выполняет запрос на производство. Клиенты-производители отправляют записи брокеру для сохранения, чтобы позднее клиенты-потребители могли прочитать их.

На рис. 2.2 показан производитель, отправляющий записи брокеру. Важно отметить, что эта иллюстрация не отражает масштаб системы. Обычно в системе имеется

<sup>1</sup> Скотт Д., Гамов В., Клейн Д. Kafka в действии. — М., 2022.

множество клиентов, общающихся с несколькими брокерами в кластере. Один клиент может работать с несколькими брокерами. Однако чем проще иллюстрация, тем легче представить картину происходящего. Обратите внимание, что я упростил взаимодействия, но не волнуйтесь, в главе 4 мы рассмотрим больше деталей, касающихся работы клиентов.



**Рис. 2.2.** Как брокеры обрабатывают запросы на производство

Рассмотрим шаги, обозначенные цифрами на иллюстрации.

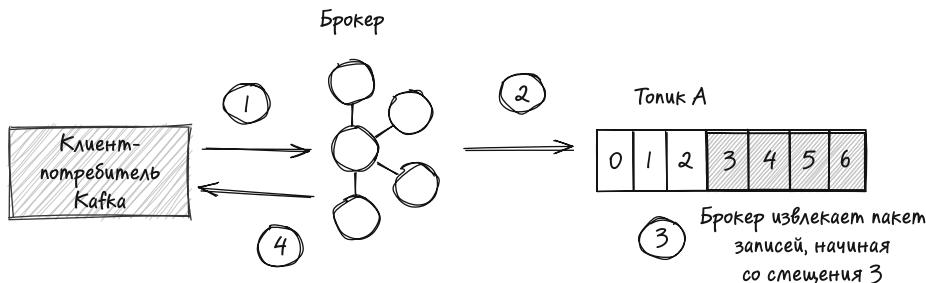
- Производитель отправляет пакет записей брокеру. Независимо от того, производитель это или потребитель, клиентские API всегда работают с коллекциями записей, посыпая передачу данных пакетами.
- Брокер извлекает запрос на производство из очереди запросов.
- Брокер сохраняет записи в топике. Внутренне топик разбит на разделы. Один пакет записей записывается в определенный раздел в топике и всегда в конец.
- После сохранения записей брокер отвечает производителю. Подробнее о том, что возвращается клиенту в ответе в случае успешного сохранения записей, мы поговорим ниже в этой главе и еще раз в главе 4.

Далее мы рассмотрим другой тип запросов — запросы на потребление (извлечение) записей, который является логической противоположностью типу запросов на производство (сохранение) записей.

## 2.3. ЗАПРОСЫ НА ИЗВЛЕЧЕНИЕ

Теперь посмотрим на другую сторону медали — на запрос на выборку. В них клиенты-потребители отправляют брокеру запросы на чтение (потребление) записей из топика. Важно понимать, что потребление записей не влияет на сохранность данных или доступность записей для других клиентов-потребителей. Брокеры Kafka могут обрабатывать сотни запросов на потребление записей из одного и того же топика, и никакой запрос не будет влиять на другие. Мы рассмотрим сохранение данных позже, но уже сейчас отмечу, что брокер выполняет эту операцию отдельно от потребления.

Имейте в виду, что производители и потребители отделены и ничего не знают друг о друге. Брокер обслуживает запросы на производство и потребление отдельно, одно никак не связано с другим. Пример на рис. 2.3 упрощен, чтобы показать работу брокера в общих чертах.



**Рис. 2.3.** Как брокеры обрабатывают запросы на потребление

Рассмотрим шаги, обозначенные цифрами на иллюстрации.

1. Потребитель отправляет запрос на выборку, указав смещение, с которого следует начать чтение записей. Более подробно смещения будут рассматриваться ниже в этой главе.
2. Брокер извлекает запрос на выборку из очереди запросов.
3. На основе смещения и раздела топика в запросе брокер извлекает пакет записей.
4. Брокер отправляет потребителю ответ с извлеченным пакетом записей.

Теперь, рассмотрев по шагам, как обрабатываются запросы двух наиболее распространенных типов, на производство и на потребление, вы наверняка заметили, что я употребил несколько терминов, которые пока не были описаны: топики, разделы и смещения. Топики, разделы и смещения — это основополагающие концепции в Kafka, поэтому давайте уделим немного времени, чтобы понять, что они означают.

## 2.4. ТОПИКИ И РАЗДЕЛЫ

Kafka обеспечивает надежное хранение ваших данных в виде неограниченной последовательности сообщений в форме пар «ключ — значение» так долго, сколько вам нужно (сообщения содержат и другие поля, например отметку времени, но мы вернемся к этим подробностям позже). Kafka реплицирует (копирует) данные между несколькими брокерами, поэтому потеря диска или даже целого брокера не приводит к потере данных.

В частности, брокеры Kafka используют в качестве хранилища файловую систему, добавляя входящие записи в конец файла в топике. Топик — это имя каталога, где находится файл, в который брокер Kafka добавляет записи.

## ПРИМЕЧАНИЕ

Kafka получает сообщения с парами «ключ — значение» в виде необработанных байтов, сохраняет их в таком виде и в таком же виде возвращает в ответ на запросы на чтение. Брокер Kafka ничего не знает о типах записей, которые обрабатывает. Работая с обычными байтами, брокеры не тратят времени на сериализацию и десериализацию данных, что обеспечивает более высокую производительность. Позднее вы увидите, как гарантировать хранение в топике байтов в ожидаемом формате, когда мы будем изучать Schema Registry в главе 3.

Топики имеют разделы, представляющие следующий уровень организации данных. Разделы идентифицируются целыми числами, начиная с 0. То есть, если топик имеет три раздела, им присваиваются номера 0, 1 и 2. Kafka добавляет номер раздела в конец имени топика, создавая для каждого раздела свой каталог с именем в форме `topic-N`, где N представляет номер раздела.

Брокеры Kafka имеют конфигурационный параметр `log.dirs`, указывающий имя каталога верхнего уровня, который будет содержать все каталоги разделов и топиков. Рассмотрим листинг 2.1. Здесь предполагается, что в параметре `log.dirs` указан путь `/var/kafka/topic-data` и имеется топик с именем `purchases` с тремя разделами.

### Листинг 2.1. Пример структуры каталогов топика

```
root@broker:/# tree /var/kafka/topic-data/purchases*
```

```
/var/kafka/topic-data/purchases-0
├── 000000000000000000000000.index
├── 000000000000000000000000.log
├── 000000000000000000000000.timeindex
└── leader-epoch-checkpoint
/var/kafka/topic-data/purchases-1
├── 000000000000000000000000.index
├── 000000000000000000000000.log
├── 000000000000000000000000.timeindex
└── leader-epoch-checkpoint
/var/kafka/topic-data/purchases-2
├── 000000000000000000000000.index
├── 000000000000000000000000.log
└── 000000000000000000000000.timeindex
    leader-epoch-checkpoint
```

Как видите, топик `purchases` с тремя разделами размещается в файловой системе в трех каталогах — `purchases-0`, `purchases-1` и `purchases-2`. Название топика — это скорее логическая группировка, а раздел — это единица хранения.

## СОВЕТ

Показанная здесь структура каталогов получена с помощью команды `tree` — инструмента командной строки, используемого для отображения всего содержимого каталога.

Несмотря на желание обсудить содержимое этих каталогов, мы должны прежде рассмотреть некоторые подробности, касающиеся разделов топиков.

Разделы являются единицей параллелизма в Kafka. В общем случае чем больше количество разделов, тем выше пропускная способность. Будучи основным средством хранения, разделы позволяют распределить сообщения по нескольким машинам. Емкость топика не ограничивается дисковым пространством, доступным одному брокеру. Кроме того, как упоминалось ранее, репликация данных по нескольким брокерам гарантирует сохранность данных при потере диска или даже целого брокера.

Позже в этой главе мы подробнее поговорим о распределении нагрузки, когда будем обсуждать репликацию, ведущие и ведомые брокеры. Мы также рассмотрим новую возможность организации многоуровневого хранилища, где данные легко перемещаются во внешнее хранилище, обеспечивая практически безграничную емкость.

Как же Kafka определяет, в каком разделе должна храниться та или иная запись? Раздел и топик определяются клиентом-производителем перед отправкой записи брокеру. Получив и обработав запись, брокер добавит ее в файл в соответствующем каталоге раздела.

Существует три способа определения раздела для записи.

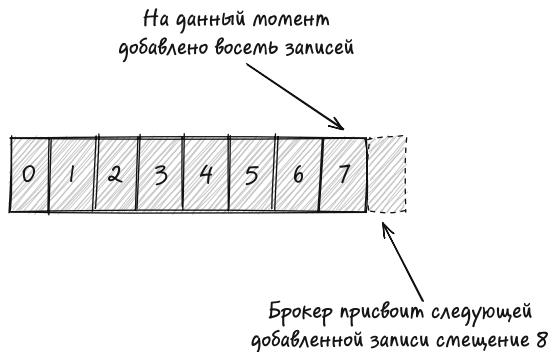
1. Kafka работает с записями, содержащими пары «ключ — значение». Предположим, что ключ не является пустым (в общем случае ключи могут быть пустыми). Тогда производитель задает раздел для сохранения записи, используя детерминированную формулу вычисления хеша и взятия модуля от количества разделов. При таком подходе записи с идентичными ключами всегда будут попадать в один и тот же раздел.
2. При создании `ProducerRecord` в приложении можно явно указать раздел для сохранения записи, который производитель затем будет использовать перед отправкой.
3. Если в сообщении ключ или раздел не указан, то пакеты чередуются. Как Kafka обрабатывает записи без ключей и назначенных разделов, я подробно расскажу в главе 4.

Теперь, рассмотрев работу разделов, вернемся к тому, что Kafka всегда добавляет записи в конец файла. Вы наверняка заметили в примере, представленном в листинге 2.1, файлы с расширением `.log` (о том, как Kafka выбирает имена для этих файлов, мы поговорим в подразделе 2.6.3). Но это не те файлы журналов, которые сразу приходят на ум большинству разработчиков. Эти *журналы* хранят не сообщения о состоянии или этапах выполнения, они хранят транзакции — последовательности событий в порядке возникновения. Таким образом, каждый каталог раздела содержит свой журнал транзакций. На этом этапе было бы справедливо задать вопрос о росте файла журнала. Подробнее о размере файла журнала и управлении им мы поговорим, когда будем рассматривать сегментирование далее в этой главе.

## 2.4.1. Смещения

Когда брокер добавляет новую запись, он назначает ей идентификатор, называемый смещением. Смещение — это число (порядковый номер, начинающийся с 0), которое брокер увеличивает на 1 для каждой следующей записи. Но смещение — это не только уникальный идентификатор, но и логическая позиция записи в файле.

Термин *логическая позиция* означает, что это  $n$ -я запись в файле, но ее физическое местоположение определяется размером в байтах предыдущих записей. В подразделе 2.6.3 мы поговорим о том, как брокеры используют смещение для поиска физической позиции записи. На рис. 2.4 демонстрируется концепция смещений для входящих записей.



**Рис. 2.4.** Назначение смещений входящим записям

Поскольку новые записи всегда добавляются в конец файла, они упорядочены по смещению. Kafka гарантирует упорядочение записей внутри раздела, но не гарантирует такого же упорядочения между разделами. Поскольку записи упорядочены по смещению, можно было бы подумать, что они также упорядочены по времени, но это не обязательно. Записи упорядочены по времени их *получения брокером*, но не обязательно по *времени события*. Мы подробнее рассмотрим семантику времени в главе 4, когда будем обсуждать отметки времени. Мы также подробно рассмотрим обработку времени события, когда доберемся до главы 9 и будем говорить о Kafka Streams.

Потребители используют смещения для отслеживания позиции записей, которые они уже прочитали, чтобы продолжить извлечение записей, начиная со смещения, которое на единицу больше последней прочитанной потребителем. Взгляните на рис. 2.5, поясняющий работу смещений.



**Рис. 2.5.** Смещение указывает, где потребитель остановился при чтении записей

Здесь показана ситуация, когда потребитель прочитал записи со смещениями 0–5. Если теперь потребитель обратится к брокеру за новыми записями, тот начнет чтение записей со смещения 6. Смещения уникальны для каждого потребителя и хранятся во внутреннем топике с именем `_consumer_offsets`. Подробнее о потребителях и смещениях мы поговорим в главе 4.

Теперь, рассмотрев топики, разделы и смещения, кратко обсудим некоторые компромиссы, касающиеся количества используемых разделов.

## 2.4.2. Выбор правильного количества разделов

Выбор количества разделов при создании топика — это отчасти искусство, отчасти наука. Одно из важнейших соображений — объем данных, поступающих в топик. Чем больше данных поступает, тем больше разделов желательно иметь, чтобы получить высокую пропускную способность. Но, как и в жизни, чем-то приходится жертвовать.

Увеличение количества разделов увеличивает количество TCP-соединений и открытых файловых дескрипторов. Время, необходимое для обработки входящей записи в потребителе, тоже будет определять пропускную способность. Если в потребителе осуществляется тяжелая обработка, то добавление большего количества разделов может способствовать увеличению общей пропускной способности, но более медленная обработка в конечном счете снизит производительность (Дзюн Рао (Jun Rao), *How to Choose the Number of Topics/Partitions in a Kafka Cluster?* <http://mng.bz/4C03>).

Вот несколько замечаний, которые следует учитывать при выборе количества разделов. Выбирайте достаточно большое число, чтобы охватить ситуации, требующие высокой пропускной способности, но все же не очень большое, чтобы не превысить предел количества разделов, которые может обработать брокер, потому что со временем число топиков будет неизменно расти. Хорошой отправной точкой может быть число 30, которое делится нацело на несколько чисел, что обеспечит более равномерное распределение ключей на уровне обработки (Майкл Нолл (Michael Noll), *Streams and Tables in Apache Kafka: Topics, Partitions and Storage Fundamentals*, <http://mng.bz/K9qg>). Подробнее о важности распределения ключей мы поговорим в главе 4, посвященной клиентам, и в главе 7, посвященной Kafka Streams.

Теперь вы знаете, что брокер обрабатывает клиентские запросы и играет роль слоя хранения данных в Kafka. Вы также узнали, что такие топики и разделы и какую роль они играют в слое хранения.

На следующем шаге мы попробуем создать и прочитать записи, чтобы увидеть эти механизмы в действии.

### ПРИМЕЧАНИЕ

О клиентах-производителях и клиентах-потребителях мы поговорим в главе 4. Для целей обучения, быстрого прототипирования и отладки прекрасно подойдут консольные клиенты. Но на практике вы будете использовать клиенты в своем коде.

## 2.5. ОТПРАВКА ПЕРВЫХ СООБЩЕНИЙ

Чтобы опробовать несколько следующих примеров, вам понадобится запустить брокер Kafka. В предыдущем издании этой книги я приводил инструкции, как загрузить и распаковать TAR-файл с двоичной версией Kafka. В этом издании я решил запустить Kafka в контейнере Docker. В частности, мы используем Docker Compose, чтобы упростить запуск многоконтейнерного приложения. Пользователи macOS и Windows могут установить Docker Desktop, который включает Docker Compose. За дополнительной информацией по установке Docker обращайтесь к документации на сайте Docker: <https://docs.docker.com/get-docker/>. Обратите внимание, что в качестве брокера для запуска приложений Kafka также можно использовать Confluent Cloud (<https://www.confluent.io/confluent-cloud/>). Подробную информацию о ресурсах Confluent вы найдете в приложении Б.

Начнем работу с брокером Kafka и попробуем создать и прочитать несколько записей.

### 2.5.1. Создание топика

Первый шаг к созданию или потреблению записей — создание топика. Для этого вам понадобится действующий брокер Kafka, поэтому сейчас займемся им. Я предполагаю, что вы уже установили Docker. Чтобы запустить Kafka, загрузите файл `docker-compose.yml` из репозитория с исходным кодом (<https://github.com/bbejeck/KafkaStreamsInAction2ndEdition>). Затем откройте новое окно терминала, перейдите в каталог с файлом `docker-compose.yml` и выполните команду `docker-compose up -d`.

#### СОВЕТ

Команда `docker-compose` с флагом `-d` запустит службы docker в фоновом режиме. Docker Compose можно запустить и без флага `-d`, но контейнеры выводят в терминал довольно много разных сообщений, поэтому для выполнения дальнейших операций вам придется открыть новое окно терминала.

Подождите несколько секунд, а затем выполните следующую команду, чтобы открыть командную оболочку в контейнере с брокером: `docker-compose exec broker bash`.

В этой командной оболочке выполните следующую команду, чтобы создать топик:

```
kafka-topics --create --topic first-topic\   | Имя хоста и номер порта
--bootstrap-server localhost:9092\    | для подключения к брокеру
--replication-factor 1\    | Коеффициент репликации
--partitions 1\    | Количество разделов
```

#### ПРИМЕЧАНИЕ

Несмотря на то что мы запустили Kafka в контейнере Docker, команды для создания топиков и запуска консольных клиентов-производителей и клиентов-потребителей одни и те же.

Поскольку для опробования мы используем локальный брокер, нам достаточно коэффициента репликации, равного 1. То же касается и количества разделов: на данном этапе для локального опробования нам нужен только один раздел.

Теперь, создав топик, сделаем в нем несколько записей.

## 2.5.2. Создание записей в командной строке

В том же окне, где вы запустили команду `--create --topic`, выполните следующую команду, чтобы запустить консольный производитель:

```
kafka-console-producer --topic first-topic\ ← Топик, созданный  
--bootstrap-server localhost:9092\ ← на предыдущем шаге  
--property parse.key=true\ ← Имя хоста и номер порта для подключения  
--property key.separator=":" ← клиента-производителя к брокеру  
← Указывает, что будет  
← предоставлен ключ  
← Указывает разделитель  
← ключа и значений
```

При использовании консольного клиента-производителя необходимо указать, будут ли предоставляться ключи. Хотя Kafka работает с парами «ключ — значение», в общем случае ключ может отсутствовать, то есть быть пустым. Поскольку ключ и значение передаются в одной строке, необходимо также указать, как будут отделяться ключи от значений, задав разделитель.

После ввода предыдущей команды вы должны увидеть приглашение к вводу. Введите текст, например такой:

```
key:my first message  
key:is something  
key:very simple
```

После ввода каждой строки нажмайте клавишу `Enter`, чтобы создать отдельные записи. Поздравляю, вы только что отправили свои первые сообщения в топик Kafka! Теперь попробуем прочитать записи, которые вы только что отправили в топик. Оставьте консольный клиент-производитель запущенным, так как мы снова воспользуемся им через несколько минут.

## 2.5.3. Потребление записей из командной строки

Попробуем прочитать только что созданные записи. Откройте новое окно терминала и выполните команду `docker-compose exec broker bash`, чтобы запустить оболочку в контейнере брокера. Затем запустите консольный клиент-потребитель следующей командой:

```
Задать топик для чтения  
kafka-console-consumer --topic first-topic\ ← Имя хоста и номер порта для подключения  
--bootstrap-server localhost:9092\ ← клиента-потребителя к брокеру  
--from-beginning\ ← Запустить потребление записей с начала журнала  
--property print.key=true\ ← Вывести ключи  
--property key.separator="-"\ ← Использовать символ «-» как  
← разделитель ключей и значений
```

В консоли должен появиться следующий вывод:

```
key-my first message
key-is something
key-very simple
```

Я должен кратко объяснить причину использования флага `--from-beginning`. Мы создали значения до запуска клиента-потребителя, поэтому не увидели бы эти сообщения, запустив чтение с конца топика. Параметр `--from-beginning` устанавливает позицию чтения в начало топика. Теперь вернитесь в окно производителя и введите новую пару «ключ — значение». Содержимое консоли с клиентом-потребителем обновится, и в конец добавится только что введенная запись.

На этом ваш первый пример заканчивается, но давайте разберем еще один пример и посмотрим, как работают разделы.

## 2.5.4. Разделы в действии

В предыдущем примере мы создали и прочитали несколько записей, но топик имел только один раздел, поэтому мы не увидели влияния разделов. Рассмотрим еще один пример, но на этот раз создадим новый топик с двумя разделами, добавим в него записи с разными ключами и оценим результат.

У вас еще должны быть запущены консольный производитель и потребитель. Завершите их оба, нажав `Ctrl+C` в обоих терминалах. Теперь создадим новый топик с разделами. Выполните следующую команду в одном из терминалов, которые вы использовали для создания или чтения записей:

```
kafka-topics --create --topic second-topic \
  --bootstrap-server localhost:9092 \
  --replication-factor 1 \
  --partitions 2
```

Затем запустите консольный потребитель:

```
kafka-console-consumer --topic second-topic \
  --bootstrap-server broker:9092 \
  --property print.key=true \
  --property key.separator="-" \
  --partition 0
```



Эта команда похожа на ту, что использовалась выше, но на этот раз в ней указан раздел, откуда будут потребляться записи. После запуска этой команды в консоли будут появляться записи, которые вы начнете вводить на следующем шаге. Теперь запустите консольный производитель:

```
kafka-console-producer --topic second-topic \
  --bootstrap-server localhost:9092 \
  --property parse.key=true \
  --property key.separator ":"
```

Затем в терминале производителя введите следующие пары «ключ — значение»:

```
key1:The lazy
key2:brown fox
key1:jumped over
key2:the lazy dog
```

В терминале потребителя должны появиться следующие записи:

```
key1:The lazy  
key1:jumped over
```

Вы не увидите других записей, потому что производитель записал их в раздел 1. Убедиться в этом можно, нажав Ctrl+C в терминале потребителя и запустив следующую команду:

```
kafka-console-consumer --topic second-topic\  
--bootstrap-server broker:9092\  
--property print.key=true\  
--property key.separator="-"\  
--partition 1\  
--from-beginning
```

После этого должны появиться следующие строки:

```
key2:brown fox  
key2:the lazy dog
```

Если перезапустить предыдущий потребитель, опустив указание раздела, то он выведет все записи. Более подробно о потребителях и разделах мы поговорим в главе 4.

На этом примеры закончены, так что можете завершить работу производителя и потребителя, нажав Ctrl+C в обоих терминалах, а затем остановить все контейнеры Docker командой `docker-compose down`.

Чтобы подвести итоги примера по опробованию базовой функциональности Kafka, кратко перечислю, что мы сейчас проделали. Мы создали несколько записей в топике, а потом в другом процессе прочитали их. На практике вы будете использовать топики с большим количеством разделов, создавать и читать гораздо большее число сообщений и использовать что-то более сложное, чем консольные инструменты, но суть останется той же самой.

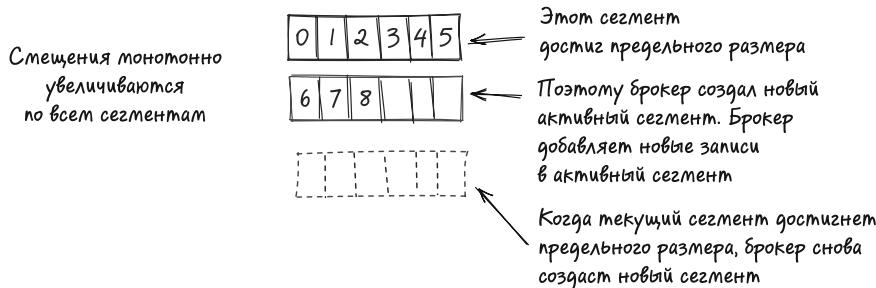
Мы также рассмотрели базовую единицу хранения, используемую брокером, — разделы. Узнали, что каждой входящей записи Kafka назначает уникальный (в пределах раздела) идентификатор и смещение и всегда добавляет записи в конец журнала раздела топика. Возникает закономерный вопрос: по мере поступления большего количества данных эти файлы продолжают расти бесконечно? Ответ на этот вопрос — нет, и в следующем разделе мы узнаем, как брокеры управляют данными.

## 2.6. СЕГМЕНТЫ

К настоящему моменту вы узнали, что брокеры добавляют входящие записи в файл раздела топика. Но добавление не всегда производится в один и тот же файл, потому что иначе получились бы огромные монолитные файлы. В действительности брокеры разбивают файлы на отдельные части, называемые сегментами. Сегментирование упрощает применение настроек хранения данных и извлечение записей по смещению.

Выше в этой главе я утверждал, что брокер записывает сообщение в раздел — добавляет запись в конец файла. Но точнее было бы сказать, что брокер добавляет запись в конец *активного сегмента*. Когда файл журнала достигает определенного

размера (по умолчанию 1 Мбайт), брокер создает новый сегмент, но продолжает использовать ранее созданные сегменты для обслуживания запросов потребителей. Рассмотрим иллюстрацию этого процесса, представленную на рис. 2.6.



**Рис. 2.6.** Создание новых сегментов

Согласно рисунку, брокер добавляет входящие записи в конец текущего активного сегмента. А как только его размер достигает установленного порога, брокер создает новый сегмент и назначает его активным. Этот процесс повторяется бесконечно.

Предельный размер сегмента определяется параметром `log.segment.bytes`, который имеет значение по умолчанию 1 Мбайт. Кроме того, брокер с течением времени будет создавать новые сегменты. Параметры `log.roll.ms` и `log.roll.hours` определяют максимальное время, через которое брокер создаст новый сегмент. Параметр `log.roll.ms` — основной, но он не имеет значения по умолчанию. Параметр `log.roll.hours` имеет значение по умолчанию 168 часов (7 дней). Когда брокер создает новый сегмент по времени, новая запись в сегменте получает отметку времени больше, чем самая ранняя отметка времени в текущем активном сегменте, плюс значение параметра `log.roll.ms` или `log.roll.hours`. Она не зависит от системного времени или времени последнего изменения файла.

#### ПРИМЕЧАНИЕ

Глядя на рис. 2.6, можно подумать, что все сегменты содержат одинаковое количество записей, но это не так. На практике количество записей в разных сегментах может различаться. Не забывайте, что брокер может создавать новые сегменты не только по достижении предельного размера файла, но и по истечении установленного времени.

Теперь, зная, как брокеры создают сегменты, можно перейти к обсуждению их роли в хранении данных.

### 2.6.1. Хранение данных

Поскольку записи могут поступать непрерывным потоком, брокерам необходимо удалять старые записи, чтобы не исчерпать пространство файловой системы. Для удаления данных используется двухуровневый подход: по времени и по размеру. Прежде всего Kafka удаляет записи старше заданного порога, основываясь на отметках времени в записях. Если бы брокер помещал все записи в один большой файл, ему пришлось бы сканировать файл в поисках всех записей, подлежащих удалению. Но

благодаря организации журнала в сегменты брокер может удалять сегменты целиком, в которых последняя запись, если судить по ее отметке времени, хранится дольше настроенного времени хранения. Ниже в порядке приоритетов перечислены три конфигурационных параметра, управляющих удалением данных:

- `log.retention.ms` — продолжительность хранения файла журнала в миллисекундах;
- `log.retention.minutes` — продолжительность хранения файла журнала в минутах;
- `log.retention.hours` — продолжительность хранения файла журнала в часах.

По умолчанию только параметр `log.retention.hours` имеет значение по умолчанию — 168 часов (семь дней). В Kafka также есть параметр `log.retention.bytes`, ограничивающий хранение на основе размера. По умолчанию он имеет значение -1. Если настроить ограничение хранения на основе размера и времени одновременно, то брокеры будут удалять сегменты, соответствующие любому из условий.

До сих пор мы обсуждали удаление данных целыми сегментами. Как вы наверняка помните, в Kafka записи хранят пары «ключ — значение». Возникает вопрос: можно ли сохранить последнюю запись для каждого ключа? Это означало бы не удаление сегментов целиком, а удаление только самых старых записей для каждого ключа. Kafka поддерживает такую возможность, предоставляя механизм сжатых топиков.

## 2.6.2. Сжатые топики

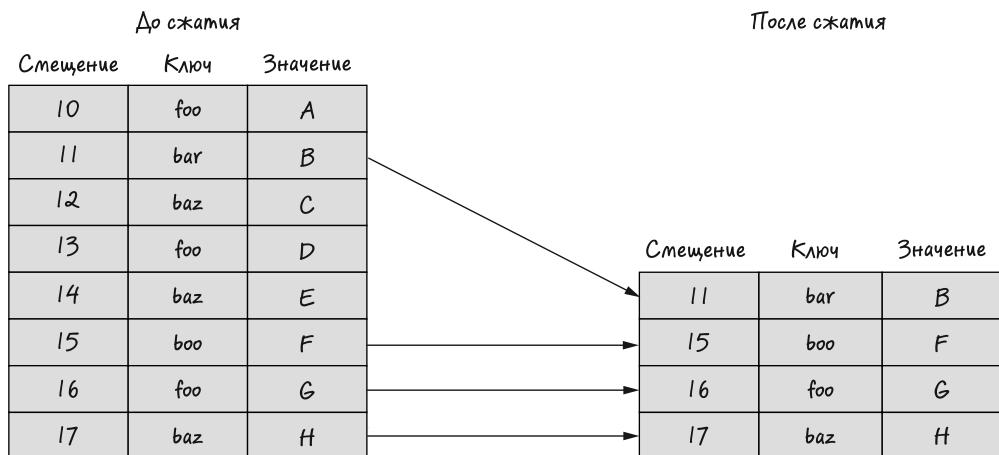
Рассмотрим случай, когда есть данные с ключом и эти данные периодически обновляются, то есть добавляются новые записи с тем же ключом, но с обновленным значением. Например, роль ключа может играть биржевой символ, а роль значения — цена за акцию, которая регулярно обновляется. Представьте, что вы используете эту информацию для отображения стоимости акций и у вас происходит сбой или перезагрузка. Вам нужна возможность начать резервное копирование с последними данными для каждого ключа (см. документацию Kafka, *Log Compaction*, <http://kafka.apache.org/documentation/#compaction>).

Если используется политика удаления, то брокер может удалить сегмент, созданный между последним обновлением и сбоем или перезапуском приложения. Тогда при запуске вы получите не все записи. Сохранение последнего известного значения для данного ключа — это, пожалуй, лучше, чем обработка последовательности записей с одинаковыми ключами и обновление таблицы в базе данных.

Обновление записей по ключу — это поведение, которое поддерживают сжатые топики (журналы). Вместо удаления целых сегментов на основе времени или размера сжатие действует более избирательно и удаляет старые записи *по ключам*. Механизм очистки журнала (пул потоков выполнения) работает в фоновом режиме, копируя файлы сегментов журнала и удаляя записи, если позже в журнале встречается событие с тем же ключом. На рис. 2.7 показано, как сжатие журнала сохраняет самое последнее сообщение для каждого ключа.

Этот подход гарантирует сохранность последней записи для данного ключа в журнале. Есть возможность по-разному настроить сохранение журнала для разных

топиков, то есть для одного топика можно настроить сохранение на основе времени, а для других топиков использовать сжатие.



**Рис. 2.7.** Слева — журнал до сжатия. Здесь можно видеть дубликаты ключей с разными значениями. Эти дубликаты — обновления. Справа — журнал после сжатия, здесь сохранились только самые последние значения для каждого ключа, благодаря чему размер журнала уменьшился

По умолчанию механизм очистки журнала включен. Чтобы применить сжатие к топику, необходимо при его создании указать свойство `log.cleanup.policy=compact`.

Сжатие используется в Kafka Streams при использовании хранилищ состояний, но вам не придется создавать эти журналы/топики — фреймворк сам решит эту задачу. Однако важно понимать, как работает сжатие. Сжатие журналов — это обширная тема, и мы лишь слегка коснулись ее. За дополнительной информацией обращайтесь к документации Kafka: <http://kafka.apache.org/documentation/#compaction>.

### ПРИМЕЧАНИЕ

У вас наверняка рано или поздно возникнет вопрос: как удалить запись из журнала при использовании настройки `log.cleanup.policy=compact`? В таком случае нужно добавить новую запись с тем же ключом и значением `null`, создав маркер «надгробия» (`tombstone`). Встретив такой маркер в следующий раз, механизм сжатия удалит предыдущие записи с тем же ключом, а позже Kafka удалит и маркер «надгробия», чтобы освободить место (подробнее об этом — в главе 5).

Главный вывод этого раздела: если у вас есть независимые, автономные события или сообщения, то используйте политику удаления. Если события или сообщения — обновляемые и ценность имеют только последние их значения, то используйте политику сжатия журнала.

Теперь, узнав, как брокеры Kafka управляют данными с помощью сегментов, вернемся чуть назад и поговорим о содержимом каталогов разделов топиков.

### 2.6.3. Содержимое каталога раздела топика

Выше в этой главе я говорил, что топики служат для логической группировки записей, а раздел — это фактическая физическая единица хранения. Брокеры Kafka добавляют каждую новую входящую запись в файл в каталоге, соответствующем топику и разделу, указанным в записи. Для примера в листинге 2.2 показано содержимое раздела топика.

#### Листинг 2.2. Содержимое каталога раздела топика

```
/var/kafka/topic-data/purchases-0
├── 00000000000000000000.index
├── 00000000000000000000.log
└── 00000000000000000000.timeindex
```

#### ПРИМЕЧАНИЕ

На практике, вероятнее всего, вы не будете взаимодействовать с брокером Kafka на этом уровне. Я рассказываю все эти детали, только чтобы вы могли лучше понять, как работает хранилище брокера.

Мы уже знаем, что файл с расширением `log` хранит записи Kafka, но какую роль играют файлы с расширениями `index` и `timeindex`? Когда брокер добавляет запись, то вместе с ключом и значением он сохраняет другие поля. Вот три из них:

- смещение (о котором мы уже говорили);
- размер;
- физическое положение записи в сегменте.

Файл с расширением `index` — это файл, отображаемый в память, который хранит соответствие между смещениями и фактическими позициями. Файл с расширением `timeindex` — это еще один файл, отображаемый в память, хранящий соответствие между отметкой времени и смещением. Сначала рассмотрим файл с расширением `index` (рис. 2.8).

000000000.index	000000000.log
смещение, позиция	..., смещение, позиция, размер...
0, 0	0, 0, 71
1, 71	1, 71, 80
2, 151	2, 151, 85



Рис. 2.8. Поиск начальной точки чтения записи со смещением 2

Брокеры используют индексные файлы, чтобы определить начальную точку для извлечения записи по заданному смещению. Брокеры выполняют двоичный поиск в файле с расширением `index`, отыскивают пару «индекс – позиция» с наибольшим смещением, которое меньше или равно целевому смещению. Смещение, хранящееся в индексном файле, является относительным к базовому смещению. Это означает, что если базовое смещение равно 100, то смещение 101 хранится как 1; смещение 102 — как 2 и т. д. Благодаря использованию относительных смещений

в индексном файле можно использовать две 4-байтные записи для хранения смещения и позиции. Базовое смещение — это число, используемое как имя файла, и вскоре мы рассмотрим его.

Файл с расширением `timeindex` — это отображаемый в память файл, хранящий соответствие между отметкой времени и смещением (рис. 2.9).

```
000000000.timeindex
смещение, отмечка времени
122456789, 0
...
123985789, 0
```

**Рис. 2.9.** Файл с расширением `timeindex`

### ПРИМЕЧАНИЕ

Файл, отображаемый в память, — это особый файл в Java, часть которого хранится в памяти, что позволяет быстрее читать данные из него. Более подробное описание вы найдете в замечательной статье *What Is Memory-Mapped File in Java* на сайте GeeksForGeeks (<http://mng.bz/wj47>).

Каждая запись в этом файле содержит 8-байтную отметку времени и 4-байтное значение относительного смещения. Выполняя поиск, брокер извлекает отметку времени из самого раннего сегмента. Если она меньше искомой отметки времени, то затем брокер выполняет двоичный поиск ближайшей записи в файле `timeindex`.

А что насчет имен? Брокер дает этим файлам имена на основе первого смещения в файле журнала. Сегмент в Kafka включает файлы `log`, `index` и `timeindex`. Итак, в нашем предыдущем примере (см. листинг 2.2) каталог содержит один активный сегмент. После того как брокер создаст новый сегмент, каталог будет выглядеть примерно так, как показано в листинге 2.3.

### Листинг 2.3. Содержимое каталога после создания нового сегмента

```
/var/kafka/topic-data/purchases-0
├── 000000000000000000000000.index
├── 000000000000000000000000.log
├── 000000000000000000000000.timeindex
├── 000000000000037348.index
├── 000000000000037348.log
└── 000000000000037348.timeindex
```

Судя по именам файлов в этом каталоге, первый сегмент содержит записи со смещениями 0–37 347, а второй сегмент — записи со смещениями, начиная с 37 348.

Файлы в каталоге раздела топика имеют двоичный формат, и их нельзя просмотреть с помощью простого текстового редактора. Как я уже говорил, вам не придется напрямую взаимодействовать с этими файлами, но иногда вам может потребоваться просмотреть их содержимое.

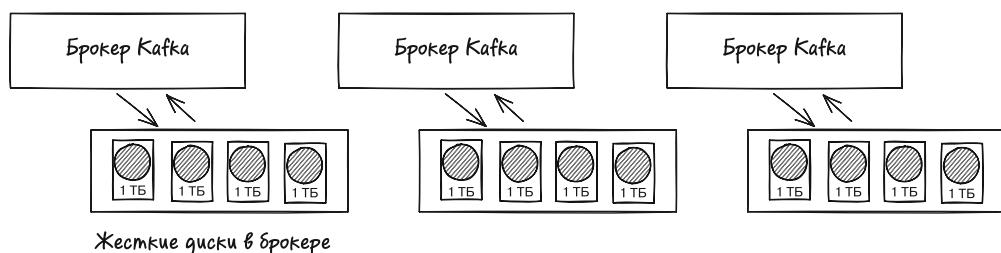
### ВНИМАНИЕ

*Никогда не изменяйте* файлы в каталоге раздела топика и *не обращайтесь к ним напрямую*.

А для просмотра их содержимого используйте инструменты, предоставляемые Kafka.

## 2.7. МНОГОУРОВНЕВОЕ ХРАНИЛИЩЕ

Мы узнали, что брокеры являются слоем хранения в архитектуре Kafka, хранят данные в неизменяемых файлах, предназначенных только для добавления, и ограничивают рост объема данных, удаляя сегменты, когда данные достигают возраста, превышающего настроенное время хранения. Но, так как Kafka может использоваться в качестве центральной нервной системы для данных, когда все ваши данные поступают в Kafka, требования к дисковому пространству будут продолжать расти. Этую проблему иллюстрирует рис. 2.10.



**Рис. 2.10.** Брокеры Kafka хранят все данные на локальном диске, и объем данных со временем будет увеличиваться

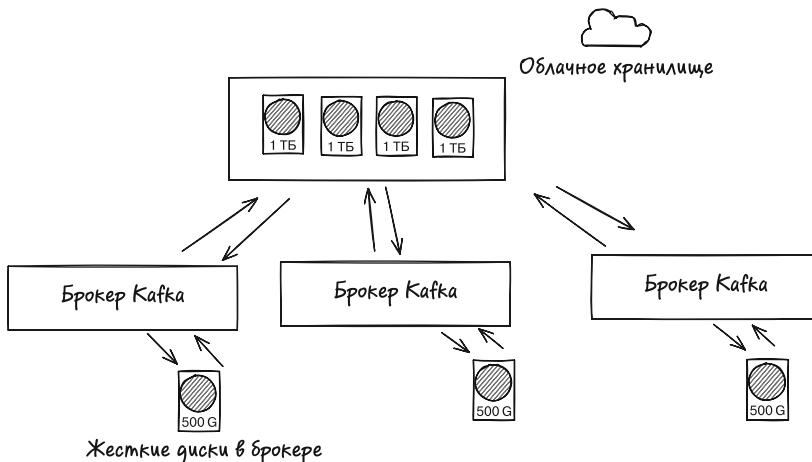
Как видите, чтобы хранить данные дольше, вам придется добавить больше дисков и получить больше места для новых записей. Это означает, что пользователи Kafka, желающие хранить данные дольше обычного, должны выгрузить эти данные из кластера в более масштабируемое хранилище. Для перемещения данных можно использовать Kafka Connect (о котором мы поговорим в главе 5), но использование внешнего хранилища требует создания различных приложений для доступа к данным в нем.

Новая возможность в Kafka, получившая название Tiered Storage (многоуровневое хранилище), отделяет уровни вычислений и хранения. Здесь я дам только краткое описание, а за более подробной информацией советую обратиться к KIP-405 (<http://mng.bz/qjZK>). Суть предложения заключается в том, чтобы добавить в брокеры Kafka концепцию локального и удаленного хранилища. Локальное хранилище — это то же самое хранилище, которое брокеры используют в настоящее время, а удаленное хранилище — это нечто более масштабируемое, скажем S3, но доступное для управления брокерам Kafka. На рис. 2.11 показана еще одна иллюстрация, демонстрирующая, как брокеры могут сохранять данные, используя идею многоуровневого хранения.

Идея заключается в том, чтобы дать брокерам возможность переносить старые данные в удаленное хранилище. Этот многоуровневый подход к хранению необходим по двум причинам.

- Миграция данных осуществляется как часть нормальной работы брокеров Kafka. Никаких дополнительных процессов для перемещения старых данных не требуется.
- Более старые данные остаются доступными через брокеров Kafka, поэтому для их обработки дополнительные приложения не требуются. Многоуровневое хранили-

ще будет бесшовно функционировать для клиентских приложений. Они не будут и не должны знать, откуда получены потребляемые записи, из локального или внешнего хранилища.



**Рис. 2.11.** Брокеры Kafka хранят локально только «горячие» данные, а все «теплые» и «холодные» записи переносятся в облачное хранилище

Многоуровневая организация хранения открывает перед брокерами Kafka бесконечные возможности хранения. Еще одно преимущество многоуровневого хранения, неочевидное на первый взгляд, — улучшение эластичности. До внедрения многоуровневого хранения при добавлении нового брокера требовалось перемещать по сети целые разделы. Помните, как выше мы говорили, что Kafka распределяет разделы топиков между брокерами. Поэтому добавление нового брокера означает необходимость перераспределения связей и соответствующее перемещение данных. При многоуровневом хранении большинство сегментов, кроме активных, будет находиться во внешнем хранилище. Это означает, что перемещать придется гораздо меньше данных и изменение количества брокеров будет происходить намного быстрее.

На момент написания книги (октябрь 2023 года) в версии Apache Kafka 3.6.0 появилась предварительная поддержка многоуровневого хранилища. В настоящий момент ее не рекомендуется использовать в промышленных приложениях. И снова всем, кто интересуется подробностями, связанными с функцией многоуровневого хранилища, я рекомендую обратиться к KIP-405 (<http://mng.bz/qjZK>).

## 2.8. МЕТАДАННЫЕ КЛАСТЕРА

Kafka — это распределенная система, которой для управления всеми действиями и состояниями в кластере необходимы метаданные. Наличие метаданных для хранения состояния кластера является неотъемлемой частью архитектуры Kafka. Исторически для управления метаданными в Kafka применялся ZooKeeper (<https://zookeeper.apache.org/>). Но теперь, с выходом KIP-500, для той же цели могут

использоваться брокеры Kafka, при этом серверы Kafka работают в режиме KRaft. Описание подробностей вы найдете в KIP-500 (<http://mng.bz/7vPx>). В своей статье *Apache Kafka Needs No Keeper: Removing the Apache ZooKeeper Dependency* (<http://mng.bz/mjrn>) Колин Маккейб (Colin McCabe) описывает, как и когда происходят изменения в Kafka.

В настоящее время вы можете настроить запуск Kafka в режиме ZooKeeper или KRaft, но предпочтительнее, конечно, выбрать режим KRaft, потому что с выпуском версии 4.0 планируется удалить поддержку ZooKeeper. Поскольку целевая аудитория этой книги – разработчики, а не администраторы кластеров, то вам будет достаточно лишь в общих чертах знать, *как* Kafka использует метаданные. Хранение и использование метаданных позволяет Kafka делить брокеров на ведущих и ведомых и выполнять такие действия, как отслеживание репликации топиков.

Метаданные кластера используются в Kafka для поддержки таких возможностей, как:

- *членство в кластере* – присоединение и поддержание членства в кластере. Если брокер становится недоступным, то ZooKeeper исключает брокер из состава членов кластера;
- *конфигурация топика* – отслеживание топиков в кластере, определение ведущего брокера для топика, выяснение количества разделов и любые конкретные переопределения настроек для топика;
- *управление доступом* – назначение пользователям (людям или другому программному обеспечению) привилегий на чтение и запись в определенные топики.

Это был краткий обзор управления метаданными в Kafka. Я не хочу слишком подробно рассказывать об управлении метаданными, поскольку в этой книге стою на позиции разработчика, а не администратора, управляющего кластером Kafka. Теперь, когда мы кратко обсудили потребность Kafka в метаданных и как они используются, продолжим обсуждение брокеров и поговорим об их делении на ведущих и ведомых и их роли в репликации.

## 2.9. ВЕДУЩИЕ И ВЕДОМЫЕ БРОКЕРЫ

К настоящему моменту мы обсудили назначение топиков в Kafka и почему топики имеют разделы. Вы узнали, что разделы не обязательно должны находиться на одной машине и могут быть распределены между брокерами по всему кластеру. Теперь пришло время посмотреть, как Kafka обеспечивает доступность данных в условиях сбоев целых серверов.

В кластере Kafka для каждого раздела топика один брокер является ведущим, а остальные – ведомыми (рис. 2.12).

На рис. 2.12 показано упрощенное представление идеи «ведущий – ведомый». Ведущий брокер раздела топика обрабатывает все запросы на производство и потребление (хотя потребители могут работать и с ведомыми брокерами, как вы увидите в главе 4, посвященной клиентам). Ведомые брокеры получают записи от ведущего для данного раздела топика. Kafka использует эту связь «ведущий – ведомый», чтобы

обеспечить целостность данных. Напомню, что ведущий брокер играет особую роль для топика, поэтому разделы распределяются по всему кластеру — никакой отдельный брокер не может стать ведущим для всех разделов заданного топика.

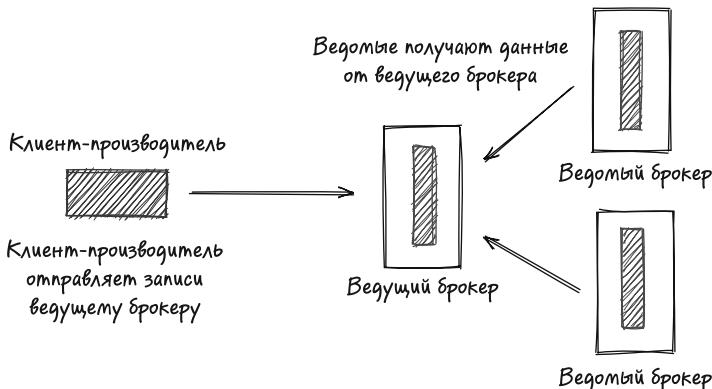


Рис. 2.12. Пример ведущих и ведомых брокеров

Но прежде, чем обсуждать работу ведущих и ведомых брокеров, а также репликацию, мы должны рассмотреть, что делает Kafka, чтобы обеспечить репликацию между ведущими и ведомыми брокерами.

## 2.9.1. Репликация

В предыдущем разделе о ведущих и ведомых брокерах я отметил, что каждый раздел топика обслуживается одним ведущим брокером и несколькими ведомыми брокерами. Эту идею иллюстрирует рис. 2.12. После того как ведущий брокер добавит записи в журнал, ведомые брокеры читают эти данные через ведущий.

Kafka реплицирует (копирует) записи между брокерами, чтобы обеспечить их сохранность в случае сбоя брокера в кластере. На рис. 2.13 показан пример потока репликации между брокерами. Уровень репликации определяется пользовательскими настройками, но вообще рекомендуется использовать число 3. При коэффициенте репликации 3 ведущий брокер считается репликой 1, а два ведомых — репликами 2 и 3.

Процесс репликации Kafka прост. Ведомые брокеры потребляют сообщения от ведущего брокера. После того как ведущий добавит новые записи в журнал, ведомые получат их от ведущего и добавят в свои журналы. После того как ведомые добавят записи, их журналы будут представлять собой точные копии (реплики) журнала ведущего брокера с теми же данными и смещениями. Когда они полностью догонят ведущий, эти ведомые брокеры будут считаться синхронизированными репликами (*in-sync replica, ISR*).

Когда производитель отправляет пакет записей, ведущий должен добавить их в журнал до передачи ведомым. Есть небольшой промежуток времени, когда ведущий будет впереди ведомых, как показано на рис. 2.14.

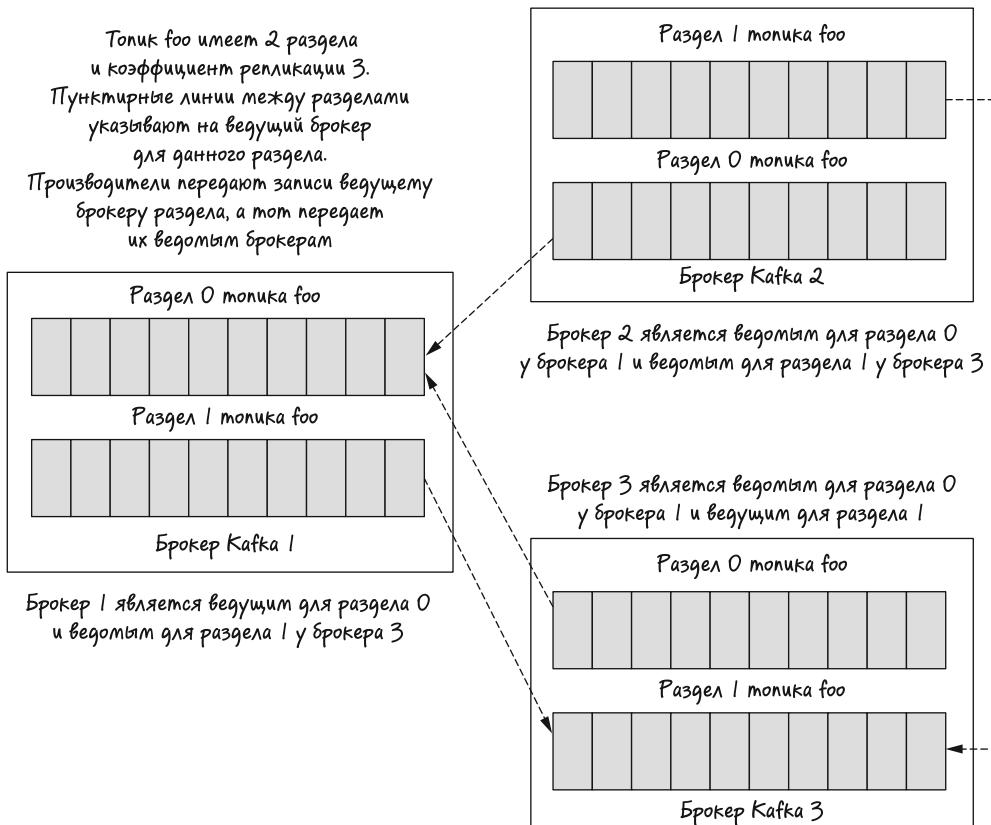


Рис. 2.13. Процесс репликации в Kafka

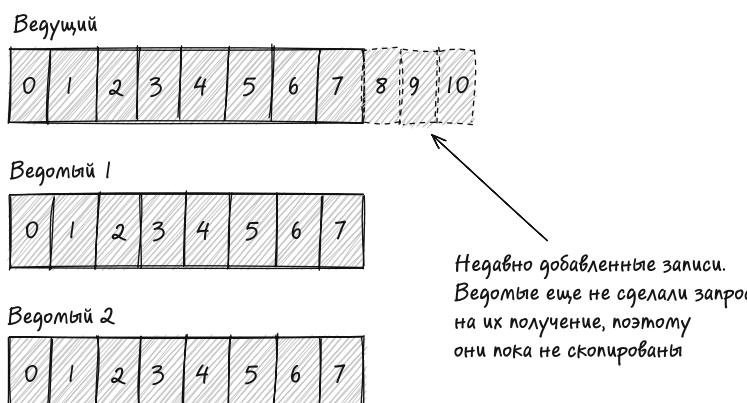


Рис. 2.14. В разделе у ведущего брокера может иметься несколько еще не реплицированных сообщений

На практике эта небольшая задержка репликации не является проблемой. Но мы должны убедиться, что репликация не слишком сильно отстает, так как это может указывать на проблемы с ведомыми. Итак, как нам определить, что отставание не слишком большое? У брокеров Kafka есть конфигурационный параметр `replica.lag.time.max.ms` (рис. 2.15).



**Рис. 2.15.** Ведомые должны отправить запрос на выборку и уложиться в заданное в конфигурации время задержки

Этот параметр определяет предельное время, в течение которого ведомые должны получить полную копию журнала ведущего. Если ведомым не удается сделать это в течение заданного времени, то они считаются слишком сильно отстающими и удаляются из списка ISR.

Как я уже говорил, ведомые брокеры, догнав свой ведущий брокер, считаются синхронизированными. Синхронизированные брокеры могут быть избраны ведущими, если текущий ведущий брокер выйдет из строя или станет недоступным (см. раздел *Replication* в документации Kafka, <http://kafka.apache.org/documentation/#replication>).

В Kafka потребители никогда не видят записи, которые не были записаны всеми синхронизированными брокерами. Смещение последней записи, сохраненной всеми репликами, известно как наивысшая отметка и является наибольшим смещением, доступным потребителям. Это свойство Kafka означает, что потребители не беспокоятся об исчезновении недавно прочитанных записей. В качестве примера рассмотрим ситуацию, изображенную на рис. 2.15. Поскольку смещения 8–10 еще не были записаны во все реплики, наибольшим смещением, доступным потребителям топика, считается смещение 7.

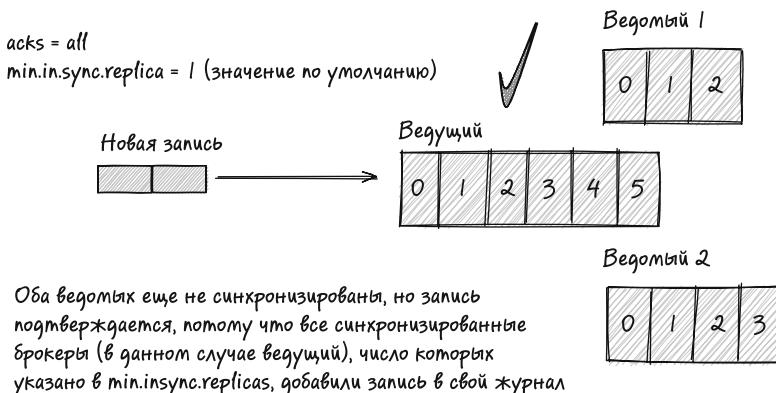
Если ведущий брокер станет недоступным или выйдет из строя до того, как записи 8–10 сохранятся, то производитель не получит подтверждения и отправит записи повторно. В данном сценарии есть еще кое-что, но об этом мы поговорим в главе 4, посвященной клиентам.

В иных случаях выхода ведущего брокера из строя у ведомого будет иметься полная реплика журнала. Но давайте исследуем, как связаны между собой ведущие, ведомые и реплики.

## Репликация и подтверждения

Отправляя записи в Kafka, производитель ждет подтверждения их получения и сохранения ни в одной, в некоторых или во всех синхронизированных репликах. Эти различные настройки позволяют производителю выбирать компромисс между задержкой и сохранностью данных. Но есть один важный момент, который следует учитывать.

Ведущий брокер сам считается репликой. Конфигурация `min.insync.replicas` определяет, сколько реплик должно быть синхронизировано, чтобы считать запись зафиксированной. Значение по умолчанию для `min.insync.replicas` равно единице. Если предположить, что размер кластера и фактор репликации равны 3, то с настройкой `acks=all` достаточно подтверждения только от ведущего брокера. Этот сценарий демонстрирует рис. 2.16.



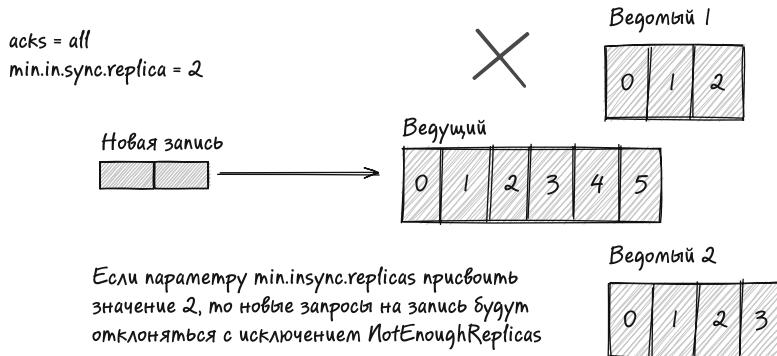
**Рис. 2.16.** Параметр `acks` со значением `all` и параметр `min.insync.replicas` со значением по умолчанию

Как такое возможно? Представьте, что двое ведомых временно отстают достаточно сильно, что контроллер удалил их из списка синхронизированных реплик. Это означает, что даже при установке `acks=all` на стороне производителя существует потенциальный риск потери данных, если вдруг ведущий брокер выйдет из строя до того, как ведомые восстановят работоспособность и синхронизируются.

Чтобы предотвратить такой сценарий, нужно установить параметр `min.insync.replicas=2`. В этом случае ведущий брокер проверит количество синхронизированных реплик перед добавлением новой записи в свой журнал и обработает запрос на запись, только если достигнуто требуемое количество синхронизированных реплик. В противном случае выдаст `NotEnoughReplicasException` и производитель повторит запрос.

Рассмотрим еще одну ситуацию, изображенную на рис. 2.17, чтобы получить более четкое представление о происходящем. Как показано здесь, ведущий брокер получает пакет записей, но он не будет добавлять их в свой журнал, потому что не достигнуто достаточное количество синхронизированных реплик. Такое поведение улучшит сохранность данных, потому что запрос на запись будет обработан только

после появления достаточного количества синхронизированных реплик. Здесь мы обсудили подтверждение приема сообщений в зависимости от числа синхронизированных реплик с точки зрения брокера. В главе 4 мы рассмотрим эту идею с точки зрения клиента-производителя и обсудим компромиссы производительности.



**Рис. 2.17.** Присваивание параметру `min.insync.replicas` значения больше чем 1 улучшает сохранность данных

## 2.10. ПРОВЕРКА РАБОТОСПОСОБНОСТИ БРОКЕРА

В начале главы мы узнали, что брокер Kafka обрабатывает запросы клиентов в порядке поступления. Брокеры Kafka обрабатывают несколько типов запросов, включая, например, запросы:

- *на запись* — запрос на добавление записей в журнал;
- *на извлечение* — запрос на потребление записей, начиная с заданного смещения;
- *на получение метаданных* — запрос текущего состояния кластера: ведущие брокеры для разделов, доступные разделы и т. д.

Это небольшое подмножество всех запросов, которые можно послать брокеру. Брокер обрабатывает запросы в порядке «первым пришел, первым вышел», передавая их соответствующим обработчикам в зависимости от типа запроса.

Проще говоря, клиент посылает запрос, а брокер отвечает. Если запросы поступают чаще, чем брокер может ответить, то они выстраиваются в очередь. Внутри Kafka есть пул потоков, предназначенный для обработки входящих запросов. Этот процесс приводит нас к первой линии проверки проблем, от которых может пострадать производительность кластера Kafka.

В распределенной системе отказ следует воспринимать как норму. Но это не означает, что система должна отключаться при первых признаках проблемы. Проблемы с сетевыми разделами в распределенной системе — обычное дело, и они часто быстро разрешаются. Поэтому особый смысл приобретает понятие повторяющихся ошибок, отличное от понятия фатальных ошибок. Но если возникли проблемы с Kafka, например, попытки чтения/записи данных стали завершаться по тайм-ауту, то где в первую очередь следует искать причину?

## 2.10.1. Процент простоя обработчика запросов

При возникновении проблем с приложением Kafka прежде всего следует проверить JMX-метрику `RequestHandlerAvgIdlePercent`. Она показывает среднюю долю времени (от 0 до 1), в течение которого простоявали потоки, обрабатывающие запросы. В нормальных условиях можно ожидать коэффициент простоя 0,7–0,9, который указывает, что брокер обрабатывает запросы быстро. Если доля времени, приходящаяся на простоя, достигает нуля, то это означает, что потоки-обработчики не успевают обрабатывать входящие запросы и очередь запросов продолжает расти. Огромная очередь запросов является признаком проблем и причиной более длительного времени ответа и возможных тайм-аутов.

## 2.10.2. Процент простоя сетевого обработчика

JMX-метрика `NetworkProcessorAvgIdlePercent` аналогична предыдущей метрике, но измеряет среднее время простоя сетевых обработчиков. Хорошим считается значение выше 0,5. Если оно постоянно ниже 0,5, то это свидетельствует о проблеме.

## 2.10.3. Несинхронизированные разделы

JMX-метрика `UnderReplicatedPartitions` представляет количество разделов, принадлежащих брокеру, удаленному из списка синхронизированных реплик. Синхронизацию и репликацию мы обсудили в подразделе 2.9.1. Значение выше нуля означает, что брокер Kafka не справляется с репликацией разделов, назначенных ему как ведомому. Ненулевое значение метрики `UnderReplicatedPartitions` может указывать на проблемы с сетью или на то, что брокер перегружен и не справляется со своей работой. Обратите внимание, что всегда желательно видеть в метрике `UnderReplicatedPartitions` число 0.

# ИТОГИ ГЛАВЫ

- Брокер Kafka — это слой хранения, обрабатывающий клиентские запросы на производство (запись) и потребление (чтение) записей.
- Брокеры Kafka получают записи в виде байтов, сохраняют их в том же формате и возвращают в ответ на запросы чтения.
- Брокеры Kafka надежно хранят записи в топиках.
- Топики — это каталоги в файловой системе, разделенные на разделы, то есть записи, помещаемые в топик, сохраняются в разных разделах.
- Kafka использует разделы для увеличения пропускной способности и распределения нагрузки по нескольким брокерам.
- Брокеры Kafka для большей надежности копируют данные друг друга.

## Часть II

В первой части вы познакомились с платформой потоковой передачи событий Apache Kafka и исследовали различные компоненты верхнего уровня. Затем вы узнали, как работает брокер Kafka и какие функции он выполняет, выступая в роли центральной нервной системы для данных. В этой части вам предстоит узнать, как загружать данные в Kafka.

Сначала мы рассмотрим Schema Registry — компонент, помогающий обеспечить соблюдение подразумеваемого контракта между производителями и потребителями Kafka (если вы еще не знаете, что такое «подразумеваемый контракт», то не волнуйтесь, далее я объясню это понятие). Вы можете сказать: «Я не использую схемы». Однако это не совсем так: вы всегда используете хотя бы одну схему, просто она может быть неявной. К концу главы 3 вы поймете, что я имею в виду и как Schema Registry решает эту проблему.

Затем мы перейдем к рабочим лошадкам платформы Kafka — к клиентам производителя и потребителя. Вы увидите, как производители загружают данные в Kafka и как потребители извлекают их. Изучение клиентов Kafka имеет важное значение, так как в Kafka есть важные инструменты, являющиеся абстракциями над производителями и потребителями, поэтому очень важно понимать особенности их работы.

И под конец второй части вы откроете для себя Kafka Connect. Архитектура Kafka Connect, основывающаяся на клиентах-производителях и клиентах-потребителях, служит мостом между внешним миром и Kafka. Коннекторы-источники могут загружать данные в Kafka из реляционных баз данных и практически любых других внешних хранилищ или систем, таких как ElasticSearch, Snowflake или S3. Коннекторы-приемники выполняют обратную операцию — экспортируют данные из Kafka во внешние системы.

# 3

## *Schema Registry*

### **В этой главе**

- ✓ Использование байтов подразумевает правила сериализации.
- ✓ Что такое схема и почему ее нужно использовать.
- ✓ Что такое Schema Registry.
- ✓ Обеспечение совместимости с изменениями — эволюция схемы.
- ✓ Особенности именования субъектов.
- ✓ Повторное использование схем со ссылками.

В главе 2 вы познакомились с брокерами Kafka, находящимися в сердце платформы. В частности, вы узнали, что брокер — это слой хранения, добавляющий входящие сообщения в топик, играющий роль неизменяемого распределенного журнала событий. Топик — это каталог, содержащий файл (-ы) журнала.

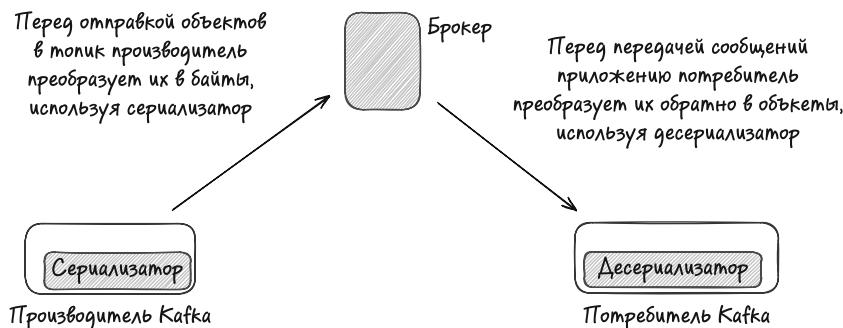
Поскольку производители отправляют сообщения по сети, эти сообщения необходимо сначала сериализовать в массив двоичных байтов. Брокер Kafka никак не изменяет сообщения и сохраняет их в исходном формате. То же происходит, когда брокер выполняет запросы на выборку от потребителей, — он извлекает уже сериализованные сообщения и отправляет их по сети.

Работая с сообщениями как с массивами байтов, брокер никак не зависит от типов данных, составляющих сообщения, от приложений, производящих и потребляющих сообщения, а также от языков программирования, на которых написаны

эти приложения. Независимость брокера от форматов представления данных позволяет любому клиенту, использующему протокол Kafka, производить или потреблять сообщения.

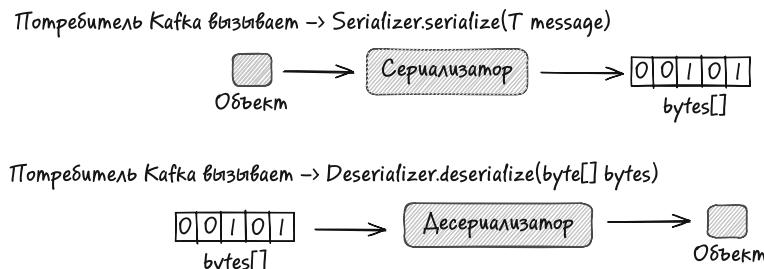
### 3.1. ОБЪЕКТЫ

Байты необходимы для хранения и передачи данных по сети, однако гораздо эффективнее работать с данными на более высоком уровне абстракции — когда они принимают форму объектов. Но где происходит это преобразование? На уровне клиентов — производителей и потребителей сообщений (рис. 3.1).



**Рис. 3.1.** Преобразование объектов в байты и байтов в объекты происходит на уровне клиентов

Как показано на рис. 3.1, производитель преобразует объект сообщения в байты с помощью экземпляра `Serializer` и затем передает полученные байты брокеру для помещения в топик. Потребитель выполняет обратную последовательность операций: получает байты из топика и с помощью экземпляра `Deserializer` преобразует их обратно в формат объекта. Производитель и потребитель не связаны с сериализатором и десериализатором и просто вызывают метод `serialize` или `deserialize` (рис. 3.2).



**Рис. 3.2.** Сериализатор и десериализатор не зависят от производителя и потребителя и выполняют ожидаемые действия при вызове их методов `serialize` и `deserialize`

Как показано на рис. 3.2, производитель предполагает, что работает с экземпляром интерфейса `Serializer`. Он вызывает метод `Serializer.serialize`, передает ему объект заданного типа и получает байты. Потребитель работает с интерфейсом `Deserializer` и вызывает его метод `Deserializer.deserialize`, которому передает массив байтов и получает объект заданного типа.

Производитель и потребитель получают экземпляры сериализатора и десериализатора через параметры конфигурации. Соответствующие примеры вы увидите далее в этой главе.

#### ПРИМЕЧАНИЕ

Я упоминаю здесь и далее производители и потребители, но мы не будем вдаваться в детали и познакомимся с их особенностями ровно настолько, сколько нужно, чтобы понять главную тему этой главы. Подробнее о производителях и потребителях рассказывается в следующей главе.

Я хочу особо подчеркнуть, что тип объекта, который производитель сериализует для передачи в заданный топик, совпадает с типом объекта, который десериализует потребитель. Поскольку производители и потребители никак не связаны друг с другом, типы сообщений или объектов предметных событий представляют собой неявный контракт между отправителем и получателем.

Возникает естественный вопрос: есть ли что-то, что разработчики производителей и потребителей могут использовать для выяснения правильной структуры сообщений? Ответ на этот вопрос — да, схема.

## 3.2. ЧТО ТАКОЕ СХЕМА И ЗАЧЕМ ОНА НУЖНА

Когда разработчик слышит слово «схема», то первое, что приходит ему в голову, — это схема базы данных. Схема базы данных описывает ее структуру, включая имена и типы столбцов в таблицах и отношения между таблицами. Но здесь я говорю о совершенно иной схеме, хотя она и похожа по назначению на схему базы данных.

В нашем случае под схемой подразумевается описание объекта, не зависящее от языка, включая имя объекта, а также имена и типы его полей. В листинге 3.1 представлен пример схемы в формате JSON.

#### Листинг 3.1. Пример простой схемы в формате JSON

```
{sdd
"name": "Person", ← Имя объекта
"fields": [ ← Определение полей объекта
    {"name": "name", "type": "string"}, ←
    {"name": "age", "type": "int"}, ← Имена полей и их типы
    {"name": "email", "type": "string"}
]
}
```

Вымышленная схема в этом примере описывает объект с именем Person с полями, которые мы ожидаем найти в таком объекте. Теперь у нас есть структурированное описание объекта, которое производители и потребители могут использовать в качестве соглашения или контракта о том, как объект должен выглядеть до и после сериализации. В подразделе 3.2.9 я расскажу подробнее об использовании схем при построении и (де)сериализации сообщений.

А пока я хотел бы коснуться ключевых моментов, которые мы выяснили на данный момент:

- брокер Kafka работает только с сообщениями в двоичном формате (массивами байтов);
- производители и потребители Kafka отвечают за (де)сериализацию сообщений, кроме того, поскольку они ничего не знают друг о друге, записи формируют контракт между ними.

Мы также узнали, что с помощью схемы можем явно объявить контракт между производителями и потребителями. Итак, теперь мы знаем, *зачем* использовать схему, но все, что мы выяснили до сих пор, выглядит немного абстрактным и потому нам нужно ответить на следующие вопросы *«как»*.

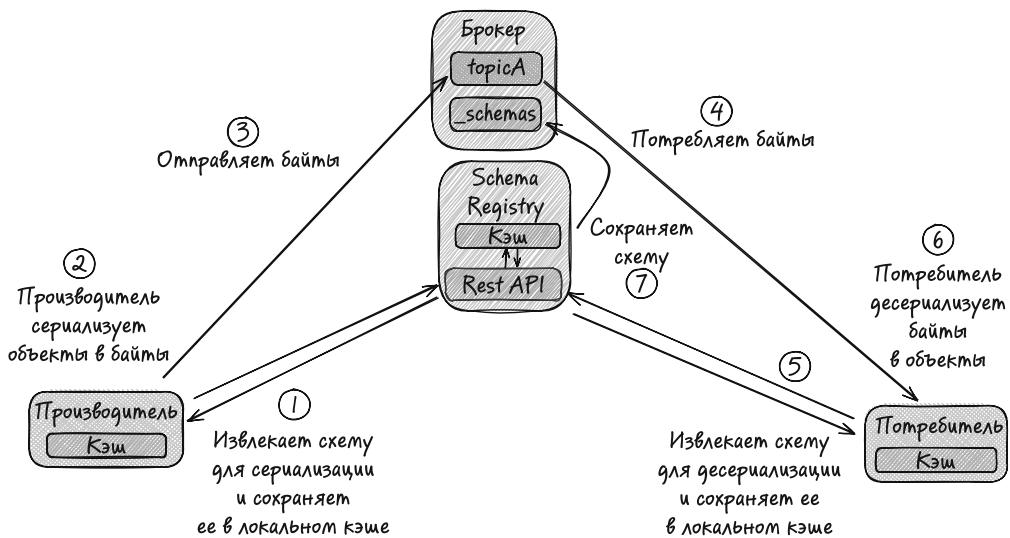
- Как использовать схемы в жизненном цикле разработки приложений?
- Как производители и потребители Kafka могут использовать сериализацию и гарантировать правильный формат сообщений, учитывая, что они отделены от этих операций?
- Как обеспечить использование правильной версии схемы? В конце концов, схемам свойственно меняться.

Ответом на эти вопросы *«как»* является Schema Registry.

### 3.2.1. Что такое Schema Registry

Schema Registry предоставляет приложение для централизованного хранения схем, их проверки и эволюции (изменения структуры сообщений). Что особенно важно, Schema Registry служит гарантом истинности схем, которую клиенты-производители и клиенты-потребители могут быстро проверить. Schema Registry предоставляет сериализаторы и десериализаторы для настройки производителей и потребителей Kafka, что упрощает разработку приложений, работающих с Kafka.

Код сериализации в Schema Registry поддерживает схемы в фреймворках сериализации Avro (<https://avro.apache.org/docs/current/>) и Protocol Buffers (<https://developers.google.com/protocol-buffers>). Далее я буду называть Protocol Buffers как Protobuf. Кроме того, Schema Registry поддерживает схемы, написанные с использованием JSON Schema (<https://json-schema.org/>), но это скорее спецификация, чем фреймворк. Далее я покажу, как использовать Avro, Protobuf и JSON Schema, а пока рассмотрим в общих чертах, как работает Schema Registry (рис. 3.3).



**Рис. 3.3.** Schema Registry обеспечивает согласованность формата данных между производителями и потребителями

Давайте кратко рассмотрим, как работает Schema Registry, на примере этой иллюстрации.

1. Когда производитель вызывает метод `serialize`, то сериализатор, поддерживающий Schema Registry, извлекает схему (через HTTP) и сохраняет ее в своем локальном кэше.
2. Сериализатор, встроенный в производитель, сериализует запись.
3. Производитель отправляет сериализованное сообщение (байты) в Kafka.
4. Потребитель читает байты.
5. Десериализатор в потребителе, поддерживающий Schema Registry, извлекает схему и сохраняет ее в своем локальном кэше.
6. Потребитель десериализует байты, согласно схеме.
7. Серверы Schema Registry создают сообщение, а схема сохраняется в топике `_schemas`.

#### СОВЕТ

Я представляю Schema Registry как неотъемлемую часть платформы потоковой передачи событий Kafka, но в действительности это не так. Помните, производители и потребители Kafka отделены от используемых ими сериализаторов и десериализаторов. С тем же успехом производителю и потребителю можно передать экземпляр нестандартного класса, реализующего соответствующий интерфейс. Правда, при этом вы потеряете проверки достоверности, выполняемые при использовании Schema Registry. О сериализации без Schema Registry я расскажу в конце этой главы.

Предыдущая иллюстрация дает неплохое представление о том, как работает Schema Registry, но есть важная деталь, на которую я хотел бы обратить внимание. Сериализатор и десериализатор действительно обращаются к Schema Registry за

получением схемы для объектов данного типа, но делается это *только один раз* — в первый раз, когда пересыпается объект, для которого у них нет схемы. После этого схема, необходимая для операций (де)сериализации, извлекается из локального кэша.

### 3.2.2. Запуск Schema Registry

Наш первый шаг — запустить Schema Registry. Для ускорения процесса обучения и разработки снова используем Docker Compose. С этой целью возьмите файл `docker-compose.yml` из корневого каталога в репозитории с исходным кодом книги.

Этот файл похож на файл `docker-compose.yml`, который мы использовали в главе 2. Но в дополнение к образу Kafka в нем есть запись для развертывания образа Schema Registry. Далее запустите команду `docker-compose up -d`. Напомню, что ключ `-d` в командах Docker означает «отсоединенный» (*detached*) режим, то есть контейнеры docker, запущенные с этим флагом, работают в фоновом режиме, освобождая окно терминала, в котором выполнялась команда.

### 3.2.3. Архитектура

Прежде чем мы перейдем к изучению приемов работы с Schema Registry, желательно получить общее представление о его архитектуре. Schema Registry — это распределенное приложение, находящееся за пределами брокеров Kafka. Клиенты взаимодействуют с Schema Registry через REST API. Клиентом может быть сериализатор (производитель), десериализатор (потребитель), плагин инструмента сборки или инструмент командной строки `curl`, посылающий запросы. Об использовании плагинов инструмента сборки — в данном случае Gradle — я расскажу в подразделе 3.2.6.

Schema Registry использует Kafka в качестве хранилища (журнала опережающей записи) для всех своих схем, создавая для этого сжатый топик `_schemas` с одним разделом, и имеет простую архитектуру, в которой только один ведущий узел, а остальные узлы являются ведомыми.

#### ПРИМЕЧАНИЕ

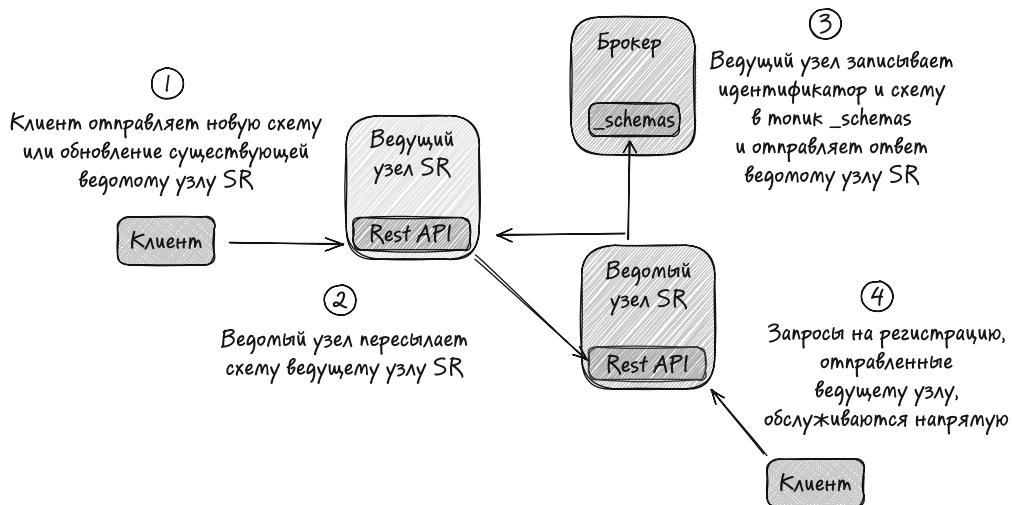
В соответствии с соглашениями об именовании, принятыми в Kafka, двойные символы подчеркивания (`__`) обозначают внутренние топики, не предназначенные для публичного использования. С этого момента мы будем называть этот топик просто `schemas`.

В этой простой архитектуре только ведущий узел пишет в топик `_schemas`. Любой другой узел примет запрос на сохранение или обновление схемы и перешлет его ведущему узлу. Этот процесс показан на рис. 3.4.

Каждый раз, когда клиент регистрирует или обновляет схему, ведущий узел создает запись в топике `_schemas`. Для записи в топик Schema Registry использует производитель Kafka, а для чтения обновлений все узлы используют потребитель. Как видите, Schema Registry создает резервную копию своего локального состояния в топике Kafka, что обеспечивает надежное хранение схем.

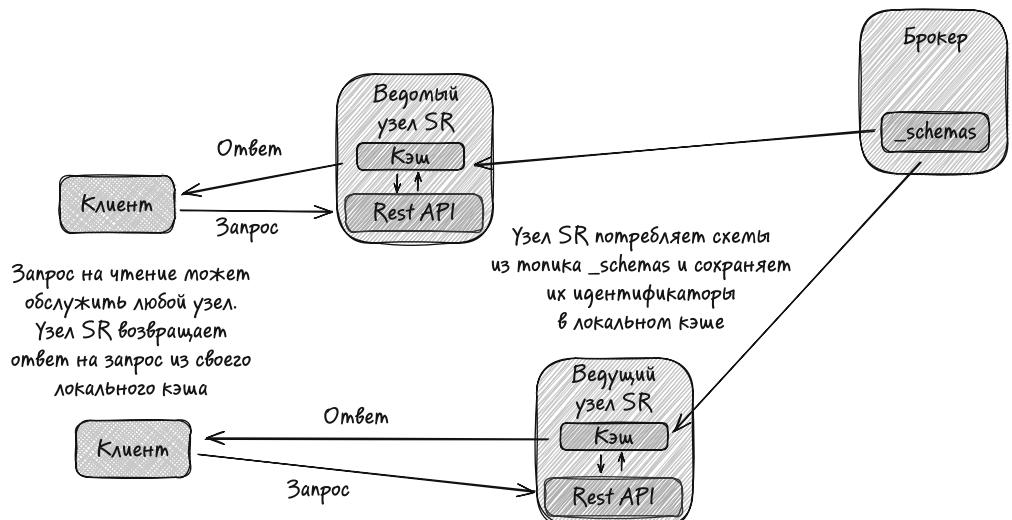
#### ПРИМЕЧАНИЕ

При работе с Schema Registry во всех примерах в книге используется окружение с единственным узлом, прекрасно подходящее для локальной разработки.



**Рис. 3.4.** Schema Registry — это распределенное приложение, в котором только ведущий узел взаимодействует с Kafka

При этом все узлы Schema Registry полноценно обслуживают запросы на чтение. Если какой-то ведомый узел получает запрос на регистрацию или обновление, они пересыпают его ведущему узлу и возвращают его ответ клиенту. Рассмотрим схему этой архитектуры на рис. 3.5, чтобы закрепить понимание особенностей ее работы.



**Рис. 3.5.** Запросы на чтение обслуживаются всеми узлами Schema Registry

Теперь, сделав краткий обзор архитектуры, приступим к работе и выполним несколько базовых команд Schema Registry с использованием REST API.

### 3.2.4. Использование Schema Registry REST API

Вы узнали, как должен работать Schema Registry в теории. Теперь пришло время увидеть, как он действует на практике. Для этого загрузим схему и выполним несколько дополнительных команд, чтобы получить информацию об этой схеме. На первых порах для выполнения команд мы используем инструменты командной строки curl и jq.

curl (<https://curl.se/>) — это утилита командной строки для работы с данными через URL-адреса. jq (<https://stedolan.github.io/jq/>) — это процессор JSON. Чтобы установить jq для вашей операционной системы, перейдите на страницу загрузки jq: <https://stedolan.github.io/jq/download/>. Утилита curl устанавливается по умолчанию в Windows 10+ и macOS, а в Linux ее можно установить с помощью менеджера пакетов. Если вы используете macOS, то обе утилиты сможете установить с помощью homebrew (<https://brew.sh/>).

Позднее для взаимодействия с Schema Registry мы будем использовать плагин Gradle. Получив представление о том, как работают различные вызовы REST API, мы используем плагины Gradle и несколько простых реализаций производителей и потребителей, чтобы увидеть, как действует сериализация.

Обычно для выполнения действий с Schema Registry вы будете использовать плагины инструмента сборки. Во-первых, они делают процесс разработки намного быстрее, чем выполнение запросов к API из командной строки, а во-вторых, они автоматически генерируют исходный код из схем. Использование плагинов инструмента сборки мы рассмотрим в подразделе 3.2.6.

#### ПРИМЕЧАНИЕ

Для работы с Schema Registry существуют плагины Maven и Gradle, но в примерах исходного кода для книги используется Gradle, поэтому вам понадобится именно этот плагин.

### 3.2.5. Регистрация схемы

Прежде чем начать, не забудьте запустить экземпляр Schema Registry командой docker-compose up -d. Сразу после запуска в реестре не будет зарегистрировано ни одной схемы, поэтому первым делом зарегистрируем схему. Сделаем пример чуть более забавным и создадим схему для супергероев из комикса Marvel «Мстители». Для создания первой схемы используем Avro, но давайте отвлечемся ненадолго, чтобы обсудить ее формат.

**Листинг 3.2.** Схема Avro для представления «Мстителей»

```
{"namespace": "bbejeck.chapter_3",  
 "type": "record",  
 "name": "Avenger",  
 "fields": [  
     {"name": "name", "type": "string"},  
     {"name": "real_name", "type": "string"},  
     {"name": "movies", "type": {"type": "array", "items": "string"}},  
     "default": []  
 ]}
```

Пространство имен уникально идентифицирует схему. Для генерированного кода на Java пространство имен — это имя пакета

Тип — запись, то есть сложный тип. Другими сложными типами являются перечисления, массивы, словари, объединения и фиксированные типы. Позже в этой главе мы еще поговорим о типах Avro

Описания отдельных полей. Поля в Avro бывают простыми или сложными

Значение по умолчанию. Avro использует значение по умолчанию при десериализации, если сериализованное представление не содержит этого поля

Здесь мы определили схему Avro в формате JSON. Этот файл схемы мы еще раз будем использовать в подразделе 3.2.6 при обсуждении применения плагина Gradle для генерации кода и взаимодействия с Schema Registry. Поскольку Schema Registry поддерживает форматы Protobuf и JSON Schema, рассмотрим, как определяется тот же тип объекта в этих форматах схем.

### Листинг 3.3. Схема Protobuf для представления «Мстителей»

```
syntax = "proto3";           ← Определение в формате Protobuf;
                             ← в этой книге используется версия 3

package bbejeck.chapter_3.proto; ← Объявление имени пакета
option java_multiple_files = true; ← Настраивает Protobuf для генерирования
                                   ← отдельных файлов для сообщения

message Avenger {             ← Определение сообщения
    string name = 1;           ← Уникальный номер поля
    string real_name = 2;
    repeated string movies = 3; ← Поле со спецификатором repeated
                                ← соответствует списку
}

```

Схема Protobuf выглядит ближе к обычному коду, так как этот формат отличается от формата JSON. Для идентификации полей в двоичном сообщении Protobuf использует числовые номера. В отличие от формата Avro, позволяющего задавать значения по умолчанию, в Protobuf (версии 3) каждое поле считается необязательным, но при этом значения по умолчанию для них не указываются, а определяются автоматически, исходя из их типов. Например, значение по умолчанию для числового поля — 0; для строк — пустая строка, а для полей со спецификатором `repeated` — пустой список.

### ПРИМЕЧАНИЕ

Protobuf — это обширная тема, однако, учитывая, что эта книга посвящена платформе потоковой передачи событий Kafka, я расскажу о спецификации Protobuf не больше, чем нужно, чтобы вы чувствовали себя более или менее комфортно при ее использовании. Полную информацию о ней можно получить в руководстве, доступном по адресу <http://mng.bz/5oB1>.

Теперь рассмотрим версию JSON Schema.

### Листинг 3.4. Схема JSON Schema для представления «Мстителей»

```
{
  "$schema": "http://json-schema.org/draft-07/schema#", ← Ссылка на спецификацию
  "title": "Avenger",                                     ← схемы
  "description": "A JSON schema of Avenger object",
  "type": "object",           ← Тип объекта
  "javaType": "bbejeck.chapter_3.json.SimpleAvengerJson", ← Java-тип, используемый
  "properties": {                                         ← для десериализации
    "name": {                                              ← Список полей объекта
      "type": "string"
    },
  }
}
```

```

    "realName": {
        "type": "string"
    },
    "movies": {
        "type": "array",
        "items": {
            "type": "string"
        },
        "default": [] ← Значение по умолчанию
    }
},
"required": [
    "name",
    "realName"
]
}

```

Схема в формате JSON Schema напоминает версию в формате Avro, так как оба формата используют нотацию JSON. Наиболее существенное различие между ними — в схеме JSON поля объекта перечисляются внутри элемента `properties`, а не в массиве `fields`, и сами поля объявляются простым указанием имени, то есть без элемента `name`.

### ПРИМЕЧАНИЕ

Обратите внимание на разницу между схемой в формате JSON и схемой в формате JSON Schema. JSON Schema — это «словарь, обеспечивающий согласованность, достоверность и совместимость данных JSON в масштабе». Как и в случае с Avro и Protobuf, я опишу формат JSON Schema лишь в той степени, чтобы вы могли использовать его в своих проектах, а за более подробной информацией обращайтесь на сайт проекта <https://json-schema.org/>.

Я показал разные форматы схем для сравнения. Но в остальной части главы я обычно буду показывать только одну версию каждой схемы, чтобы сэкономить место.

Теперь, познакомившись с форматами схем, продолжим и зарегистрируем схему в формате Avro. Запрос к REST API для регистрации схемы из командной строки показан в листинге 3.5.

### Листинг 3.5. Регистрация схемы из командной строки

```

jq '. | {schema: toJSON}' src/main/avro/avenger.avsc | \
curl -s -X POST http://localhost:8081/subjects/avro-avengers-value/versions \
-H "Content-Type: application/vnd.schemaregistry.v1+json" \
-d @- \
| jq

```

Annotations for the command line:

- Вызывается функция `jq toJSON` для форматирования файла `avenger.avsc` (переносы строк не допускаются в JSON) перед загрузкой, а затем результат передается команде `curl`**
- POST URL для добавления схемы; флаг `-s` подавляет вывод утилитой `curl` информации о ходе выполнения**
- Флаг `-d` указывает, что в запросе передаются данные, а `@-` означает чтение из STDIN (то есть данные, возвращаемые командой `jq`, которая предшествует команде `curl`)**
- Ответ JSON передается команде `jq`, чтобы получить красиво оформленный вывод**
- Заголовок, определяющий тип содержимого в запросе**

Результат выполнения этой команды должен выглядеть так, как показано в листинге 3.6.

**Листинг 3.6.** Ожидаемый ответ на попытку зарегистрировать схему

```
{  
  "id": 1  
}
```

В ответе на запрос POST возвращается идентификатор, назначенный новой схеме. Каждой новой схеме Schema Registry присваивает уникальный идентификатор (монотонно увеличивающееся число). Клиенты используют этот идентификатор для хранения схем в своем локальном кэше.

Прежде чем перейти к другой команде, я хочу обратить ваше внимание на листинг 3.5, в частности на имя субъекта схемы `subjects/avro-avengers-value/`. Schema Registry использует имя субъекта для управления видимостью изменений в схеме. В данном случае `avro-avengers-value` означает, что значения (в парах «ключ — значение»), входящие в топик `avro-avengers`, должны иметь формат зарегистрированной схемы. Подробнее об именах субъектов и их роли с точки зрения внесения изменений в схемы мы поговорим в разделе 3.3.

Теперь рассмотрим несколько команд для извлечения информации из Schema Registry. Представьте, что вы работаете над созданием нового приложения, работающего с Kafka. Вы слышали о Schema Registry и хотели бы посмотреть, как выглядит некоторая схема, разработанная одним из ваших коллег, но не можете вспомнить ее название, а так как сейчас выходные, то не хотите никого беспокоить своими вопросами. В таком случае вы можете перечислить все субъекты зарегистрированных схем, выполнив команду из листинга 3.7.

**Листинг 3.7.** Получение списка субъектов зарегистрированных схем

```
curl -s "http://localhost:8081/subjects" | jq
```

В ответ эта команда возвращает массив JSON со всеми субъектами. Поскольку мы зарегистрировали только одну схему, результат должен выглядеть так:

```
[  
  "avro-avengers-value"  
]
```

Отлично, мы нашли то, что искали, — схему, зарегистрированную для топика `avro-avengers`.

Теперь предположим, что в последней схеме произошли некоторые изменения и вам захотелось увидеть предыдущую версию. Проблема в том, что вы не знаете историю версий. Следующий запрос (листинг 3.8) вернет список всех версий для данной схемы.

**Листинг 3.8.** Получение всех версий заданной схемы

```
curl -s "http://localhost:8081/subjects/avro-avengers-value/versions" | jq
```

Эта команда вернет JSON-массив версий заданной схемы. В нашем случае результаты должны выглядеть так:

```
[  
  1  
]
```

Теперь, имея номер нужной версии, можно выполнить другую команду и получить схему определенной версии, как показано в листинге 3.9.

#### **Листинг 3.9.** Получение конкретной версии схемы

```
curl -s "http://localhost:8081/subjects/avro-avengers-value/versions/1"\  
| jq .'
```

Эта команда должна вывести примерно следующее:

```
{  
  "subject": "avro-avengers-value",  
  "version": 1,  
  "id": 1,  
  "schema": "{\"type\":\"record\", \"name\":\"AvengerAvro\",  
             \"namespace\":\"bbejeck.chapter_3.avro\", \"fields\""  
            :[{"name": "name", "type": "string"}, {"name"  
              :"real_name", "type": "string"}, {"name"  
                :"movies", "type": {"type": "array"  
                               , "items": "string"}, "default": []}]}"  
}
```

Значение поля `schema` форматируется как строка, поэтому кавычки экранируются, а все символы перевода строки удаляются. С помощью пары коротких команд мы смогли найти схему, определить историю версий и получить схему определенной версии.

Отмечу, что если вас не интересуют предыдущие версии схемы и нужна только последняя, то нет необходимости узнавать ее фактический номер. Можно просто использовать вызов REST API, показанный в листинге 3.10, чтобы получить последнюю версию схемы.

#### **Листинг 3.10.** Получение последней версии схемы

```
curl -s "http://localhost:8081/subjects/avro-avengers-value/  
versions/latest" | jq .'
```

Я не буду показывать здесь результат выполнения этой команды, так как он идентичен результату предыдущей команды.

Это был краткий обзор некоторых команд, доступных в Schema Registry REST API. Полный список доступных команд и их описание вы найдете, перейдя по ссылке <http://mng.bz/OZEo>.

Далее для работы со схемами Avro, Protobuf и JSON Schema в Schema Registry и мы будем использовать плагины Gradle.

### 3.2.6. Плагины и инструменты сериализации

К настоящему моменту вы узнали, что объекты событий, отправляемые производителями и получаемые потребителями, представляют собой контракт между клиентами-производителями и клиентами-потребителями, что этот контракт можно выразить явно в форме конкретной схемы и что для хранения схем и передачи их клиентам-производителям и клиентам-потребителям, чтобы те могли правильно сериализовать и десериализовать записи, можно использовать Schema Registry.

В следующих разделах вы познакомитесь с некоторыми другими возможностями Schema Registry, такими как тестирование схем на совместимость, поддержка разных режимов совместимости и процесса относительно безболезненного для клиентов изменения и развития схем.

До сих пор мы работали только с файлом схемы, и потому их применение все еще выглядит для вас немного абстрактно. Как упоминалось выше в этой главе, в своих приложениях разработчики работают с объектами. Поэтому далее мы посмотрим, как преобразовать файлы схем в конкретные объекты, которые можно использовать в приложении.

Schema Registry поддерживает схемы в форматах Avro, Protobuf и JSON Schema. Avro и Protobuf — это платформы сериализации, предоставляющие инструменты для работы со схемами в соответствующих форматах. Одним из самых важных инструментов является возможность генерировать объекты из схем.

Поскольку JSON Schema — это стандарт, а не библиотека или платформа, нам понадобится инструмент, который будет генерировать код. В этой книге мы будем использовать проект <https://github.com/eirnym/js2p-gradle>. Для (дес)сериализации без Schema Registry я рекомендую использовать ObjectMapper из проекта <https://github.com/FasterXML/jackson-databind>.

Генерация кода из схемы облегчит вашу жизнь как разработчика, автоматизировав повторяющийся шаблонный процесс создания предметных объектов. Кроме того, хранение схемы в системе управления исходным кодом (в нашем случае Git) поможет существенно уменьшить вероятность ошибки, например сгенерировать строковое поле вместо поля типа long при создании предметных объектов, и даст возможность быстро распространить внесенные изменения между разработчиками.

Для управления исходным кодом примеров в этой книге используется инструмент сборки Gradle (<https://gradle.org/>). К счастью, в Gradle есть плагины для работы с Schema Registry, Avro, Protobuf и JSON Schema. В частности, мы будем использовать следующие плагины:

- <https://github.com/ImFlog/schema-registry-plugin> — для взаимодействия с Schema Registry (то есть регистрации, тестирования и настройки совместимости схем);
- <https://github.com/davidmc24/gradle-avro-plugin> — для генерации кода на Java из файлов схемы Avro (.avsc);
- <https://github.com/google/protobuf-gradle-plugin> — для генерации кода на Java из файлов схемы Protobuf (.proto);
- <https://github.com/eirnym/js2p-gradle> — для генерации кода на Java из схем, оформленных с использованием спецификации JSON Schema.

## ПРИМЕЧАНИЕ

Важно отметить различие между файлами схем в формате JSON, такими как схемы Avro и JSON Schema (<https://json-schema.org/>). Файлы Avro содержат код JSON, но следуют спецификации Avro. Файлы JSON Schema следуют официальной спецификации для JSON Schema.

При использовании плагинов Gradle для поддержки Avro, Protobuf и JSON Schema вам не придется учиться пользоваться отдельными инструментами для каждого компонента, плагины сами выполняют всю работу. Плагин Gradle мы также будем использовать для обработки большинства взаимодействий с Schema Registry. Начнем с выгрузки схемы с помощью консольной команды Gradle вместо запроса к REST API.

### 3.2.7. Выгрузка файла схемы

Первым делом зарегистрируем схему с помощью Gradle. Мы будем использовать ту же схему Avro, что использовали выше. Чтобы выгрузить схему, обязательно измените текущий каталог (`cd`), перейдя в корневой каталог проекта, и выполните эту команду Gradle:

```
./gradlew streams:registerSchemasTask
```

После запуска этой команды в консоли должен появиться текст **BUILD SUCCESSFUL** (Сборка выполнена успешно). Обратите внимание, что в командной строке достаточно ввести только имя задачи Gradle (из плагина поддержки Schema Registry), а задача зарегистрирует все схемы, перечисленные внутри блока `register { }` в файле `streams/build.gradle`.

Теперь рассмотрим конфигурацию плагина поддержки Schema Registry в файле `streams/build.gradle` (листинг 3.11).

**Листинг 3.11.** Конфигурация плагина поддержки Schema Registry в файле `streams/build.gradle`

```
schemaRegistry { ← Начало блока конфигурации
    Schema Registry в файле build.gradle
    url = 'http://localhost:8081' ← URL для подключения
                                    к Schema Registry
    register {
        subject('avro-avengers-value', ← Регистрирует схему
            'src/main/avro/avenger.avsc', ← по имени субъекта
            'AVRO') ← Тип регистрируемой схемы
        // другие элементы опущены для простоты
    }
    // другие настройки опущены для простоты
}
```

В блоке `register` указывается та же информация, что и в запросе к REST API, только в форме вызова метода, а не URL. Внутри плагина использует все тот же Schema Registry REST API через `SchemaRegistryClient`. Обратите внимание, что

в исходном коде примера в блоке `register` несколько записей. Мы будем использовать их все при изучении примеров далее.

Чуть позже мы рассмотрим другие задачи Gradle Schema Registry, а пока попробуем сгенерировать код из схемы.

### 3.2.8. Генерация кода из схем

Как я уже говорил, одним из важных преимуществ платформ Avro и Protobuf является наличие инструментов генерации кода. Использование плагина Gradle немного повышает удобство, позволяя абстрагироваться от деталей использования отдельных инструментов. Чтобы сгенерировать объекты, представленные схемами, нужно просто запустить задачу Gradle, как показано в листинге 3.12.

#### Листинг 3.12. Генерирование объекта модели

```
./gradlew clean build
```

Эта команда сгенерирует код на Java для всех схем, перечисленных в проекте, — Avro, Protobuf и JSON Schema. Теперь поговорим о том, где в проекте следует размещать схемы. По умолчанию схемы Avro и Protobuf сохраняются в каталогах `src/main/avro` и `src/main/proto` соответственно. Местом хранения по умолчанию для схем JSON Schema является каталог `src/main/json`, но его необходимо явно настроить в файле `build.gradle`, как показано в листинге 3.13.

#### Листинг 3.13. Настройка каталога со схемами JSON Schema

```
jsonSchema2Pojo {  
    source = files("${project.projectDir}/src/main/json") // Конфигурационный параметр source определяет,  
    targetDirectory = file("${project.buildDir}/generated-main-json-java") // где инструменты генерации могут найти схемы  
    // другие конфигурации опущены для простоты  
}
```

Параметр `targetDirectory` определяет место, куда инструмент записывает сгенерированные объекты Java

#### ПРИМЕЧАНИЕ

Все приведенные примеры относятся к схемам в подкаталоге `streams`, если явно не указано иное.

В листинге 3.13 показана конфигурация входных и выходных каталогов для плагина `js2p-gradle`. Плагин Avro по умолчанию помещает сгенерированные файлы в подкаталог `generated-main-avro-java` в каталоге `build`. Для Protobuf выходной каталог настраивается по аналогии с шаблоном JSON Schema и Avro в блоке `protobuf` в файле `build.gradle`, как показано в листинге 3.14.

#### Листинг 3.14. Настройка каталога вывода для Protobuf

```
protobuf {  
    protoc {  
        artifact = 'com.google.protobuf:protoc:3.25.0' // Местоположение компилятора protoc  
    }  
}
```

Я хотел коротко обсудить аннотацию в листинге 3.14. Для использования Protobuf необходимо установить компилятор `protoc`. По умолчанию плагин ищет выполняемый файл `protoc`. Но вообще можно использовать предварительно скомпилированную версию `protoc` из Maven Central, что избавляет от необходимости устанавливать компилятор. Но если вы предпочитаете использовать локальную установку, то укажите путь в блоке `protoc` в формате `path = путь/к/компилятору/protoc`.

Итак, мы сгенерировали код из схем. Теперь запустим полный пример.

### 3.2.9. Полный пример

Соберем вместе все, что вы узнали, и запустим полный пример. К настоящему моменту мы уже зарегистрировали схемы и сгенерировали необходимые файлы с кодом на Java. Поэтому далее выполним следующие шаги.

1. Создадим несколько предметных объектов из сгенерированных файлов Java.
2. Отправим их в соответствующий топик Kafka.
3. Извлечем только что отправленные объекты из того же топика Kafka.

Шаги 2 и 3 больше связаны с клиентами, чем с Schema Registry, тем не менее мы рассмотрим их именно с этой точки зрения. Мы будем создавать экземпляры объектов Java, полученных из файлов схемы, поэтому обратите внимание на поля и как объекты соответствуют структуре схемы. Также сосредоточьтесь на элементах конфигурации, определяющих сериализатор и десериализатор, а также URL для связи с Schema Registry.

#### ПРИМЕЧАНИЕ

В этом примере используются производитель и потребитель Kafka, но я не буду раскрывать подробности работы с ними, так как мы только знакомимся с клиентами-производителями и клиентами-потребителями. Более подробно о производителях и потребителях я расскажу в следующей главе, а пока просто пройдемся по примерам как есть.

Если вы еще не зарегистрировали файлы схемы и не сгенерировали код на Java, то сделаем это сейчас (листинг 3.15). Но перед этим не забудьте запустить брокер Kafka и Schema Registry, выполнив команду `docker-compose up -d`.

#### Листинг 3.15. Регистрация схем и генерирование файлов с кодом на Java

```
./gradlew streams:registerSchemasTask ← Зарегистрировать файлы схем  
./gradlew clean build ← Генерировать Java-объекты на основе схем
```

Теперь сосредоточимся на конфигурации Schema Registry. Перейдите в папку с исходным кодом примеров и посмотрите на определение класса `bvejeck.chapter_3.producer.BaseProducer`. Сейчас нас интересуют только следующие две конфигурации; другие конфигурации, касающиеся производителя, мы рассмотрим в следующей главе:

```
producerProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,  
    keySerializer); ← Задает используемый сериализатор  
producerProps.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,  
    "http://localhost:8081"); ← Определяет местоположение Schema Registry
```

Первая конфигурация определяет, какой сериализатор будет использовать производитель. Не забывайте, что `KafkaProducer` отделен от типа `Serializer`; он просто вызывает метод `serialize` и получает обратно массив байтов для отправки. Поэтому вы несете всю ответственность за предоставление правильного класса, реализующего интерфейс `Serializer`.

В данном случае мы будем работать с объектами, сгенерированными из схемы Avro, поэтому используем `KafkaAvroSerializer`. Если заглянуть в определение класса `bbejeck.chapter_3.producer.avro.AvroProducer` (который расширяет класс `BaseProducer`), то увидите, что он передает `KafkaAvroSerializer.class` конструктору родительского объекта. Вторая конфигурация задает конечную точку HTTP, которую сериализатор использует для связи с Schema Registry. Эти конфигурации позволяют выполнять взаимодействия, изображенные на рис. 3.3.

Теперь кратко рассмотрим создание объекта (листинг 3.16).

**Листинг 3.16.** Создание экземпляра объекта с использованием сгенерированного кода

```
var blackWidow = AvengerAvro.newBuilder()
    .setName("Black Widow")
    .setRealName("Natasha Romanova")
    .setMovies(List.of("Avengers", "Infinity Wars",
        "End Game")).build();
```

Возможно, вы уже подумали: «Ну хорошо, этот код создает объект, а в чем, собственно, проблема?» То, что я пытаюсь донести, может показаться незначительным, тем не менее имейте в виду, что вы можете заполнить ожидаемые поля только значениями правильных типов, как определено контрактом на создание записей в желаемом формате. При необходимости вы можете обновить схему и повторно сгенерировать код.

Но при внесении изменений вам придется зарегистрировать новую схему, а изменения должны соответствовать текущему формату совместимости для имени субъекта. Теперь вы можете видеть, как Schema Registry обеспечивает соблюдение «контракта» между производителями и потребителями. Режимы совместимости и разрешенные изменения мы рассмотрим в разделе 3.4.

Выполним следующую команду Gradle, чтобы создать объекты для топика `avro-avengers` (листинг 3.17).

**Листинг 3.17.** Запуск AvroProducer

```
./gradlew streams:runAvroProducer
```

После выполнения этой команды вы увидите примерно такой вывод:

```
DEBUG [main] bbejeck.chapter_3.producer.BaseProducer - Producing records
[{"name": "Black Widow", "real_name": "Natasha Romanova", "movies": ["Avengers", "Infinity Wars", "End Game"]}, {"name": "Hulk", "real_name": "Dr. Bruce Banner", "movies": ["Avengers", "Ragnarok", "Infinity Wars"]}, {"name": "Thor", "real_name": "Thor", "movies": ["Dark Universe", "Ragnarok", "Avengers"]}]
```

Произведя несколько записей, приложение завершит работу.

### ПРИМЕЧАНИЕ

Вводите команду точно так, как показано здесь, включая предшествующий символ :. В примерах работы с Schema Registry используются три разных модуля Gradle. Мы должны гарантировать, что запускаемые команды предназначены для определенного модуля. В данном случае : запускает только основной модуль. Без этого символа запустится производитель для всех модулей и пример завершится ошибкой.

Эта команда не делает ничего особенного, зато наглядно показывает, насколько просто осуществляется сериализация с использованием Schema Registry. Производитель извлекает схему, сохраняет ее локально и отправляет записи в Kafka в правильном сериализованном формате — и все это без необходимости писать какой-либо код сериализации или конструировать предметные модели. Поздравляю, вы отправили сериализованные записи в Kafka!

### СОВЕТ

Часто полезно заглянуть в файл журнала, сгенерированный при запуске этой команды. Он находится в каталоге `streams/logs/` в папке с примерами исходного кода для книги. Конфигурация `log4j` перезаписывает файл журнала при каждом запуске, поэтому загляните в него перед выполнением следующего шага.

Теперь запустим потребитель (листинг 3.18), который извлекает и десериализует записи. Как и в случае с производителем, мы сосредоточимся исключительно на конфигурации, необходимой для десериализации и работы с Schema Registry.

**Листинг 3.18.** Конфигурация потребителя, извлекающего записи в формате Avro

```
consumerProps.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,  
    KafkaAvroDeserializer.class); ← Использовать десериализатор Avro  
consumerProps.put(KafkaAvroDeserializerConfig.SPECIFIC_AVRO_READER_CONFIG,  
    true); ← Конфигурация для использования SpecificAvroReader  
consumerProps.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,  
    "http://localhost:8081"); ← Адрес Schema Registry  
                                в формате host:port
```

Обратите внимание на параметр `SPECIFIC_AVRO_READER_CONFIG` со значением `true`. Какую роль он играет? Чтобы ответить на этот вопрос, коротко обсудим работу с сериализованными объектами Avro, Protobuf и JSON Schema.

При десериализации одного из объектов — Avro, Protobuf или JSON Schema — десериализуется конкретный тип объекта или неспецифический объект-контейнер. Например, если параметр `SPECIFIC_AVRO_READER_CONFIG` имеет значение `true`, то десериализатор внутри потребителя вернет объект конкретного типа `AvroAvenger`. Однако если параметру `SPECIFIC_AVRO_READER_CONFIG` присвоить значение `false`, то десериализатор вернет объект типа `GenericRecord`, который по-прежнему следует той же схеме и имеет то же содержимое, но сам по себе лишен какой-либо

информации о типе. Как следует из его имени, это просто обобщенный контейнер для полей. Пример в листинге 3.19 поясняет то, о чём я говорю.

#### Листинг 3.19. Записи конкретного типа и записи типа GenericRecord

```
AvroAvenger avenger = // Вернет потребитель
                      // с SPECIFIC_AVRO_READER_CONFIG=true
avenger.getName();
avenger.getRealName();   ← Доступ к полям объекта
avenger.getMovies();

GenericRecord genericRecord = // Вернет потребитель
                             // с SPECIFIC_AVRO_READER_CONFIG=false
if (genericRecord.hasField("name")) {
    genericRecord.get("name");
}

if (genericRecord.hasField("real_name")) { ← Доступ к полям объекта
    genericRecord.get("real_name");
}

if (GenericRecord.hasField("movies")) {
    genericRecord.get("movies");
}
```

Этот простой пример кода показывает различия между конкретным и универсальным типом возвращаемого значения. Имея объект `AvroAvenger`, можно напрямую обращаться к его свойствам, поскольку объект «знает», как он устроен, и предоставляет методы для доступа к этим полям. Но при работе с объектом `GenericRecord` сначала нужно запросить его, содержит ли в нем конкретное поле, прежде чем пытаться обращаться к нему.

#### ПРИМЕЧАНИЕ

Конкретная версия схемы Avro — это не просто POJO (plain old Java object — старый добрый Java-объект), она расширяет класс `SpecificRecordBase`.

Обратите внимание, что при работе с `GenericRecord` имя поля должно указываться точно так же, как оно определено в схеме, тогда как конкретная версия предлагает более привычную «верблюжью» нотацию. Разница между ними в том, что при работе с конкретным типом вы знаете структуру объекта, тогда как универсальный тип может представлять любой произвольный объект, и потому вы должны запросить его поля, чтобы определить структуру. Работа с `GenericRecord` во многом напоминает работу с `HashMap`.

Однако не обязательно работать вслепую. Вызовом `GenericRecord.getSchema().getFields()` можно получить список полей, а затем выполнить цикл по списку объектов `Field` и получить имена отдельных полей вызовом `Fields.name()`. Аналогично вызовом `GenericRecord.getSchema().getFullName()` можете получить имя схемы. Обновление значения поля производится похожим образом, как показано в листинге 3.20.

**Листинг 3.20.** Изменение значения поля в записи конкретного и универсального типа

```
avenger.setRealName("updated name")
genericRecord.put("real_name", "updated name")
```

Итак, этот небольшой пример показал нам, что объект конкретного типа поддерживает знакомую функциональность методов записи (сеттеров). А при использовании объекта универсального типа нужно явно объявить имя изменяемого поля. И снова такое поведение универсальной версии может напомнить вам поведение `HashMap`.

Protobuf предоставляет похожую функциональность для работы с конкретными и произвольными типами. Для работы с произвольным типом в Protobuf используется класс `DynamicMessage`. Как и в случае с `GenericRecord` в Avro, `DynamicMessage` поддерживает функции для определения типа записи и составляющих ее полей. В JSON Schema конкретные типы — это просто объекты, сгенерированные плагином Gradle. Они не связаны с кодом поддержки, как в Avro или Protobuf. Универсальная версия — это тип `JsonNode`, поскольку для сериализации и десериализации используется библиотека `jackson-databind` (<https://github.com/FasterXML/jackson-databind>).

### ПРИМЕЧАНИЕ

Исходный код для этой главы содержит примеры работы с конкретными и универсальными типами Avro, Protobuf и JSON Schema.

Итак, когда лучше использовать конкретный тип, а когда универсальный? Конкретную версию можно использовать, если через топик Kafka передаются записи только одного типа. Но если в топике имеется несколько типов событий, то придется использовать универсальную версию, поскольку каждая потребляемая запись может оказаться другого типа. Мы обсудим передачу событий нескольких типов через один топик позже в этой главе и затем еще раз в главе 4, когда будем рассматривать клиенты Kafka.

Наконец, запомните, что для использования конкретного типа нужно обязательно присвоить параметру `KafkaAvroDeserializerConfig.SPECIFIC_AVRO_READER_CONFIG` значение `true`. По умолчанию этот параметр получает значение `false`, поэтому потребитель возвращает тип `GenericRecord`, если конфигурация не задана явно.

Завершив обсуждение различных типов записей, продолжим исследование нашего первого полного примера использования Schema Registry. Мы уже создали несколько записей, применяя схему, которую выгрузили ранее. Теперь нужно запустить потребитель и посмотреть, как происходит десериализация записей с помощью схемы. И снова довольно поучительным может оказаться просмотр файла журнала. В нем вы увидите, что встроенный десериализатор загружает схему только для первой записи, а затем сохраняет ее в локальном кэше.

Я также должен отметить, что следующий пример `bbejeck.chapter_3.consumer.avro.AvroConsumer` использует как конкретный тип, так и тип `GenericRecord`. В процессе выполнения код примера выводит тип потребляемой записи.

### ПРИМЕЧАНИЕ

Исходный код содержит аналогичные примеры для Protobuf и JSON Schema.

Запустим пример потребителя, выполнив команду из листинга 3.21 в корневой папке с примерами исходного кода для книги.

#### Листинг 3.21. Запуск AvroConsumer

```
./gradlew streams:runAvroConsumer
```

#### ПРИМЕЧАНИЕ

И снова напомню, что вы должны вводить команду точно так, как показано здесь, включая предшествующий символ :, в противном случае будет запущен потребитель для всех модулей и пример завершится ошибкой.

*AvroConsumer* выводит полученные записи и завершает работу. Поздравляю, вы только что сериализовали и десериализовали записи с помощью Schema Registry!

К настоящему моменту мы рассмотрели фреймворки сериализации, поддерживаемые Schema Registry, вы узнали, как оформить и выгрузить файл схемы, и увидели простой пример использования схемы. Выше в главе я упомянул термин «субъект» (subject) и отметил, что он определяет область видимости эволюционирующей схемы. В следующем разделе я расскажу, как использовать различные стратегии именования субъектов.

## 3.3. СТРАТЕГИИ ИМЕНОВАНИЯ СУБЪЕКТОВ

Для управления областью видимости изменений в схемах Schema Registry использует концепцию субъекта. Субъект можно рассматривать как пространство имен конкретной схемы. Другими словами, с развитием бизнес-требований вам придется изменить файлы схемы, чтобы внести поправки в предметные объекты. Например, представьте, что мы решили удалить из нашего предметного объекта *AvroAvenger* настоящее (гражданское) имя героя и добавить список его полномочий.

Для поиска существующей схемы и сравнения изменений с новой схемой Schema Registry использует субъект. Эта проверка выполняется, чтобы гарантировать соответствие изменений текущему набору режимов совместимости. Подробнее о режимах совместимости мы поговорим в разделе 3.4. Стратегия именования субъектов определяет область видимости, в которой Schema Registry выполняет проверки совместимости.

Существует три типа стратегий: *TopicNameStrategy*, *RecordNameStrategy* и *TopicRecordNameStrategy*. Уже по именам стратегий можно догадаться о сфере действия соответствующих пространств имен, однако не будем торопиться и рассмотрим некоторые детали. Итак, обсудим эти различные стратегии.

#### ПРИМЕЧАНИЕ

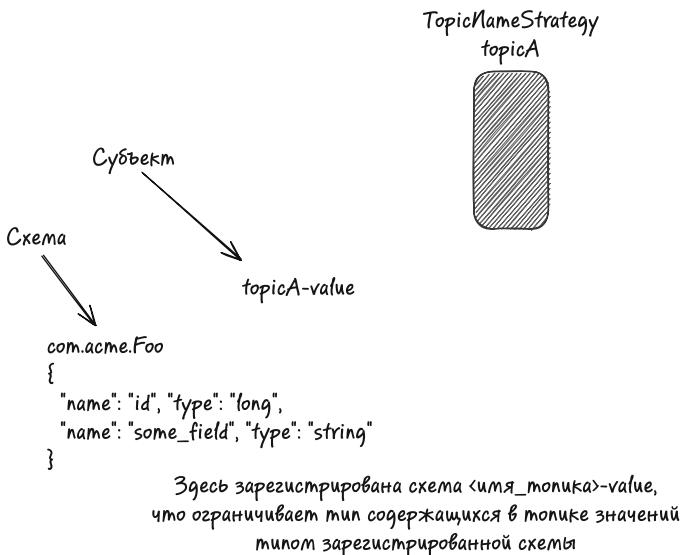
По умолчанию все сериализаторы пытаются зарегистрировать схему при сериализации, если не найдут соответствующий идентификатор у себя в локальном кэше. Автоматическая регистрация здорово помогает при разработке, но в промышленном окружении ее может потребоваться отключить в конфигурации производителя, указав параметр *auto.register.schemas=false*. Другой пример, когда автоматическая регистрация нежелательна, — использование схемы объединения Avro со ссылками. Мы рассмотрим эту тему более подробно далее в этой главе.

### 3.3.1. TopicNameStrategy

**TopicNameStrategy** — это субъект по умолчанию в Schema Registry. Имя субъекта исходит из имени топика. Мы уже видели **TopicNameStrategy** в действии выше в этой главе, когда регистрировали схему с помощью плагина Gradle. Если говорить точнее, то имя субъекта — это **<имя\_топика>-key** или **<имя\_топика>-value**, поскольку могут быть разные типы для ключа и значения, требующие разных схем.

**TopicNameStrategy** гарантирует наличие в топике только одного типа данных, поскольку невозможно зарегистрировать схему для другого типа с тем же именем топика. Наличие в каждом топике единственного типа удобно во многих случаях, например, если топику присваивается имя исходя из типа хранимых в нем событий, то есть такие топики будут хранить только один тип записей.

Еще одно преимущество **TopicNameStrategy** — ограничивая применение схемы одним топиком, можно создать другой топик, использующий тот же тип записей, но с другой схемой (рис. 3.6). Рассмотрим ситуацию, когда два отдела используют один и тот же тип записей, но разные названия топиков. Применяя **TopicNameStrategy**, эти отделы могут зарегистрировать совершенно разные схемы для одного и того же типа записи, поскольку область видимости схемы ограничена определенным топиком.



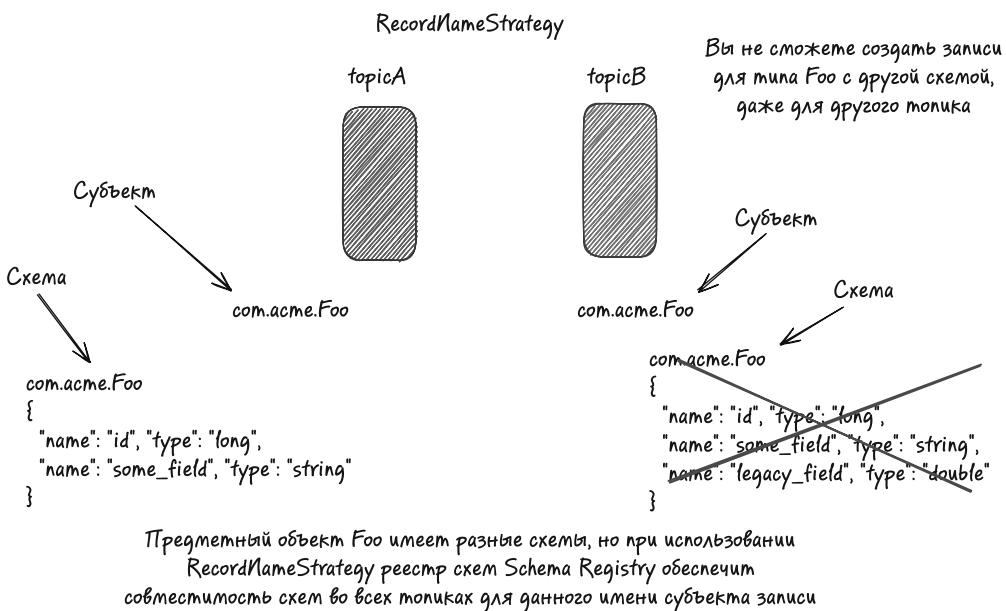
**Рис. 3.6.** **TopicNameStrategy** гарантирует использование для значения и/или ключа одного и того же типа предметного объекта, представленного зарегистрированной схемой

Поскольку **TopicNameStrategy** является значением по умолчанию, то нет необходимости явно определять какие-либо дополнительные конфигурационные параметры. При регистрации схем вы будете использовать формат **<топик>-value** в качестве субъекта для схем значений и **<топик>-key** в качестве субъекта для схем ключей. В обоих случаях на место токена **<топик>** вы должны подставить фактическое имя топика.

Но иногда могут иметься настолько тесно связанные события, что было бы желательно объединить их в один топик. В таком случае нужно выбрать стратегию, которая допускает различные типы и схемы в топике.

### 3.3.2. RecordNameStrategy

RecordNameStrategy (рис. 3.7) использует в качестве имени субъекта полное имя класса (представления Java-объекта схемы). Благодаря данной стратегии можно иметь в одном топике несколько типов записей. Но самое важное, что эти записи логически связаны друг с другом, хотя и отличаются физическими макетами.



**Рис. 3.7.** RecordNameStrategy позволяет использовать одну и ту же схему предметного объекта в разных топиках

Когда следует выбирать RecordNameStrategy? Представьте, что у вас дома установлены разные датчики, подключенные к Интернету вещей (Internet of Things, IoT). Разные датчики измеряют разные параметры, поэтому посылаемые ими записи будут иметь разную структуру, но вам хотелось бы поместить их в один топик.

Поскольку типы могут быть разными, проверки совместимости происходят между схемами с тем же именем записи. Кроме того, проверка совместимости распространяется на все топики, где используется субъект с тем же именем записи.

Стратегия RecordNameStrategy требует использовать в качестве имени субъекта полное имя класса при регистрации схемы для данного типа записи. Для объекта AvengerAvro, например, регистрация схемы должна выполняться так, как показано в листинге 3.22.

**Листинг 3.22.** Настройка плагина Gradle поддержки Schema Registry для применения RecordNameStrategy

```
subject('bbejeck.chapter_3.avro.AvengerAvro', 'src/main/avro/avenger.avsc', 'AVRO')
```

Затем необходимо настроить применение стратегии именования субъекта в производителе (листинг 3.23) и потребителе (листинг 3.24).

**Листинг 3.23.** Настройка производителя для применения RecordNameStrategy

```
Map<String, Object> producerConfig = new HashMap<>();
producerConfig.put(KafkaAvroSerializerConfig.VALUE_SUBJECT_NAME_STRATEGY,
RecordNameStrategy.class);
producerConfig.put(KafkaAvroSerializerConfig.KEY_SUBJECT_NAME_STRATEGY,
RecordNameStrategy.class);
```

**Листинг 3.24.** Настройка потребителя для применения RecordNameStrategy

```
Map<String, Object> consumerConfig = new HashMap<>();
config.put(KafkaAvroDeserializerConfig.KEY_SUBJECT_NAME_STRATEGY,
RecordNameStrategy.class);
config.put(KafkaAvroDeserializerConfig.VALUE_SUBJECT_NAME_STRATEGY,
RecordNameStrategy.class);
```

### ПРИМЕЧАНИЕ

Если Avro используется только для сериализации/десериализации значений, то нет необходимости добавлять конфигурацию для ключа. Кроме того, стратегии именования субъекта для ключей и значений не обязательно должны совпадать, я просто представил их здесь совпадающими.

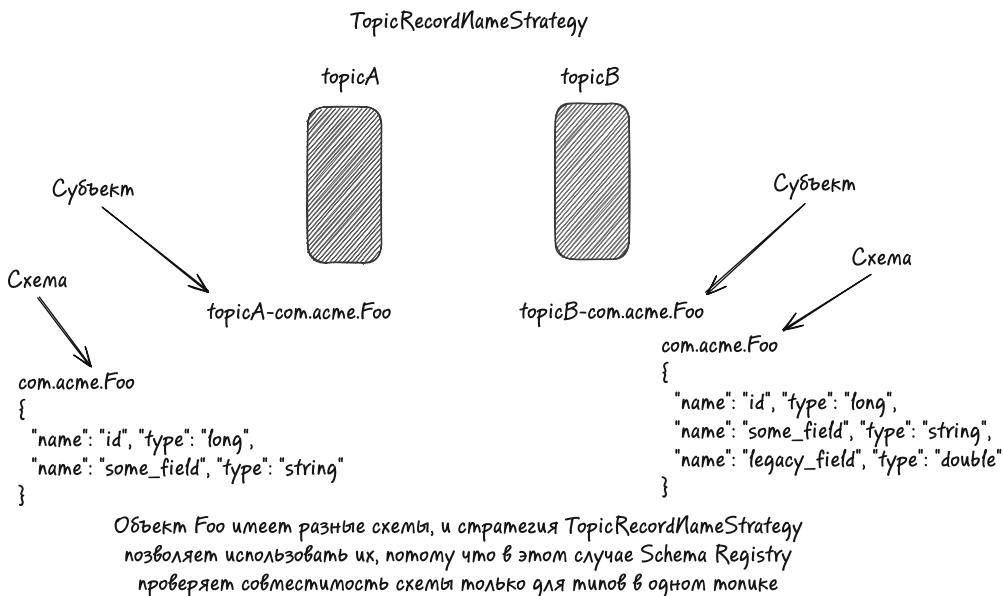
Для Protobuf используйте KafkaProtobufSerializerConfig и KafkaProtobufDeserializerConfig, а для JSON Schema используйте KafkaJsonSchemaSerializerConfig и KafkaJsonSchemaDeserializerConfig. Эти конфигурации влияют только на взаимодействие сериализатора/десериализатора с Schema Registry для поиска схем. Напомню еще раз, что сериализация отделена от процессов производства и потребления.

Помните, что при использовании только имени записи все топики должны применять одну и ту же схему. Чтобы использовать разные записи в топике и при этом учитывать только схемы для этого конкретного топика, необходимо применить другую стратегию.

### 3.3.3. TopicRecordNameStrategy

Как можно понять из названия, эта стратегия тоже позволяет иметь в одном топике записи разных типов. Однако зарегистрированные схемы для данной записи действуют только в рамках текущего топика. Взгляните на рис. 3.8, поясняющий суть.

Как показано на рис. 3.8, топик `topicA` может иметь схему для типа записи `Foo`, отличную от имеющейся в топике `topicB`. Эта стратегия позволяет иметь несколько логически связанных типов в одном топике, но они будут изолированы от других топиков, где имеется тот же тип, но используются другие схемы.



**Рис. 3.8.** TopicRecordNameStrategy допускает использование разных схем для одного и того же предметного объекта в разных топиках

В каких случаях может пригодиться TopicRecordNameStrategy? Представьте такую ситуацию: у вас есть одна версия объекта события `CustomerPurchaseEvent` в топике `interactions`, которая объединяет все типы событий клиентов (`CustomerSearchEvent`, `CustomerLoginEvent` и т. д.), и есть старый топик, `purchases`, который тоже содержит объекты `CustomerPurchaseEvent`. Однако старый топик предназначен для устаревшей системы, поэтому схема старая и содержит поля, отличные от новой. TopicRecordNameStrategy позволяет этим двум топикам содержать один и тот же тип, но с разными версиями схемы. Подобно RecordNameStrategy, эту стратегию нужно настроить, как показано в листинге 3.25.

#### Листинг 3.25. Настройка плагина Gradle поддержки Schema Registry для применения TopicRecordNameStrategy

```
subject('avro-avengers-bbejeck.chapter_3.avro.AvengerAvro',
'src/main/avro/avenger.avsc', 'AVRO')
```

Затем необходимо настроить применение стратегии именования субъекта в производителе (листинг 3.26) и потребителе (листинг 3.27).

#### Листинг 3.26. Настройка производителя для применения TopicRecordNameStrategy

```
Map<String, Object> producerConfig = new HashMap<>();
producerConfig.put(KafkaAvroSerializerConfig.VALUE_SUBJECT_NAME_STRATEGY,
TopicRecordNameStrategy.class);
producerConfig.put(KafkaAvroSerializerConfig.KEY_SUBJECT_NAME_STRATEGY,
TopicRecordNameStrategy.class);
```

**Листинг 3.27.** Настройка потребителя для применения TopicRecordNameStrategy

```
Map<String, Object> consumerConfig = new HashMap<>();
config.put(KafkaAvroDeserializerConfig.KEY_SUBJECT_NAME_STRATEGY,
TopicRecordNameStrategy.class);
config.put(KafkaAvroDeserializerConfig.VALUE_SUBJECT_NAME_STRATEGY,
TopicRecordNameStrategy.class);
```

**ПРИМЕЧАНИЕ**

Предостережение о регистрации стратегии для ключа, которое я дал выше, применимо и в этом случае. Это следует делать, только если для ключа используется отдельная схема. В этом примере я привел ее определение исключительно для полноты картины. Кроме того, стратегии именования субъектов для ключей и значений не обязательно должны совпадать.

Когда следует использовать TopicRecordNameStrategy вместо TopicNameStrategy или RecordNameStrategy? Если нужна возможность передавать в топик события нескольких типов и использовать разные версии схемы для данного типа в разных топиках.

Но ни одна из стратегий, TopicRecordNameStrategy и RecordNameStrategy, допускающих использование нескольких типов в топике, не могут ограничить топик некоторым фиксированным набором типов. Использование любой из этих стратегий именования субъектов позволяет иметь в топике неограниченное число различных типов. В разделе 3.5, где мы будем рассматривать ссылки на схемы, вы узнаете, как исправить эту ситуацию.

В табл. 3.1 приводится краткий обзор различных стратегий именования субъектов. Представьте себе стратегию именования субъектов как функцию, которая принимает аргументы с именем топика и схемой записи и возвращает имя субъекта. TopicNameStrategy использует только имя топика и игнорирует схему записи. RecordNameStrategy, наоборот, игнорирует имя топика и использует только схему записи. А TopicRecordNameStrategy использует и имя топика, и схему.

**Таблица 3.1.** Обзор стратегий именования субъектов

Стратегия	Несколько типов в топике	Разные версии схем объектов в разных топиках
TopicNameStrategy	Возможно	Да
RecordNameStrategy	Да	Нет
TopicRecordNameStrategy	Да	Да

К настоящему моменту мы рассмотрели стратегии именования субъектов и узнали, как Schema Registry использует субъекты для ограничения области видимости схем. Но управление схемами имеет еще одно измерение: развитие изменений внутри самой схемы. Как учесть такие изменения, как удаление или добавление поля? Желательна ли прямая или обратная совместимость для ваших клиентов? Подробнее о совместимости схем рассказывается в следующем разделе.

## 3.4. СОВМЕСТИМОСТЬ СХЕМ

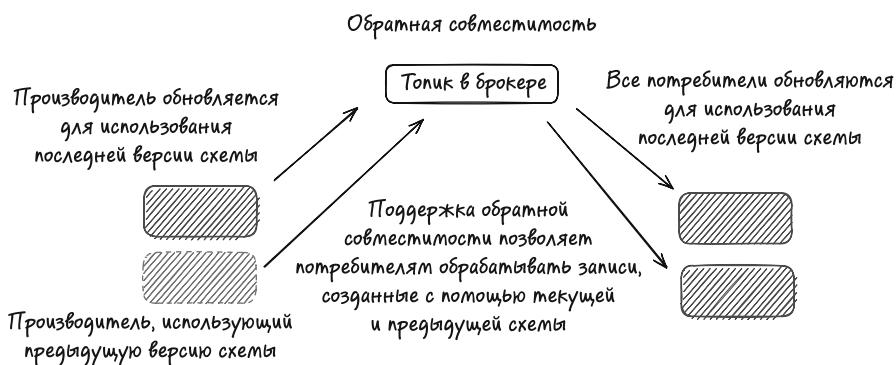
При изменении схемы необходимо учитывать совместимость с существующей схемой и клиентами-производителями и клиентами-потребителями. Если удалить поле, то как это повлияет на производителя, сериализующего записи, или потребителя, десериализующего этот новый формат?

Для решения этих проблем совместимости Schema Registry предоставляет четыре основных режима совместимости: BACKWARD, FORWARD, FULL и NONE. И три дополнительных: BACKWARD\_TRANSITIVE, FORWARD\_TRANSITIVE и FULL\_TRANSITIVE, расширяющих основной режим совместимости с тем же именем. Основные режимы совместимости гарантируют только совместимость новой схемы с предыдущей версией, непосредственно предшествующей ей. Транзитивная (переходная) совместимость указывает, что новая схема совместима со всеми предшествующими версиями данной схемы, применяющими режим совместимости. Уровень совместимости можно установить как глобально, так и для каждого субъекта в отдельности.

Далее следует описание допустимых изменений для каждого режима совместимости и иллюстрация, демонстрирующая последовательность изменений, которые нужно будет внести в реализацию производителей и потребителей. Практическое руководство по изменению схемы вы найдете в приложении В.

### 3.4.1. Обратная совместимость

Обратная совместимость – это настройка миграции по умолчанию. Для сохранения обратной совместимости сначала обновляется код потребителя, чтобы обеспечить поддержку новой схемы (рис. 3.9). Обновленные потребители могут читать записи, сериализованные с новой или предыдущей схемой.



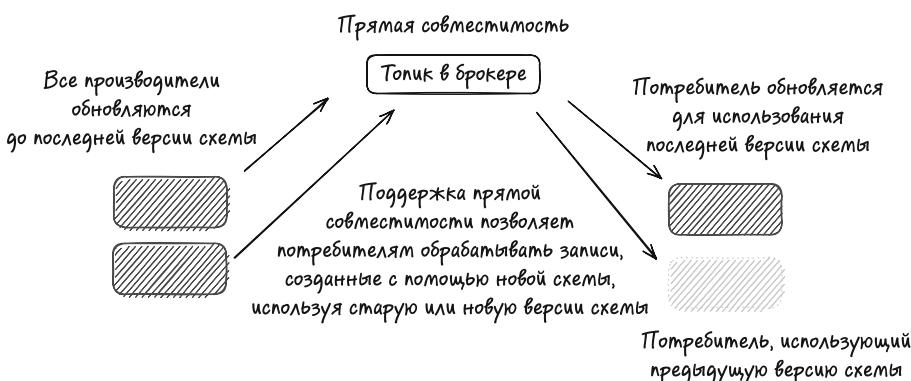
**Рис. 3.9.** В режиме обратной совместимости поддержка новой схемы сначала добавляется в потребитель. Обновленные потребители могут читать записи, сериализованные с новой или предыдущей схемой

Как показано на рис. 3.9, потребитель может работать с предыдущей и новой схемами. В режиме обратной совместимости разрешается удалять поля и добавлять необязательные поля. Поле считается необязательным, если схема определяет

значение по умолчанию. Когда сериализованное представление не содержит необязательное поле, десериализатор использует указанное значение по умолчанию при десериализации байтов обратно в объект.

### 3.4.2. Прямая совместимость

Прямая совместимость — это зеркальное отражение обратной совместимости относительно изменений полей. При поддержке прямой совместимости можно добавлять поля и удалять необязательные поля (рис. 3.10).



**Рис. 3.10.** В режиме сохранения прямой совместимости поддержка новой схемы сначала добавляется в производитель, а потребители могут читать записи, используя новую или предыдущую версию схемы

Обновление кода производителя до обновления потребителей гарантирует правильное заполнение новых полей и их доступность только в записях в новом формате. Потребители могут продолжать использовать старую схему, которая будет игнорировать новые поля, а удаленные поля будут заполнять значениями по умолчанию.

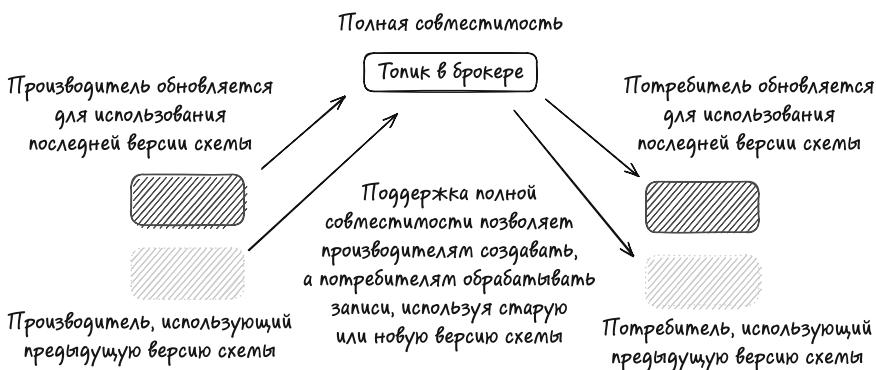
Итак, вы познакомились с двумя типами совместимости: обратной и прямой. Как следует из названий совместимостей, при их поддержке изменения должны учитываться в одном направлении. При поддержке обратной совместимости сначала обновляются потребители, потому что записи могут поступать как в новом, так и в старом формате. При поддержке прямой совместимости сначала обновляются производители, чтобы гарантировать, что к моменту обновления потребителей все записи будут создаваться в новом формате. Последний режим совместимости, который нам предстоит изучить, — режим полной совместимости.

### 3.4.3. Полная совместимость

В режиме полной совместимости разрешается добавлять и удалять поля, но только необязательные (рис. 3.11). Необязательное поле — это поле, для которого в определении схемы указано значение по умолчанию. Это значение будет представлено автоматически, если в десериализованной записи не окажется этого конкретного поля.

## ПРИМЕЧАНИЕ

Avro и JSON Schema поддерживают явное определение значений по умолчанию. В Protocol Buffers версии 3 (версия, используемая в книге) каждое поле автоматически получает значение по умолчанию, определяемое его типом. Например, поля числовых типов по умолчанию получают значение 0, строковых — пустую строку, коллекций — пустые коллекции и т. д.



**Рис. 3.11.** Полная совместимость позволяет производителям отправлять, а потребителям обрабатывать записи с использованием новой или предыдущей схемы

Поскольку поля в обновленной схеме являются необязательными, то любые изменения будут совместимы с существующими производителями и потребителями. Соответственно, порядок обновления в этом случае зависит от вас. Потребители смогут благополучно обрабатывать записи, созданные с помощью новой или старой схемы.

### 3.4.4. Отсутствие совместимости

Выбор режима совместимости `NONE` требует от Schema Registry пропустить проверку на совместимость. Это означает возможность добавления новых, удаления существующих и изменения типов полей. Любые изменения будут приниматься успешно.

Отсутствие проверок на совместимость дает больше свободы, но тогда вы оказываетесь уязвимыми для критических изменений, которые могут остаться незамеченными до самого неподходящего момента: внедрения изменений в производство.

Чтобы избежать проблем, можно после каждого обновления схемы обновлять все производители и потребители или создать новый топик, который приложения смогут использовать, не опасаясь, что в нем появятся записи, созданные с применением старой, несовместимой схемы.

Теперь вы знаете, как обновить версию схемы в различных режимах поддержки совместимости. В табл. 3.2 приводится краткий обзор различных типов совместимости.

**Таблица 3.2.** Обзор режимов совместимости схем

Режим	Допустимые изменения	Порядок обновления клиентов	Гарантированная совместимость с предыдущими версиями
Обратная совместимость	Удаление полей и добавление новых необязательных полей	Потребители, производители	Предшествующая версия
Обратная транзитивная совместимость	Удаление полей и добавление новых необязательных полей	Потребители, производители	Все предыдущие версии
Прямая совместимость	Добавление полей, удаление необязательных полей	Производители, потребители	Предшествующая версия
Прямая транзитивная совместимость	Добавление полей, удаление необязательных полей	Производители, потребители	Все предыдущие версии
Полная совместимость	Добавление и удаление необязательных полей	Не имеет значения	Предшествующая версия
Полная транзитивная совместимость	Добавление и удаление необязательных полей	Не имеет значения	Все предыдущие версии

Но есть еще кое-что, что можно делать со схемами. Подобно программным объектам, схемы могут иметь общий код, что помогает уменьшить дублирование и сделать обслуживание более управляемым. Этой цели служат ссылки на схемы.

## 3.5. ССЫЛКИ НА СХЕМЫ

Ссылка на схему — это именно ссылка, указывающая на другую схему внутри текущей. Повторное использование кода является одним из основных принципов разработки программного обеспечения, потому что возможность использовать то, что уже создано, помогает решить две проблемы. Во-первых, позволяет сэкономить время на переписывании существующего кода. Во-вторых, когда обновляется оригинальный код (что случается почти всегда), все остальные компоненты, использующие этот код, автоматически обновляются.

Когда может понадобиться ссылка на схему? Представьте, что у вас есть приложение, сообщающее информацию о коммерческих предприятиях и университетах. Для моделирования коммерческих предприятий вы используете схему `Company`, а для моделирования университетов — схему `College`. В компании есть руководители, а в колледже — профессора. И того и другого вы решили представить с помощью

вложенного объекта Person. Схемы будут выглядеть примерно так, как показано в листинге 3.28.

#### Листинг 3.28. Схема College

```
"namespace": "bbejeck.chapter_3.avro",
"type": "record",
"name": "CollegeAvro",
"fields": [
    {"name": "name", "type": "string"},
    {"name": "professors", "type": "array",
        "items": {                                     ← Массив с данными о профессорах
            "namespace": "bbejeck.chapter_3.avro",
            "name": "PersonAvro",                      ←
            "fields": [
                {"name": "name", "type": "string"},      | Элемент массива
                {"name": "address", "type": "string"},    | имеет тип Person
                {"name": "age", "type": "int"}             ]
            }
        },
        "default": []
    }
]
```

Как видите, в схеме колледжа имеется вложенный тип записи, что вполне обычное дело. Теперь посмотрим на схему компании (листинг 3.29).

#### Листинг 3.29. Схема Company

```
{
  "namespace": "bbejeck.chapter_3.avro",
  "type": "record",
  "name": "CompanyAvro",
  "fields": [
    {"name": "name", "type": "string"},
    {"name": "executives", "type": "array",
        "items": {                                     ← Массив с данными о руководителях
            "type": "record",
            "namespace": "bbejeck.chapter_3.avro",
            "name": "PersonAvro",                      ←
            "fields": [
                {"name": "name", "type": "string"},      | Элемент имеет
                {"name": "address", "type": "string"},    | тип PersonAvro
                {"name": "age", "type": "int"}             ]
            }
        },
        "default": []
    }
}
```

И снова мы видим вложенный тип в массиве внутри схемы. Моделирование руководителя или профессора в виде отдельного типа — естественное решение,

поскольку такой шаг позволит инкапсулировать все детали в объект. Но, как видите, из-за этого в наших схемах появился повторяющийся код. Если нам понадобится изменить схему описания человека, то мы будем вынуждены обновить каждый файл, содержащий вложенное определение. Кроме того, по мере добавления дополнительных определений схемы могут быстро стать громоздкими из-за наличия определений вложенных типов.

Было бы лучше указать в определении массива ссылку на тип. Для этого мы поместим вложенную запись `PersonAvro` в файл схемы `person.avsc`.

Я не буду приводить содержимое этого файла здесь, поскольку в определении типа ничего не изменилось. Теперь посмотрим, как изменились файлы схем `college.avsc` (листинг 3.30) и `company.avsc` (листинг 3.31).

### Листинг 3.30. Обновленная схема College

```
{
  "namespace": "bbejeck.chapter_3.avro",
  "type": "record",
  "name": "CollegeAvro",
  "fields": [
    {"name": "name", "type": "string"},
    {"name": "professors", "type": "array",
      "items": "bbejeck.chapter_3.avro.PersonAvro"}, ←
      "default": []
    }
  ]
}
```

Это новый элемент,  
ссылка на объект PersonAvro

### ПРИМЕЧАНИЕ

Схема, на которую указывает ссылка, должна быть того же типа, что и ссылающаяся на нее схема. Например, нельзя вставить в схему Protocol Buffers ссылку на схему Avro или JSON Schema; ссылка должна указывать на схему Protocol Buffers.

Здесь мы сделали код чище, использовав ссылку на объект, созданный схемой `person.avsc`. Теперь посмотрим на обновленную схему `Company`.

### Листинг 3.31. Обновленная схема Company

```
{
  "namespace": "bbejeck.chapter_3.avro",
  "type": "record",
  "name": "CompanyAvro",
  "fields": [
    {"name": "name", "type": "string"},
    {"name": "executives", "type": "array",
      "items": "bbejeck.chapter_3.avro.PersonAvro"}, ←
      "default": []
    }
  ]
}
```

Это новый элемент, ссылка  
на объект PersonAvro

Теперь обе схемы ссылаются на один и тот же объект, созданный с помощью схемы PersonAvro. Для полноты картины посмотрим, как реализовать ссылку на схему в JSON Schema и в Protocol Buffers. Сначала рассмотрим версию JSON Schema (листинг 3.32).

#### Листинг 3.32. Схема Company в формате JSON Schema со ссылкой

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Exchange",
  "description": "A JSON schema of a Company using Person refs",
  "javaType": "bbejeck.chapter_3.json.CompanyJson",
  "type": "object",
  "properties": {
    "name": {
      "type": "string"
    },
    "executives": {
      "type": "array",
      "items": {
        "$ref": "person.json" ← Ссылка на схему
      }                                объекта Person
    }
  }
}
```

В JSON Schema ссылки оформляются точно так же, с той лишь разницей, что необходимо добавить явный спецификатор `$ref`, указывающий, что это ссылка на файл схемы. Предполагается, что файл, на который указывает ссылка, находится в том же каталоге, что и ссылающаяся на него схема.

Теперь взглянем на эквивалентную ссылку в схеме Protocol Buffers (листинг 3.33).

#### Листинг 3.33. Схема Company в формате Protocol Buffers со ссылкой

```
syntax = "proto3";

package bbejeck.chapter_3.proto;
import "person.proto"; ← Импортирование схемы
                                для ссылки

option java_outer_classname = "CompanyProto";
option java_multiple_files = true;

message Company {
  string name = 1;
  repeated Person executives = 2; ← Ссылка на схему person.proto
}
```

В схемах Protocol Buffers необходимо сделать один небольшой дополнительный шаг — добавить инструкцию импорта файла, содержащего объект, на который далее в коде дается ссылка.

Теперь возникает вопрос: как (де)серIALIZаторы определяют, каким образом сериализовать и десериализовать объект в правильный формат. Мы удалили определение

из файла, а значит, сериализаторам и десериализаторам также нужно получить ссылку на схему. К счастью, Schema Registry поддерживает такую возможность.

Сначала нужно зарегистрировать схему для объекта, представляющего человека, чтобы при регистрации схем `College` и `Company` мы ссылались на уже зарегистрированную схему.

Gradle-плагин `schema-registry` упрощает решение этой задачи. В листинге 3.34 показаны настройки, необходимые для использования ссылок на схемы.

#### **Листинг 3.34.** Настройка поддержки ссылок на схемы в плагине Gradle

```
register {
    subject('person','src/main/avro/person.avsc', 'AVRO') ← Регистрация
    subject('college-value','src/main/avro/college.avsc', 'AVRO') ← схемы person
        .addReference("bbejeck.chapter_3.avro.PersonAvro", "person", 1) ←
    subject('company-value','src/main/avro/company.avsc', 'AVRO') ←
        .addReference("bbejeck.chapter_3.avro.PersonAvro", "person", 1) ←
}
```

Регистрация схемы Company  
и добавление ссылки на схему person

Регистрация схемы College  
и добавление ссылки на схему person

Здесь мы сначала зарегистрировали файл `person.avsc`, и в этом случае субъект получает простое имя `person`, потому что схема не связана напрямую с каким-то одним топиком. Затем мы регистрируем схемы `College` и `Company`, использовав шаблон `<имя-топика>-value`, потому что эти две схемы привязаны к топикам с одинаковыми названиями и используют стратегию именования по умолчанию (`TopicNameStrategy`). Метод `addReference` принимает три параметра:

- имя ссылки; в формате Avro роль имени ссылки играет полное имя схемы, в Protobuf — имя файла схемы, а в JSON Schema — URL;
- имя субъекта для зарегистрированной схемы;
- номер версии для справки.

Теперь, когда схемы со ссылками зарегистрированы, ваши клиенты-производители и клиенты-потребители смогут правильно сериализовать и десериализовать объекты.

В репозитории к книге есть примеры запуска производителя и потребителя, демонстрирующие схемы со ссылками в действии. Мы уже запустили `./gradlew streams:registerSchemasTask` для основного модуля, а значит, настроили свои ссылки. Чтобы увидеть, как действуют ссылки на практике, запустите код из листинга 3.35.

#### **Листинг 3.35.** Задачи для опробования ссылок на схемы

```
./gradlew streams:runCompanyProducer
./gradlew streams:runCompanyConsumer

./gradlew streams:runCollegeProducer
./gradlew streams:runCollegeConsumer
```

## 3.6. ССЫЛКИ НА СХЕМЫ И НАЛИЧИЕ НЕСКОЛЬКИХ СОБЫТИЙ В ТОПИКЕ

Мы узнали, что стратегии именования субъектов `RecordNameStrategy` и `TopicRecordNameStrategy` позволяют создавать записи различных типов для одного топика. Но с помощью `RecordNameStrategy` любой имеющийся топик должен использовать одну и ту же версию схемы для данного типа. Если вы решите изменить схему, то все топики должны использовать новую схему. Стратегия `TopicRecordNameStrategy` допускает наличие нескольких типов событий в топике и ограничивает видимость схемы рамками одного топика, что позволяет разрабатывать схемы независимо от других топиков.

Но ни один из подходов не позволяет контролировать количество типов, создаваемых в топике. Если кто-то решит создать запись другого нежелательного типа, то вы никак не сможете помешать этому.

Однако, используя ссылки на схемы, можно создать несколько типов событий для топика и ограничить набор типов записей, которые можно создать в нем. Используя `TopicNameStrategy` в сочетании со ссылками на схемы, можно ограничить все записи одним объектом. Другими словами, ссылки на схемы позволяют иметь несколько типов, но только те, на которые ссылается схема. Чтобы лучше понять идею, рассмотрим простой пример.

Представьте, что вы занимаетесь развитием интернет-магазина и разработали систему точного отслеживания посылок, отправленных покупателям. У вас есть парк грузовиков и самолетов, которые доставляют посылки в любую точку страны. Каждый раз, когда посылка минует пункт обработки на маршруте, она сканируется в вашей системе и для нее генерируется одно из трех возможных событий: `PlaneEvent`, `TruckEvent` или `DeliveryEvent`.

Это разные события, но они тесно связаны. Кроме того, поскольку порядок этих событий важен, они должны создаваться в одном топике, чтобы все связанные события хранились в последовательности их возникновения. В главе 4 я расскажу подробнее о том, как объединение связанных событий в одном топике помогает их упорядочить. Далее, создав схемы `PlaneEvent`, `TruckEvent` и `DeliveryEvent`, вы могли бы создать схему (листинг 3.36), содержащую ссылки на различные типы событий.

**Листинг 3.36.** Схема `all_events.avsc` в формате Avro с несколькими событиями

```
[ "bbejeck.chapter_3.avro.TruckEvent", ← | Тип-объединение в Avro с несколькими
  "bbejeck.chapter_3.avro.PlaneEvent", | различными событиями
  "bbejeck.chapter_3.avro.DeliveryEvent"
]
```

Файл схемы `all_events.avsc` представляет объединение Avro, массив возможных типов событий. Объединение используется, когда поле или, как в данном случае, схема может быть более чем одного типа.

Поскольку все ожидаемые типы определяются в одной схеме, топик теперь может содержать несколько типов событий, но только те, что перечислены в схеме. При использовании ссылок на схемы в таком виде в Avro важно всегда устанавливать в конфигурации производителя Kafka параметры `auto.register.schemas=false` и `use.latest.version=true`. Далее я расскажу, почему это необходимо.

Когда сериализатор Avro переходит к сериализации объекта, то не находит схему, потому что она находится в схеме-объединении. В результате он регистрирует схему отдельного объекта, затирая схему-объединение. Установка параметра, отвечающего за автоматическую регистрацию схем, в значение `false` позволяет избежать затирания схемы. Кроме того, благодаря установке параметра `use.latest.version = true` сериализатор получит последнюю версию схемы (схему-объединение) и будет использовать ее для сериализации. В противном случае он будет искать тип события в названии топика и, не найдя его, потерпит сбой.

## СОВЕТ

При использовании поля `oneOf` со ссылками в формате Protocol Buffers указанные схемы автоматически регистрируются рекурсивно, поэтому вы можете использовать конфигурационный параметр `auto.register.schemas` со значением `true`. То же относится к полю `oneOf` в формате JSON Schema.

Теперь посмотрим, как зарегистрировать схему со ссылками (листинг 3.37).

### Листинг 3.37. Регистрация схемы all\_events со ссылками

```
subject('truck_event',
       'src/main/avro/truck_event.avsc', 'AVRO')           | Регистрация отдельных схем, на которые
                                                               | ссылается файл all_events.avsc
subject('plane_event','src/main/avro/plane_event.avsc', 'AVRO')
subject('delivery_event','src/main/avro/delivery_event.avsc', 'AVRO')

subject('inventory-events-value',
       'src/main/avro/all_events.avsc','AVRO')             | Регистрация схемы all_events
.addReference("bbejeck.chapter_3.avro.TruckEvent",
              "truck_event", 1)                            | Добавление ссылок
                                                               | на отдельные схемы
.addReference("bbejeck.chapter_3.avro.PlanEvent", "plane_event", 1)
.addReference("bbejeck.chapter_3.avro.DeliveryEvent", "delivery_event", 1)
```

Как было показано в разделе 3.5, при использовании формата Avro необходимо сначала зарегестрировать отдельные схемы, а уж затем схему-объединение со ссылками.

В формате Protobuf используется не тип объединения, а поле `oneOf`, что, по сути, одно и то же. Однако в Protobuf поле `oneOf` не может располагаться на верхнем уровне; оно должно находиться в сообщении Protobuf. Например, представьте, что вам требуется отслеживать взаимодействия с клиентами — вход в систему, поисковые запросы и покупки — как отдельные события. Но, поскольку они тесно взаимосвязаны, важно сохранить их очередность, поэтому вы решили поместить их в один топик. В листинге 3.38 показан файл в формате Protobuf, содержащий ссылки.

**Листинг 3.38.** Файл в формате Protobuf со ссылками

```
syntax = "proto3";

package bbejeck.chapter_3.proto;

import "purchase_event.proto"; ← Импорт отдельных
import "login_event.proto";
import "search_event.proto";

option java_multiple_files = true;
option java_outer_classname = "EventsProto";

message Events {
    oneof type { ← Поле oneOf, которое может иметь
        PurchaseEvent purchase_event = 1;
        LoginEvent login_event = 2;
        SearchEvent search_event = 3;
    }
    string key = 4;
}
```

Выше в этой главе вы уже видели схему Protobuf, поэтому я не буду приводить все ее части, а повторю только наиболее критичное для этого примера поле `oneOf`, которое может иметь тип `PurchaseEvent`, `LoginEvent` или `SearchEvent`. На этапе регистрации схема Protobuf имеет достаточно информации для рекурсивной регистрации всех схем, на которые даны ссылки, поэтому можно смело устанавливать конфигурационный параметр `auto.register` в значение `true`. Аналогичным образом можно структурировать ссылки Avro, как показано в листинге 3.39.

**Листинг 3.39.** Схема в формате Avro со ссылками, использующими внешний класс

```
{
    "type": "record",
    "namespace": "bbejeck.chapter_3.avro", ← Имя внешнего
    "name": "TransportationEvent",
    "fields" : [ ← Поле с именем "event"
        {"name": "event", "type": [ ←
            "bbejeck.chapter_3.avro.TruckEvent",
            "bbejeck.chapter_3.avro.PlaneEvent",
            "bbejeck.chapter_3.avro.DeliveryEvent"
        ]} ← Объединение,
    ]
}
```

Итак, основное отличие этой схемы Avro от предыдущей заключается в том, что у этой схемы есть внешний класс, а ссылки являются полем в классе. Кроме того, когда внешнему классу предоставляются ссылки Avro, как в этом примере, можно установить конфигурационный параметр `auto.register` равным `true`, но при этом вы все равно должны заранее зарегистрировать схемы объектов, на которые ссылается основная схема, поскольку Avro, в отличие от Protobuf, не располагает достаточной информацией для рекурсивной регистрации объектов по ссылкам.

Есть еще ряд важных вопросов, касающихся использования нескольких типов с производителями и потребителями. Я имею в виду универсальные типы, которые можно использовать в Java-клиентах, и порядок выбора действий с объектом, в зависимости от соответствующего ему имени конкретного класса. Однако эти вопросы лучше обсуждать вместе с обсуждением самих клиентов, поэтому мы рассмотрим их в следующей главе.

К настоящему моменту вы познакомились с различными стратегиями совместности схем, узнали, как работать со схемами и как использовать ссылки. Во всех представленных примерах использовались встроенные сериализаторы и десериализаторы, предоставляемые Schema Registry. В следующем разделе мы рассмотрим настройку (де)сериализаторов для производителей и потребителей, но только параметры, связанные с (де)сериализаторами, а не настройку производителя и потребителя в целом, которую мы обсудим в следующей главе.

## 3.7. СЕРИАЛИЗАТОРЫ И ДЕСЕРИАЛИЗАТОРЫ SCHEMA REGISTRY

В начале главы я отметил, что при создании записей в Kafka их необходимо сериализовать для транспортировки по сети и хранения в Kafka. И наоборот, перед потреблением записей их нужно десериализовать, чтобы иметь возможность напрямую работать с объектами.

Для сериализации и десериализации нужно настроить производитель и потребитель с необходимыми классами. Schema Registry предоставляет классы сериализатора, десериализатора и Serde (используется в Kafka Streams) для всех трех поддерживаемых форматов (Avro, Protobuf, JSON).

Предоставление инструментов сериализации – веский аргумент в пользу Schema Registry, который я рассматривал ранее в этой главе. Отсутствие необходимости писать свой код сериализации ускоряет разработку и повышает стандартизацию в рамках организации. Кроме того, использование стандартных инструментов сериализации уменьшает количество ошибок и вероятность, что одна команда реализует свою платформу сериализации.

### ПРИМЕЧАНИЕ

Что такое Serde? Класс Serde содержит сериализатор и десериализатор для заданного типа. Классы Serde вы будете использовать при работе с Kafka Streams, где нет возможности получить доступ к встроенному производителю и потребителю. Следовательно, предоставление класса с правильными сериализатором и десериализатором имеет смысл. Порядок использования классов Serde мы обсудим в главе 6, когда начнем работать с Kafka Streams.

В следующих разделах я расскажу о настройках использования сериализаторов и десериализаторов в Schema Registry. При этом важно помнить, что эти настройки не настраивают сериализаторы напрямую. Конфигурация сериализатора определяется при настройке KafkaProducer или KafkaConsumer. Если что-то в следующих разделах покажется вам непонятным, то не переживайте, потому что в следующей главе я расскажу о клиентах (производителях и потребителях) более подробно.

### 3.7.1. Сериализаторы и десериализаторы Avro

Для сериализации и десериализации записей Avro можно использовать классы KafkaAvroSerializer и KafkaAvroDeserializer. При настройке потребителя нужно включить дополнительное свойство KafkaAvroDeserializerConfig.SPECIFIC\_AVRO\_READER\_CONFIG = true, указывающее, что десериализатор должен создавать экземпляр SpecificRecord. В противном случае будет возвращаться экземпляр GenericRecord.

В листинге 3.40 показано, как добавить эти свойства в конфигурацию производителя и потребителя. Обратите внимание, что в примере показаны только настройки, касающиеся сериализации. Другие настройки я для простоты опустил. Более подробно настройка производителей и потребителей будет рассматриваться в главе 4.

#### Листинг 3.40. Обязательные настройки для Avro

```
// свойства производителя
producerProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
    StringSerializer.class);                                ← СерIALIZATOR для ключа
producerProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
    KafkaAvroSerializer.class);                            ← SERIALIZATOR для значения
producerProps.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
    "http://localhost:8081");                                ← URL для доступа к сериализатору

// свойства потребителя настраиваются отдельно для каждого потребителя
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
    StringDeserializer.class);                            ← DESERIALIZATOR для ключа
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
    KafkaAvroDeserializer.class);                        ← DESERIALIZATOR для значения
props.put(KafkaAvroDeserializerConfig.SPECIFIC_AVRO_READER_CONFIG,
    true);                                              ← Требует создавать экземпляр записи конкретного типа
props.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
    "http://localhost:8081");                                ← URL для доступа к десериализатору
```

Далее рассмотрим настройки для работы с записями Protobuf.

### 3.7.2. Protobuf

Для работы с записями Protobuf можно использовать классы KafkaProtobufSerializer и KafkaProtobufDeserializer. При использовании Protobuf в комплексе с Schema Registry нелишним будет установить в схеме Protobuf параметрам java\_outer\_classname и java\_multiple\_files значение true. Если вы используете RecordNameStrategy с Protobuf, то обязательно настройте эти свойства, чтобы десериализатор мог определить конкретный тип при конструировании экземпляра из сериализованных байтов.

Выше в этой главе уже говорилось, что сериализаторы, встроенные в Schema Registry, будут пытаться зарегистрировать новую схему. Если ваша схема Protobuf ссылается на другие схемы через инструкции import, то эти схемы тоже будут регистрироваться. Однако такую возможность поддерживает только Protobuf; Avro и JSON Schema не регистрируют автоматически схемы в ссылках. Если вы не желаете, чтобы регистрация схем происходила автоматически, то деактивируйте ее, задав параметр auto.schema.registration = false.

Рассмотрим аналогичный пример с соответствующими настройками Schema Registry для работы с записями Protobuf (листинг 3.41).

#### Листинг 3.41. Обязательные настройки для Protobuf

```
// свойства производителя
producerProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
    StringSerializer.class);                                ← Сериализатор для ключа
producerProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
    KafkaProtobufSerializer.class);                        ← Сериализатор Protobuf для значения
producerProps.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
    "http://localhost:8081");                                ← URL для доступа к Schema Registry со стороны потребителя

// свойства потребителя настраиваются отдельно для каждого потребителя
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
    StringDeserializer.class);                                ← Десериализатор для ключа
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
    KafkaProtobufDeserializer.class);                      ← Десериализатор Protobuf для значения
props.put(KafkaProtobufDeserializerConfig.SPECIFIC_PROTOBUF_VALUE_TYPE,
    AvengerSimpleProtos.AvengerSimple.class);             ← Конкретный класс, экземпляры
props.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
    "http://localhost:8081");                                ← которого должен создавать
                                                               десериализатор
```

По аналогии с десериализатором Avro, вы должны потребовать создавать экземпляры конкретного класса. Но в этом случае вместо установки логического флага, сообщающего, что вам нужен экземпляр конкретного типа, указывается фактическое имя класса. Если не указать конкретный тип значения, то десериализатор вернет экземпляр типа `DynamicRecord` (этот тип подробно рассматривается в приложении, в подразделе B.2.5).

Пример создания и потребления записи в формате Protobuf вы найдете в классе `bvejeck.chapter_3.ProtobufProduceConsumeExample` в примерах исходного кода к книге. Теперь перейдем к примеру настройки Schema Registry для работы с последним поддерживаемым форматом схем — JSON Schema.

### 3.7.3. JSON Schema

Для работы с объектами JSON Schema в Schema Registry имеются классы `KafkaJsonSchemaSerializer` и `KafkaJsonSchemaDeserializer`. Настройки для этого формата (листинг 3.42) похожи на настройки как для Avro, так и для Protobuf.

#### ПРИМЕЧАНИЕ

В Schema Registry также имеются классы `KafkaJsonSerializer` и `KafkaJsonDeserializer`. Несмотря на сходство с именами классов, упомянутых выше, эти (де)сериализаторы предназначены для работы с объектами Java и преобразования их в формат JSON и обратно без участия JSON Schema. Учитывая такое сходство в именах, проверяйте и перепроверяйте себя, что используете сериализатор и десериализатор со словом `Schema` в имени. Об универсальных сериализаторах JSON мы поговорим в следующем разделе.

**Листинг 3.42.** Обязательные настройки для JSON Schema

```
// свойства производителя
producerProps.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
    "http://localhost:8081"); ← URL для доступа к Schema Registry со стороны производителя
producerProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
    StringSerializer.class); ← Сериализатор для ключа
producerProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
    KafkaJsonSchemaSerializer.class); ← Сериализатор JSON Schema для значения

// свойства потребителя
props.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
    "http://localhost:8081"); ← URL для доступа к Schema Registry со стороны потребителя
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
    StringDeserializer.class); ← Десериализатор для ключа
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
    KafkaJsonSchemaDeserializer.class); ← Десериализатор JSON Schema для значения
props.put(KafkaJsonDeserializerConfig.JSON_VALUE_TYPE,
    SimpleAvengerJson.class); ← Конкретный класс, экземпляры которого
                                | должны создавать десериализатор
```

Здесь в последней строке можно заметить сходство с настройками для Protobuf: определение класса, экземпляры которого десериализатор должен создавать из сериализованной формы. Если не указать тип значения, то десериализатор вернет Map, универсальный результат десериализации JSON Schema. То же относится к ключам. Если ключ является объектом JSON Schema, то вы должны указать в параметре `KafkaJsonDeserializerConfig.JSON_KEY_TYPE`, экземпляры какого класса будет создавать десериализатор.

В примерах исходного кода, в `bbejeck.chapter_3.JsonSchemaProduceConsumeExample`, демонстрируется простая реализация производителя и потребителя для работы с объектами JSON Schema. Как и в других примерах производителей и потребителей, здесь есть разделы, демонстрирующие приемы работы с конкретными и универсальными возвращаемыми типами. Подробнее об универсальном типе, используемом в JSON Schema, рассказывается в подразделе В.3.5.

Мы рассмотрели различные сериализаторы и десериализаторы для каждого вида сериализации, поддерживаемого в Schema Registry. Несмотря на то что использовать Schema Registry желательно, это совершенно не обязательно. В следующем разделе я расскажу, как организовать сериализацию и десериализацию объектов Java без Schema Registry.

## 3.8. СЕРИАЛИЗАЦИЯ БЕЗ SCHEMA REGISTRY

В начале этой главы я заявил, что объекты событий, точнее, схемы, представляющие их, являются контрактом между производителями и потребителями платформы потоковой передачи событий Kafka. Schema Registry предоставляет центральный репозиторий для хранения схем, обеспечивая соблюдение этих контрактов в организации. Кроме того, Schema Registry предоставляет удобные встроенные сериализаторы и десериализаторы для работы с данными, избавляющие от необходимости писать свой код сериализации.

Означает ли это, что использование Schema Registry обязательно? Конечно же нет. Иногда Schema Registry может быть недоступным или могут иметься другие препятствия к его использованию. Создать свои сериализаторы и десериализаторы довольно просто. Помните, что производители и потребители отделены от реализации (де)сериализатора; вы лишь указываете в конфигурации имя нужного класса. Однако не стоит забывать, что при использовании Schema Registry те же схемы можно применять при работе с Kafka Streams, Connect и ksqlDB.

Итак, чтобы создать сериализатор и десериализатор, нужно написать классы, реализующие интерфейсы `org.apache.kafka.common.serialization.Serializer` и `org.apache.kafka.common.serialization.Deserializer`. Интерфейс `Serializer` определяет только один обязательный для реализации метод: `serialize`. Интерфейс `Deserializer` тоже определяет только один обязательный метод: `deserialize`. Оба интерфейса имеют дополнительные методы по умолчанию (`configure`, `close`), которые можно переопределить при необходимости. В листинге 3.43 показан фрагмент нестандартного сериализатора, использующего `jackson-databind``objectMapper` (некоторые детали опущены для простоты).

#### Листинг 3.43. Метод `serialize` в нестандартном сериализаторе

```
@Override
public byte[] serialize(String topic, T data) {
    if (data == null) {
        return null;
    }
    try {
        return objectMapper.writeValueAsBytes(data); ← | Преобразует заданный
    } catch (JsonProcessingException e) {           | объект в массив байтов
        throw new SerializationException(e);
    }
}
```

Здесь вызывается метод `objectMapper.writeValueAsBytes()`, который возвращает сериализованное представление переданного объекта. Теперь рассмотрим пример (листинг 3.44) соответствующего десериализатора (некоторые детали опущены для простоты).

#### Листинг 3.44. Метод `deserialize` в нестандартном десериализаторе

```
@Override
public T deserialize(String topic, byte[] data) {
    try {
        return objectMapper.readValue(data, objectClass); ← | Преобразует байты обратно
    } catch (IOException e) {                         | в объект, заданный
        throw new SerializationException(e);          | параметром objectClass
    }
}
```

Пакет `bbejeck.serializers` содержит сериализаторы и десериализаторы, показанные здесь, а также дополнительные сериализаторы и десериализаторы для Protobuf. Вы можете использовать эти сериализаторы и десериализаторы в любых примерах из книги, но помните, что они не используют Schema Registry. Их также можно взять за основу при разработке своих (де)сериализаторов.

В этой главе вы узнали, что объекты событий, точнее, их схемы представляют контракты между производителями и потребителями. Мы обсудили, как Schema Registry хранит эти схемы и гарантирует выполнение подразумеваемого контракта на платформе Kafka. Наконец, мы рассмотрели поддерживаемые форматы сериализации Avro, Protocol Buffers и JSON. В следующей главе продолжим исследовать платформу потоковой передачи событий, вы познакомитесь с клиентами Kafka: `KafkaProducer` и `KafkaConsumer`. Если Kafka рассматривать как центральную нервную систему для данных, то клиенты — это ее нервные окончания, входы и выходы.

## ИТОГИ ГЛАВЫ

- Схемы определяют контракт между производителями и потребителями. Даже если вы не используете явные схемы, у вас всегда есть подразумеваемая схема с предметными объектами, поэтому разработка способа соблюдения этого контракта имеет решающее значение.
- Schema Registry хранит ваши схемы, обеспечивая управление данными, поддерживая версионирование и предлагая три стратегии совместимости схем: обратную, прямую и полную. Стратегии совместимости помогают гарантировать совместимость новой схемы с ее непосредственной предшественницей, а в некоторых случаях и с более старыми схемами. Для совместимости со всеми предшествующими версиями следует использовать обратную транзитивную, прямую транзитивную и полную транзитивную стратегию. Schema Registry предоставляет удобный REST API для загрузки, просмотра и тестирования совместимости схем.
- Schema Registry поддерживает три формата сериализации: Avro, Protocol Buffers и JSON Schema, а также предоставляет встроенные сериализаторы и десериализаторы, которые вы можете подключить к своим экземплярам `KafkaProducer` и `KafkaConsumer` для бесшовной поддержки всех трех поддерживаемых типов. Встроенные (де)сериализаторы кэшируют схемы локально и извлекают их из Schema Registry, только когда не удается найти схему в кэше.
- Генерирование кода с использованием таких инструментов, как Avro и Protocol Buffers, или плагинов с открытым исходным кодом, поддерживающих JSON Schema, помогает ускорить разработку и исключить человеческие ошибки. Плагины, которые интегрируются с Gradle и Maven, также поддерживают возможность тестирования и выгрузки схем в процессе разработки.

# *Клиенты Kafka*

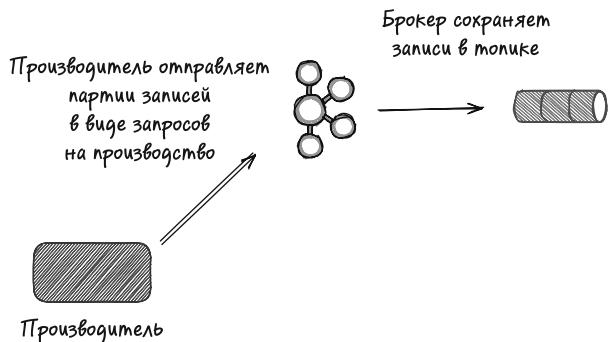
## **В этой главе**

- ✓ Создание записей с помощью KafkaProducer.
- ✓ Семантика доставки сообщений.
- ✓ Потребление записей с помощью KafkaConsumer.
- ✓ Изучение семантики потоковой передачи «точно один раз» в Kafka.
- ✓ Использование Admin API для программного управления топиками.
- ✓ Обработка нескольких типов событий в одном топике.

В этой главе мы берем то, что узнали из предыдущих двух глав, и применяем это здесь, чтобы начать создавать приложения потоковой передачи событий. Мы начнем с индивидуальной работы с клиентами — производителями и потребителями, чтобы понять, как работает каждый из них.

## **4.1. ЗНАКОМСТВО С КЛИЕНТАМИ KAFKA**

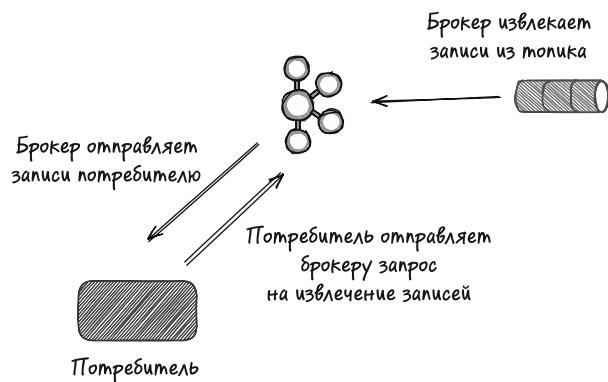
В упрощенном представлении клиенты работают следующим образом: производители отправляют записи брокеру, брокер сохраняет их в топике, потребители отправляют запросы на извлечение, а брокер извлекает записи из топика и возвращает их потребителям (рис. 4.1). В разговорах о платформе потоковой передачи событий Kafka мы часто упоминаем производители и потребители. В конце концов, можно с уверенностью предположить, что вы производите данные для кого-то, кто потребляет их. Но важно понимать, что производители и потребители ничего не знают друг о друге. Между ними нет никакой связи.



**Рис. 4.1.** Производители отправляют партии записей в Kafka в виде запросов на производство

У KafkaProducer одна задача: отправлять записи брокеру. Сами записи содержат всю информацию, необходимую для их хранения.

KafkaConsumer, с другой стороны, только читает, или потребляет, записи из топика (рис. 4.2). Кроме того, как упоминалось в главе 1, посвященной брокерам Kafka, брокер управляет хранением записей. Потребление записей не влияет на то, как долго брокер их хранит.



**Рис. 4.2.** Потребители отправляют запросы на извлечение записей из топика, а брокер извлекает эти записи, чтобы выполнить запрос

В этой главе мы сначала возьмемся за изучение KafkaProducer, исследуем основные конфигурационные параметры и рассмотрим примеры создания записей для брокера Kafka. Изучение особенностей работы KafkaProducer — важнейшая отправная точка для всех, кто намеревается создавать приложения потоковой передачи событий в Kafka.

Затем перейдем к KafkaConsumer, точно так же исследуем основные конфигурационные параметры и рассмотрим некоторые примеры, демонстрирующие, как в приложении потоковой передачи событий осуществляется непрерывное потребление записей из брокера Kafka. Вы начали свой путь к потоковой передаче событий, отправив данные в Kafka, а научившись потреблять эти данные, вы сможете приступить к созданию ценных приложений.

Далее мы опробуем интерфейс `Admin`. Как можно догадаться по имени, это клиент, позволяющий программно выполнять операции по администрированию.

После этого перейдем к более сложной теме, такой как настройка идемпотентного производителя, который гарантирует однократную доставку сообщений в каждом разделе, и транзакционного производителя для однократной доставки в группе из нескольких разделов.

К концу этой главы вы будете знать, как создавать приложения, поддерживающие потоковую передачу событий с использованием клиентов `KafkaProducer` и `KafkaConsumer`. Познакомитесь с особенностями их работы и сможете распознавать ситуации, когда их можно включить в свое приложение. Научитесь настраивать клиенты и тем самым гарантировать надежность своих приложений и их способность справляться с ситуациями, когда что-то идет не так, как ожидалось.

Итак, обозначив маршрут, начнем нашу экскурсию по миру клиентов и посмотрим, как они выполняют свою работу. Сначала мы поговорим о производителе, а затем о потребителе. По пути мы будем делать остановки и уделять некоторое время более глубокому изучению деталей, а затем возобновлять экскурсию.

## 4.2. ПРОИЗВОДСТВО ЗАПИСЕЙ С KAFKAPRODUCER

Вы уже встречались с `KafkaProducer` в главе 3, когда знакомились с Schema Registry, но тогда я не вдавался в подробности работы производителя. Давайте сделаем это сейчас.

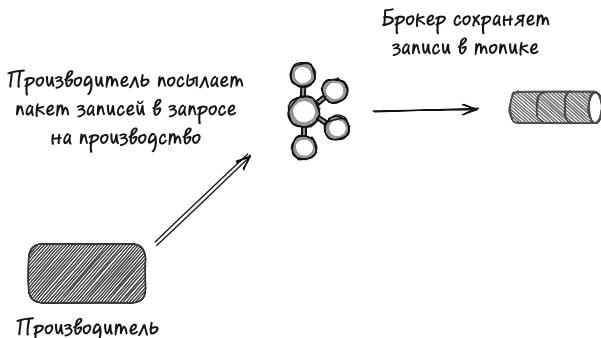
Представьте, что мы работаем в компании оптовой торговли среднего размера и занимаемся сбором данных. Вы получаете данные о сделках через службу продаж, и к этим данным хотели бы получить доступ некоторые другие отделы компании, например, для составления отчетов, управления запасами, выявления тенденций и т. д.

Наша задача — разработать быстрый и надежный способ предоставления этой информации любому сотруднику компании, который желает получить ее. Наша вымышленная компания Vandelay Industries использует Kafka для потоковой передачи событий, и для нас это хорошая возможность включиться в работу. Данные о продажах содержат такие поля, как:

- название товара;
- цена за единицу;
- количество единиц в заказе;
- дата и время заказа;
- имя клиента.

На данном этапе нам ничего не нужно делать с данными о продажах, кроме их отправки в раздел Kafka, который обеспечивает их доступность для использования любым сотрудником компании (рис. 4.3).

Чтобы гарантировать единообразие использования структуры данных, мы создали схему записей и опубликовали ее в Schema Registry. После этого нам осталось лишь написать код, который получает записи о продажах и с помощью `KafkaProducer` отправляет их в Kafka.



**Рис. 4.3.** Отправка данных в топик Kafka

В листинге 4.1 показан наш код. Для простоты я опустил некоторые детали. Полный исходный код можно найти в папке `bbejeck.chapter_4.sales.SalesProducerClient`.

#### Листинг 4.1. KafkaProducer

```

try (
    Producer<String, ProductTransaction> producer =
        new KafkaProducer<>(producerConfigs)) {
    while(keepProducing) {
        Collection<ProductTransaction> purchases =
            salesDataSource.fetch();
        purchases.forEach(purchase -> {
            ProducerRecord<String, ProductTransaction> producerRecord =
                new ProducerRecord<>(topicName, purchase.getCustomerName(),
                    purchase);
            producer.send(producerRecord,
                (RecordMetadata metadata, Exception exception) -> {
                    if (exception != null) {
                        LOG.error("Error producing records ", exception);
                    } else {
                        LOG.info("Produced record at offset {} with timestamp {}",
                            metadata.offset(), metadata.timestamp());
                    }
                });
    };
}
}

```

Создается экземпляр KafkaProducer с помощью оператора `try-with-resources`, чтобы производитель автоматически закрывался после выхода из блока кода

Источник данных о сделках купли-продажи

Отправка записи брокеру Kafka и передача лямбда-выражения в качестве экземпляра Callback

Из входных данных создается запись ProducerRecord

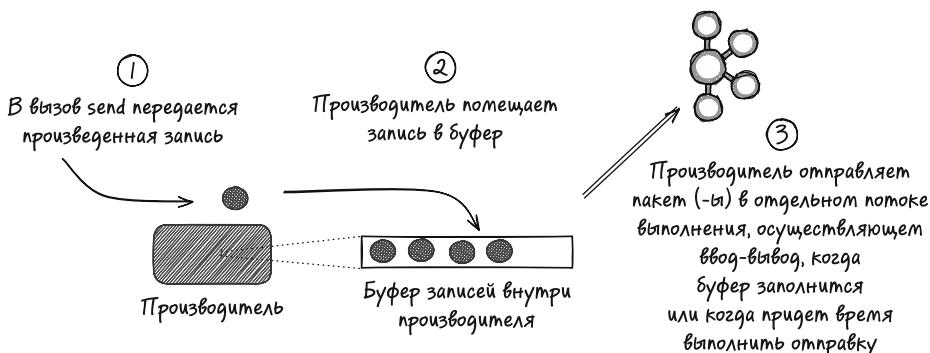
В случае неудачи записать в журнал сообщение об ошибке

В случае успеха возвращаются смещение и отметка времени записи в топике, которые мы сохраняем в журнале

Обратите внимание, что конструктор `KafkaProducer` принимает Map с конфигурацией (некоторые наиболее важные конфигурационные параметры `KafkaProducer` мы обсудим в подразделе 4.2.1). В этом примере мы используем генератор данных, с помощью которого создаются имитации записей. Мы берем список объектов `ProductTransaction` и используем Java Stream API для отображения каждого объекта в объект `ProducerRecord`.

Каждый созданный экземпляр `ProducerRecord` передается как аргумент в вызов метода `KafkaProducer.send()`. Однако производитель не отправляет запись брокеру немедленно, а пытается объединить записи в пакет. При работе в пакетном режиме производитель отправляет меньше запросов, что повышает производительность

как брокера, так и клиента-производителя. Вызов `KafkaProducer.send()` действует асинхронно, что позволяет постоянно добавлять записи в пакет. Производитель запускает отдельный поток выполнения (осуществляющий операции ввода-вывода), который отправляет записи по заполнении пакета или когда придет время выполнить отправку (рис. 4.4).



**Рис. 4.4.** Производитель помещает записи в пакет и отправляет его брокеру по заполнении пакета или когда придет время выполнить отправку

Метод `send` имеет две сигнатуры. Версия, использованная в примере, принимает объекты `ProducerRecord` и `Callback`. Однако, поскольку интерфейс `Callback`, известный также как функциональный интерфейс, содержит только один метод, вместо конкретной реализации можно передать лямбда-выражение. Поток выполнения, осуществляющий ввод-вывод в производителе, выполняет `Callback`, когда брокер подтверждает, что запись сохранена.

Метод `Callback.onCompletion`, представленный здесь как лямбда-выражение, принимает два параметра: `RecordMetadata` и `Exception`. Объект `RecordMetadata` содержит метаданные записи, подтвержденной брокером. Стоит отметить, что если установить `acks=0`, то поле `RecordMetadata.offset` получит значение `-1`, потому что производитель не будет ждать подтверждений от брокера и, соответственно, ему не нужно смещение, назначенное записи. В случае ошибки параметр `Exception` получит непустое значение.

Поскольку обратный вызов выполняется в отдельном потоке ввода-вывода производителя, лучше воздержаться от выполнения в нем тяжеловесных операций, так как это задержит отправку записей. Другая версия метода `KafkaProducer.send()` принимает только параметр `ProducerRecord` и возвращает `Future<RecordMetadata>`. Метод `Future.get()` блокируется до тех пор, пока брокер не подтвердит запись (обработку запроса). Обратите внимание, что вызов метода `get` генерирует исключение, если во время отправки возникнет ошибка.

В общем случае лучше использовать более удобную версию метода `send` с параметром `Callback`, которая использует отдельный поток выполнения для ввода-вывода и обрабатывает результаты отправки асинхронно, избавляя от необходимости отслеживать каждый `Future`, возвращаемый другой версией `send`.

К данному моменту мы рассмотрели основные черты поведения `KafkaProducer`, но, прежде чем перейти к потреблению записей, уделим время обсуждению других важных тем, связанных с производителем: конфигурации, семантики доставки, назначения разделов и отметок времени.

## 4.2.1. Конфигурации производителя

Ниже перечислены некоторые наиболее важные конфигурационные параметры производителя.

- `bootstrap.servers` — одна или несколько пар значений в формате «хост:порт», определяющих адреса брокеров, к которым должен подключиться производитель. В нашем примере используется только одно значение, потому что наш код работает с одним брокером в окружении разработки. В промышленной среде можно перечислить все брокеры в кластере, разделяя их запятыми.
- `key.serializer` — сериализатор для ключей. В нашем примере ключ имеет тип `String`, поэтому можно использовать класс `StringSerializer`. В пакете `org.apache.kafka.common.serialization` имеются сериализаторы для типов `String`, `Integer`, `Double` и т. д. Для сериализации ключей также можно использовать сериализаторы Avro, Protobuf или JSON Schema.
- `value.serializer` — сериализатор для значений. Здесь мы используем объект, сгенерированный из схемы Avro. А так как мы задействовали Schema Registry, то будем использовать `KafkaAvroSerializer`, который вы уже видели в главе 3. Но значение также может быть строкой, целым числом и т. д., и для сериализации таких значений можно использовать сериализаторы из пакета `org.apache.kafka.common.serialization`.
- `acks` — количество подтверждений, необходимых для того, чтобы считать обработку запроса на производство успешной. Допустимые значения: `0`, `1` и `all`. Параметр `acks` — один из самых важных, поскольку он напрямую влияет на сохранность данных. Рассмотрим различные его настройки.
  - Ноль (`acks=0`) — означает, что производитель не будет ждать подтверждения о сохранении записей и считает отправку успешной сразу после передачи записи брокеру. Значение `acks=0` можно рассматривать как настройку типа «запустил и забыл». Эта настройка обеспечивает самую высокую пропускную способность, но и самую низкую гарантию сохранности данных.
  - Единица (`acks=1`) — означает, что производитель ждет уведомления об успешном сохранении записи от ведущего брокера раздела топика. Но производитель не ждет подтверждения о том, что какой-то из ведомых брокеров тоже сохранил запись. С этой настройкой вы получаете дополнительную уверенность в сохранности записи, но она будет потеряна, если ведущий брокер выйдет из строя до того, как ведомые успеют скопировать ее.
  - Все (`acks=all`) — дает наивысшую гарантию сохранности данных. В этом случае производитель ждет подтверждения от ведущего брокера, что он успешно сохранил запись в своем журнале и что ведомые брокеры тоже скопировали ее к себе. Эта настройка дает самую низкую пропускную способность, но самую

высокую гарантию сохранности. При использовании параметра `acks=all` лучше всего установить параметр `min.insync.replicas` ваших топиков в значение выше, чем значение по умолчанию 1. Например, при коэффициенте репликации 3 установка `min.insync.replicas=2` означает, что производитель вызовет исключение при недостаточном количестве реплик. Мы более подробно рассмотрим этот сценарий далее в этой главе.

- `delivery.timeout.ms` — это максимальное время ожидания ответа после вызова `KafkaProducer.send()`. Поскольку Kafka — распределенная система, в ней неизбежно будут происходить сбои при доставке записей брокеру. Но во многих случаях эти ошибки временные, и, следовательно, попытки могут быть повторены. Например, производитель может столкнуться с проблемами подключения из-за фрагментации сети. Но фрагментация сети может быть временной проблемой, поэтому производитель повторит отправку пакета, и во многих случаях повторная отправка увенчается успехом. Однако часто желательно, чтобы после некоторого момента производитель прекратил попытки и выдал ошибку, поскольку длительные проблемы с подключением означают, что им нужно уделить внимание. Обратите внимание, что если производитель сталкивается с фатальной ошибкой, то он выдаст исключение до истечения этого времени ожидания.
- `retries` — когда производитель сталкивается с нефатальной ошибкой, он будет пытаться снова и снова отправить пакет записей. Попытки будут повторяться, пока не истечет тайм-аут `delivery.timeout.ms`. Параметр `retries` по умолчанию получает значение `INTEGER.MAX_VALUE`. Как правило, предпочтительнее не менять этот параметр. А если вы решите ограничить количество повторных попыток, то лучше уменьшите значение `delivery.timeout.ms`. При ошибках и повторах записи могут поступать не по порядку. Предположим, производитель отправляет пакет записей, но ошибка заставляет повторить попытку. В то же время производитель отправил второй пакет, который благополучно достиг брокера. При повторной попытке и первый пакет был успешно отправлен, но теперь он добавится в топик после второго пакета. Чтобы избежать этой проблемы, можно задать параметр `max.in.flight.requests.per.connection=1`. Другой подход, позволяющий избежать нарушения порядка следования пакетов, — использование идемпотентного производителя, о котором рассказывается в подразделе 4.4.1.

Теперь, когда вы познакомились с идеей повторных попыток и подтверждений записи, рассмотрим семантику доставки сообщений.

### 4.2.2. Семантика доставки в Kafka

Kafka поддерживает три разные семантики доставки: «не менее одного раза», «не более одного раза» и «точно один раз». Давайте обсудим каждую из них.

- «*Не менее одного раза*» — при использовании этой семантики записи никогда не теряются, но могут быть доставлены несколько раз. Такое может произойти, например, когда производитель отправляет пакет записей брокеру, брокер добавляет записи в раздел топика, но производитель не получает подтверждения вовремя и повторно отправляет тот же пакет записей. Или когда потребитель обработал полученные записи, но столкнулся с ошибкой, прежде чем успел подтвердить

доставку. В таком случае приложение повторно обработает данные, начиная с последнего подтвержденного смещения, включая уже обработанные, но не подтвержденные записи. По умолчанию Kafka использует семантику доставки «не менее одного раза».

- «*Не более одного раза*» — при использовании этой семантики записи успешно доставляются, но могут теряться из-за ошибок. На стороне производителя семантику «не более одного раза» можно активировать установкой параметра `acks=0`. Поскольку производитель не ждет подтверждения, то после отправки записи его уже не интересует, получил ли брокер запись и сохранил ли ее в топике, и потому не будет повторять попытки отправить запись. На стороне потребителя эта семантика обеспечивается подтверждением получения перед обработкой записей. В этом случае потребитель не получит повторно записи, при обработке которых возникла ошибка, потому что он уже подтвердил их получение и зафиксировал смещения. Итак, чтобы активировать семантику «не более одного раза», нужно на стороне производителя установить параметр `acks = 0`, а на стороне потребителя фиксировать смещения перед обработкой.
- «*Точно один раз*» — при использовании этой семантики записи доставляются не более одного раза и не теряются. Для ее достижения Kafka использует транзакции. Если Kafka прервет транзакцию, то потребитель проигнорирует прерванные данные, его внутренняя позиция продолжит продвигаться, но сохраненные смещения не будут видны другим потребителям с настройкой `read_committed`.

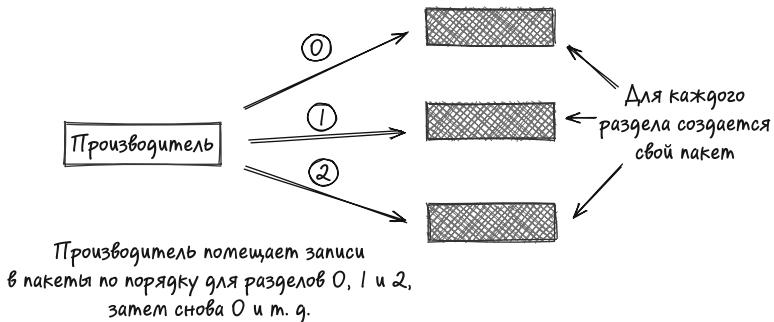
Далее мы поговорим о двух важнейших элементах Kafka — о разделах и отметках времени. Разделы определяют уровень параллелизма и позволяют Kafka распределять загрузку записей раздела между несколькими брокерами в кластере. Брокер использует отметки времени, чтобы определить, какие сегменты журнала он будет удалять. В Kafka Streams отметки времени управляют продвижением записей по топологии (мы еще вернемся к отметкам времени в главе 9).

### 4.2.3. Назначение разделов

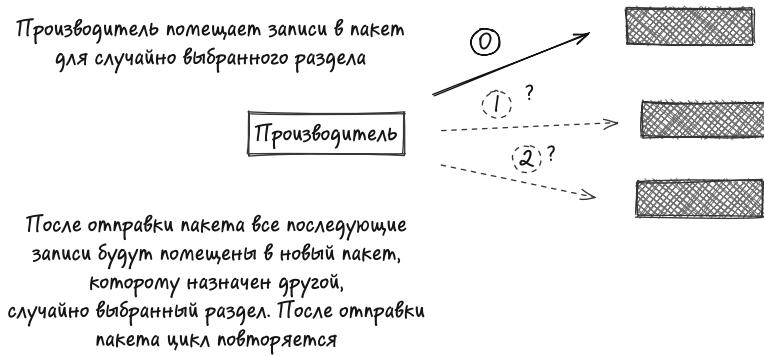
Когда дело доходит до назначения разделов для сохранения записи, у нас есть на выбор три варианта.

1. Указать действительный номер раздела.
2. Указать ключ. В этом случае производитель сможет выбрать номер раздела, взяв хеш ключа и затем остаток от деления хеша на количество разделов.
3. Не указывать ни номер раздела, ни ключ. В этом случае `KafkaProducer` будет выбирать разделы по очереди.

Подход к назначению разделов без ключей несколько изменился с течением времени. До Kafka 2.4 разделы по умолчанию назначались циклическим перебором. То есть производитель выбирал раздел для сохранения записи, увеличивая номер раздела, куда была сохранена предыдущая запись. При таком циклическом подходе брокеру отправлялось несколько уменьшенных пакетов. А из-за большего количества отправляемых запросов увеличивалась нагрузка на брокер. Понять происходящее вам поможет рис. 4.5.

**Рис. 4.5.** Циклическое назначение разделов

Но теперь, если не указать ключ или номер раздела для записи, раздел будет назначаться для каждого пакета. Когда производитель очищает свой буфер и отправляет записи брокеру, пакет помещается в один раздел, для чего достаточно одного запроса. На рис. 4.6 наглядно показано, как это работает.

**Рис. 4.6.** Назначение случайного раздела

После отправки пакета механизм назначения разделов выбирает следующий случайный раздел и назначает его следующему пакету. С течением времени должно быть достигнуто довольно равномерное распределение записей по всем разделам, но с уровнем детализации, равным размеру пакета.

Иногда встроенный механизм может не соответствовать вашим требованиям и нужен более тонкий контроль над назначением разделов. В таких случаях можно реализовать свой механизм.

#### 4.2.4. Реализация своего механизма назначения разделов

Вернемся к приложению производителя из раздела 4.1. Ключом в нем служит имя клиента, но некоторые заказы, в частности заказы с именем клиента `CUSTOM`, обрабатываются нестандартным способом. Для нас желательно помещать все такие заказы в раздел с номером 0, а все остальные — в разделы с номерами 1 и выше.

Чтобы обеспечить такое поведение, нам нужно написать свой механизм назначения разделов, который будет просматривать ключ и возвращать соответствующий номер раздела.

В листинге 4.2 показан такой механизм. `CustomOrderPartitioner` из `src/main/java/bbejeck/chapter_4/sales/CustomOrderPartitioner.java` проверяет ключ и определяет, какой раздел использовать (некоторые детали опущены для простоты).

#### Листинг 4.2. `CustomOrderPartitioner` — механизм назначения разделов

```
public class CustomOrderPartitioner implements Partitioner {

    @Override
    public int partition(String topic,
                         Object key,
                         byte[] keyBytes,
                         Object value,
                         byte[] valueBytes,
                         Cluster cluster) {

        Objects.requireNonNull(key, "Key can't be null");
        int numPartitions = cluster.partitionCountForTopic(topic); ← Получение числа разделов в топике
        String strKey = (String) key;
        int partition;

        if (strKey.equals("CUSTOM")) { ← Для клиента с именем CUSTOM нужно вернуть 0
            partition = 0;
        } else {
            byte[] bytes = strKey.getBytes(StandardCharsets.UTF_8);
            partition = Utils.toPositive(Utils.murmur2(bytes)) %
                        (numPartitions - 1) + 1; ← Определение номера раздела для других записей
        }
        return partition;
    }
}
```

Механизм выбора раздела должен реализовать интерфейс `Partitioner`, который определяет три метода: `partition`, `configure` и `close`. В листинге 4.2 показан только метод `partition`, так как два других метода в этой реализации ничего не делают. Логика работы проста: если в записи указано имя клиента `CUSTOM`, то возвращается номер 0 раздела. Иначе номер раздела определяется как обычно, но с небольшим изменением. Сначала мы вычитаем единицу из числа кандидатов на разделы, поскольку раздел 0 зарезервирован для других нужд. Затем увеличиваем полученный номер раздела на 1, чтобы гарантировать, что для обычных заказов всегда будет возвращаться номер раздела 1 или больше.

#### ПРИМЕЧАНИЕ

Этот пример представляет нетипичный вариант использования и приводится, только чтобы показать, как реализовать свой механизм назначения разделов. В большинстве случаев лучше использовать один из стандартных классов.

Теперь, узнав, как создать свой механизм назначения разделов, посмотрим, как подключить его к производителю.

## 4.2.5. Подключение механизма назначения разделов

Итак, настроим производитель так, чтобы он использовал наш механизм вместо стандартного. Вот как это делается:

```
producerConfigs.put(ProducerConfig.PARTITIONER_CLASS_CONFIG,  
    CustomOrderPartitioner.class);
```

В примере `bbejeck.chapter_4.sales.SalesProducerClient` используется `CustomOrderPartitioner`, но вы можете закомментировать строку с этой настройкой, если пожелаете применить механизм назначения разделов по умолчанию. Обратите внимание, что выбор механизма назначения разделов является настройкой производителя, поэтому вам придется настроить каждый производитель, который должен использовать нестандартный механизм назначения.

## 4.2.6. Отметки времени

Объект `ProducerRecord` содержит поле типа `Long`, которое хранит отметку времени. Если вы не укажете отметку времени, то производитель `KafkaProducer` сам подставит текущее время системы, на которой он работает. Отметки времени играют важную роль в Kafka. Брокер с их помощью определяет, когда удалять записи, извлекая самую старую отметку времени в сегменте и сравнивая ее с текущим временем. Сегмент удаляется, если разница превышает настроенное время хранения. Kafka Streams и `ksqlDB` тоже в значительной степени полагаются на отметки времени, но подробнее об этом я расскажу в соответствующих главах.

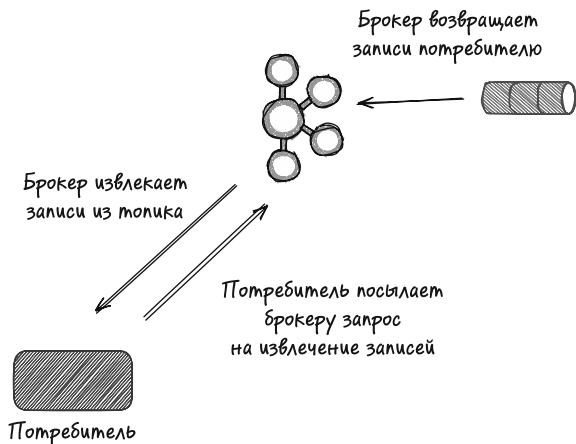
Kafka использует две возможные отметки времени, в зависимости от конфигурации топика. В Kafka топики имеют конфигурационный параметр `message.timestamp.type`, который может принимать значение `CreateTime` или `LogAppendTime`. Со значением `CreateTime` в этом параметре брокер будет сохранять записи с отметками времени, предоставленными производителем, а со значением `LogAppendTime` — подставлять свое текущее системное время. На практике разница между этими отметками времени должна быть небольшой. Отмечу также, что время события можно встроить в полезную нагрузку записи при ее создании.

На этом мы завершаем обсуждение вопросов, связанных с производителем, и переходим к противоположной стороне Kafka — стороне потребления записей.

# 4.3. ПОТРЕБЛЕНИЕ ЗАПИСЕЙ С KAFKA CONSUMER

Итак, мы продолжаем работать в Vandelay Industries, и теперь перед нами стоит новая задача. Наше приложение-производитель запущено и работает, успешно отправляя записи о продажах в топик. Но теперь нас просят разработать приложение `KafkaConsumer`, которое будет служить моделью для потребления записей из топика Kafka.

`KafkaConsumer` отправляет брокеру запрос на извлечение записей из топиков, на которые он подписан (рис. 4.7). Но отправка запроса не обязательно приводит к извлечению новых записей. Иногда брокер может извлекать записи, кэшированные предыдущим вызовом.



**Рис. 4.7.** Потребитель посылает запросы на извлечение, а брокер извлекает записи из топика и возвращает их потребителю

### ПРИМЕЧАНИЕ

Клиенты-производители и клиенты-потребители доступны также в других языках программирования, но в этой книге мы рассматриваем только клиенты на Java, которые доступны в дистрибутиве Apache Kafka. Список клиентов на других языках программирования вы найдете по адресу <http://mng.bz/Y7qK>.

Для начала рассмотрим код, создающий экземпляр `KafkaConsumer` (листинг 4.3). Некоторые детали для простоты опущены.

#### Листинг 4.3. Код создания KafkaConsumer из bbejeck.chapter\_4.sales.SalesConsumerClient

```
try {
    final Consumer<String, ProductTransaction> consumer = new KafkaConsumer<>(
        consumerConfigs); // Создание нового экземпляра потребителя
    consumer.subscribe(topicNames); // Подписка на топик (-и)
    while (keepConsuming) {
        ConsumerRecords<String, ProductTransaction> consumerRecords =
            consumer.poll(Duration.ofSeconds(5)); // Запрос на извлечение записей
        consumerRecords.forEach(record -> { // Обработка каждой полученной записи
            ProductTransaction pt = record.value();
            LOG.info("Sale for {} with product {} for a total sale of {}",
                record.key(),
                pt.getProductName(),
                pt.getQuantity() * pt.getPrice());
        });
    }
}
```

В этом примере с помощью оператора `try-with-resources` создается экземпляр `KafkaConsumer`. Затем производится подписка на топик или топики и начинается обработка записей, возвращаемых методом `KafkaConsumer.poll`. В этом примере мы просто выводим подробности продаж.

**СОВЕТ**

Закончив использовать экземпляр `KafkaProducer` или `KafkaConsumer`, его нужно закрыть, чтобы удалить все потоки выполнения и закрыть сетевые соединения. Оператор `try-with-resources` (<http://mng.bz/GZxR>) в Java гарантирует своевременное освобождение ресурсов, созданных в разделе `try` в конце оператора. Использование `try-with-resources` — хорошая практика, так как иначе легко забыть добавить вызов метода `close` производителя или потребителя.

Обратите внимание, что, как и в случае с производителем, мы создаем ассоциативный массив `Map` с конфигурацией и передаем его конструктору. Ниже перечислены некоторые из наиболее важных параметров конфигурации.

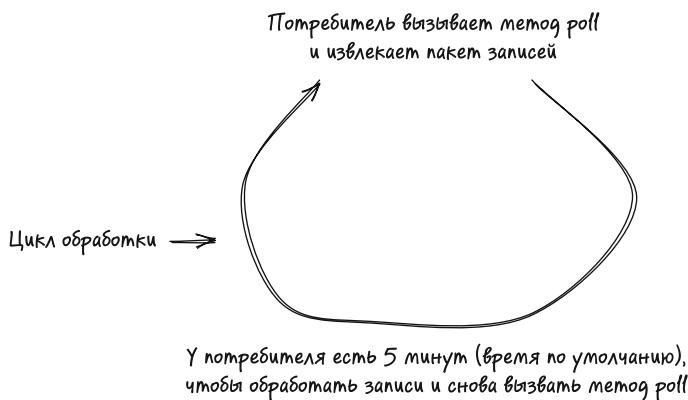
- `bootstrap.servers` — одна или несколько пар значений в формате «хост:порт», определяющих адреса брокеров, к которым должен подключиться потребитель. В нашем примере используется только одно значение, но вообще можно через запятую перечислить несколько значений.
- `max.poll.interval.ms` — максимальное время между вызовами `KafkaConsumer.poll()`, при превышении которого потребитель считается неактивным и запускается перебалансировка. Подробнее о координаторах групп потребителей и перебалансировках мы поговорим ниже в этом разделе.
- `group.id` — произвольное строковое значение, используемое для объединения отдельных потребителей в одну группу. Kafka использует концепцию группы потребителей для логического представления нескольких потребителей как одного потребителя.
- `enable.auto.commit` — логический флаг, который определяет, будет ли потребитель автоматически фиксировать смещения. Если присвоить этому параметру значение `false`, то код приложения должен вручную фиксировать смещения записей, которые считаются успешно обработанными.
- `auto.commit.interval.ms` — временной интервал для автоматической фиксации смещений.
- `auto.offset.reset` — при запуске потребитель возобновляет потребление с последнего зафиксированного смещения. Если смещения недоступны, то этот параметр указывает, откуда начать потребление записей, с самого раннего или самого позднего доступного смещения. В последнем случае первой доступной считается запись, поступившая после запуска потребителя.
- `key.deserializer.class` — имя класса десериализатора, преобразующего ключи записей в объект желаемого типа.
- `value.deserializer.class` — имя класса десериализатора, преобразующего значения записей в объекты желаемого типа. В этом примере мы используем встроенный десериализатор `KafkaAvroDeserializer`, который настраивается с помощью параметра `schema.registry.url`, присутствующего в нашей конфигурации.

Код нашего первого приложения-потребителя прост, но это не главное. В других ваших приложениях бизнес-логика (то есть код обработки данных) будет отличаться в каждом конкретном случае.

Гораздо важнее понять, как работает `KafkaConsumer` и как на его работе отражаются различные значения конфигурационных параметров. Зная это, вы будете лучше понимать, как писать код, обрабатывающий потребляемые записи. Поэтому, так же как в примере с производителем, мы отвлечемся от нашего повествования и исследуем влияние различных настроек потребителя.

### 4.3.1. Интервал опроса

Первым обсудим роль параметра `max.poll.interval.ms`. Взгляните на рис. 4.8, чтобы увидеть, как действует интервал опроса, и получить полное представление о нем.



**Рис. 4.8.** Параметр `max.poll.interval.ms` определяет, как долго может простоять потребитель между вызовами `KafkaConsumer.poll()`, прежде чем он станет считаться неактивным и будет удален из группы потребителей

На этой иллюстрации цикл обработки потребителя начинается с вызова `KafkaConsumer.poll(Duration.ofSeconds(5))`. В вызов `poll(Duration)` передается максимальное время, в течение которого потребитель будет ждать появления новых записей, в данном случае 5 секунд. Если вызов `poll(Duration)` возвращает какие-либо записи, то далее цикл `for` выполняет обход списка записей `ConsumerRecords` и обрабатывает каждую из них. Если вызов не вернул ни одной записи, то управление передается внешнему циклу `while` и выполняется следующий вызов `poll(Duration)`.

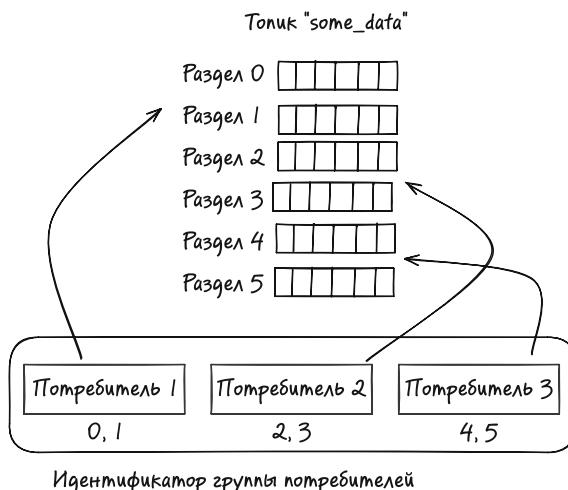
Обход и обработка записей должны быть завершены до истечения времени `max.poll.interval.ms`. По умолчанию это время составляет 5 минут, поэтому если обработка займет больше времени, то этот конкретный потребитель будет считаться неактивным и произойдет перебалансировка. Я упомянул несколько новых терминов: координатор группы и перебалансировка. Мы рассмотрим их в следующем разделе, посвященном параметру конфигурации `group.id`.

Если обработка занимает больше времени, чем `max.poll.interval.ms`, то попробуйте оптимизировать обработку записей, чтобы ускорить ее. Если оптимизировать обработку не получится, то уменьшите максимальное количество записей, извлекаемых потребителем одним вызовом `poll`. Для этого можно уменьшить значение

параметра `max.poll.records`, которое по умолчанию равно 500. В этом случае у меня нет хороших рекомендаций и вам придется подобрать оптимальное число экспериментальным путем.

### 4.3.2. Идентификатор группы

Параметр `group.id` требует более обстоятельного обсуждения групп потребителей в Kafka. Потребители имеют параметр `group.id`, который Kafka использует для объединения всех потребителей с одинаковым значением `group.id` в одну группу потребителей. Потребители, объединенные в группу, рассматриваются как один логический потребитель. Схема на рис. 4.9 показывает, как работает группировка.



**Рис. 4.9.** Группировка потребителей позволяет назначать разделы топиков сразу нескольким потребителям

Схема на рис. 4.9 изображает конфигурацию с одним топиком и шестью разделами. В группе имеется три потребителя, каждому из которых назначены два раздела. Kafka гарантирует, что каждый раздел топика будет назначен только одному потребителю. Если бы раздел назначался более чем одному потребителю, то это привело бы к неопределенному поведению.

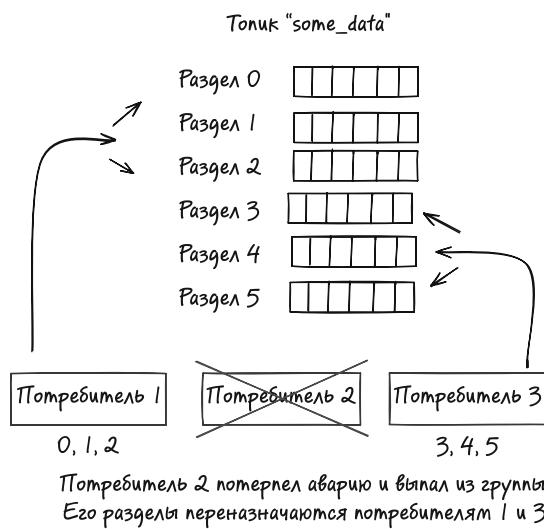
Опыт использования распределенных систем показывает, что отказов следует не избегать, а принимать, применяя разумные практики, помогающие справиться с отказами при их возникновении. Итак, что случится в нашем сценарии, если один из потребителей в группе выйдет из строя, столкнувшись с исключением или превысив тайм-аут, как обсуждалось, когда мы рассматривали параметр `max.poll.interval.ms`? В таком случае Kafka выполнит протокол перебалансировки, как показано на рис. 4.10.

На рис. 4.10 мы видим, что потребитель 2 потерпел аварию и больше не может функционировать. В этом случае механизм перебалансировки возьмет разделы, назначенные потребителю 2, и переназначит их двум другим потребителям, по одному каждому. Если потребитель 2 вернется в строй (или к группе присоединится другой

потребитель), то произойдет еще одна перебалансировка и разделы будут переназначены активным потребителям. Каждый член группы снова будет опрашивать по два раздела топика.

### ПРИМЕЧАНИЕ

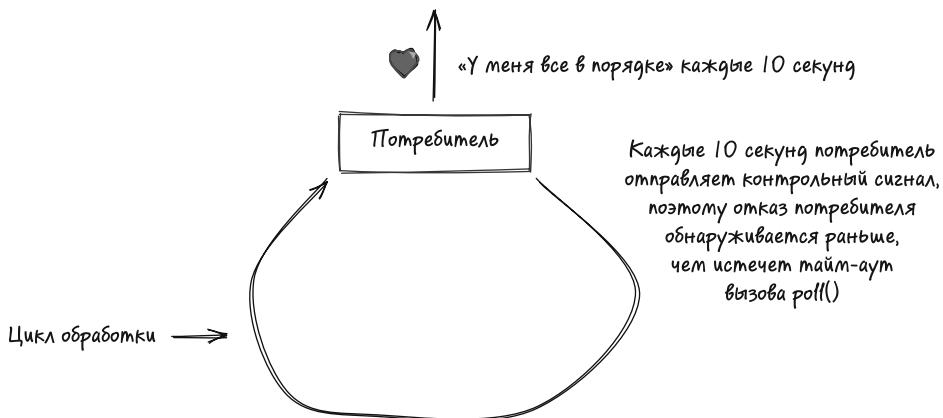
Количество разделов ограничивает количество активных потребителей. В данном примере можно запустить до шести потребителей в группе. Если потребителей окажется больше, то первые шесть будут работать, а остальные — простоять. Обратите также внимание, что разные группы не влияют друг на друга; каждая из них действует независимо.



**Рис. 4.10.** Протокол перебалансировки Kafka переназначает освободившиеся разделы между активными потребителями

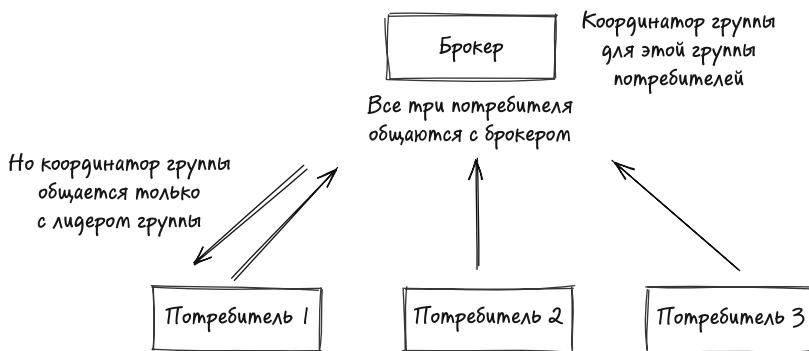
Выше я говорил, что невыполнение вызова `poll()` в течение указанного тайм-аута приведет к исключению потребителя из группы, последующей перебалансировке и назначению его разделов другим потребителям. Но, как вы помните, по умолчанию `max.poll.interval.ms` получает значение, равное 5 минутам. Означает ли это, что для исключения потенциально аварийного потребителя из группы и переназначения его разделов потребуется до 5 минут? Ответ: нет. Давайте снова посмотрим на схему, иллюстрирующую работу интервала опроса, но на этот раз дополним ее тайм-аутами сеанса (рис. 4.11).

Существует еще один настраиваемый тайм-аут — параметр `session.timeout.ms` — со значением по умолчанию 45 секунд. Каждый `KafkaConsumer` запускает отдельный поток выполнения для отправки контрольных сигналов, сообщающих, что он все еще работоспособен. Если потребитель не отправит тактовый импульс в течение 45 секунд, то он отмечается как аварийный и удаляется из группы, что в свою очередь, запускает перебалансировку. Такой двухуровневый подход к подтверждению активности потребителя помогает обеспечить функционирование всех потребителей. Он также позволяет переназначать разделы другим членам группы в случае сбоя одного из них, чтобы обеспечить непрерывную обработку.



**Рис. 4.11.** Потребитель должен не только вызывать метод poll() в течение тайм-аута, но еще и отправлять контрольный сигнал каждые 10 секунд

Давайте обсудим новые термины «координатор группы», «перебалансировка» и «лидер группы», упоминавшиеся выше, чтобы вы могли наглядно представить, как работают группы. Начнем с визуального представления взаимосвязей между частями (рис. 4.12).



**Рис. 4.12.** Координатор группы — это брокер, на который возложена роль отслеживания подмножества групп потребителей, а лидер группы — это потребитель, взаимодействующий с координатором групп

Координатор группы — это брокер, управляющий составом подмножества всех доступных групп потребителей. Ни один брокер не будет действовать как координатор группы в одиночку — ответственность распределяется между разными брокерами. Координатор группы отслеживает состав групп потребителей с помощью запросов на присоединение к группе или когда член группы не выходит на связь в течение заданных тайм-аутов.

Обнаружив изменение состава группы, координатор группы запускает перебалансировку для перераспределения разделов между оставшимися членами. Суть

перебалансировки заключается в том, что все участники группы отсоединяются и присоединяются к группе снова, чтобы ресурсы группы (разделы топиков) были равномерно распределены между всеми участниками. Когда присоединяется новый участник, у некоторых или у всех существующих участников группы отбираются какие-то разделы топиков и назначаются новому участнику. Когда существующий участник покидает группу, происходит обратный процесс: его разделы переназначаются другим активным участникам.

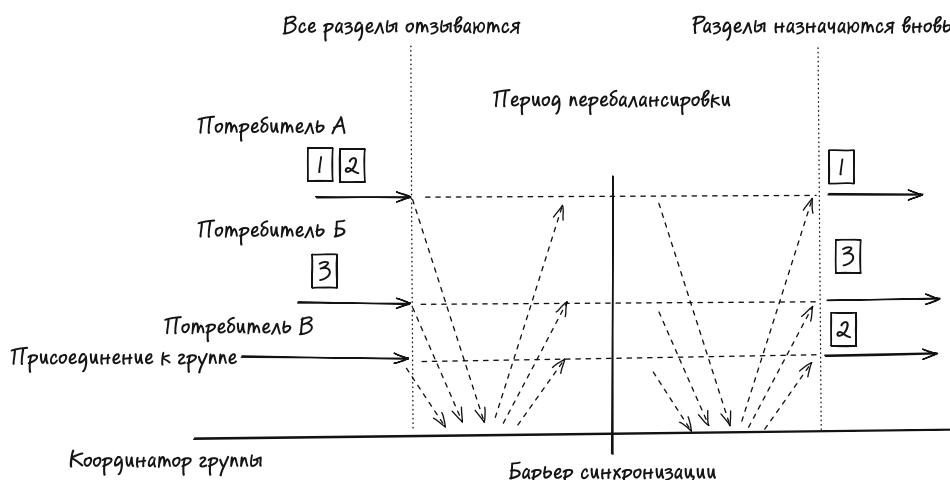
Процесс перебалансировки довольно прост, но занимает время, в течение которого все потребители простоят, ожидая его завершения. Часто этот процесс называют «остановкой мира» или *жадной перебалансировкой*. Однако в версии Kafka 2.4 появилась возможность использовать новый протокол перебалансировки — *кооперативную перебалансировку*.

Рассмотрим оба протокола, начав с жадной перебалансировки.

### Жадная перебалансировка

Когда координатор группы обнаруживает изменение в составе группы, он запускает перебалансировку. Это касается обоих протоколов перебалансировки, которые мы обсуждаем.

С началом процесса перебалансировки каждый член группы отказывается от владения всеми назначенными ему разделами топиков. Затем они отправляют запрос `JoinGroup` контроллеру. Запрос включает разделы топиков, интересующие потребителя, от которых он только что отказался. В результате отказа потребителей от разделов обработка останавливается (рис. 4.13).



**Рис. 4.13.** Жадная перебалансировка, вызывающая «остановку мира». Обработка всех разделов прекращается до завершения переназначения, при этом большинство разделов остаются за потребителем, который владел ими прежде

Контроллер собирает всю информацию о разделах группы и отправляет ответ `JoinGroup`, а лидер группы получает всю информацию о разделах топиков.

#### **ПРИМЕЧАНИЕ**

Напомню, что, как обсуждалось в главе 2, все действия представляют собой последовательность запросов и ответов.

На основе этой информации лидер группы назначает разделы топиков всем членам группы и посыпает информацию о назначении координатору в запросе `SyncGroup`. Обратите внимание, что другие члены группы тоже отправляют запросы `SyncGroup`, но не включают в него информацию о назначении. Получив информацию о назначении от лидера, контроллер назначает новые наборы разделов членам группы через ответ `SyncGroup`.

После этого все участники группы возобновляют обработку. Обратите внимание, что обработка приостанавливается в момент отправки участниками группы запроса `JoinGroup` и возобновляется после получения ответа `SyncGroup` с назначениями. Этот разрыв в обработке называют барьером синхронизации, и он совершенно необходим, потому что очень важно гарантировать, что у каждого раздела топика будет только один владелец-потребитель. Если бы у раздела топика было несколько владельцев, то это привело бы к неопределенному поведению.

#### **ПРИМЕЧАНИЕ**

Во время всего этого процесса клиенты-потребители общаются только с координатором группы. Кроме того, только один член группы, лидер, назначает разделы топиков и отправляет информацию о назначении координатору.

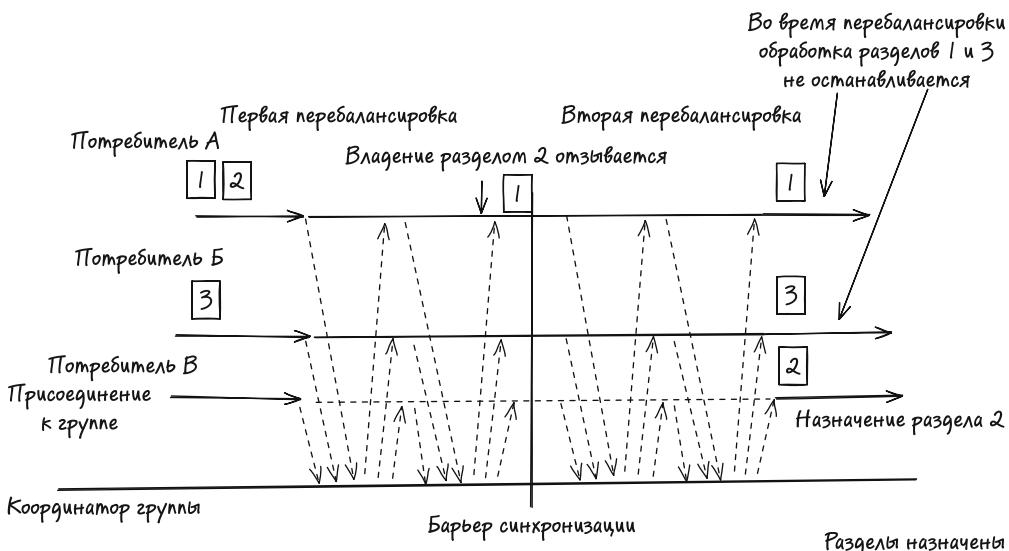
Протокол жадной перебалансировки перераспределяет ресурсы и гарантирует, что только один потребитель будет владеть каждым конкретным разделом топика, но он сопровождается простое, потому что каждый потребитель приостанавливается после отправки запроса `JoinGroup` и возобновляет работу только после получения ответа `SyncGroup`. В небольших приложениях этот период может быть незначительным, но в приложениях с несколькими потребителями и большим количеством разделов в топиках продолжительность простоя увеличивается. К счастью, есть другой подход к перебалансировке, направленный на исправление этой ситуации.

### **Инкрементальная кооперативная перебалансировка**

В Kafka 2.4 был реализован протокол инкрементальной кооперативной перебалансировки, избавляющий от необходимости останавливать обработку. В этом подходе используется другой взгляд на перебалансировку.

1. Потребители не отказываются автоматически от права владения всеми своими разделами.
2. Лидер группы определяет конкретные разделы, которые требуется переназначить.
3. Обработка разделов, которые не меняют владельца, продолжается без перерыва.

Последний пункт — самый большой плюс кооперативной перебалансировки. На этот раз останавливается обработка не «всего мира», а только переназначаемых разделов (рис. 4.14). Другими словами, барьер синхронизации получается намного меньше.



**Рис. 4.14.** Процесс кооперативной перебалансировки приостанавливает обработку только тех разделов, которые должны быть переназначены

Давайте кратко рассмотрим процесс инкрементальной кооперативной перебалансировки. Как и прежде, когда контроллер группы обнаруживает изменение в составе группы, он запускает перебалансировку. Каждый член группы все так же перечисляет свои текущие подписки на разделы в запросе `JoinGroup`, но на этот раз владение разделами сохраняется.

Координатор группы собирает всю информацию о подписках и посыпает ответ `JoinGroup` лидеру группы с назначениями. Лидер группы определяет, какие разделы должны быть переданы в новое владение, удаляет все эти разделы из назначений и отправляет обновленные подписки координатору через запрос `SyncGroup`. Как и прежде, каждый член группы отправляет запрос `SyncGroup`, но только запрос лидера содержит информацию о подписке.

### ПРИМЕЧАНИЕ

Все члены группы получают ответ `JoinGroup`, но только ответ лидеру группы содержит информацию о назначении. Аналогично каждый член группы отправляет запрос `SyncGroup`, но только лидер отправляет информацию о новых назначениях разделов. В ответе `SyncGroup` все члены получают список назначенных разделов, возможно обновленный.

Члены группы получают ответ `SyncGroup` с двумя списками старых и новых назначений, по которым определяют, какие новые разделы им назначены или какие

старые отозваны. Разделы, включенные в оба списка, старых и новых назначений, не требуют никаких действий.

Затем члены группы запускают вторую перебалансировку, но в нее включаются только разделы, меняющие владельца. Эта вторая перебалансировка действует как барьер синхронизации, как и в жадном подходе, но, поскольку в нее вовлечены только разделы, получающие новых владельцев, она выполняется намного быстрее. Кроме того, разделы, не меняющие владельца, продолжают обрабатываться!

Теперь, обсудив различные подходы к перебалансировке, рассмотрим некоторую более общую информацию о доступных стратегиях назначения разделов и способах их применения.

### 4.3.3. Применение стратегий назначения разделов

Мы уже знаем, что брокер выступает в роли координатора группы для некоторого подмножества групп потребителей. Поскольку две разные группы потребителей могут по-разному распределять ресурсы (разделы), ответственность за выбор подхода целиком ложится на клиента.

#### ПРИМЕЧАНИЕ

Для Kafka Connect и Kafka Streams, которые являются абстракциями поверх производителей и потребителей Kafka, используйте протоколы кооперативной перебалансировки с настройками по умолчанию. Далее вы узнаете, какие стратегии доступны для приложений, напрямую использующих KafkaConsumer.

Выбор стратегии назначения разделов для экземпляров KafkaConsumer в группе производится с помощью конфигурационного параметра `partition.assignment.strategy`, которому присваивается список поддерживаемых стратегий назначения разделов. Все доступные механизмы назначения разделов реализуют интерфейс `ConsumerPartitionAssignor`. Ниже приводится список доступных стратегий с кратким описанием.

- `RangeAssignor` — стратегия по умолчанию. `RangeAssignor` сортирует разделы в числовом порядке и назначает их потребителям путем деления количества доступных разделов на количество потребителей. Назначение разделов потребителям происходит в лексикографическом порядке.
- `RoundRobinAssignor` — циклически обходит список разделов и членов группы и назначает очередной раздел следующему доступному члену группы.
- `StickyAssignor` — стремится назначать разделы максимально сбалансированно и пытается максимально сохранить существующие назначения. `StickyAssignor` следует протоколу жадной перебалансировки.
- `CooperativeStickyAssignor` — действует подобно `StickyAssignor`, но использует протокол кооперативной перебалансировки.

Сложно дать конкретный совет по выбору той или иной стратегии, потому что каждый вариант использования требует тщательного анализа его уникальных потребностей. Однако в общем случае в новых приложениях желательно выбирать `CooperativeStickyAssignor` по причинам, описанным в разделе об инкрементальной кооперативной перебалансировке.

**СОВЕТ**

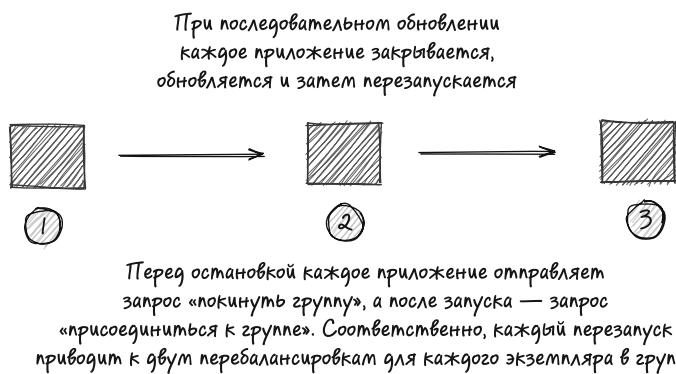
При обновлении версии Kafka 2.3 или более ранней необходимо следовать определенному алгоритму, описанному в документации по обновлению до версии 2.4 ([https://kafka.apache.org/documentation/#upgrade\\_240\\_notable](https://kafka.apache.org/documentation/#upgrade_240_notable)), чтобы безопасно перейти на протокол кооперативной перебалансировки.

На этом мы заканчиваем рассмотрение групп потребителей и протокола перебалансировки. Далее мы рассмотрим другую конфигурацию — статическое членство, — не предусматривающую перебалансировки, когда потребитель покидает группу.

#### 4.3.4. Статическое членство

В предыдущем разделе мы узнали, что при отключении экземпляра потребителя отправляет контроллеру запрос на выход из группы. Контроллер тоже может посчитать клиент зависшим и сам удалит его из группы потребителей. В обоих случаях исключение из группы приводит к одному результату: контроллер запускает перебалансировку для переназначения ресурсов (разделов) оставшимся членам группы.

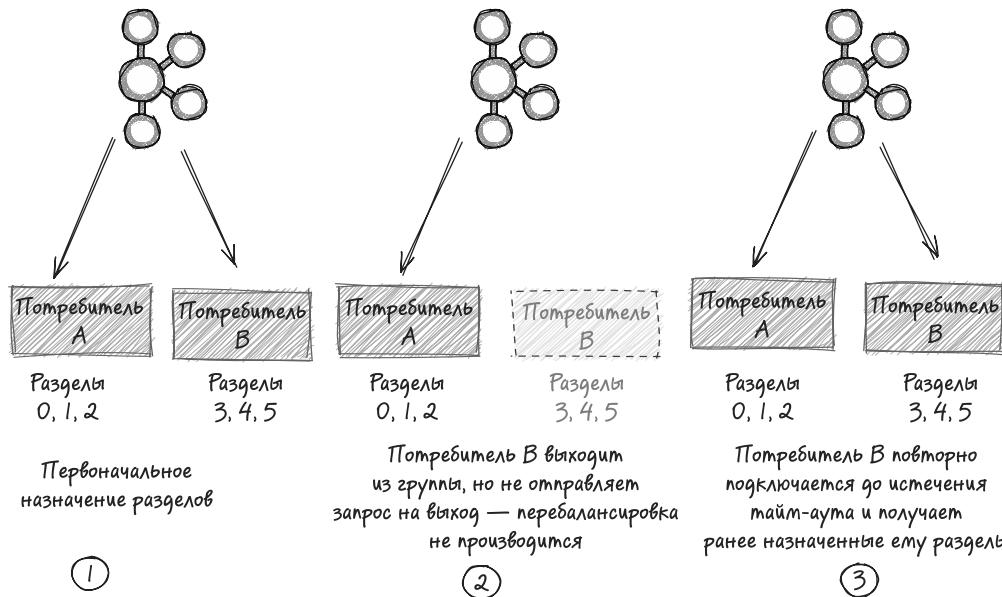
Часто такое поведение именно то, что нам нужно, так как оно обеспечивает надежность приложений, но в некоторых ситуациях может оказаться более удобным иное поведение. Например, в конфигурации с несколькими потребительскими приложениями можно производить последовательное их обновление и перезапуск, когда это требуется (рис. 4.15).



**Рис. 4.15.** Последовательное обновление запускает несколько перебалансировок

Для обновления приложения нужно остановить экземпляр 1, обновить и перезапустить его, затем остановить, обновить и перезапустить экземпляр 2 и т. д., пока не будут обновлены все приложения. Последовательное обновление не увеличивает потери времени обработки по сравнению с одновременным закрытием всех приложений. Но при последовательном обновлении для каждого экземпляра выполняются две перебалансировки: одна при завершении приложения и другая при запуске. Представьте также облачную среду, где узел приложения может отключиться в любой момент и запуститься снова после обнаружения его отказа.

Даже с улучшениями, которые привносит кооперативная перебалансировка, было бы выгоднее вообще не запускать перебалансировку для таких кратковременных действий. Идея статического членства была реализована в версии Apache Kafka 2.3. Обсудим ее, руководствуясь иллюстрацией на рис. 4.16.



**Рис. 4.16.** Статические члены не отправляют запросы на выход из группы при остановке, а статический идентификатор позволяет контроллеру запомнить их

На высоком уровне в конфигурационном параметре потребителя `group.instance.id` задается уникальный идентификатор. Потребитель передает этот идентификатор контроллеру, когда присоединяется к группе, и контроллер сохраняет его у себя. Когда потребитель покидает группу, он не отправляет запрос на выход из группы, а когда присоединяется вновь, то передает идентификатор членства контроллеру, а тот отыскивает его у себя и возвращает разделы, назначенные этому потребителю, не выполняя повторной балансировки! Недостаток статического членства заключается в необходимости присвоить параметру `session.timeout.ms` значение, превышающее значение по умолчанию 10 секунд, так как по истечении этого времени контроллер исключит потребитель из группы и запустит перебалансировку.

Ваше значение должно быть достаточно большим, чтобы учесть время на остановку, обновление и запуск и не вызывать перебалансировку, но не настолько большим, чтобы не затягивать обработку реального сбоя. Поэтому если для вас допустима продолжительность частичной недоступности 10 минут, то установите тайм-аут сеанса равным 8 минутам. Несмотря на то что статическое членство выглядит неплохим вариантом для тех, кто запускает приложения KafkaConsumer в облачной среде, важно учитывать последствия для производительности, прежде чем использовать его.

Кроме того, чтобы воспользоваться преимуществами статического членства, брокеры и клиенты Kafka должны быть версии 2.3.0 или выше.

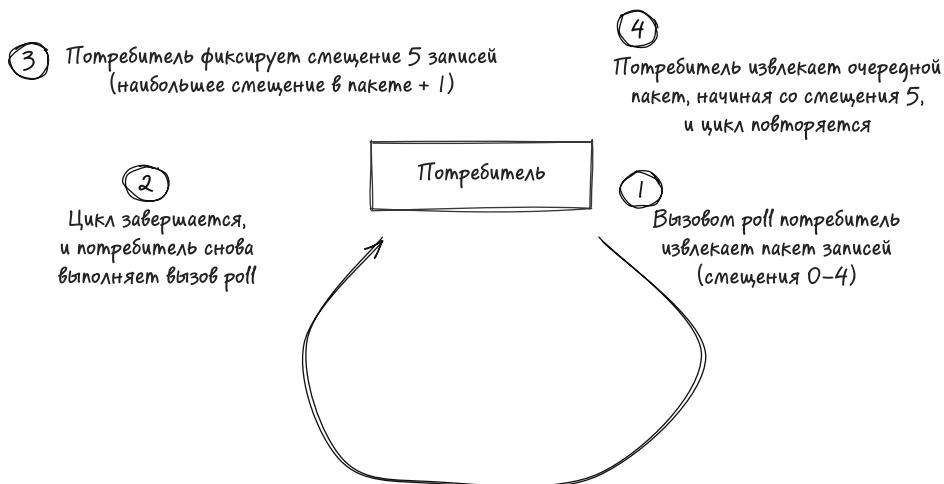
Далее мы рассмотрим важную тему фиксации смещений сообщений при использовании `KafkaConsumer`.

### 4.3.5. Фиксация смещений

В главе 2 мы узнали, как брокер назначает входящим записям номера, называемые смещениями. Для каждой следующей входящей записи брокер увеличивает смещение на единицу. Смещения играют важную роль, определяя логическую позицию записи в топике. `KafkaConsumer` использует смещения, чтобы узнать, какую последнюю запись он потребил. Например, если потребитель извлекает пакет записей со смещениями от 10 до 20, то начальное смещение следующего пакета записей, которые он извлечет, будет равно 21.

Чтобы гарантировать продвижение вперед в случае повторного запуска или сбоя, потребитель должен периодически фиксировать смещение последней успешно обработанной записи. Потребители Kafka имеют механизм автоматической фиксации смещений. Включить этот механизм можно, присвоив конфигурационному параметру `enable.auto.commit` значение `true`. Вообще это значение присваивается по умолчанию, но упомянул о нем, чтобы обсудить работу автоматической фиксации. Также нам нужно обсудить понятие позиции потребителя, отличающееся от понятия последнего зафиксированного смещения. Сопутствующий конфигурационный параметр `auto.commit.interval.ms` определяет интервал времени, через который потребитель обязан фиксировать смещения. Он основан на системном времени потребителя.

Но сначала посмотрим, как работает автоматическая фиксация (рис. 4.17).

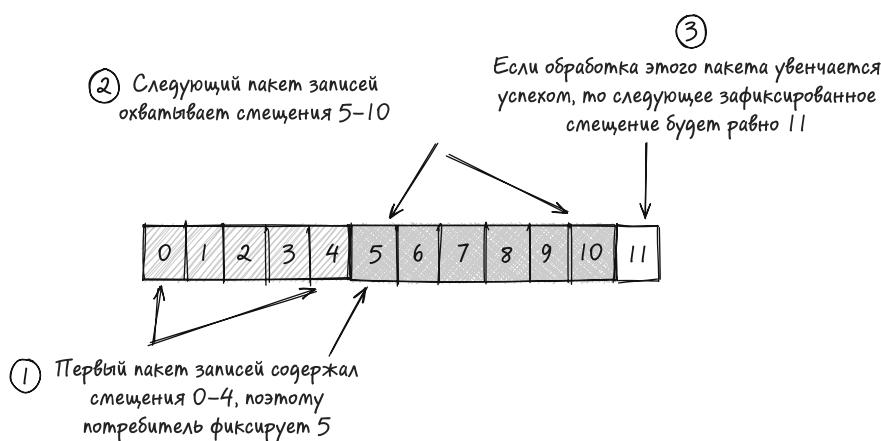


**Рис. 4.17.** Если включена автоматическая фиксация и истек интервал времени `auto.commit.interval.m`, то перед возвратом к началу цикла опроса потребитель фиксирует наибольшее смещение предыдущей партии +1

Как показано на рис. 4.17, потребитель извлекает пакет записей, вызывая `poll(Duration)`. Затем код выполняет итерации по `ConsumerRecords` и обрабатывает каждую запись. После этого выполнение возвращается в начало цикла опроса и производится следующая попытка извлечь другие записи. Но перед извлечением записей, если в настройках потребителя включена автоматическая фиксация и время, прошедшее с момента последней проверки автофиксации, больше интервала `auto.commit.interval.ms`, потребитель фиксирует смещения записей из предыдущего пакета. Фиксируя смещения, он помечает записи как потребленные и в обычных условиях больше не будет обрабатывать эти записи снова. Что под этим имеется в виду, я опишу позже.

Что подразумевается под фиксацией смещения? Kafka поддерживает внутренний топик с именем `_consumer_offsets`, где хранит смещения, зафиксированные потребителями. Когда мы говорим, что потребитель фиксирует смещение, то это означает, что он сохраняет наибольшее смещение в разделе плюс один.

Например, рассмотрим рис. 4.17 и предположим, что извлеченные записи в пакете имели смещения 0–4. Поэтому, когда потребитель зафиксирует смещение после обработки записей, он тем самым сохранит смещение 5 (рис. 4.18).



**Рис. 4.18.** Зафиксированная позиция потребителя — это наибольшее смещение потребленной к текущему моменту записи плюс один

Итак, зафиксированная позиция — это смещение последней успешно обработанной записи (плюс один), определяющее первую запись в следующем пакете, который получит потребитель. На рис. 4.18 — это число 5. Если в этом примере приложение потерпит аварию или будет перезапущено вручную, то потребитель получит записи, начиная со смещения 5, поскольку он не смог зафиксировать смещение до сбоя или перезапуска.

Потребление с последнего зафиксированного смещения гарантирует, что ни одна запись не будет пропущена из-за сбоя или перезапуска приложения. Но это также означает, что запись может быть обработана более одного раза (рис. 4.19).

Предположим, что потребитель обработал некоторые записи со смещениями больше последнего зафиксированного, но по какой-то причине не смог зафиксировать их. В этом случае, когда приложение возобновит работу, оно вновь получит записи, начиная с последнего зафиксированного смещения, поэтому некоторые записи будут обработаны повторно. Эта возможность обработки более чем один раз известна как семантика «не менее одного раза». Мы рассматривали эту семантику в подразделе 4.2.2.



**Рис. 4.19.** Перезапуск потребителя после обработки без фиксации повлечет повторную обработку некоторых записей

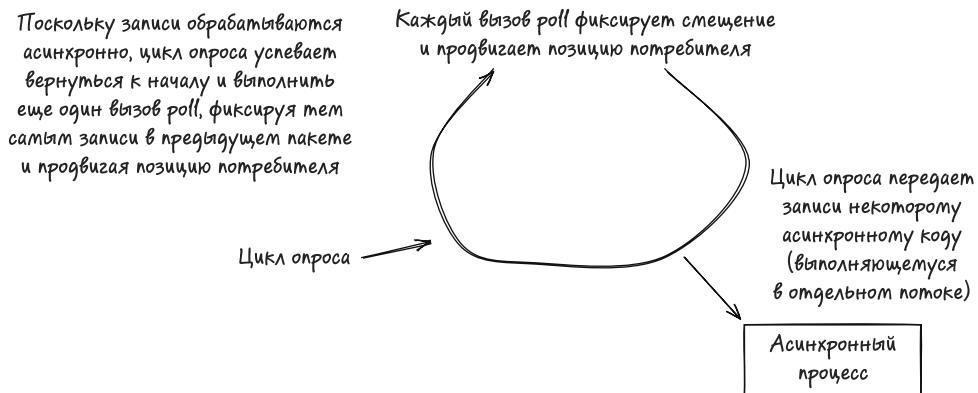
Чтобы избежать повторной обработки записей, можно вручную фиксировать смещения сразу после получения пакета записей, что даст вам семантику доставки «не более одного раза». Но в таком случае есть риск потерять некоторые записи, если потребитель столкнется с ошибкой после фиксации и до обработки записей.

### Соображения по поводу фиксации

При использовании автоматической фиксации важно убедиться, что код полностью обработал все извлеченные записи до того, как он вернется в начало цикла опроса. Это легко гарантировать, если записи обрабатываются синхронно, то есть если код ждет завершения обработки каждой записи. Однако если записи передаются другому потоку для асинхронной обработки позже, то тогда есть риск, что какие-то записи не успеют обработать до фиксации (рис. 4.20). Я объясню, как такое может произойти.

После передачи записей асинхронному процессу код в цикле опроса не будет ждать завершения обработки каждой записи. Когда цикл вернется к началу и приложение снова вызовет метод `poll()`, тот зафиксирует текущую позицию, то есть наибольшее смещение плюс один для каждого раздела топика, откуда были получены записи в предыдущем пакете. Но асинхронный процесс может не успеть завершить обработку всех записей к моменту фиксации. Представьте, что потребительское приложение столкнулось с ошибкой или было остановлено по какой-либо причине. В таком случае после перезапуска оно начнет с последнего зафиксированного

смещения и в результате пропустит записи, которые не были обработаны в предыдущем сеансе работы.



**Рис. 4.20.** Асинхронная обработка с автоматической фиксацией может привести к потенциальной потере записей

Чтобы избежать преждевременной фиксации записей до их полной обработки, нужно отключить автоматическую фиксацию, присвоив параметру `enable.auto.commit` значение `false`. Но зачем тогда возиться с асинхронной обработкой при ручной фиксации? Вот простой пример: к потребляемым записям применяется довольно продолжительная обработка (до 1 секунды), при этом через топик идет большой объем трафика, и вы хотите оставаться в курсе событий. Поэтому вы решаете сразу же передавать полученные пакеты асинхронному процессу и немедленно возвращаться к вызову `poll` для получения следующего пакета.

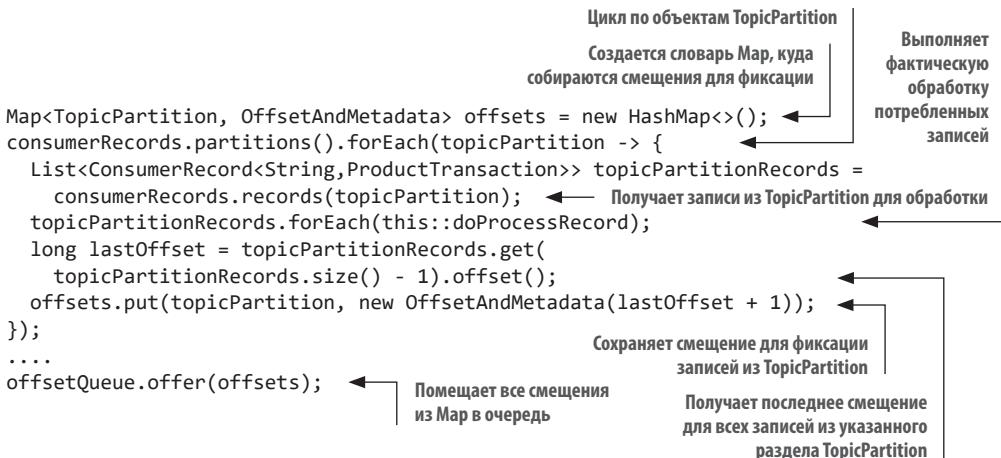
Такой подход называется *конвейеризацией*. Используя его, вы должны гарантировать, что фиксироваться будут только смещения успешно обработанных записей. Для этого нужно отключить автоматическую фиксацию и создать свой механизм, который будет фиксировать только те записи, которые были полностью обработаны. Код в листинге 4.4 демонстрирует одно из возможных решений. Обратите внимание, что здесь показаны только ключевые детали, а чтобы увидеть весь пример целиком, обращайтесь к исходному коду в репозитории (`bbejeck.chapter_4.pipeline.PipliningConsumerClient`).

#### Листинг 4.4. Код потребителя

```
ConsumerRecords<String, ProductTransaction> consumerRecords = consumer.poll(
    Duration.ofSeconds(5));
if (!consumerRecords.isEmpty()) {
    recordProcessor.processRecords(consumerRecords); ← После извлечения пакет записей
    Map<TopicPartition, OffsetAndMetadata> offsetsAndMetadata =
        recordProcessor.getOffsets(); ← Проверка смещений обработанных записей
    if (offsetsAndMetadata != null) {
        consumer.commitSync(offsetsAndMetadata); ← Если словарь Map не пустой,
    }                                            то фиксируются смещения обработанных
}                                                на данный момент записей
```

Важно отметить, что в этом коде вызов `RecordProcessor.processRecords()` возвращает управление немедленно, поэтому следующий вызов `RecordProcessor.getOffsets()` возвращает смещения из предыдущего пакета записей, которые полностью обработаны. Здесь я хочу особо подчеркнуть, что код сначала передает новые записи в обработку, а затем собирает смещения обработанных записей для фиксации. Взглянем на код обработчика, чтобы увидеть, как это делается (листинг 4.5). Полный код см. в `bbejeck.chapter_4.piplining.ConcurrentRecordProcessor`.

#### Листинг 4.5. Асинхронный обработчик записей



Суть здесь заключается в том, что при переборе записей с помощью `TopicPartition` код создает в `Map` элемент со смещением для фиксации. После обработки всех записей остается только получить самое последнее смещение. Внимательный читатель может спросить: «Почему код добавляет 1 к последнему смещению?» Дело в том, что фиксация выполняется для смещения записи, которая следует за последней обработанной. Например, если последняя обработанная запись имеет смещение 5, то вы должны зафиксировать 6. Поскольку записи со смещениями 0–5 приложение уже потребило, в следующей итерации оно должно начать со смещения 6.

Далее в верхней части цикла `TopicPartition` используется в качестве ключа, а объект `OffsetAndMetadata` — в качестве значения. Когда смещения извлекаются из очереди, их можно фиксировать без опаски, потому что приложение уже обработало соответствующие записи. Главное в этом примере то, что этот код гарантированно фиксирует только те записи, которые были асинхронно обработаны вне цикла `Consumer.poll`. Обратите внимание, что в этом примере используется *только один поток* для обработки записей, то есть код по-прежнему обрабатывает записи по порядку, поэтому фиксация смещений по мере их возврата не таит никаких опасностей.

#### ПРИМЕЧАНИЕ

Более полный пример потоковой передачи и использования `KafkaConsumer` вы найдете в статье Энтони Стаббса (Anthony Stubbs) *Introducing the Confluent Parallel Consumer* (<http://mng.bz/z8xX>) и в репозитории <https://github.com/confluentinc/parallel-consumer>.

## Когда смещения недоступны

Выше я упоминал, что Kafka хранит смещения во внутреннем топике `_consumer_offsets`. Но что произойдет, если потребитель не сможет найти свои смещения? Возьмем случай запуска нового потребителя для существующего топика. Новый `group.id` не будет иметь никаких фиксаций, связанных с ним. Возникает вопрос: с какого места начать потребление, если для данного потребителя не будет найдено зафиксированных смещений? Для таких случаев в `KafkaConsumer` предусмотрен конфигурационный параметр `auto.offset.reset`, позволяющий указать относительную позицию для начала потребления.

Он может принимать три значения:

- `earliest` — сбрасывает смещение на самое раннее;
- `latest` — сбрасывает смещение на самое последнее;
- `none` — генерирует исключение в потребителе.

Значение `earliest` подразумевает, что обработка начнется с начала топика, то есть потребитель получит все доступные в данный момент записи. Значение `latest` показывает, что потребитель начнет получать записи, которые поступят в топик после начала работы потребителя, а все предыдущие записи, уже имеющиеся в топике, будут пропущены. Значение `none` означает, что потребитель генерирует исключение и, в зависимости использования в нем блоков `try/catch`, он сможет завершить работу.

Выбор настройки полностью зависит от конкретного варианта использования. Но в общем случае после запуска потребителя предпочтительнее начать читать последние записи, иначе обработка всех имеющихся записей может оказаться слишком затратной.

Вы прошли длинный путь, но он стоил затраченных усилий, и теперь вы знаете некоторые критические аспекты работы с `KafkaConsumer`. Вы узнали, как создавать потоковые приложения с использованием `KafkaProducer` и `KafkaConsumer`. Мы обсудили ситуации, в которых желательно использовать семантику «не менее одного раза». Но в некоторых случаях нужно гарантировать обработку каждой записи ровно один раз. Для таких случаев Kafka предлагает семантику доставки «точно один раз».

## 4.4. СЕМАНТИКА ДОСТАВКИ «ТОЧНО ОДИН РАЗ»

В версии Apache Kafka 0.11 в `KafkaProducer` была реализована семантика доставки сообщений «точно один раз». Для поддержки этой семантики в `KafkaProducer` поддерживаются два режима: идемпотентный и транзакционный.

### ПРИМЕЧАНИЕ

Операция называется идемпотентной, если при многократном выполнении она выдает один и тот же результат.

Идемпотентный производитель гарантирует доставку сообщений по порядку и только один раз в раздел топика. Транзакционный производитель позволяет производить сообщения для нескольких топиков атомарно, то есть либо все сообщения будут доставлены во все топики, либо ни одно. В следующих разделах мы обсудим идемпотентный и транзакционный производители.

#### 4.4.1. Идемпотентный производитель

Чтобы получить идемпотентный производитель, достаточно присвоить конфигурационному параметру `enable.idempotence` значение `true`, которое в настоящее время является значением по умолчанию. Важную роль играют еще несколько параметров.

1. Значение `max.in.flight.requests.per.connection` не должно превышать 5 (значение по умолчанию — 5).
2. Значение `retries` должно быть больше 0 (значение по умолчанию — `Integer.MAX_VALUE`).
3. Параметр `acks` должен иметь значение `all`.

Рассмотрим листинг 4.6 (некоторые детали опущены для простоты).

##### Листинг 4.6. Настройка идемпотентности в KafkaProducer

```
Map<String, Object> producerProps = new HashMap<>();
// Стандартные настройки
producerProps.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "somehost:9092");
producerProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, ...);
producerProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, ...);

// Настройки, связанные с идемпотентностью
producerProps.put(ProducerConfig.ACKS_CONFIG, "all"); ← Включить ожидание
producerProps.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, true); ← Включить идемпотентность
producerProps.put(
    ProducerConfig.RETRIES_CONFIG, Integer.MAX_VALUE); ← Задать максимальное число повторов равным
producerProps.put(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION, 5); ← Задать максимальное количество неподтвержденных
                                                                           запросов на соединение равным 5. Это значение
                                                                           по умолчанию и показано здесь лишь для полноты
                                                                           и показано здесь лишь для полноты
```

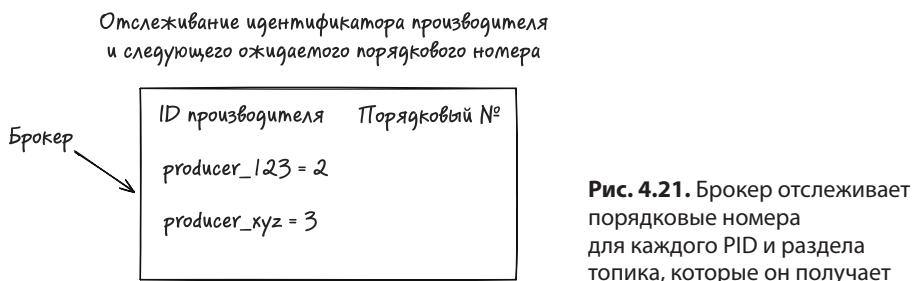
В обсуждении `KafkaProducer` выше мы разобрали ситуацию, когда пакеты записей могут сохраняться в разделе не по порядку из-за ошибок и повторных попыток, и узнали, что присвоение конфигурационному параметру `max.inflight.requests.per.connection` значения 1 позволит избежать ее. Использование идемпотентного производителя устраняет необходимость в настройке этого параметра. В подразделе 4.2.2 о семантике доставки сообщений также говорилось, что нужно присвоить 0 параметру `retries`, чтобы предотвратить возможное дублирование записей ценой вероятной потери данных.

Идемпотентный производитель решает обе проблемы: сохранение записей не по порядку и возможное их дублирование из-за повторных попыток. Если вам требуется строго соблюдать порядок следования записей в разделе и гарантировать отсутствие дублирующих записей, то использование идемпотентного производителя становится обязательным.

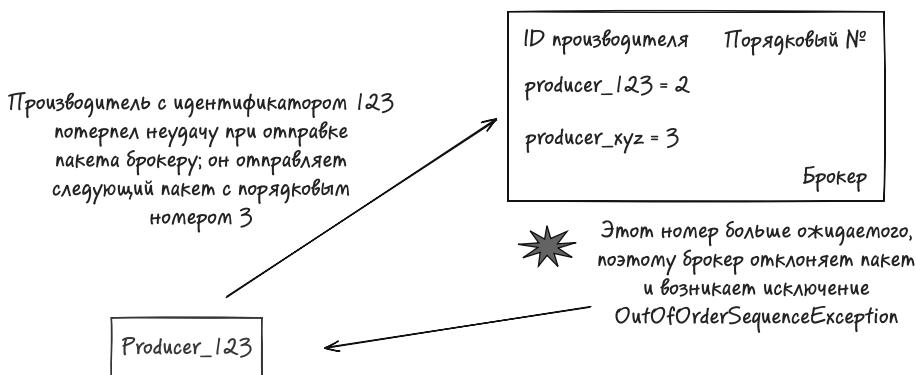
### ПРИМЕЧАНИЕ

Начиная с версии Apache Kafka 3.0, производители по умолчанию имеют настройки идемпотентности, поэтому вы получите все преимущества их использования сразу после установки без какой-либо дополнительной настройки.

Идемпотентный производитель использует две концепции для достижения упорядоченности и семантики доставки «точно один раз»: уникальные идентификаторы производителей (producer ID, PID) и порядковые номера сообщений. При инициализации идемпотентному производителю присваивается PID. Поскольку каждый новый идемпотентный производитель получает новый PID, идемпотентность производителей гарантируется только в границах одного сеанса работы производителя. Для заданного PID каждому пакету сообщений назначается монотонно возрастающий идентификатор (начиная с 0). Для каждого раздела, куда производитель отправляет записи, также назначается порядковый номер (рис. 4.21).



Брокер хранит в своей памяти список с порядковыми номерами для каждого раздела топика и PID. Если брокер получит порядковый номер, не превышающий *ровно на единицу* порядковый номер последней зафиксированной записи для указанного PID и раздела топика, то он отклонит запрос на производство (рис. 4.22).



Если порядковый номер пакета меньше ожидаемого, то это расценивается как ошибка дублирования, которую производитель игнорирует. Если порядковый номер больше ожидаемого, то запрос на производство приводит к исключению `OutOfOrderSequenceException`. Для идемпотентного производителя `OutOfOrderSequenceException` не является фатальной ошибкой, и он продолжит повторные попытки. Если при этом в процессе выполнения имеется более одного запроса, то брокер отклонит последующие запросы, а производитель вернется к началу и повторит отправку.

Итак, если вам требуется строгий порядок следования записей в разделе, то использование идемпотентного производителя является обязательным условием. А как быть, если возникнет необходимость атомарно выполнять запись в несколько разделов? В таком случае следует использовать транзакционный производитель, о котором мы поговорим далее.

#### 4.4.2. Транзакционный производитель

Транзакционный производитель позволяет выполнять запись в несколько разделов атомарно, когда запись во все разделы завершается успехом, или все они отменяются, если хотя бы одна потерпит неудачу. Когда лучше всего использовать транзакционный производитель? В любом сценарии, где недопустимы дублирующие записи, как, например, в финансовой отрасли.

Чтобы использовать транзакционный производитель, нужно присвоить уникальное значение конфигурационному параметру `transactional.id` производителя. Брокеры Kafka используют `transactional.id` для поддержки транзакций, выполняемых одним и тем же экземпляром производителя. Поскольку идентификатор должен быть уникальным для каждого производителя, а приложения могут иметь несколько производителей, то желательно придумать стратегию, согласно которой идентификаторы производителей будут представлять сегменты приложения, в которых они работают.

#### ПРИМЕЧАНИЕ

Транзакции Kafka — это обширная тема, заслуживающая отдельной главы. Поэтому я не буду подробно рассказывать, как работают транзакции. Те, кому это интересно, могут ознакомиться с оригинальным KIP (KIP расшифровывается как Kafka Improvement Proposal — «предложение по улучшению Kafka»): <http://mng.bz/9Qqq>.

Включение поддержки транзакций в производителе автоматически делает его идемпотентным. Вы можете создать идемпотентный производитель без поддержки транзакций, но обратное — создание транзакционного производителя без поддержки идемпотентности — невозможно. Рассмотрим пример в листинге 4.7. Возьмем наш предыдущий код и добавим в него поддержку транзакций.

После создания экземпляра транзакционного производителя необходимо вызвать метод `initTransactions()`, который отправит сообщение координатору транзакций (координатор транзакций — это брокер, обрабатывающий транзакции производителей), чтобы тот зарегистрировал `transactional.id` производителя для управления его последующими транзакциями.

### Листинг 4.7. Простой транзакционный KafkaProducer

```

HashMap<String, Object> producerProps = new HashMap<>();
producerProps.put("transactional.id", "set-a-unique-transactional-id"); ←

Producer<String, String> producer = new KafkaProducer<>(producerProps);
producer.initTransactions(); ← Вызов initTransactions

try {
    producer.beginTransaction(); ← Начинает транзакцию,
    producer.send(topic, "key", "value"); ← но пока не запускает отсчет
    producer.commitTransaction(); ← тайм-аута транзакции
} catch (ProducerFencedException | OutOfOrderSequenceException
| AuthorizationException e) {
    producer.close(); ← Отправка записи: на практике
} catch (KafkaException e) { ← вы будете отправлять несколько,
    producer.abortTransaction(); ← но здесь для простоты
    // в этой точке можно повторить попытку ← отправляется только одна
}
}

Фиксация
транзакции
после отправки
всех записей

```

Настраойка уникального идентификатора производителя. Обратите внимание, что этот идентификатор должен задать пользователь

Начинает транзакцию, но пока не запускает отсчет тайм-аута транзакции

Отправка записи: на практике вы будете отправлять несколько, но здесь для простоты отправляется только одна

Обработка фатальных исключений. В этом случае у вас только один выбор — закрыть и заново создать экземпляр производителя

Обработав нефатальное исключение, можно начать новую транзакцию с тем же производителем и повторить попытку

Если перед вызовом этого метода была запущена другая транзакция, то вызов блокируется до завершения этой транзакции. Внутренне он также извлекает некоторые метаданные, включая значение `epoch` (номер эпохи), которое этот производитель будет использовать в своих будущих транзакциях.

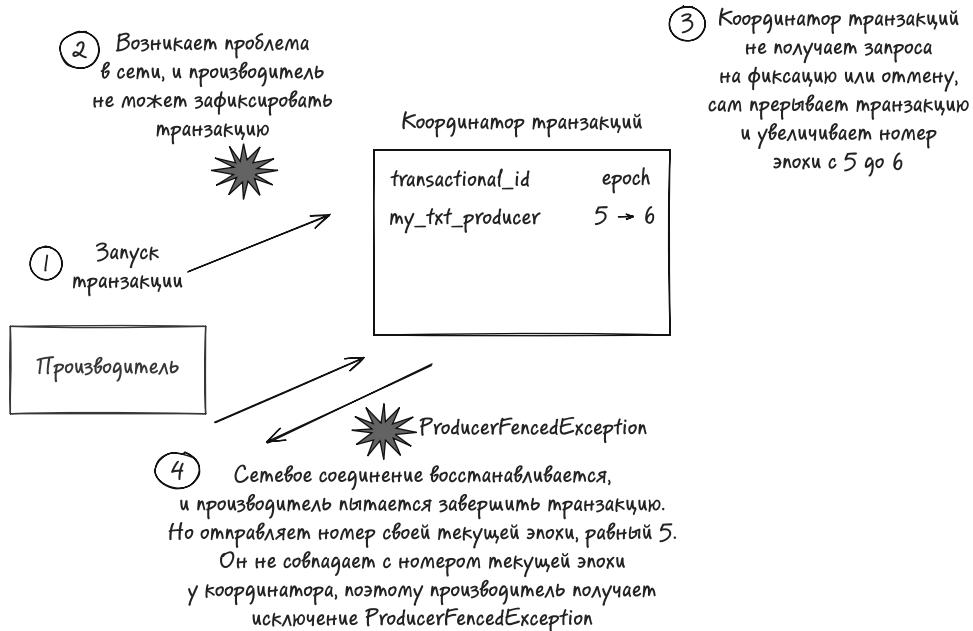
Перед отправкой записей нужно вызвать метод `beginTransaction()`, запускающий транзакцию. После запуска транзакции координатор будет ждать ее фиксации или отмены в течение периода, определяемого параметром `transaction.timeout.ms` (по умолчанию 1 минута). Если этого не произойдет, то координатор сам прервет транзакцию. Однако отсчет тайм-аута начинается только с момента отправки записей. Завершив создание и отправку записей, производитель должен зафиксировать транзакцию.

Обратите внимание на разницу в обработке ошибок между транзакционным и предыдущим нетранзакционным примером. При использовании транзакционного производителя не нужно проверять наличие ошибки ни с помощью `Callback`, ни путем проверки возвращаемого `Future`, потому что транзакционный производитель напрямую передаст их вашему коду для обработки.

Важно отметить, что любые исключения в первом блоке `catch` являются фатальными и, столкнувшись с ними, следует закрыть производитель. Чтобы продолжить работу, приложение должно создать новый экземпляр. Но при любом другом исключении можно попробовать повторить попытку, при этом следует прервать текущую транзакцию и начать отправку заново.

К фатальным исключениям относится `OutOfOrderSequenceException`, которое мы уже обсудили в подразделе 4.4.1, а также `AuthorizationException`. Теперь кратко обсудим `ProducerFencedException`. Kafka строго требует, чтобы в приложении имелся только один экземпляр производителя с данным `transactional.id`. Когда

запускается новый транзакционный производитель, он «отгораживается» (fences) от любого предыдущего производителя с тем же идентификатором и должен закрыться. Однако, как показано на рис. 4.23, `ProducerFencedException` можно получить еще в одном случае.



**Рис. 4.23.** Транзакции, заранее прерванные координатором транзакций, приводят к увеличению значения epoch, связанного с идентификатором транзакции

Когда производитель вызывает метод `producer.initTransactions()`, координатор транзакций увеличивает номер эпохи производителя — число, которое координатор транзакций связывает с идентификатором транзакции. Когда производитель запускает транзакцию, он передает свой идентификатор транзакции и номер эпохи. Если эпоха в запросе не совпадет с текущей эпохой, то координатор транзакций отклонит запрос с исключением `ProducerFencedException`.

Если производитель не сможет связаться с координатором до истечения тайм-аута, то, как уже говорилось, координатор прервет транзакцию и увеличит номер эпохи для этого идентификатора. Когда производитель восстановит соединение и попытается продолжить работу, то он окажется огражденным и его необходимо закрыть и запустить новый экземпляр.

#### ПРИМЕЧАНИЕ

Пример транзакционных производителей в форме вы найдете в файле `src/test/java/bvejcek/chapter_4/TransactionalProducerConsumerTest.java`.

Итак, я рассказал, как создавать транзакционные записи, а теперь перейдем к их потреблению.

### 4.4.3. Потребители в транзакциях

Потребители Kafka могут подписываться сразу на несколько топиков, часть из которых хранит транзакционные записи, а другая часть — нет. Потребляться должны только транзакционные записи, полученные в успешных транзакциях. К счастью, это легко реализовать простыми настройками. Чтобы настроить потребители для извлечения транзакционных записей, нужно присвоить конфигурационному параметру `isolation.level` значение `read_committed`, как показано в листинге 4.8 (некоторые детали опущены для простоты).

**Листинг 4.8.** Настройка KafkaConsumer для потребления транзакционных записей

```
HashMap<String, Object> consumerProps = new HashMap<>();
consumerProps.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
consumerProps.put(ConsumerConfig.GROUP_ID_CONFIG, "the-group-id");

consumerProps.put(ConsumerConfig.ISOLATION_LEVEL_CONFIG,
    "read_committed"); ← Настройка параметра изоляции в потребителе

consumerProps.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
    StringDeserializer.class);
consumerProps.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
    IntegerDeserializer.class);
```

Настройка этого параметра гарантирует, что потребитель будет извлекать только успешно зафиксированные транзакционные записи. Если в `isolation.level` задать значение `read_uncommitted`, то потребитель будет извлекать записи и из успешно зафиксированных, и из прерванных транзакций. Независимо от значения этого параметра потребитель гарантированно будет извлекать нетранзакционные записи. В режиме `read_committed` максимальное смещение извлекаемых записей может отличаться. Взгляните на рис. 4.24, поясняющий эту особенность.



**Рис. 4.24.** Высшая точка и последнее стабильное смещение в транзакционной среде

В Kafka существует понятие последнего стабильного смещения (last stable offset, LSO), выше которого не существует «определенных» (зафиксированных

транзакционных записей). Есть также понятие высшей точки. Высшая точка — это самое значительное смещение, успешно записанное во все реплики. В нетранзакционной среде LSO совпадает с высшей точкой, все записи считаются определенными или хранимыми и записываются немедленно. Но в случае с транзакциями смещение не может считаться определенным, пока транзакция не будет зафиксирована или отменена, поэтому LSO — это смещение первой открытой транзакции минус 1.

В нетранзакционной среде потребитель может вызовом `poll()` получить записи вплоть до высшей точки. Но в транзакционной среде он получит данные только до LSO.

#### ПРИМЕЧАНИЕ

В файле `src/test/java/bbejeck/chapter_4/TransactionalProducerConsumerTest.java` имеется несколько тестов, демонстрирующих поведение потребителя с конфигурациями `read_committed` и `read_uncommitted`.

К настоящему моменту мы рассмотрели использование производителя и потребителя по отдельности. Но есть еще один случай: совместное использование потребителя и производителя в транзакционном окружении.

### 4.4.4. Производители и потребители в транзакционном окружении

При создании приложений Kafka общепринятой практикой является потребление из топика, применение некоторых преобразований к записям и возврат результатов обратно в Kafka в другой топик. Записи считаются потребленными, когда потребитель фиксирует смещения. Как вы помните, фиксация смещений — это просто запись в топик (`_consumer_offsets`).

При выполнении цикла «потребление — преобразование — производство» следует убедиться, что фиксация смещений тоже производится в рамках транзакции. Иначе можно оказаться в ситуации, когда смещение зафиксировано, а транзакция отменяется. В таком случае после перезапуска приложение пропустит недавно обработанные записи, поскольку потребитель зафиксировал смещения.

Представьте, что у вас есть приложение, составляющее отчет по акциям и вам нужно сформировать отчет о соответствии брокера. Отчеты о соответствии должны отправляться только один раз, поэтому наилучшим подходом будет потребление транзакционных записей с информацией об акциях и составление отчета в рамках транзакции. Такой подход гарантирует, что отчет будет отправлен только один раз. Рассмотрим листинг 4.9 с кодом из файла `src/test/java/chapter_4/TransactionalConsumeTransformProduceTest.java` (некоторые детали опущены для простоты).

Самое существенное отличие этого кода от нетранзакционного приложения «потребление — преобразование — производство» заключается в отслеживании объектов `TopicPartition` и смещений записей. Это делается потому, что нам нужно передать смещения только что обработанных записей в вызов метода

`KafkaProducer.sendOffsetsToTransaction`. В приложениях, реализующих цикл «потребление — преобразование — производство», который выполняется в рамках транзакции, производитель отправляет смещения координатору группы потребителей, чем гарантирует фиксацию смещений только в случае успеха транзакции. Если транзакция завершается неудачей или прерывается, то смещения не фиксируются. Благодаря тому что производитель фиксирует смещения, отпадает необходимость в координации между производителем и потребителем в случаях отката транзакций.

**Листинг 4.9.** Пример цикла «потребление — преобразование — производство» в транзакционной среде

```
Map<TopicPartition, OffsetAndMetadata> offsets = new HashMap<>(); ← Создание HashMap для хранения смещений, подлежащих фиксации
producer.beginTransaction(); ← Запуск транзакции
consumerRecords.partitions().forEach(topicPartition -> {
    consumerRecords.records(topicPartition).forEach(record -> {
        lastOffset.set(record.offset());
        StockTransaction stockTransaction = record.value();
        BrokerSummary brokerSummary = BrokerSummary.newBuilder() ← Преобразование объекта StockTransaction в BrokerSummary
            .build();
        producer.send(new ProducerRecord<>(outputTopic, brokerSummary));
    });
    offsets.put(topicPartition, ← Сохранение TopicPartition и OffsetAndMetadata в HashMap
        new OffsetAndMetadata(lastOffset.get() + 1L)); ← Фиксация смещений записей,
}); try { ← потребленных в транзакции
    producer.sendOffsetsToTransaction(offsets, ← Фиксация транзакции
        consumer.groupMetadata());
    producer.commitTransaction();
}
```

Мы рассмотрели использование производителей и потребителей для отправки записей в топик Kafka и получения их из него. Но существует другой тип клиентов, использующих Admin API, позволяющий программно выполнять административные функции, связанные с топиками и группами потребителей.

## 4.5. ИСПОЛЬЗОВАНИЕ ADMIN API ДЛЯ ПРОГРАММНОГО УПРАВЛЕНИЯ ТОПИКАМИ

В Kafka имеется административный клиент для проверки топиков, брокеров, списков управления доступом (access control list, ACL) и конфигураций. В нем есть несколько функций для администрирования клиентов потребителей и производителей, однако я сосредоточусь на администрировании топиков и записей. Обычно в организации имеется отдельная группа, отвечающая за управление брокерами Kafka в промышленном окружении. Поэтому здесь я покажу, что вы сможете сделать, чтобы упростить тестирование или прототипирование приложений Kafka в среде разработки.

Для создания топиков с помощью административного клиента достаточно создать его экземпляр и выполнить команду создания топика (листинг 4.10).

#### Листинг 4.10. Создание топика

```
Map<String, Object> adminProps = new HashMap<>();
adminProps.put("bootstrap.servers", "localhost:9092");
try (Admin adminClient = Admin.create(adminProps)) { ← Создание экземпляра Admin. Обратите внимание на использование блока try-with-resources
    final List<NewTopic> topics = new ArrayList<>(); ← Список для хранения объектов NewTopic
    topics.add(new NewTopic("topic-one", 1, 1)); ← Создание объектов NewTopic
    topics.add(new NewTopic("topic-two", 1, 1));
    adminClient.createTopics(topics); ← Выполнение команды создания топиков
}
```

#### ПРИМЕЧАНИЕ

Под административным клиентом я подразумеваю интерфейс `Admin`. Существует также абстрактный класс `AdminClient`, но его рекомендуется избегать и использовать интерфейс `Admin`. В будущем класс `AdminClient` может быть удален из Kafka.

Этот код может пригодиться при прототипировании и создании новых приложений, потому что он гарантирует наличие топиков перед запуском кода. Давайте расширим этот пример и посмотрим, как получить список топиков и удалить один из них, если понадобится (листинг 4.11).

#### Листинг 4.11. Дополнительные операции с топиками

```
Map<String, Object> adminProps = new HashMap<>();
adminProps.put("bootstrap.servers", "localhost:9092");
try (Admin adminClient = Admin.create(adminProps)) { ← Код в этом примере перечисляет все топики в кластере, кроме внутренних. Обратите внимание, что для включения в список внутренних топиков нужно использовать объект ListTopicOptions и вызвать его метод ListTopicOptions.listInternal(true)
    Set<String> topicNames = adminClient.listTopics().names.get(); ← Вывод имен найденных топиков
    System.out.println(topicNames);
    adminClient.deleteTopics(Collections.singletonList("topic-two")); ← Удаление топика и повторное получение списка всех топиков. На этот раз в списке не должно быть недавно удаленного топика
}
```

Дополнительное замечание: `Admin.listTopics()` возвращает объект `ListTopicResult`. Получить имена топиков можно вызовом метода `ListTopicResult.names()`, который возвращает `KafkaFuture<Set<String>>`, поэтому далее вызывается метод `get()`, который блокируется до получения ответа на запрос административного клиента. В нашем примере эта команда выполняется немедленно, так как мы используем контейнер брокера, работающий на вашей локальной машине.

Можно также вызвать ряд других методов административного клиента, например, чтобы удалить записи и создать описание топиков. Они используются похожим образом, поэтому я не буду перечислять их здесь, но вы можете заглянуть в `src/test/java/bvejeck/chapter_4/AdminClientTest.java`, чтобы увидеть больше примеров использования административного клиента.

#### СОВЕТ

Мы работаем с брокером Kafka, запущенным в контейнере Docker на локальной машине, поэтому без всякого риска можем выполнять все операции с топиками и записями. Однако будьте осторожны при работе в сетевой среде и старайтесь не создавать проблем другим разработчикам. Кроме того, помните, что у вас может не быть возможности использовать административный клиент в промышленной среде. Никогда не пытайтесь изменять топики на лету в промышленных средах.

На этом мы завершаем знакомство с Admin API. В следующем и последнем разделе этой главы мы обсудим вопрос создания событий нескольких типов в одном топике.

## 4.6. ОБРАБОТКА НЕСКОЛЬКИХ ТИПОВ СОБЫТИЙ В ОДНОМ ТОПИКЕ

Допустим, мы создаем приложение для отслеживания активности на коммерческом сайте. Мы должны отследить последовательности щелчков кнопкой мыши, такие как вход, поиск и покупка. Здравый смысл требует поместить различные события (вход, поиск) и покупки в отдельные топики, поскольку они являются независимыми событиями. Но нам нужно узнать, как эти связанные события происходили в последовательности.

Для этого мы должны извлечь записи из разных топиков, а затем попытаться «сшить» их вместе в правильном порядке. Напомню, что Kafka гарантирует порядок следования записей в разделе топика, но не между разделами того же топика, не говоря уже о других топиках.

Есть ли другой подход? Да, есть: можно поместить эти различные события в один топик. Если предположить, что мы предоставляем согласованный ключ для всех типов событий, то мы сможем получить различные события по порядку в одном и том же разделе топика.

В конце главы 3 я рассказал, как использовать несколько типов событий в топике, но отложил подробное обсуждение, чтобы показать пример с производителями и потребителями. Теперь мы выясним, как безопасно производить и потреблять события нескольких типов с помощью Schema Registry.

В главе 3, а именно в разделе 3.6 о ссылках на схемы и нескольких событиях в топике, мы обсудили использование Schema Registry для поддержки нескольких типов событий в одном топике. В том разделе я не приводил пример использования производителя или потребителя, так как он более уместен в этой главе. Вот такой пример мы сейчас и рассмотрим.

**ПРИМЕЧАНИЕ**

Schema Registry уже обсуждался в главе 3, поэтому здесь я не буду делать никаких обзоров. Я буду упоминать некоторые термины, введенные в той главе, поэтому, если почувствуете, что вам нужно освежить память, возвращайтесь к их описанию там.

Начнем со стороны производителя.

### 4.6.1. Создание нескольких типов событий

Мы будем использовать следующую схему Protobuf:

```
syntax = "proto3";  
  
package bbejeck.chapter_4.proto;  
  
import "purchase_event.proto";  
import "login_event.proto";  
import "search_event.proto";  
  
option java_multiple_files = true;  
option java_outer_classname = "EventsProto";  
  
message Events {  
    oneof type {  
        PurchaseEvent purchase_event = 1;  
        LogInEvent login_event = 2;  
        SearchEvent search_event = 3;  
    }  
    string key = 4;  
}
```

Что получится, если сгенерировать код из этого определения Protobuf? Получится объект `Events` (листинг 4.12), содержащий одно поле `type`, принимающее один из трех возможных объектов событий (поле Protobuf `oneof`). Некоторые детали опущены для простоты.

**Листинг 4.12.** Пример создания KafkaProducer с использованием схемы Protobuf с полем `oneof`

```
...  
producerConfigs.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,  
    StringSerializer.class);  
producerConfigs.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,  
    KafkaProtobufSerializer.class); ← Настройка производителя на использование  
    ...                                         сериализатора Protobuf  
  
Producer<String, Events> producer = new KafkaProducer<>(  
    producerConfigs)); ← Создание экземпляра  
                           KafkaProducer
```

Поскольку Protobuf не позволяет использовать поле `oneof` как элемент верхнего уровня, производимые события всегда заключены во внешний контейнер. В результате код производителя выглядит так же, как при производстве событий одного типа.

Таким образом, общий тип для KafkaProducer и ProducerRecord — это класс внешнего сообщения Protobuf (в данном случае Events). Напротив, если бы мы использовали объединение Avro для схемы, как в примере в листинге 4.13, оно могло бы быть элементом верхнего уровня само по себе.

#### **Листинг 4.13.** Схема Avro для типа объединения

```
[  
    "bbejeck.chapter_3.avro.TruckEvent",  
    "bbejeck.chapter_3.avro.PlaneEvent",  
    "bbejeck.chapter_3.avro.DeliveryEvent"  
]
```

При использовании общего типа интерфейса для всех сгенерированных классов Avro ваш код производителя изменится, как показано в листинге 4.14 (некоторые детали опущены для простоты).

#### **Листинг 4.14.** Настройка KafkaProducer для использования схемы Avro типа объединения

```
producerConfigs.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,  
    StringSerializer.class);  
producerConfigs.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,  
    KafkaAvroSerializer.class);           ← Отключение автоматической регистрации схемы в производителе  
producerConfigs.put(AbstractKafkaSchemaSerDeConfig.AUTO_REGISTER_SCHEMAS,  
    false);                            ← Настройка выбора сериализатора Kafka Avro  
producerConfigs.put(AbstractKafkaSchemaSerDeConfig.USE_LATEST_VERSION,  
    true);    ← Настройка использования последней версии схемы  
Producer<String, SpecificRecord> producer = new KafkaProducer<>(  
    producerConfigs()) ← Создается экземпляр производителя
```

У нас нет внешнего класса, поэтому в данном примере каждое событие в схеме является конкретным классом: `TruckEvent`, `PlaneEvent` или `DeliveryEvent`. Для удовлетворения требований к универсальным классам в `KafkaProducer` необходимо использовать интерфейс `SpecificRecord`, поскольку его реализует каждый класс, сгенерированный Avro. Как мы видели в главе 3, при использовании ссылок на схемы Avro с объединением в роли записи верхнего уровня важно не забыть отключить автоматическую регистрацию схем и разрешить использование последней версии схемы.

Теперь перейдем на другую сторону уравнения — к потреблению событий нескольких типов.

### **4.6.2. Потребление событий нескольких типов**

В зависимости от выбранного подхода может потребоваться создать экземпляр `KafkaConsumer` с типом общего базового класса или интерфейса, реализующим все записи с разными типами событий, которые могут присутствовать в топике.

Первым рассмотрим использование Protobuf. Поскольку в этом случае всегда будет определяться класс-обертка, который будет указываться в параметре универсального типа, в данном примере — в параметре значения, как показано в листинге 4.15 (некоторые детали конфигурации для простоты опущены).

**Листинг 4.15.** Настройка потребителя для работы с несколькими типами событий в формате Protobuf

```
consumerProps.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
    KafkaProtobufDeserializer.class); ← Использовать десериализатор Protobuf
consumerProps.put(
    KafkaProtobufDeserializerConfig.SPECIFIC_PROTOBUF_VALUE_TYPE,
    Events.class); ← Установка десериализатора Protobuf, возвращающего конкретный тип

Consumer<Events> consumer = new KafkaConsumer<>(
    consumerProps); ← Создание KafkaConsume
```

Как уже демонстрировалось ранее, при настройке потребителя нужно указать, какой десериализатор использовать для получения определенного типа (в данном случае класса `Events`). При использовании схемы Protobuf с полем `oneof` сгенерированный код на Java включает методы, помогающие определить тип поля с помощью методов `hasXXX`. В нашем случае объект `Events` получит следующие три метода:

```
hasSearchEvent()
hasPurchaseEvent()
hasLoginEvent()
```

Сгенерированный код на Java также содержит перечисление с именем `<имя поля oneof>Case`. В этом примере мы дали полю `oneof` имя `type`, поэтому перечисление получит имя `TypeCase` и мы сможем читать его вызовом `Events.getTypeCase()`. Вот как можно использовать перечисление для быстрого определения типа базового объекта (некоторые детали опущены для простоты):

```
switch (event.getTypeCase()) { ← Инструкции case основаны
    case LOGIN_EVENT -> { ← на перечислении
        logins.add(event.getLoginEvent()); ← Вызовом метода getXXX извлекается
    }
    case SEARCH_EVENT -> { ← объект конкретного типа из числа
        searches.add(event.getSearchEvent()); ← перечисленных в поле oneof
    }
    case PURCHASE_EVENT -> {
        purchases.add(event.getPurchaseEvent());
    }
}
```

Выбор подхода для определения типа — это вопрос вкуса.

В листинге 4.16 мы настроим потребитель для работы с несколькими типами с помощью схемы-объединения Avro (некоторые детали конфигурации для простоты опущены).

**Листинг 4.16.** Настройка потребителя для работы со схемой-объединением Avro

```
consumerProps.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
    KafkaAvroDeserializer.class); ← Использовать десериализатор Avro
consumerProps.put(KafkaAvroDeserializerConfig.SPECIFIC_AVRO_READER_CONFIG,
    true); ← Установка десериализатора Avro,
    возвращающего конкретный тип
Consumer<SpecificRecord> consumer = new KafkaConsumer<>(consumerProps); ← Создание KafkaConsume
```

В конфигурации мы указали, что в роли десериализатора должен использоваться `KafkaAvroDeserializer`. Мы уже знаем, что в отличие от Protobuf и JSON Schema при

настройке десериализатора Avro мы требуем вернуть определенный тип класса, но не указываем имя класса. Поэтому при использовании Avro с топиками, в которых имеется несколько типов событий, потребитель должен использовать интерфейс `SpecificRecord` подобно тому, как показано в листинге 4.16.

При использовании интерфейса `SpecificRecord` для извлечения записей из вызова `Consumer.poll` нужно определить конкретный тип, чтобы выполнить какую-либо работу с ним, как показано в листинге 4.17 (некоторые детали для простоты опущены).

#### **Листинг 4.17.** Определение конкретного типа записи с использованием схемы-объединения Avro

```
SpecificRecord avroRecord = record.value();
if (avroRecord instanceof PlaneEvent) {
    PlaneEvent planeEvent = (PlaneEvent) avroRecord;
    ...
} else if (avroRecord instanceof TruckEvent) {
    TruckEvent truckEvent = (TruckEvent) avroRecord;
    ...
} else if (avroRecord instanceof DeliveryEvent) {
    DeliveryEvent deliveryEvent = (DeliveryEvent) avroRecord;
    ...
}
```

Этот подход напоминает работу с Protobuf, но на уровне класса, а не на уровне поля. При использовании Avro также можно смоделировать подход Protobuf и определить запись с полем, представляющим объединение, как показано в листинге 4.18.

#### **Листинг 4.18.** Схема Avro со встроенным в запись полем-объединением

```
{
    "type": "record",
    "namespace": "bbejeck.chapter_4.avro",
    "name": "TransportationEvent", ← Определение внешнего класса
    "fields" : [
        {"name": "txntype", "type": [ ← Тип-объединение Avro
            "bbejeck.chapter_4.avro.TruckEvent", ← на уровне поля
            "bbejeck.chapter_4.avro.PlaneEvent",
            "bbejeck.chapter_4.avro.DeliveryEvent"
        ]}
    ]
}
```

В этом случае сгенерированный код на Java содержит единственный метод `getTxnType()`, но он возвращает значение типа `Object`. По этой причине нужно использовать тот же подход с проверкой типа экземпляра, как и при применении схемы-объединения, только теперь задача определения типа записи переносится с уровня класса на уровень поля.

#### **ПРИМЕЧАНИЕ**

В Java 16 появилась поддержка ключевого слова `instanceof` в операции сопоставления с шаблоном, которая устраниет необходимость приведения типа объекта после проверки `instanceof`.

## ИТОГИ ГЛАВЫ

- Производители Kafka отправляют записи брокерам Kafka пакетами и продолжают повторять попытки отправки неудачных пакетов, пока не истечет время, указанное в конфигурационном параметре `delivery.timeout.ms`. В производителе Kafka можно настроить идемпотентный режим работы, который гарантирует отправку записей точно один раз и строго по порядку. Производители Kafka также имеют транзакционный режим, гарантирующий доставку записей в несколько топиков точно один раз. Поддержка транзакционного режима включается в производителях с помощью конфигурационного параметра `transactional.id`, которому в каждом производителе нужно присвоить уникальный идентификатор. При настройке потребителей для работы в транзакционном режиме нужно присвоить параметру `isolation.level` значение `read_committed`, чтобы они потребляли только зафиксированные записи из транзакционных топиков.
- Потребители Kafka читают записи из топиков. При наличии нескольких потребителей с одинаковым идентификатором группы между ними распределяются разделы для чтения, и они работают вместе как один логический потребитель. Если один член группы выйдет из строя, то назначенные ему разделы топиков будут перераспределены между другими членами группы с помощью процесса, известного как перебалансировка. Потребители периодически фиксируют смещения потребляемых записей, поэтому при перезапуске после выключения они возобновляют обработку с того места, где остановились.
- Производители и потребители Kafka поддерживают три семантики доставки: «не менее одного раза», «не более одного раза» и «точно один раз». Семантика «не менее одного раза» означает, что никакие записи не будут потеряны, но из-за повторных попыток отправки могут быть получены дубликаты. Семантика «не более одного раза» означает, что никаких дубликатов записей не будет, но некоторые записи могут быть потеряны из-за ошибок. Семантика доставки «точно один раз» означает, что никаких дубликатов записей не будет и никакие записи не потеряются из-за ошибок.
- Статическое членство обеспечивает стабильность в условиях, когда потребители часто отключаются, а затем через довольно короткий промежуток времени возвращаются.
- `CooperativeStickyAssignor` обеспечивает более оптимальное поведение перебалансировки. Кооперативный протокол перебалансировки — лучший выбор в большинстве случаев, так как он значительно сокращает время простоя во время перебалансировки.
- `Admin API` предоставляет возможность программного управления топиками, разделами и записями.
- Если имеются события нескольких типов, но эти события тесно взаимосвязаны и их обработка по порядку имеет большое значение, то стоит рассмотреть возможность их размещения в одном топике. Вы можете настроить производитель Kafka как идемпотентный производитель, что означает, что он гарантирует отправку записей только один раз и по порядку для заданного раздела.

# 5

## *Kafka Connect*

### **В этой главе**

- ✓ Начало работы с Kafka Connect.
- ✓ Применение преобразований к отдельным сообщениям.
- ✓ Создание и развертывание собственного коннектора.
- ✓ Динамическое управление коннектором с помощью потока мониторинга.
- ✓ Создание нестандартного преобразования.

В этой главе вы узнаете, как быстро перемещать события в платформу Apache Kafka и из нее. Несмотря на то что Kafka может функционировать как центральная нервная система для данных, в первую очередь она обеспечивает изолированный и централизованный доступ к данным. Однако есть и другие важные сервисы, такие как полнотекстовый поиск, генерация отчетов и анализ данных, которые могут предоставляться приложениями, предназначенными для этих целей. Важно понимать, что никакая технология или приложение в одиночку не сможет удовлетворить все потребности бизнеса или организации.

В предыдущих главах мы установили, что Kafka упрощает архитектуру технологической компании, получая события один раз, предоставляя возможность любым группам в организации потреблять эти события независимо. В ранее упомянутых случаях, когда потребителем является другое приложение, нужно написать потребитель специально для этого приложения. Вам придется повторно написать большой объем кода, если у вас несколько таких приложений. Было бы здорово иметь фреймворк, который бы обрабатывал передачу данных из внешнего приложения в Kafka или из Kafka во внешнее приложение. И такой фреймворк существует! Это важнейший компонент Apache Kafka: Kafka Connect.

## 5.1. ВВЕДЕНИЕ В KAFKA CONNECT

Фреймворк Kafka Connect – это часть проекта Apache Kafka. Он обеспечивает возможность интеграции Kafka с другими системами, такими как реляционные базы данных, поисковые системы, хранилища NoSQL, облачные хранилища и системы хранилищ данных. Connect позволяет быстро передавать большие объемы данных в Kafka и из нее. Возможность такой потоковой интеграции имеет решающее значение для устаревших систем в современных приложениях потоковой передачи событий.

С помощью Connect можно организовать двунаправленную передачу данных между существующими архитектурами и новыми приложениями. Например, можете передать входящие события из Kafka в типичное приложение Model – View – Controller (MVC), используя коннектор для записи результатов в реляционную базу данных. То есть мы можем рассматривать Kafka Connect как своего рода «клей», позволяющий бесшовно «склеивать» различные приложения с новыми источниками данных и событий.

Один из конкретных примеров применения Kafka Connect – захват изменений в таблице базы данных по мере их появления, называемый *захватом изменений в данных* (change data capture, CDC). CDC экспортирует изменения в таблице базы данных (`INSERT`, `UPDATE` и `DELETE`) в другие приложения. С помощью CDC можно сохранять изменения в топике Kafka, делая их доступными для потребителей. Одна из лучших сторон этой интеграции – отсутствие необходимости менять старое приложение.

В настоящий момент вы уже можете реализовать это самостоятельно, используя клиенты-производители и клиенты-потребители. Но при этом вам придется проделать массу работы, чтобы подготовить приложение к работе в промышленной среде. Не говоря уже о том, что каждая система, из которой вы потребляете или в которую записываете данные, потребует индивидуального подхода.

Потребление изменений из реляционной базы данных отличается от потребления из хранилища NoSQL, а создание записей в ElasticSearch отличается от создания записей в хранилище Amazon S3. По этой причине имеет смысл использовать проверенное решение с несколькими готовыми компонентами. В Confluent (<https://www.confluent.io/hub/>) вы найдете сотни коннекторов. Некоторые из них являются коммерческими и требуют оплаты, но существует более 100 коннекторов, доступных бесплатно.

Connect запускается в кластере Kafka как один или несколько экземпляров приложения и экспортирует коннекторы как плагины, которые настраиваются в пути поиска классов classpath. Для запуска коннектора не требуется писать код, достаточно передать серверу Connect файл конфигурации в формате JSON. Если для ваших целей нет доступного коннектора, то вы можете написать свой коннектор. Создание собственного коннектора мы рассмотрим в последнем разделе этой главы.

Важно понимать, что коннекторы делятся на два типа: источники и приемники. Коннектор-источник будет потреблять данные из внешнего источника, такого как база данных Postgres или MySql, MongoDB или контейнер S3, и записывать их в топики Kafka. Коннектор-приемник выполняет обратную операцию, извлекая события из топика Kafka и пересылая их во внешнее приложение, такое как Elasticsearch или Google BigQuery. Кроме того, благодаря конструкции Kafka можно одновременно

иметь несколько коннекторов-приемников, экспортирующих данные из топика Kafka. То есть можно взять существующее приложение и с помощью комбинации коннекторов-источников и коннекторов-приемников передать его данные другим системам, не внося никаких изменений в исходное приложение.

Наконец, Connect позволяет изменять данные, поступающие в Kafka или извлекаемые из нее. Преобразования отдельных сообщений (single message transforms, SMT) позволяют изменять формат записей при их сохранении в Kafka. Если понадобится изменить формат записи так, чтобы он соответствовал требованиям целевой системы, то в таких случаях тоже можно использовать SMT. Например, при импорте данных клиентов из таблицы в БД можно замаскировать конфиденциальные сведения с помощью преобразования `MaskField`. Также есть возможность изменять формат данных.

Представьте, что вы используете формат Protobuf в кластере Kafka. У вас есть таблица в БД, которая передает данные в топик через коннектор-источник, а для целевого топика у вас также есть коннектор-приемник, пересылающий записи в хранилище ключей и значений Redis. Ни база данных, ни Redis не поддерживают формат Protobuf. Но с помощью преобразования значений можно легко преобразовать в формат Protobuf записи, поступающие из коннектора-источника. Коннектор-приемник в этой схеме будет использовать другой преобразователь, превращающий исходящие записи из формата Protobuf обратно в простой текст.

Наконец, в этой главе вы узнаете, как развернуть и использовать Kafka Connect для интеграции внешних систем с Kafka, что позволит вам создавать конвейеры потоковой передачи событий, как применять SMT для изменения входящих или исходящих записей и как реализовать свои преобразования. Наконец, мы рассмотрим создание собственного коннектора, если ни один из существующих не соответствует вашим требованиям.

## 5.2. ИНТЕГРАЦИЯ ВНЕШНИХ ПРИЛОЖЕНИЙ С KAFKA

К настоящему моменту вы узнали, что такое Connect и что он действует как «клей», позволяющий «склеивать» различные системы вместе, а теперь рассмотрим его работу на примере.

Представьте, что вы отвечаете за координацию регистрации новых студентов на ознакомительные лекции в колледже. С помощью веб-формы, которой колледж пользуется уже некоторое время, студенты указывают, какие ознакомительные лекции они будут посещать. Раньше считалось, что по прибытии студентов на ознакомление с ними должны встретиться представители факультетов, жилищной службы, службы питания и руководства. Анкета с информацией, указанной студентами, распечатывалась и раздавалась сотрудникам каждого факультета.

Однако этот процесс был чреват ошибками и мог занять много времени, поскольку другие сотрудники видели информацию о студентах впервые, а определение логистики для каждого студента на месте требует времени. Большим облегчением было бы наличие процесса, позволяющего передавать информацию сразу же после регистрации студента. Вы узнали, что недавно университет запустил процесс технологической модернизации, положив в основу платформу Kafka.

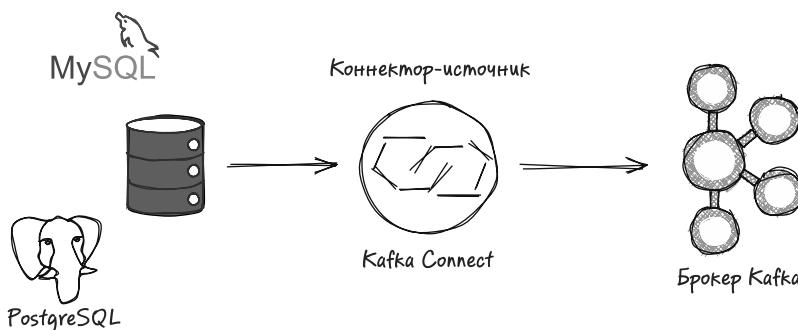
В настоящее время отдел по работе со студентами, ответственный за получение ориентировочной информации, использует базовое веб-приложение, работающее с базой данных PostgreSQL. Его сотрудники выразили нежелание что-то менять, и это поначалу показалось вам проблемой, но потом вы поняли, что их данные довольно легко можно интегрировать в новую архитектуру потоковой передачи событий.

Информацию о владельце регистрации можно отправлять непосредственно в раздел Kafka, используя коннектор-источник JDBC. После этого другие отделы смогут настроить приложения-потребители для получения этих данных.

### ПРИМЕЧАНИЕ

Точка интеграции — это топик Kafka, когда для получения данных из других источников используется Kafka Connect. Это означает, что *любое* приложение, использующее KafkaConsumer (включая Kafka Streams), сможет использовать импортированные данные.

На рис. 5.1 показано, как работает интеграция базы данных с Kafka. В этом случае Kafka Connect применяется для мониторинга таблицы базы данных и потоковой передачи обновлений в топик Kafka.



**Рис. 5.1.** Интеграция таблицы БД и топика Kafka с помощью Kafka Connect

Теперь, приняв решение использовать Kafka Connect для передачи информации о регистрации студентов в новую платформу потоковой обработки событий, развернутую в университете, рассмотрим простой рабочий пример (см. следующий раздел). Затем мы более подробно исследуем работу Connect и обсудим основные концепции.

## 5.3. НАЧАЛО РАБОТЫ С KAFKA CONNECT

Kafka Connect работает в двух режимах: распределенном и автономном. Распределенный режим хорошо подходит для большинства промышленных окружений, позволяя использовать параллелизм и отказоустойчивость, которые обеспечивает запуск нескольких экземпляров Connect. Автономный режим лучше подходит для разработки на локальной машине или ноутбуке.

## НАСТРОЙКА KAFKA CONNECT

Мы не будем устанавливать и запускать Kafka Connect на локальном компьютере, а используем образ Docker. Помните, Kafka Connect работает независимо от брокера Kafka, поэтому нам нужно настроить новый файл `docker-compose` и добавить в него сервис Connect. Это упростит разработку и позволит сосредоточиться на обучении.

При настройке Kafka Connect нужно предоставить два набора конфигурации. Один предназначен для сервера Connect (worker — «рабочий процесс»), а другой — для отдельного коннектора. Я только что употребил новый термин: рабочий процесс Connect. Мы рассмотрим его значение, а также ряд базовых понятий Connect в следующем разделе.

Теперь перечислю некоторые ключевые конфигурационные параметры Kafka Connect.

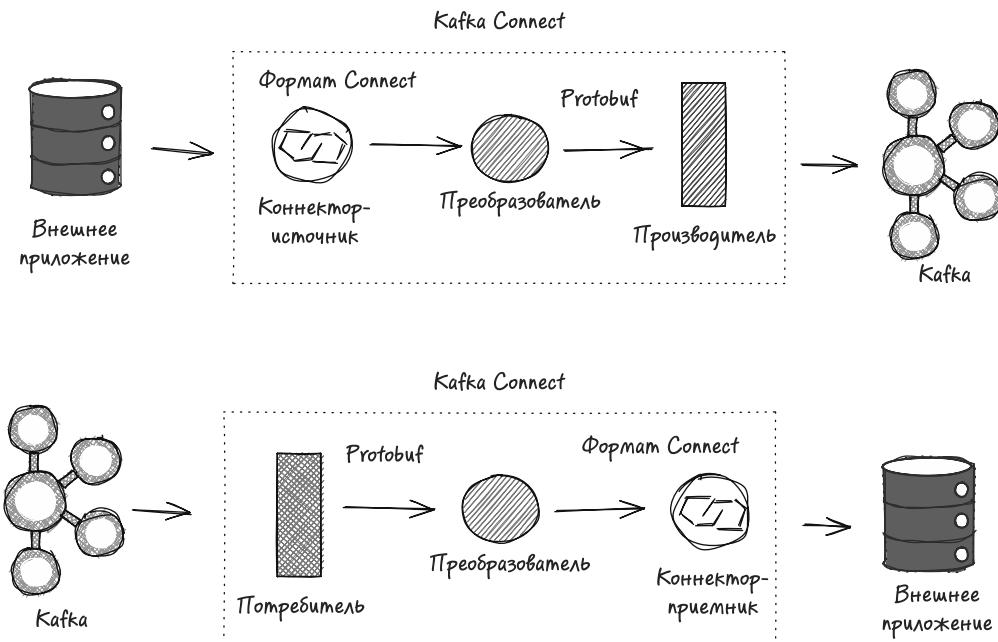
- `key.converter` — класс преобразователя, управляющего сериализацией ключа из формата Connect в формат, в котором данные записываются в Kafka. В нашем примере мы будем использовать встроенный `org.apache.kafka.connect.storage.StringConverter`. Этот параметр задает преобразователь по умолчанию для коннекторов, созданных на этом сервере Connect, но имейте в виду, что сами коннекторы могут переопределять это значение.
- `value.converter` — класс преобразователя, управляющего сериализацией значения из формата Connect в формат, в котором данные записываются в Kafka. В нашем примере мы будем использовать встроенный `org.apache.kafka.connect.json.JsonConverter`. Как и в случае с преобразователем ключей, этот параметр задает преобразователь по умолчанию, но сами коннекторы могут использовать другую настройку.
- `plugin.path` — сообщает Connect местоположение плагинов, таких как коннекторы и преобразователи, и их зависимости от сервера.
- `group.id` — идентификатор всех потребителей в подключенном кластере; этот параметр используется только для коннекторов-источников.

## ПРИМЕЧАНИЕ

Эти конфигурационные параметры рассматриваются здесь лишь для полноты картины. Вам не придется настраивать их, так как они уже присутствуют в файле `docker-compose`. Параметры в файле определяются под именами, такими как `CONNECT_PLUGIN_PATH` для `plugin.path`. Файл `docker-compose` также запустит экземпляр БД Postgres и автоматически заполнит таблицу значениями, необходимыми для опробования примера.

Хочу кратко объяснить, что такое преобразователь. Connect использует преобразователь для изменения формата данных, полученных и созданных коннектором-источником, в формат Kafka или для преобразования из формата Kafka в формат, поддерживаемый внешней системой, как в случае с коннектором-приемником (рис. 5.2).

Поскольку эту настройку можно изменить индивидуально для отдельных коннекторов, любой коннектор может работать с любым форматом сериализации. Например, один коннектор может использовать Protobuf, а другой — JSON.



**Рис. 5.2.** Конвертер Connect преобразует данные, перед тем как они попадут в Kafka или после того как они покинут ее

Далее рассмотрим некоторые конфигурации для коннекторов. В нашем примере мы используем коннектор JDBC, поэтому в конфигурации коннектора необходимо указать параметры подключения к БД, например имя пользователя и пароль, а также задать условие, руководствуясь которым коннектор будет определять, какие записи импортируются в Kafka. Рассмотрим наиболее важные конфигурационные параметры коннектора JDBC:

- `connector.class` — класс коннектора;
- `connection.url` — URL подключения к базе данных;
- `mode` — метод отслеживания изменений, который будет использовать коннектор-источник JDBC;
- `timestamp.column.name` — имя столбца, по которому отслеживаются изменения;
- `topic.prefix` — Connect записывает все таблицы в топики с названиями «`топик.префикс + имя_таблицы`».

Большинство из этих настроек довольно просты, но две из них — `mode` и `timestamp.column.name` — нам придется обсудить подробнее, поскольку они играют важную роль в работе коннектора. Коннектор-источник JDBC по значению параметра `mode` определяет, какие записи загружать.

В этом примере мы используем значение `timestamp` в параметре `mode`. Соответственно, выбор записей для загрузки будет осуществляться с помощью параметра `timestamp.column.name`, ссылающегося на столбец с отметкой времени. Мы знаем, что оператор `INSERT` автоматически устанавливает отметку времени, но дополнительно

добавили триггер в Docker-образ базы данных, который обновляет отметку времени при выполнении операторов `UPDATE`. Используя эту отметку, коннектор будет извлекать из таблицы только те записи, отметка времени в которых больше последней, сохраненной в предыдущем сеансе импортирования.

В параметре `mode` можно также указать значение `incrementing`. В этом случае выборка записей будет производиться по столбцу с автоматически увеличивающимся числовым значением, то есть выбираться будут только новые записи. Коннектор JDBC определяет записи для импортирования, используя несколько конфигурационных параметров. За кулисами коннектор будет запрашивать базу данных и передавать результаты в Kafka. В этой главе я не буду вдаваться в подробности о коннекторе JDBC. Но это не значит, что мы рассмотрели все, что нужно знать. Коннектору JDBC можно было бы посвятить целую главу. Наиболее важным моментом является необходимость определить конфигурацию для каждого отдельного коннектора, и многие из них предлагают богатый набор параметров управления их поведением.

Теперь посмотрим, как запустить отдельный коннектор. Это легко сделать с помощью Connect REST API. В листинге 5.1 показан пример запуска коннектора.

#### Листинг 5.1. Вызов REST API для запуска коннектора

```
curl -i -X PUT http://localhost:8083/connectors/
  jdbc_source_connector_example/config \
  -H "Content-Type: application/json" \
  -d '{  

    "connector.class": "io.confluent.connect.jdbc.JdbcSourceConnector",
    "connection.url": "jdbc:postgresql://postgres:5432/postgres",
    "connection.user": "postgres",
    "connection.password": "postgres",
    "mode": "timestamp",
    "timestamp.column.name": "ts",  

    "topic.prefix": "postgres_",
    "value.converter": "org.apache.kafka.connect.json.JsonConverter", ←
    "value.converter.schemas.enable": "false", ←
    "tasks.max": "1" ←
  }'  

  ↑ Предписывает использовать  

  JsonConverter для преобразования  

  входных записей в форматJSON  

  ↑ Отключает использование  

  схемы для записей  

  ↑ Определяет максимальное  

  количество задач для коннектора
```

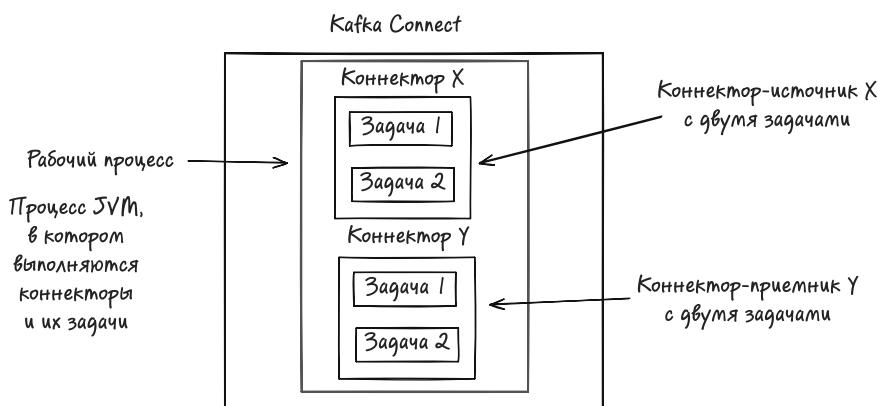
Этот вызов REST API запустит коннектор JDBC. Здесь есть несколько конфигурационных параметров, на которые я хотел бы обратить ваше внимание. Параметр `value.converter` указывает, какой преобразователь использовать для преобразования входных записей в формат JSON. Здесь также присутствует параметр `value.converter.schemas.enable`, которому присваивается значение `false`, что означает, что преобразователь не будет сохранять схему из коннектора в содержимом сообщения.

Напомню, что при использовании преобразователя JSON схема прикрепляется к каждой входной записи, что может значительно увеличить ее размер. Мы можем отключить добавление схемы, выявленной преобразователем, потому что запись осуществляется в Kafka. Но при потреблении из топика Kafka для записи во внешнюю систему, в зависимости от коннектора, необходимо включить автоматическое определение схемы, чтобы Connect мог правильно интерпретировать массивы байтов, хранящиеся в Kafka. Лучшим подходом для преобразователя значений было бы

использовать Schema Registry (и форматы Avro, Protobuf или JSONSchema). Мы рассмотрели схемы и Schema Registry в главе 3, поэтому я не буду снова рассматривать эти детали здесь.

В коде также можно видеть параметр `tasks.max`. На текущий момент мы знаем, что Kafka Connect будет использоваться для импортирования данных из внешней системы в Kafka или для экспортации данных из Kafka в другое приложение. Вы только что рассмотрели преобразователь JSON, необходимый коннектору, но он не выполняет извлечение или передачу данных, а отвечает за запуск ряда задач, занимающихся перемещением данных.

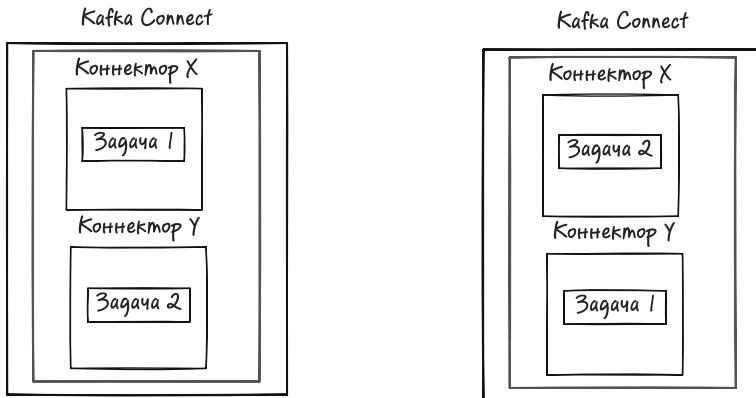
Выше в главе упоминались два типа коннекторов: `SourceConnector` и `SinkConnector`. Они используют два соответствующих вида задач: `SourceTask` и `SinkTask`. Основная задача коннектора — генерировать конфигурации для задач. При работе в распределенном режиме фреймворк Connect будет распределять и запускать их в разных рабочих процессах в кластере Connect. Каждый экземпляр задачи будет работать в своем собственном потоке. Обратите внимание, что параметр `tasks.max` не гарантирует запуск указанного числа задач — коннектор сам определит нужное количество. Теперь было бы полезно взглянуть на схему, иллюстрирующую взаимосвязи между коннекторами и задачами (рис. 5.3).



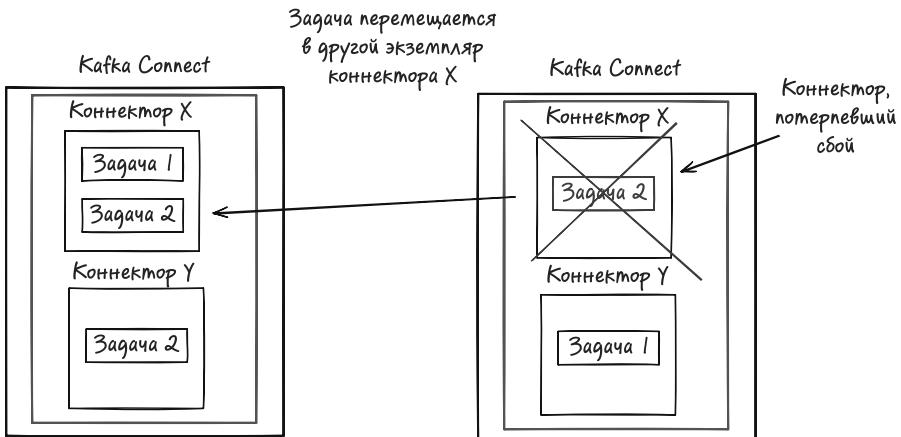
**Рис. 5.3.** Connect в автономном режиме; все задачи находятся в одном рабочем процессе

Здесь мы рассматриваем работу Kafka Connect в автономном режиме. У нас есть один рабочий процесс, процесс JVM, отвечающий за запуск коннекторов и их задач. Теперь рассмотрим распределенный режим (рис. 5.4).

Как показано здесь, в распределенном режиме задачи распределяются между рабочими процессами в кластере Connect. Распределенный режим не только дает более высокую пропускную способность за счет распределения нагрузки, но и обеспечивает возможность продолжать обработку в случае отказа коннектора. Рассмотрим еще одну схему, иллюстрирующую, что это означает (рис. 5.5).



**Рис. 5.4.** Connect в распределенном режиме; задачи распределяются по нескольким экземплярам коннектора



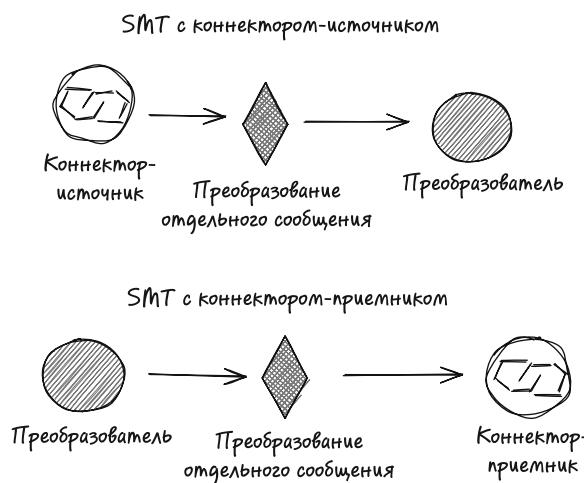
**Рис. 5.5.** Connect в распределенном режиме обеспечивает отказоустойчивость

Как можно судить по этой иллюстрации, если рабочий процесс Kafka Connect перестает работать, то экземпляры задач коннектора из этого рабочего процесса будут назначены другим рабочим процессам в кластере. Из всего вышесказанного можно сделать следующий вывод: автономный режим отлично подходит для прототипирования и разработки с использованием Kafka Connect, но в промышленных системах лучше использовать распределенный режим как обеспечивающий высокую отказоустойчивость — задачи из неисправных рабочих процессов назначаются оставшимся рабочим процессам в кластере. Обратите внимание, что в распределенном режиме необходимо выполнить вызов REST API для запуска коннектора на каждой машине в кластере Connect.

Но вернемся к нашему примеру. Мы запустили коннектор, но есть кое-что, что хотелось бы сделать по-другому. Во-первых, входные записи не имеют ключей, а это проблема, потому что записи, соответствующие одному и тому же студенту, могут оказаться в разных разделах. Поскольку Kafka гарантирует упорядоченность только в пределах раздела, несколько обновлений, сделанных студентом, могут быть обработаны не по порядку. Во-вторых, студенты вводят свои номера социального страхования. Важно ограничить видимость этих данных, поэтому было бы хорошо изменить или замаскировать их до того, как они попадут в Kafka. К счастью, Connect предлагает простое, но мощное решение: преобразования.

## 5.4. ПРИМЕНЕНИЕ ПРЕОБРАЗОВАНИЙ ОТДЕЛЬНЫХ СООБЩЕНИЙ

Под преобразованием отдельных сообщений (single message transforms, SMT) понимаются простые изменения записей до того, как они попадут в Kafka или будут переданы внешним системам, в зависимости от того, применяется ли преобразование в коннекторе-источнике или коннекторе-приемнике. Главное в преобразованиях — выполняемая работа должна быть простой, то есть они применяются только к одной записи за раз (без объединений или агрегирования), а вариант использования должен быть применен исключительно для коннекторов, а не для пользовательских приложений производителей или потребителей. Для чего-то более сложного лучше использовать Kafka Streams или ksqlDB, специально созданные для сложных операций. На рис. 5.6 показано, как происходят преобразования.



**Рис. 5.6.** Преобразователь изменяет формат данных до того, как они попадут в Kafka, или после того, как покинут ее

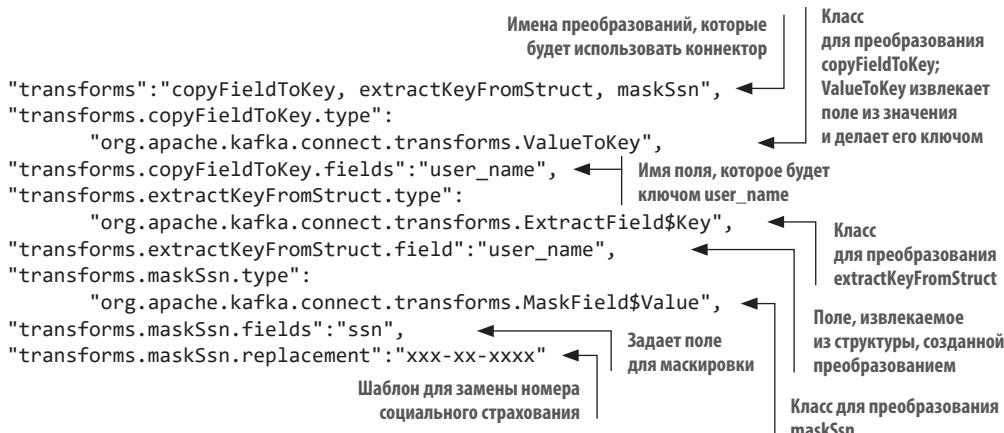
Как видите, роль SMT — находится между коннектором и преобразователем. Для коннектора-источника операция преобразования применяется к записи до того,

как она попадет в преобразователь, а для коннектора-приемника — после выхода ее из преобразователя. В обоих случаях SMT работают с данными в одном и том же формате, поэтому большинство SMT одинаково хорошо работают как с коннекторами-источниками, так и с коннекторами-приемниками.

Connect имеет несколько встроенных преобразований SMT, охватывающих широкий спектр вариантов использования. Например, есть SMT для фильтрации записей, развертывания вложенных структур или удаления поля. Я не буду перечислять здесь все, а желающие смогут найти полный список SMT в документации Kafka Connect (<http://mng.bz/84VP>).

Мы будем использовать три преобразования: `ValueToKey`, `ExtractField` и `MaskField`. Для использования встроенных SMT необходимо добавить некоторую конфигурацию JSON. Если встроенных преобразований окажется недостаточно, то можно написать свое преобразование. Немного ниже в этой главе мы рассмотрим пример создания своего SMT. В листинге 5.2 представлен полный код JSON, который мы будем использовать для добавления необходимых преобразований.

### Листинг 5.2. Конфигурация преобразования JSON



Большая часть конфигурации преобразования выглядит просто. В ней есть список имён, разделенных запятыми, каждое из которых представляет одно преобразование.

#### СОВЕТ

Порядок имён в элементе `transforms` отнюдь не произвольный. Имена преобразований перечислены в порядке их применения, поэтому важно учитывать, как каждое преобразование изменит данные, проходящие через цепочку, и как их порядок повлияет на результат.

Затем для каждого имени в списке указывается класс, реализующий преобразование, и поле, к которому оно будет применяться. Но есть кое-что, на что я хотел бы обратить особое внимание. С помощью преобразования `copyFieldToKey` мы указали,

что в роли ключа каждой записи в Kafka будет использоваться столбец `user_name`. Но в результате будет создана структура с одним полем, как показано в листинге 5.3.

### Листинг 5.3. Структура

```
Struct {"user_name" : "artv"}
```

Но нам нужно значение поля `user_name` структуры, поэтому мы также применяем преобразование `ExtractField`. В конфигурации мы должны указать преобразование, извлекающее ключ, например `ExtractField$Key`. Connect применит второе преобразование, и в итоге получится ключ для входной записи в виде одиночного значения.

Я хочу отметить еще кое-что касающееся преобразований, что может остаться незамеченным. Для обработки одного и того же поля можно применить несколько преобразований, как показано в нашем примере, где сначала поле копируется в структуру, а затем извлекается из нее для копирования в ключ. Но здесь важно найти правильный баланс: если цепочка включает более двух преобразований, то имеет смысл подумать об использовании Kafka Streams для выполнения преобразований, так как это будет более эффективно.

Последнее преобразование, которое мы используем, — это `MaskField`, применяемое к полю с номером социального страхования студента. И снова, как можно видеть в конфигурации, мы указали, что хотим замаскировать значение с помощью преобразования `MaskField$value`. В данном случае мы указали, что строка номера социального страхования должна замещаться строкой символов `x`, то есть в результате должна получаться строка `xxx-xx-xxxx`. В преобразовании `MaskField` также можно не указывать конкретную замену, и тогда оно будет использовать пустое значение в зависимости от типа поля — пустую строку для строкового поля или `0` для числового.

Теперь мы получили полностью настроенный коннектор, который будет проверять наличие изменений в базе данных и импортировать их в Kafka, давайте сделаем нашу реляционную базу данных частью платформы потоковой обработки событий!

### ПРИМЕЧАНИЕ

Выше мы говорили о коннекторе JDBC. Я хочу отметить, что есть несколько крайних случаев, когда коннектор JDBC не сможет переместить последние изменения в Kafka. Я не буду вдаваться в них здесь, но рекомендую взглянуть на коннектор Debezium, предназначенный для интеграции реляционных баз данных с Apache Kafka (<http://mng.bz/E9JJ>). Вместо использования поля с монотонно увеличивающимся значением или отметкой времени Debezium использует журнал изменений базы данных для выборки изменений, которые должны попасть в Kafka.

Чтобы выполнить только что описанный пример, обратитесь к файлу `README` в каталоге с исходным кодом примеров для главы 5. Там вы найдете также пример коннектора-приемника, но мы не будем его рассматривать, потому что он развертыивается, настраивается и запускается аналогично.

К настоящему моменту мы настроили коннектор-источник, но нам также понадобится отправлять события из Kafka во внешнюю систему. Для этого мы используем коннектор-приемник, а именно коннектор-приемник для Elastic Search.

## 5.5. ДОБАВЛЕНИЕ КОННЕКТОРА-ПРИЕМНИКА

Еще одна возможность, которую вы предложили добавить, — возможность для поступающих студентов найти потенциального соседа по комнате на основе входных данных. Когда студенты записываются на ознакомительные лекции, то частью процесса можно сделать ввод нескольких ключевых слов и получение ссылок на других студентов, имеющих похожие предпочтения.

Предложенная вами идея была воспринята с энтузиазмом, но вопросы о том, как получить данные, показались слишком сложными. Команда хотела бы избежать создания нового конвейера для сбора информации, поэтому они отложили идею. Но теперь, после добавления Kafka и Connect для импорта входящей информации о студентах, чтобы предоставить такую возможность поиска с помощью Elasticsearch (<https://www.elastic.co/>), достаточно лишь добавить коннектор-приемник.

К счастью, для Elasticsearch уже есть готовый коннектор, поэтому нам остается только установить JAR-файлы в работающий кластер Connect. Итак, чтобы начать работу с коннектором для Elasticsearch, достаточно выполнить вызов REST API, как показано в листинге 5.4.

### Листинг 5.4. Вызов REST API для запуска коннектора для Elasticsearch

```
$ curl -i -X PUT localhost:8083/connectors/
  student-info-elasticsearch-connector/config \
  -H "Content-Type: application/json" \
  -d '{

    "connector.class": "io.confluent.connect.elasticsearch.ElasticsearchSinkConnector",
    "connection.url": "http://elasticsearch:9200",
    "tasks.max": "1",
    "topics": "postgres_orientation_students", ← Топик для импортирования записей
    "type.name": "_doc",
    "value.converter": "org.apache.kafka.connect.json.JsonConverter",
    "value.converter.schemas.enable": "false",
    "schema.ignore": "true",
    "key.ignore": "false",
    "errors.tolerance": "all", ← Настройка допустимости ошибок
    "errors.deadletterqueue.topic.name": "orientation_student_dlq",
    "errors.deadletterqueue.context.headers.enable": "true",
    "errors.deadletterqueue.topic.replication.factor": "1" ←

    }' ← Установка коэффициента репликации
          равным 1 для разработки
          }' ← Включать
          заголовки
          для описания
          ошибки
```

Конфигурация в этом примере по большей части похожа на конфигурацию коннектора JDBC, за исключением специфических настроек. Здесь задается имя топика, откуда коннектор-приемник будет читать записи и отправлять их в Elasticsearch.

Но есть три конфигурации, начиная с `errors`, которые мы не видели раньше, и мы должны их обсудить. Поскольку коннектор-приемник пытается отправлять события во внешнюю систему, то есть вероятность ошибок. В конце концов, сбои в распределенных приложениях неизбежны, и мы должны определить, как реагировать на них.

## ПРИМЕЧАНИЕ

Код в листинге 5.4 устанавливает коэффициент репликации топика для очереди недоставленных сообщений (dead letter queue, DLQ). В кластере с одним узлом, как в нашей среде разработки Docker, нужно установить его равным 1. В противном случае коннектор не запустится, так как нет достаточного числа брокеров для коэффициента репликации со значением по умолчанию 3.

Параметр `errors.tolerance` в листинге 5.4 определяет, как коннектор будет реагировать на возникающие ошибки. Здесь вы использовали настройку `a11`, то есть коннектор будет продолжать работать независимо от ошибок, возникающих во время преобразования и передачи записей в приемник. В этом случае вам нужно будет просмотреть журналы этого конкретного коннектора, чтобы определить, что пошло не так, и выбрать соответствующее решение. Настройка `a11` позволяет коннектору продолжать работу, однако вы все равно должны знать обо всех случаях, когда записи не были доставлены. Эта настройка создает очередь недоставленных сообщений (DLQ), куда Connect сможет сохранять записи, которые ему не удалось доставить.

Однако при включении поддержки топика DLQ Connect сохраняет только запись с ошибкой. Чтобы узнать причину сбоя, нужно разрешить сохранять дополнительную информацию в заголовке записи, что мы и сделали. Ранее мы уже рассматривали заголовки записей, поэтому желающие могут вернуться к главе 4, чтобы вспомнить, как читать заголовки записей.

Использование DLQ имеет свои компромиссы, которые следует учитывать. Настройка `none` в параметре `error.tolerance` и отключение при ошибке выглядят слишком жесткой мерой, особенно в промышленной системе, однако в этом случае вы быстро сможете обнаружить проблему и исправить ее. Сравните `none` с настройкой `a11`, при которой работа будет продолжаться независимо от любых ошибок, с которыми может столкнуться коннектор. Поэтому чрезвычайно важно отслеживать любые ошибки, поскольку бесконечная работа с неисправленными ошибками может привести к еще более худшим последствиям, чем отключение. Другими словами, «если дерево падает в лесу, где никого нет, чтобы услышать треск, то издает ли падающее дерево звук?» Если ошибки поступают в DLQ, но никто на них не реагирует, то это то же самое, что и отсутствие ошибок.

Итак, с включением DLQ вам нужно настроить некий мониторинг (например, `KafkaConsumer`), который будет оповещать о любых ошибках и, возможно, предпринимать некоторые действия, например отключать проблемный коннектор при наличии постоянных проблем. Упрощенная реализация, демонстрирующая этот тип функциональности, включена в примеры исходного кода для книги.

Итак, теперь вы знаете, как использовать коннектор-источник для импорта данных в Kafka и коннектор-приемник — для экспорта из Kafka во внешнюю систему. Однако этот простой рабочий пример не отражает всей полезности Kafka Connect. Во-первых, можетиться несколько коннекторов-приемников, записывающих записи, импортированные из внешних систем, и с помощью Kafka можно создать платформу потоковой передачи событий, эффективно объединяющую все внешние системы в один центральный поток данных.

На практике в системе может иметься несколько разных коннекторов-приемников, импортирующих данные, и, скорее всего, потребуется более глубокая обработка

входящих записей, чем в простом конвейере данных, показанном в этой главе. Например, представьте, что произвольное количество клиентских приложений может полагаться на топик с данными, которые предоставляет коннектор-источник, и каждое приложение может генерировать уникальные результаты для другого топика. Затем некоторое количество коннекторов-приемников может экспортировать обновленные записи обратно во внешние системы.

Connect играет важную роль в интеграции внешних приложений данных с платформой потоковой обработки событий Kafka. С помощью Connect к платформе можно подключить любое приложение и использовать Kafka в качестве центрального узла для всех входных данных.

Но что делать, если у вас есть система — источник или приемник, для которой нет готового коннектора? В настоящее время существуют сотни готовых коннекторов, но нужный вам может отсутствовать. Не отчайтайтесь, с помощью Connect API вы сможете реализовать свой коннектор, чем мы и займемся в следующем разделе.

## 5.6. СОЗДАНИЕ И РАЗВЕРТЫВАНИЕ СОБСТВЕННОГО КОННЕКТОРА

В этом разделе мы поговорим о том, что нужно сделать, чтобы создать свой коннектор. Допустим, вы работаете в финансово-технологической компании и в вашей компании решили, помимо финансовых данных институционального уровня, инвесторам и инвестиционным фирмам предоставлять также анализ на основе модели подписки.

С этой целью компания создала несколько отделов для приема биржевых сводок в реальном времени. Получение всего потока данных в реальном времени стоит дорого, и стоимость потока для каждого отдела будет непомерно высокой. Но вы понимаете, что, создав свой коннектор-источник, вы сможете потреблять поток один раз и каждый отдел сможет настроить свое клиентское приложение для потребления потока из вашего кластера Kafka практически в реальном времени. А теперь с мыслью об этой идеи перейдем к реализации своего коннектора.

### 5.6.1. Реализация коннектора

Для создания своего коннектора мы используем интерфейсы `Connector` и `Task`. В частности, расширим абстрактный класс `SourceConnector`, который, в свою очередь, расширяет класс `Connector`. Нам нужно будет реализовать несколько абстрактных методов, но мы рассмотрим только самые важные. Полную реализацию вы найдете в примерах исходного кода для книги.

Начнем с конфигурации. Для перемещения данных класс `Connector` почти ничего не делает. Его основная обязанность — правильная настройка каждого экземпляра `Task`, поскольку именно `Task` перемещает данные в Kafka или из нее. Соответственно, внутри нашего коннектора мы создадим класс `StockTickerSourceConnectorConfig`. Класс, осуществляющий настройку, содержит экземпляр `ConfigDef` (листинг 5.5) для задания необходимых конфигурационных параметров.

**Листинг 5.5.** Настройка экземпляра ConfigDef

```

public class StockTickerSourceConnector extends SourceConnector {
    private static final ConfigDef CONFIG_DEF = new ConfigDef()
        .define(API_URL_CONFIG,
            ConfigDef.Type.STRING,
            ConfigDef.Importance.HIGH,
            "URL for the desired API call")
        .define(TOPIC_CONFIG,
            ConfigDef.Type.STRING,
            ConfigDef.Importance.HIGH,
            "The topic to publish data to")

    ... далее следуют остальные настройки
}

```

Создание экземпляра ConfigDef

Определяет конфигурацию экземпляра API URL

Добавление имени топика в конфигурацию

Здесь, как видите, мы добавляем настройки, используя текущий стиль, — связываем вызовы методов в единую цепочку. Обратите внимание, что в реализации имеются и другие настройки, но они здесь не показаны, так как в этом нет необходимости. Несколько примеров достаточно, чтобы было понятно, как это делается. Создав экземпляр `ConfigDef`, коннектор «знает», какие конфигурационные параметры можно ожидать. Метод `ConfigDef.define` позволяет также указать значения по умолчанию. Таким образом, если вы не зададите какие-то из параметров, то будут использоваться соответствующие значения по умолчанию. Но если не задать параметр, не имеющий значения по умолчанию, то на запуске коннектор сгенерирует исключение `ConfigException` и завершит работу. Далее мы посмотрим, как коннектор определяет конфигурацию для каждого экземпляра Task (листинг 5.6).

**Листинг 5.6.** Настройка экземпляров Task

```

@Override
public List<Map<String, String>> taskConfigs(int maxTasks) {
    List<Map<String, String>> taskConfigs = new ArrayList<>();
    List<String> symbols = monitorThread.symbols(); ← Получает список биржевых символов из потока мониторинга
    int numTasks = Math.min(symbols.size(), maxTasks);
    List<List<String>> groupedSymbols =
        ConnectorUtils.groupPartitions(symbols, numTasks); ← Разбивает биржевые символы по количеству заданий
    for (List<String> symbolGroup : groupedSymbols) {
        Map<String, String> taskConfig = new HashMap<>(); ← Создает Map с настройками для каждой задачи
        taskConfig.put(TOPIC_CONFIG, topic);
        taskConfig.put(API_URL_CONFIG, apiUrl);
        taskConfig.put(TOKEN_CONFIG, token);
        taskConfig.put(TASK_BATCH_SIZE_CONFIG, Integer.toString(batchSize));
        taskConfig.put(TICKER_SYMBOL_CONFIG, String.join(", ", symbolGroup)); ← Сохраняет биржевые символы в виде строки через запятую
        taskConfigs.add(taskConfig);
    }
    return taskConfigs;
}

```

Обратите внимание, что мы получаем список биржевых символов из экземпляра с именем `monitorThread`. Об этом экземпляре мы поговорим ниже в этой главе, когда будем обсуждать тему мониторинга (см. подраздел 5.6.2). А пока достаточно знать, что метод `symbols()` этого объекта возвращает список сводок, необходимых для запуска анализа.

Выше в главе мы рассматривали конфигурационный параметр коннекторов `max.tasks`. Он определяет максимальное количество задач, которые коннектор может запустить для перемещения данных. Наш сервис может извлекать информацию об акциях компаний, используя список из 100 символов, разделенных запятыми. Нам нужно разбить общий список на несколько более коротких списков, чтобы `Connect` мог равномерно распределить работу между задачами.

Например, если вы указали максимум две задачи, а всего имеется десять биржевых символов, то коннектор разделит их на два списка по пять символов в каждом. Для группировки символов используется вспомогательный метод `ConnectionUtils.groupPartitions`. Это практически все, что требуется для реализации коннектора. А теперь перейдем к реализации задачи.

Поскольку мы создаем `SourceConnector`, для реализации своей задачи мы должны расширить абстрактный класс `SourceTask`. Назовем нашу реализацию `StockTickerSourceTask`. В классе есть несколько методов, которые нужно переопределить, но мы рассмотрим только метод `poll()`, поскольку он лежит в основе всего, что делает `SourceTask`: получает данные из внешнего источника и загружает их в Kafka.

Рассмотрим поведение `SourceTask` в общих чертах. После запуска коннектора он создаст требуемое количество задач и запустит их. В процессе выполнения периодически будут вызываться методы `SourceTask.poll()` задач для извлечения записей из заданного внешнего приложения. Общего конфигурационного параметра, определяющего частоту выполнения метода `poll`, нет. Но даже в отсутствие явного контроля над частотой вызова метода `poll` задачи у нас есть возможность добавить некоторое регулирование в реализацию задачи и заставить ее ждать желаемое количество времени перед выполнением логики `SourceTask`.

В нашем примере `SourceTask` присутствует простейшее регулирование. Зачем это может понадобиться? В нашем случае предполагается отправлять запрос конечной точке HTTP API. Многие API ограничивают частоту обращений к сервису или общее количество запросов в день. Таким образом, добавив регулирование, можно гарантировать, что количество вызовов API останется в пределах установленных ограничений.

Нам также нужно обработать ситуацию, когда никаких записей не возвращается. Было бы хорошей идеей подождать немного времени, например 1–2 секунды, чтобы позволить источнику подготовить новые данные. При этом важно вернуть управление вызывающему потоку с возвращаемым значением `null`, чтобы рабочий процесс мог ответить на команды приостановки или завершения работы.

Теперь рассмотрим работу метода `StockTickerSourceTask.poll` (листинг 5.7). Мы разобьем код на части, чтобы было проще понять, как работает логика опроса.

#### Листинг 5.7. Начало метода poll класса StockTickerSourceTask

```
public List<SourceRecord> poll() throws InterruptedException {
    final long nextUpdate = lastUpdate.get() + timeBetweenPoll; ← Вычисляет время ожидания до следующего опроса
    final long now = sourceTime.milliseconds(); ← Вычисляет время следующего опроса
    final long sleepMs = nextUpdate - now; ← Определяет текущее время
    if (sleepMs > 0) {
        LOG.trace("Waiting {} ms to poll API feed next", sleepMs);
        sourceTime.sleep(sleepMs);
    }
}
```

В самом начале метода вычисляется время, когда должен произойти наш следующий вызов API. Напомню: это делается для того, чтобы не превысить ограничения, установленные тарифным планом использования сервиса. Если текущее время меньше времени последнего вызова плюс желаемый интервал между вызовами, то мы приостанавливаем поток опроса.

По истечении времени ожидания выполняется основная логика, которая получает результаты по акциям (некоторые детали для простоты опущены) (листинг 5.8).

#### Листинг 5.8. Основная логика опроса

```

HttpRequest request = HttpRequest.newBuilder() ← Конструирует объект HttpRequest
    .uri(uri)
    .GET()
    .headers("Content-Type", "text/plain;charset=UTF-8")
    .build();
HttpResponse<String> response;
try {
    response = httpClient.send(request,
        HttpResponse.BodyHandlers.ofString()); ← Отправка запроса на получение данных
    AtomicLong counter = new AtomicLong(0);
    JsonNode apiResult = objectMapper.readTree(response.body()); ← Код JSON преобразуется в JsonNode
    ArrayNode tickerResults = (ArrayNode) apiResult.get(resultNode);
    LOG.debug("Retrieved {} records", tickerResults.size());
    Stream<JsonNode> stockRecordStream =
        StreamSupport.stream(tickerResults.spliterator(), false);

    List<SourceRecord> sourceRecords = stockRecordStream.map(entry -> { ←
        Map<String, String> sourcePartition =
            Collections.singletonMap("API", apiUrl); ← Каждая из полученных сводок в формате JSON отображается в SourceRecord
        Map<String, Long> sourceOffset =
            Collections.singletonMap("index", counter.getAndIncrement()); ← Создается схема для записи
        Schema schema = getValueSchema(entry); ←
        Map<String, Object> resultsMap = toMap(entry); ← Запись в формате JSON преобразуется в Map
        return new SourceRecord(sourcePartition,
            sourceOffset,
            topic,
            null,
            schema,
            toStruct(schema, resultsMap)); ← Конструируется структура для значения SourceRecord
    }).toList();
    lastUpdate.set(sourceTime.currentTimeMillis()); ← Устанавливается отметка времени последнего обновления
}
return sourceRecords; ← Возвращается список записей SourceRecord
}

```

Логика опроса достаточно прозрачна. Создается объект запроса `HttpRequest`, который затем отправляется конечной точке API. Далее тело полученного ответа читается в строку. Результаты представлены в формате JSON и содержат информацию об акциях во вложенном массиве. Код извлекает этот массив, и каждый его элемент отображает в запись типа `SourceRecord` для отправки в Kafka.

Здесь нужно обратить внимание только на один аспект: конструктор `SourceRecord` принимает два параметра, раздел и смещение источника, представленные экземплярами `Map`. Понятие раздела и смещения источника может показаться немного странным, если учесть, что источником для `SourceConnector` будет не Kafka. Однако в действительности экземпляр раздела источника лишь содержит общее описание местоположения, откуда коннектор получает данные, — имя таблицы БД, имя файла или, как в нашем случае, API URL. Далее конструируется схема для записи, возвращаемой вызовом API, и запись в формате JSON преобразуется в экземпляр `Map`. Эти два шага необходимы, чтобы создать структуру `Struct` для значения `SourceRecord`. Кроме того, экземпляру `SourceRecord`, который может оказаться в теме Kafka, передается генерированная схема на тот случай, если в коннекторе будет настроено включение схемы.

С учетом обобщенного определения раздела источника смещение источника определяет позицию отдельной записи в результате. Как вы наверняка помните, потребители Kafka фиксируют смещение последней полностью потребленной записи, поэтому если он по какой-либо причине отключается, то после возврата сможет возобновить операции с последнего зафиксированного смещения.

Выше в этом разделе мы видели, что экземпляр нашего коннектора использовал переменную `monitorThread` для получения списка отслеживаемых биржевых символов. В следующем разделе я объясню устройство этой переменной и с какой целью коннектор использует ее.

## **5.6.2. Создание динамического коннектора с помощью потока мониторинга**

В этом примере коннектора предположение о статическом характере списка символов выглядит разумным. А что, если бы мы захотели изменить его? Конечно, можно было бы использовать Connect REST API для обновления конфигураций, но тогда вам придется отслеживать все изменения и вручную обновлять коннектор. Избавиться от этой рутины можно, реализовав поток мониторинга для своего коннектора, который будет отслеживать любые изменения и автоматически обновлять задачи.

Поток мониторинга не является чем-то уникальным для коннектора, и, чтобы получить его, нужно лишь реализовать класс, который расширяет хорошо знакомый класс `java.lang.Thread`. Концептуально это приводит к запуску вместе с коннектором дополнительного потока, содержащего логику для проверки любых изменений и перенастройки задач.

Представьте, что у нас есть отдельный микросервис, который выясняет, какую информацию включить в конфигурацию коннектора-источника. Микросервис возвращает список биржевых символов, разделенных запятыми, а нам остается только сделать так, чтобы поток мониторинга периодически отправлял `HttpRequest` микросервису, сравнивал ответ с текущим списком символов и при наличии изменений вызывал перенастройку задач коннектора. Лучшим примером, демонстрирующим, почему предпочтительнее использовать поток мониторинга, служит коннектор JDBC. Этот коннектор можно использовать для импорта всей реляционной базы данных,

возможно содержащей множество таблиц. В любой организации реляционная база данных не является статическим ресурсом и меняется время от времени. Поэтому вам нужно будет автоматически подхватывать эти изменения, чтобы коннектор импортировал последние данные в Kafka.

Начнем анализ потока мониторинга с объявления класса и конструктора (листинг 5.9).

#### Листинг 5.9. Объявление класса и конструктор потока мониторинга

```
Передача ссылки на ConnectorContext  
из SourceConnector
public StockTickerSourceConnectorMonitorThread(  
    final ConnectorContext connectorContext,  
    final int monitorThreadCheckInterval,  
    final HttpClient httpClient,  
    final String serviceUrl) {  
    this.connectorContext = connectorContext;  
    this.monitorThreadCheckInterval = monitorThreadCheckInterval;  
    this.httpClient = httpClient;  
    this.serviceUrl = serviceUrl;  
}
```

Annotations for the constructor parameters:

- ConnectorContext: Целое число, определяющее интервал (в миллисекундах) проверки наличия изменений
- monitorThreadCheckInterval: HttpClient для создания запросов на получение обновленного списка биржевых символов
- httpClient: URL микросервиса
- serviceUrl: Набор символов изменился, и требуется перенастройка

Параметры конструктора говорят сами за себя, и все же `ConnectorContext` заслуживает краткого обсуждения. Любой класс, расширяющий абстрактный класс `Connector`, будет иметь доступ к `ConnectorContext`, поэтому вам не придется беспокоиться о том, откуда он берется. Вы будете использовать его для взаимодействия со средой выполнения Connect, запрашивая перенастройку задачи при изменении источника.

Теперь определим поведение потока мониторинга в методе `Thread.run()` и добавим в него логику, которая должна выполняться в случае каких-либо существенных изменений (некоторые детали для простоты опущены) (листинг 5.10).

#### Листинг 5.10. Определение метода run потока мониторинга

```
Условие защиты в цикле while с использованием  
объекта java.util.concurrent.CountDownLatch
public void run() {  
    while (shutDownLatch.getCount() > 0) {  
        try {  
            if (updatedSymbols()) {  
                Проверка наличия изменений  
                в символах  
                connectorContext.requestTaskReconfiguration();  
            }  
            boolean isShutdown = shutDownLatch.await(monitorThreadCheckInterval,  
                TimeUnit.MILLISECONDS);  
            if (isShutdown) {  
                Если CountDownLatch закончил  
                отсчет, то прервать цикл и выйти  
            }  
        }  
    }  
}
```

Annotations for the run method logic:

- shutDownLatch.getCount(): Ожидание в течение указанного времени
- shutDownLatch.await(): Набор символов изменился, и требуется перенастройка
- updatedSymbols(): Проверка наличия изменений в символах
- shutDownLatch.getCount() > 0: Условие защиты в цикле while с использованием объекта java.util.concurrent.CountDownLatch

Как видите, `StockTickerSourceConnectorMonitorThread` просто проверяет наличие изменений в списке биржевых символов, и если изменения обнаружились, то запрашивает изменение конфигурации задач коннектора. После проверки изменений `shutDownLatch` ждет в течение интервала времени, указанного в переменной экземпляра `monitorThreadCheckInterval` при создании потока коннектором. Если

`shutDownLatch` закончил отсчет времени ожидания, то в переменной `isShutdown` возвращается `true` и поток мониторинга останавливается. В противном случае он продолжает отслеживать изменения.

### ПРИМЕЧАНИЕ

`CountDownLatch` — это класс из пакета `java.util.concurrent` и инструмент синхронизации, позволяющий одному или нескольким потокам ждать наступления определенного условия. Я не буду здесь вдаваться в подробности, а желающих узнать больше отсылаю к документации Javadoc (<http://mng.bz/j1N8>).

В завершение обсуждения темы мониторинга кратко рассмотрим способ определения каких-либо изменений (некоторые детали опущены для простоты) (листинг 5.11).

#### Листинг 5.11. Логика определения изменений в списке символов

```
List<String> maybeNewSymbols = symbols(); ← Получение набора символов
boolean foundNewSymbols = false;
if (!Objects.equals(maybeNewSymbols, this.tickerSymbols)) { ← Сравнение с имеющимся
    tickerSymbols = new ArrayList<>(maybeNewSymbols); ← набором символов
    foundNewSymbols = true;
}
return foundNewSymbols; ← Обновление списка символов
```

Чтобы определить наличие изменений, метод `symbols()` отправляет HTTP-запрос микросервису и возвращает `List<String>`, содержащийся в ответе. Если содержимое полученного списка отличается от текущего, то список экземпляров обновляется и булевой переменной `foundNewSymbols` присваивается значение `true`, что запускает перенастройку задач после возврата из метода.

На этом мы завершаем рассмотрение нашей реализации `SourceConnector`. Представленный здесь код не готов к использованию, но он дает хорошее понимание, как реализовать свой коннектор. В примерах исходного кода для книги есть инструкции, описывающие, как запустить этот коннектор локально с образом Docker.

API возвращает богатый набор полей для каждой биржевой сводки, но основной интерес представляет лишь часть этих полей. Конечно, вы можете извлечь интересующие вас поля прямо в задаче коннектора, но тогда переключение сервиса API на использование другого конструктора JSON будет означать необходимость изменения логики коннектора. Лучшим решением этой проблемы было бы использование преобразования и извлечения необходимых полей до того, как записи попадут в Kafka. Для этого вам понадобится реализовать свое преобразование, извлекающее произвольные поля из объекта JSON и возвращающее объект только с нужными полями. Именно это решение мы рассмотрим в следующем разделе: создание пользовательского преобразования SMT.

### 5.6.3. Создание пользовательского преобразования

Kafka Connect имеет несколько встроенных преобразований, но они подходят не для всех случаев. Потребность в создании своего коннектора намного ниже из-за большого количества встроенных коннекторов, поддерживающих множество внешних систем, более вероятно, что вам придется написать свое преобразование.

API, возвращающий список акций, который мы использовали в нашем коннекторе, возвращает результат, содержащий более 70 полей различных метрик для каждого вида акций. Каждое из них несет определенную пользу, но нам нужна только часть из них — максимум 5 или 6. Поэтому мы создадим преобразование, сохраняющее только поля, указанные в настраиваемом списке.

Для создания преобразования нужно реализовать интерфейс `Transformation`. Этот интерфейс имеет несколько методов, но мы сосредоточимся только на одном из них — на `apply`, поскольку именно он выполняет все действия по удалению ненужных полей.

Реализация объекта `Transformation` имеет одну необычную особенность. Выше в этой главе, запуская коннектор, мы задавали конфигурационный параметр, определяющий необходимость включения схемы для каждой записи, и нам нужно будет учесть это при реализации нашего преобразования. Но подробнее об этом мы поговорим в следующем разделе. Кроме того, нам нужно будет разрешить пользователю применять это преобразование к ключу или значению, и вы увидите, как это организовать.

А теперь перейдем к реализации `Transformation` (некоторые детали для простоты опущены) (листинг 5.12).

#### Листинг 5.12. Реализация пользовательского преобразования

```
public abstract class MultiFieldExtract<R extends ConnectRecord<R>>
    implements Transformation<R> { ← Объявление класса
    @Override
    public R apply(R connectRecord) { ← Метод apply, выполняющий преобразование
        if (operatingValue(connectRecord) == null) { ← Если значение отсутствует (null), то ничего не остается, кроме как вернуть исходную запись
            return connectRecord;
        } else if (operatingSchema(connectRecord) == null) { ← Если схема отсутствует, то преобразование применяется только к значению
            return applySchemaless(connectRecord);
        } else {
            return applyWithSchema(connectRecord); ← В противном случае нужно скорректировать схему в соответствии с новой структурой значения
        }
    }
}
```

Возможно, вы заметили, что наша реализация `Transformation` является абстрактным классом. Как отмечалось в разделе 5.3, при настройке преобразования SMT мы должны указать, что оно применяется к ключу или значению, задав параметр, такой как `MaskField$Value` (в Java символ `$` указывает на внутренний класс). То есть мы объявили класс преобразования абстрактным, потому что по соглашению он будет иметь три абстрактных метода: `operatingSchema`, `operatingValue` и `newRecord`. Реализуем эти методы в двух внутренних классах, `Key` и `Value`, представляющих преобразования для соответствующей части записи Connect. Мы не будем углубляться в детали, поэтому продолжим двигаться вперед и обсудим работу метода `apply`.

Простейший случай показан в листинге 5.12: базовое значение записи Connect равно `null`. Как вы наверняка помните, Kafka позволяет передавать `null` в ключе или значении. Ключи необязательны, а пустые (`null`) значения в сжатых топиках представляют так называемые «надгробия», сообщающие инструменту очистки

журнала, что данную запись нужно удалить из топика. В оставшейся части этого раздела я буду предполагать, что мы работаем только со значениями.

Далее мы проверяем, имеется ли в записи встроенная схема. В разделе 5.2 мы обсуждали конфигурационный параметр `value.converter.schemas.enable`. Если он имеет значение `true`, то в каждую запись, проходящую через коннектор, должна встраиваться схема. Если схема не задана, то мы применяем метод `applySchemaless` для завершения преобразования (листинг 5.13).

#### **Листинг 5.13.** Преобразование без схемы

```
private R applySchemaless(R connectRecord) {
    final Map<String, Object> originalRecord =
        requireMap(operatingValue(connectRecord), PURPOSE);
    final Map<String, Object> newRecord = new LinkedHashMap<>();
    List<Map.Entry<String, Object>> filteredEntryList =
        originalRecord.entrySet().stream()
            .filter(entry -> fieldNamesToExtract.contains(entry.getKey()))
            .toList();

    filteredEntryList.forEach(entry -> newRecord.put(entry.getKey(),
        entry.getValue()));
    return newRecord(connectRecord, null, newRecord);
}
```

Поскольку схема отсутствует, мы создаем в текущей записи `Map` с ключами `String` (для имен полей) и значениями `Object`. Затем создается пустой ассоциативный массив для новой записи и производится фильтрация полей с проверкой наличия их в настраиваемом списке. Если текущее поле присутствует в списке, то его ключ и значение помещаются в ассоциативный массив, представляющий новую запись. Для записей со схемами выполняется похожий процесс, но сначала нужно настроить схему так, чтобы она содержала только поля, которые нужно сохранить (листинг 5.14).

#### **Листинг 5.14.** Преобразование со схемой

```
private R applyWithSchema(R connectRecord) {
    final Struct value =
        requireStruct(operatingValue(connectRecord), PURPOSE);

    Schema updatedSchema = schemaUpdateCache.get(value.schema());
    if(updatedSchema == null) {
        updatedSchema = makeUpdatedSchema(value.schema());
        schemaUpdateCache.put(value.schema(), updatedSchema);
    }
    final Struct updatedValue = new Struct(updatedSchema);

    updatedValue.schema().fields()
        .forEach(field -> updatedValue.put(field.name(),
            value.get(field.name())));
    return newRecord(connectRecord, updatedSchema, updatedValue);
}
```

Когда имеется комбинация «схема — запись», сначала нужно получить структуру `Struct`, очень похожую на `HashMap` в версии без схемы, но содержащую всю

информацию о типах полей в записи. Первым делом мы проверяем, создали ли уже обновленную схему. Если нет, то создаем ее и сохраняем в кэше. Создав обновленную схему один раз, нам не нужно создавать ее повторно, поскольку все записи будут иметь одинаковую структуру. Затем мы перебираем имена полей в обновленной схеме, используя каждое из них для извлечения значения из старой записи.

Теперь вы знаете, как реализовать свое преобразование Connect. Я раскрыл далеко не все детали, поэтому за более полной информацией обращайтесь к файлу `bvejeck.chapter_5.transformer.MultiFieldExtract`.

Я хочу закрыть эту главу о Kafka Connect предупреждением о том, что созданные здесь пользовательские версии `Connector` и `Transformation` не предназначены для использования в промышленном окружении. Они показаны здесь лишь как примеры создания своих версий этих двух классов, когда стандартные классы, предоставляемые Kafka Connect, не соответствуют нашим потребностям.

## ИТОГИ ГЛАВЫ

- Kafka Connect — это ключевой элемент, предназначенный для перемещения данных из внешних систем в Kafka и из Kafka во внешние системы. Возможность перемещения данных в Kafka и из него имеет решающее значение для интеграции существующих сторонних приложений с платформой потоковой обработки событий.
- При использовании существующих коннекторов не требуется писать код. Для их запуска достаточно выгрузить конфигурацию в формате JSON. В настоящее время существуют сотни коннекторов, поэтому вам наверняка удастся найти нужный, который можете использовать «из коробки».
- Фреймворк Connect также предоставляет преобразования отдельных сообщений (single message transforms, SMT), способные применять незначительные изменения к входящим или исходящим записям.
- Если существующие коннекторы или преобразования не отвечают вашим потребностям, то есть возможность создать свой коннектор.

## Часть III

В третьей части мы углубимся в Kafka Streams. Вооруженные знаниями, полученными в первых двух частях, вы готовы взяться за дело и наконец узнать, как работает Kafka.

Здесь вы изучите уровень Kafka Streams DSL и в полной мере оцените, какие типы приложений потоковой обработки событий можно создавать. Сначала мы рассмотрим пример простого приложения Hello World в Kafka Streams, а затем быстро перейдем к более практическому примеру вымышленного розничного магазина. Продолжая в том же духе, мы создадим простое приложение DSL и постепенно будем добавлять в него новые возможности, такие как использование состояния приложения Kafka Streams. Вы будете использовать состояние всякий раз, когда понадобится «вспомнить» что-то из предыдущих событий. Следующая остановка в вашем путешествии — **KTable**. Если **KStream** — это поток событий, то **KTable** — это поток обновлений, в котором записи с одинаковым ключом являются обновлениями одной и той же записи. Далее мы перейдем к концепции, которая идет рука об руку с операциями с состоянием, — к оконным операциям. В отличие от операций с состоянием, которые продолжают охватывать все больше данных с течением времени, оконные операции позволяют сгруппировать события в дискретные временные интервалы, разбивая состояние на более удобные для анализа фрагменты.

Но так же, как в случае с любой хорошо спроектированной абстракцией, иногда Kafka Streams DSL не сможет дать вам именно то, что нужно. В таких ситуациях вам на выручку придет Kafka Streams Processor API, который, пусть и ценой дополнительных усилий, способен удовлетворить любые потребности в создании приложений потоковой обработки событий.

Затем мы выйдем за рамки Kafka Streams и перейдем к ksqlDB, инструменту разработки приложений потоковой обработки событий с использованием знакомого языка запросов SQL. После знакомства с ksqlDB мы перейдем к интеграции с фреймворком Spring, пользующимся большой популярностью среди разработчиков за то, что упрощает создание модульных и легко тестируемых приложений. Далее мы применим новые знания о Spring на практике и воспользуемся одной из уникальных возможностей Kafka Stream для создания интерактивного сервиса запросов.

В заключение мы будем учиться тестировать не только приложения Kafka Streams, но и другие части экосистемы Kafka, с которыми познакомились в этой книге. Но на этом книга не заканчивается — в ней также имеется несколько приложений с дополнительной информацией. Некоторые дополнительные сведения пригодятся вам при разработке приложений потоковой обработки событий с использованием Kafka и Kafka Streams. Однако, несмотря на ее полезность, знать эту информацию не обязательно, чтобы создавать программное обеспечение с поддержкой потоковой обработки. Итак, вот эти четыре приложения с полезной, но необязательной информацией:

- приложение А содержит практикум по Schema Registry и делится практическим опытом работы с различными режимами совместимости схем;
- приложение Б представляет информацию об использовании Confluent Cloud для разработки приложений потоковой обработки событий;
- приложение В представляет обзор работы с различными типами схем Avro, Protobuf и JSON Schema;
- приложение Г описывает архитектуру и внутреннее устройство Kafka Streams.

# Разработка приложений *Kafka Streams*

---

## В этой главе

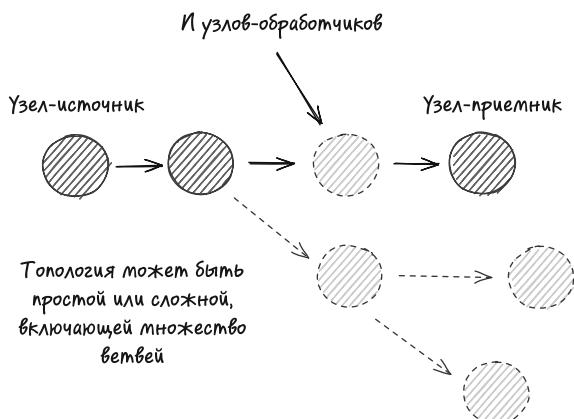
- ✓ Знакомство с Kafka Streams.
- ✓ Создание первого приложения с помощью Kafka Streams.
- ✓ Работа с данными клиентов и создание более сложных приложений.
- ✓ Разделение, слияние и разветвление потоков.

Приложение Kafka Streams представляет собой граф, в узлах которого происходит обработка и преобразование данных. В этой главе вы узнаете, как с помощью Kafka Streams построить граф, составляющий приложение потоковой обработки.

## 6.1. ОБЗОР KAFKA STREAMS

Рассмотрим рис. 6.1, который поможет понять смысл сказанного выше. Здесь показана обобщенная структура большинства приложений Kafka Streams. В приложении есть узел-источник, который получает записи событий от брокера Kafka. Есть произвольное количество узлов-обрабатчиков, каждый из которых решает отдельную узкую задачу, и, наконец, есть узел-приемник, передающий преобразованные записи обратно в Kafka. В главе 4 мы обсуждали, как использовать клиенты Kafka для производства и потребления записей с помощью Kafka. Многое из того, что вы узнали

в этой главе, применимо к Kafka Streams, потому что, по сути, Kafka Streams — это абстракция над производителями и потребителями, позволяющая сосредоточиться на требованиях к обработке потоков.



**Рис. 6.1.** Приложение Kafka Streams — это граф с узлом-источником, произвольным количеством узлов-обработчиков и узлом-приемником

#### ПРИМЕЧАНИЕ

Хотя Kafka Streams является библиотекой потоковой обработки для Apache Kafka, она работает не внутри кластера или брокеров, а подключается как клиентское приложение.

## 6.2. KAFKA STREAMS DSL

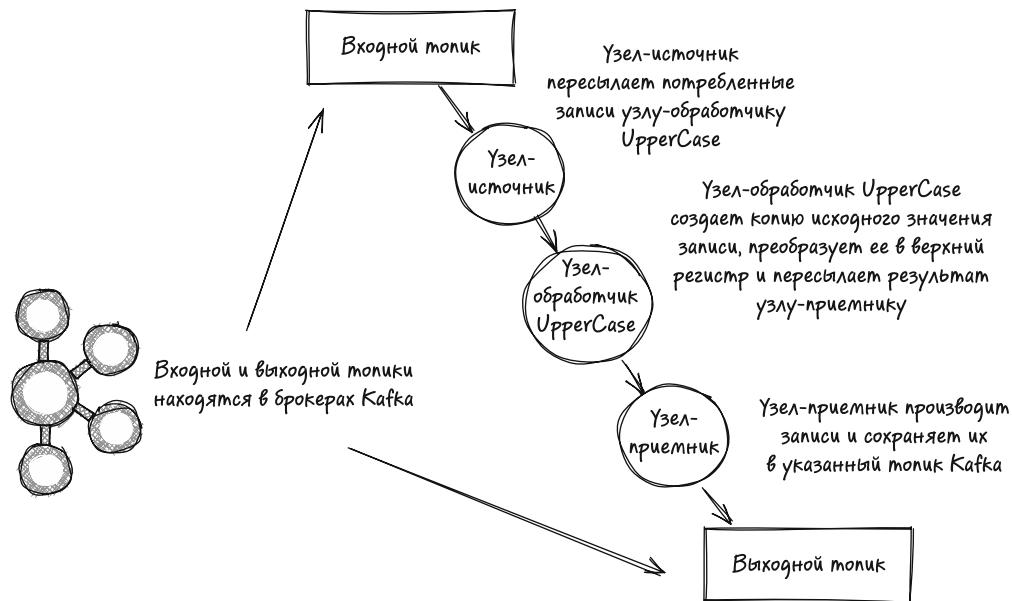
Kafka Streams DSL (предметно-ориентированный язык Kafka Streams) — это высокоуровневый API, предназначенный для быстрого создания приложений Kafka Streams. Степень продуманности этого высокоуровневого API очень высока, в нем есть готовые методы для удовлетворения большинства потребностей потоковой обработки, что позволяет создавать сложные программы потоковой обработки без особых усилий. В основе API лежит объект `KStream`, олицетворяющий потоковые записи пар «ключ — значение».

Большинство методов Kafka Streams DSL возвращает ссылку на объект `KStream`, что делает возможным стиль программирования с использованием цепочек вызовов. Кроме того, немалая доля методов интерфейса `KStream` принимает на входе типы, состоящие из интерфейсов с одним методом, что позволяет применять лямбда-выражения. Учитывая эти факторы, легко представить себе простоту и удобство создания приложений Kafka Streams.

Существует также низкоуровневый Processor API для создания узлов-обработчиков, не столь лаконичный, как Kafka Streams DSL, но предоставляющий больше возможностей контроля. Мы рассмотрим Processor API в главе 10. Теперь, после столь краткого введения, приступим к созданию нашей первой программы Hello World для Kafka Streams.

## 6.3. ПРОГРАММА HELLO WORLD ДЛЯ KAFKA STREAMS

В нашем первом примере Kafka Streams мы создадим простое и забавное приложение, что поможет нам быстрее сдвинуться с мертвой точки и увидеть, как работает Kafka Streams. Нашей первой программой будет «игрушечное» приложение, которое преобразует входящие сообщения в верхний регистр – как бы кричит на читателей сообщений. Мы назовем это приложением Yelling («Крикун»). Прежде чем углубиться в код, взглянем на топологию обработки этого приложения, показанную на рис. 6.2.



**Рис. 6.2.** Топология приложения Yelling

Как видите, этот граф обработки настолько простой, что больше напоминает связанный список узлов, чем типичный древовидный граф. Но он содержит достаточно деталей, чтобы вы могли представить, чего ожидать в коде. Итак, у нас будут узел-источник, узел-обработчик, преобразующий текст в верхний регистр, и узел-приемник, записывающий результат в топик.

Это тривиальный пример, но показанный далее код типичен для других программ Kafka Streams. Большинство из них организованы по следующему общему шаблону.

1. Определяют элементы конфигурации.
2. Создают экземпляры Serde, как пользовательских, так и предопределенных, используемых при сериализации/десериализации записей.
3. Строят топологию обработки.
4. Создают и запускают Kafka Streams.

Когда мы перейдем к более сложным примерам, они будут отличаться только более сложной топологией узлов-обрабатчиков. А теперь приступим к созданию первого приложения.

### 6.3.1. Создание топологии для приложения Yelling

Первый шаг при создании любого приложения Kafka Streams — создание узла-источника. Узел-источник отвечает за потребление протекающих через приложение записей. На рис. 6.3 в графе выделен узел-источник.

Код в листинге 6.1 создает узел-источник (родительский узел) графа.

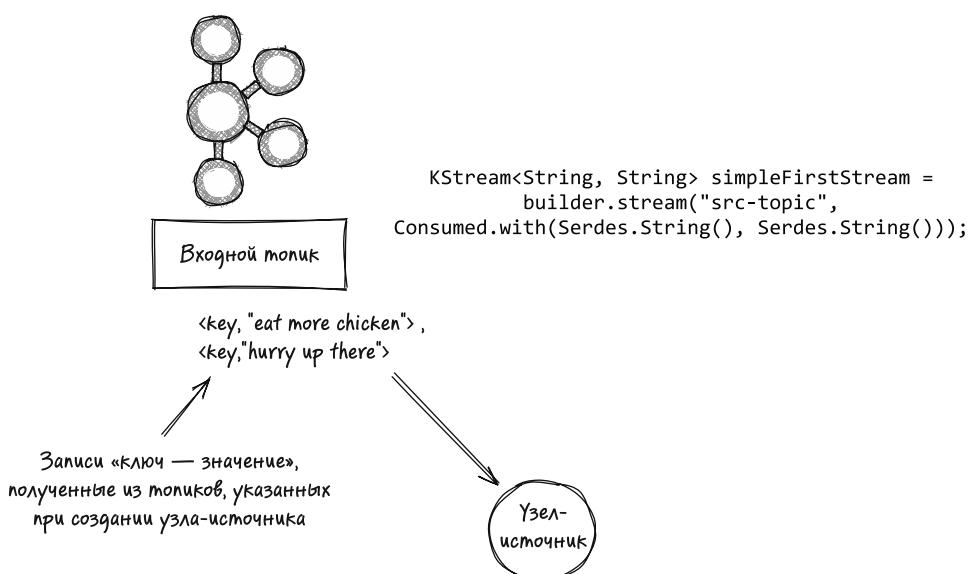


Рис. 6.3. Создание узла-источника в приложении Yelling

#### Листинг 6.1. Определение источника для потока

```
KStream<String, String> simpleFirstStream = builder.stream("src-topic",
    Consumed.with(Serdes.String(), Serdes.String()));
```

Экземпляр класса `simpleFirstStream` настроен на потребление сообщений из входного топика. Помимо имени топика, можно также передать объект `Consumed`, который Kafka Streams использует для настройки дополнительных параметров узла-источника. В этом примере мы передали экземпляры `Serde`, первый для ключа, а второй для значения. `Serde` — это объект-обертка, содержащий сериализатор и десериализатор для заданного типа.

Как рассказывалось в предыдущей главе при обсуждении клиентов-потребителей, брокер хранит и пересыпает записи в формате массива байтов. Чтобы приложение Kafka Streams могло выполнить какую-либо работу, оно должно десериализовать байты в конкретные объекты. Здесь оба объекта `Serde` предназначены для преобразования строк, поскольку в записях используются строковые ключ и значение. Kafka

Streams будет использовать Serde для десериализации ключа и значения в строковые объекты. Объекты Serde более подробно будут рассматриваться ниже в этой главе. Класс `Consumed` также можно использовать для настройки `TimestampExtractor`, сброса смещения в узле-источнике и передачи имени оператора. `TimestampExtractor` мы рассмотрим в главе 9. А сброс смещения мы обсуждали в главе 2, поэтому я не буду возвращаться к этому вопросу снова.

Так создается `KStream` для чтения из топика Kafka. Но данные для обработки могут извлекаться не только из одного топика. Давайте быстро рассмотрим некоторые другие варианты. Допустим, что у нас есть несколько топиков, на которые вы хотели бы «накричать». В этом случае мы можем подписаться на все эти топики сразу с помощью `Collection<String>`, указав в коллекции имена всех топиков, как показано в листинге 6.2.

**Листинг 6.2.** Создание приложения `Yelling`, извлекающего данные из нескольких топиков

```
KStream<String, String> simpleFirstStream =
    builder.stream(List.of("topicA", "topicB", "topicC"),
        Consumed.with(Serdes.String(), Serdes.String()))
```

Обычно этот подход используется, когда нужно применить одну и ту же обработку сразу к нескольким топикам. А как быть, если имеется длинный список топиков с похожими именами? Нужно ли выписывать их все? Ответ: нет! Вы можете использовать регулярное выражение, как показано в листинге 6.3, и подписаться на все топики, имена которых соответствуют шаблону.

**Листинг 6.3.** Использование регулярного выражения для подписки на несколько топиков

```
KStream<String, String> simpleFirstStream =
    buider.source(Pattern.compile("topic[A-C]"),
        Consumed.with(Serdes.String(), Serdes.String()))
```

Возможность использовать регулярное выражение для подписки на топики очень удобна, когда в организации используется стандартный шаблон для выбора имен топиков, отражающих их бизнес-функции. Вам нужно знать этот шаблон, чтобы оформить подписку на все нужные топики в компактном виде. Кроме того, по мере создания или удаления топиков ваша подписка будет автоматически обновляться в соответствии с изменением состава топиков.

При подписке на несколько топиков следует помнить, что ключи и значения во всех топиках должны быть одного типа; например, нельзя объединять топики, один из которых содержит ключи `Integer`, а другой — ключи `String`. Кроме того, если они содержат разное количество разделов, то вам нужно перераспределить данные перед выполнением любых операций на основе ключей, таких как агрегирование. Перераспределение мы рассмотрим в следующей главе. Наконец, важно помнить, что у вас нет никаких гарантий, касающихся упорядочения входящих записей.

## СОВЕТ

Повторю еще раз, что Kafka гарантирует упорядоченность только в пределах одного раздела топика. Поэтому при потреблении данных из нескольких топиков у вас не будет никаких гарантий упорядочения потребленных записей.

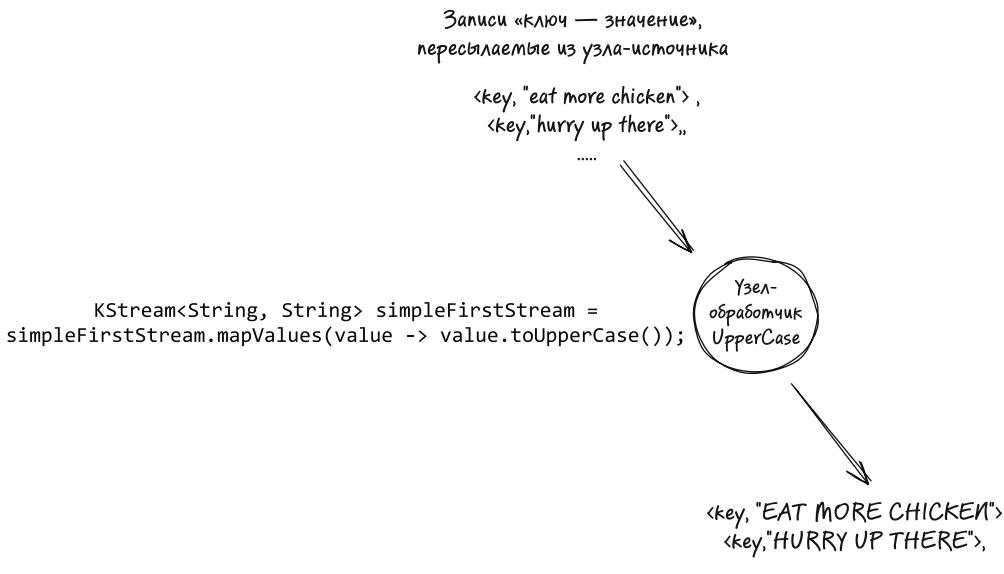
У нас теперь есть узел-источник для нашего приложения, но нам нужно присоединить к нему обрабатывающий узел, чтобы воспользоваться этими данными, как показано на рис. 6.4.

В листинге 6.4 приведен код, преобразующий текст в верхний регистр.

#### Листинг 6.4. Преобразование входного текста в верхний регистр

```
KStream<String, String> upperCasedStream =
    simpleFirstStream.mapValues(value -> value.toUpperCase());
```

Во введении к этой главе я говорил, что приложение Kafka Streams представляет собой граф с узлами-обрабатчиками, точнее говоря, ориентированный ациклический граф (directed acyclic graph, DAG). Граф конструируется добавлением узлов-обрабатчиков по одному, и каждым вызовом метода устанавливаются родительско-дочерние отношения между узлами графа. Родительско-дочерние отношения в Kafka Streams задают направление потока данных; родительские узлы пересыпают записи дочерним узлам.



**Рис. 6.4.** Добавление в приложение Yelling узла-обработчика, преобразующего текст в верхний регистр

Вызов функции `simpleFirstStream.mapValues` создает новый узел-обрабатчик, на выход которого подается запись, полученная из топика узлом-источником. Таким образом, узел-источник является родителем и пересыпает записи своему потомку, узлу-обработчику, возвращаемому из операции `mapValues`.

#### ПРИМЕЧАНИЕ

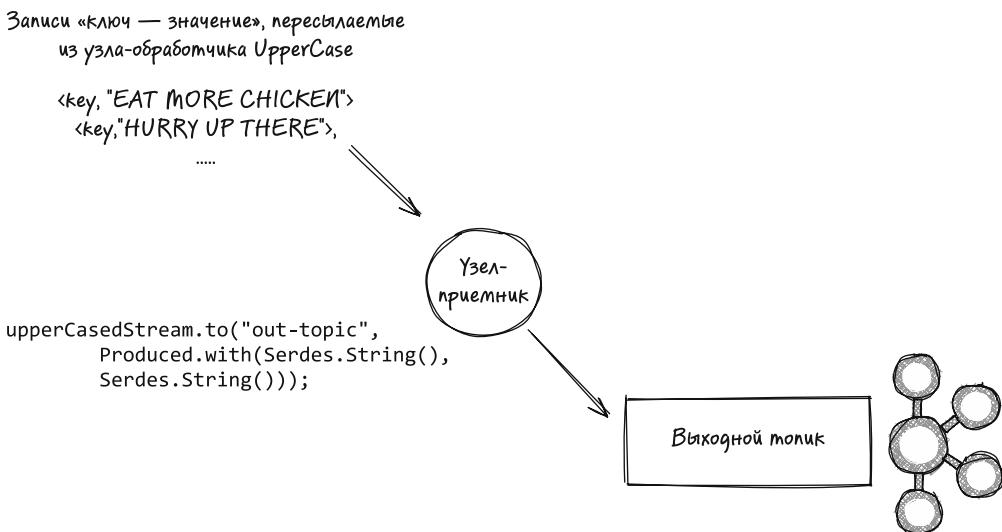
Судя по имени, `mapValues` влияет только на значение в паре «ключ — значение», но ключ исходной записи все равно передается.

Метод `mapValues()` принимает экземпляр интерфейса `ValueMapper<V, V1>`. Интерфейс `ValueMapper` имеет только один метод, `ValueMapper.apply`, что делает его идеальным кандидатом для применения лямбда-выражения, что мы и сделали, передав выражение `value -> value.toUpperCase()`.

### ПРИМЕЧАНИЕ

Существует множество руководств по лямбда-выражениям и ссылкам на методы. Для начала можно обратиться к документации Oracle по языку Java: *Lambda Expressions* (<http://mng.bz/J0Xm>) и *Method References* (<http://mng.bz/BaDW>).

Теперь наше приложение Kafka Streams умеет потреблять записи и преобразовывать их в верхний регистр. Нам осталось только добавить узел-приемник, который будет записывать результаты в топик (листинг 6.5). Рисунок 6.5 демонстрирует, до какого места в создании топологии мы добрались.



**Рис. 6.5.** Добавление узла для записи результатов приложения Yelling

### Листинг 6.5. Создание узла-приемника

```
upperCasedStream.to("out-topic",
    Produced.with(Serdes.String(), Serdes.String()));
```

Метод `KStream.to` создает в топологии узел-приемник, который записывает преобразованные записи в топик Kafka. Он является потомком узла `upperCasedStream`, а значит, принимает от него записи, являющиеся результатом операции `mapValues`.

И снова мы передаем в качестве параметров экземпляры `Serde`, на этот раз для сериализации записываемых в топик Kafka записей. Но в данном случае мы задействуем экземпляр класса `Produced`, который предоставляет необязательные параметры для создания узла-приемника в Kafka Streams. Конфигурационный объект `Produced` также

позволяет предоставлять пользовательский `StreamPartitioner`. Идею создания своего механизма назначения разделов мы рассмотрели в подразделе 4.2.4, поэтому я не буду повторно поднимать эту тему здесь. Пример использования `StreamPartitioner` можно увидеть в главе 9, ближе к концу подраздела 9.1.6.

## ПРИМЕЧАНИЕ

Передавать объекты `Serde` объектам `Consumed` и `Produced` вовсе не обязательно. Если этого не сделать, приложение будет использовать указанный в конфигурации сериализатор/десериализатор. Кроме того, с помощью классов `Consumed` и `Produced` можно задать объект `Serde` только для ключа или только для значения.

В предыдущем примере топология создается с помощью трех строк кода:

```
KStream<String, String> simpleFirstStream =  
    builder.stream("src-topic",  
        Consumed.with(Serdes.String(), Serdes.String()));  
  
KStream<String, String> upperCasedStream =  
    simpleFirstStream.mapValues(value -> value.toUpperCase());  
  
upperCasedStream.to("out-topic",  
    Produced.with(Serdes.String(), Serdes.String()));
```

Каждый шаг в коде демонстрирует отдельный этап процесса сборки. Но все методы `KStream`, кроме создающих узлы-приемники, возвращают новый экземпляр `KStream`, благодаря чему можно воспользоваться вышеупомянутым стилем программирования текущих интерфейсов. Текущий интерфейс (<https://martinfowler.com/bliki/FluentInterface.html>) позволяет объединять вызовы методов в цепочки и получать компактный и удобочитаемый код. Для демонстрации этой идеи покажу альтернативный способ создания топологии приложения `Yelling`:

```
builder.stream("src-topic", Consumed.with(Serdes.String(), Serdes.String()))  
    .mapValues(value -> value.toUpperCase())  
    .to("out-topic", Produced.with(Serdes.String(), Serdes.String()));
```

Мы сократили программу с трех строк до одной без потери ясности или функциональности. С этого момента все примеры в книге будут использовать стиль текущих интерфейсов, за исключением случаев, когда это повлекло бы за собой уменьшение ясности программы.

## СОВЕТ

Для запуска приложения Kafka Streams требуется действующий брокер Kafka. Использование Docker — очень удобный способ запуска Kafka на локальном компьютере. Примеры исходного кода включают файл `docker-compose.yml`. Чтобы запустить Kafka, используйте команду `docker compose up -d`.

Наша первая топология Kafka Streams готова, но мы обошли стороной важные шаги настройки и создания объектов `Serde`. Давайте теперь рассмотрим их.

### 6.3.2. Настройка Kafka Streams

Хотя Kafka Streams предоставляет очень широкие возможности настройки, которые можно менять под свои нужды, в нашем первом примере будут использоваться только два конфигурационных параметра — `APPLICATION_ID_CONFIG` и `BOOTSTRAP_SERVERS_CONFIG`:

```
props.put(StreamsConfig.APPLICATION_ID_CONFIG, "yelling_app_id");
props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
```

Оба параметра обязательны, поскольку у них нет значений по умолчанию. Попытка запуска программы Kafka Streams с неопределенными значениями этих двух свойств приведет к генерации исключения `ConfigException`.

Свойство `StreamsConfig.APPLICATION_ID_CONFIG` идентифицирует приложение Kafka Streams. Экземпляры Kafka Streams с одинаковым `application.id` считаются одним логическим приложением. Подробнее эта концепция рассматривается в приложении Г. `application.id` также служит префиксом для настроек встроенного клиента (`KafkaConsumer` и `KafkaProducer`). Вы можете предоставить свои настройки для встроенных клиентов, использующих одну из префиксных меток в классе `StreamsConfig`. Однако клиентские конфигурации по умолчанию в Kafka Streams выбраны так, чтобы обеспечить наилучшую производительность, поэтому будьте осторожны, изменения их.

Свойство `StreamsConfig.BOOTSTRAP_SERVERS_CONFIG` может представлять собой одну или несколько пар `имя_хоста:порт`, разделенных запятыми. С помощью `BOOTSTRAP_SERVERS_CONFIG` Kafka Streams устанавливает соединение с кластером Kafka. Мы обсудим еще несколько параметров по мере их появления в дальнейших примерах в книге.

### 6.3.3. Создание объектов Serde

Класс `Serdes` в Kafka Streams предоставляет удобные методы для создания экземпляров `Serde`:

```
Serde<String> stringSerde = Serdes.String();
```

В этой строке с помощью класса `Serdes` создается экземпляр `Serde`, необходимый для сериализации/десериализации. Полученный экземпляр сохраняется в переменной, которая будет использоваться в топологии неоднократно для ссылки на объект `Serde`. Класс `Serdes` обеспечивает реализации по умолчанию для таких типов данных, как: `String`, `ByteArray`, `Bytes`, `Long`, `Short`, `Integer`, `Double`, `Float`, `ByteBuffer`, `UUID` и `Void`.

Полеза реализаций интерфейса `Serde` заключается в наличии сериализатора/десериализатора, что избавляет от необходимости указывать четыре параметра (сериализатор ключей, сериализатор значений, десериализатор ключей и десериализатор значений) всякий раз, когда требуется передать объект `Serde` в метод класса

KStream. В следующих примерах мы будем использовать Serdes для работы с Avro, Protobuf и JSON Schema и создадим реализацию Serde, которая будет отвечать за сериализацию/десериализацию более сложных типов.

Посмотрим на нашу программу целиком (некоторые детали опущены для простоты) (листинг 6.6). Исходный код можно найти в файле `src/main/java/bbejeck/chapter_6/KafkaStreamsYellingApp.java` (доступен по адресу <https://github.com/bbejeck/KafkaStreamsInAction2ndEdition>).

#### Листинг 6.6. Hello World: приложение Yelling

```
public class KafkaStreamsYellingApp extends BaseStreamsApplication {
    private static final Logger LOG =
        LoggerFactory.getLogger(KafkaStreamsYellingApp.class);

    @Override
    public Topology topology(Properties streamProperties) {
        Serde<String> stringSerde = Serdes.String();
        StreamsBuilder builder = new StreamsBuilder();
        KStream<String, String> simpleFirstStream = builder.stream("src-topic",
            Consumed.with(stringSerde, stringSerde));
        KStream<String, String> upperCasedStream =
            simpleFirstStream.mapValues(value -> value.toUpperCase());
        upperCasedStream.to("out-topic",
            Produced.with(stringSerde, stringSerde));
        return builder.build(streamProperties);
    }

    public static void main(String[] args) throws Exception {
        Properties streamProperties = new Properties();
        streamProperties.put(StreamsConfig.APPLICATION_ID_CONFIG,
            "yelling_app_id");
        streamProperties.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
            "localhost:9092");
        KafkaStreamsYellingApp yellingApp = new KafkaStreamsYellingApp();
        Topology topology = yellingApp.topology(streamProperties);

        try(KafkaStreams kafkaStreams =
            new KafkaStreams(topology, streamProperties)) {
            LOG.info("Hello World Yelling App Started");
            kafkaStreams.start(); ← Запуск потоков выполнения в Kafka Streams
            LOG.info("Shutting down the Yelling APP now");
        }
    }
}
```

The diagram highlights several parts of the code with annotations:

- Создается объект Serdes для сериализации/десериализации ключей и значений и сохраняется в переменной**: Points to the line `Serde<String> stringSerde = Serdes.String();`.
- Создается экземпляр StreamsBuilder, используемый для конструирования топологии узлов-обработчиков**: Points to the line `StreamsBuilder builder = new StreamsBuilder();`.
- Создается фактический поток с входным топиком для чтения (родительский узел в графе)**: Points to the line `KStream<String, String> simpleFirstStream = builder.stream("src-topic", Consumed.with(stringSerde, stringSerde));`.
- Записывает преобразованный текст в другой топик (узел-приемник в графе)**: Points to the line `upperCasedStream.to("out-topic", Produced.with(stringSerde, stringSerde));`.
- Узел-обработчик, использующий лямбда-выражение (первый дочерний узел в графе)**: Points to the line `upperCasedStream.mapValues(value -> value.toUpperCase());`.

Мы создали свое первое приложение Kafka Streams. Вкратце перечислю этапы его создания, поскольку те же шаги вы будете выполнять при создании большинства приложений Kafka Streams.

1. Создание экземпляра `Properties` для определения конфигурации.
2. Создание объекта `Serde`.
3. Построение топологии обработки.
4. Запуск программы Kafka Streams.

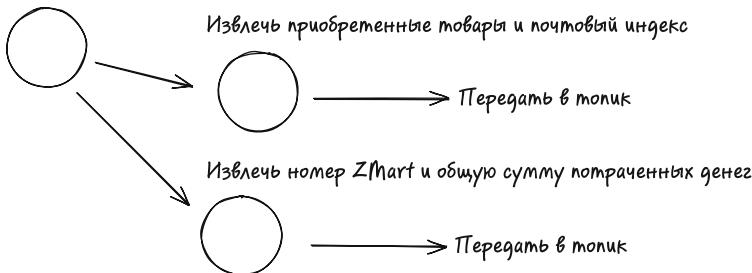
Перейдем теперь к более сложному примеру, который поможет нам изучить дополнительные инструменты Streams DSL.

## 6.4. МАСКИРОВКА НОМЕРОВ КРЕДИТНЫХ КАРТ И ОТСЛЕЖИВАНИЕ ВОЗНАГРАЖДЕНИЙ ЗА ПОКУПКИ

Представьте, что вы работаете инженером инфраструктуры в гиганте розничной торговли ZMart. В качестве платформы обработки данных в ZMart используется Kafka, и разработчики стремятся извлечь максимальную выгоду из возможности быстро и эффективно обрабатывать данные клиентов.

Итак, ваш начальник поручил вам создать приложение Kafka Streams для работы с записями о покупках, поступающими в потоковом режиме из магазинов ZMart. Диаграмма на рис. 6.6 иллюстрирует требования к программе потоковой обработки и довольно хорошо описывает, что эта программа должна делать.

*Маскировка номеров кредитных карт*



**Рис. 6.6.** Диаграмма новых требований к приложению ZMart

Вот эти требования.

1. Номера кредитных карт во всех объектах `Purchase` должны быть защищены, в данном случае путем маскирования первых 12 цифр.
2. Для определения паттернов покупок управления запасами на складах необходимо извлечь информацию о купленных товарах и почтовых индексах, которая далее будет записана в топик.

3. Необходимо собрать номера карт лояльности ZMart, а также данные о потраченных суммах и записать эту информацию в топик. Потребители топика на основе указанных данных смогут определить количество бонусных баллов.

Определившись с требованиями, приступим к созданию потокового приложения, удовлетворяющего бизнес-требованиям ZMart.

### 6.4.1. Создание узла-источника и маскирующего узла-обработчика

В первую очередь нужно создать узел-источник и первый узел-обработчик в топологии (листинг 6.7). Сделаем это, связав в цепочку вызовы двух методов `KStream`. Потомок узла-источника будет маскировать номера кредитных карт для защиты конфиденциальности клиентов.

#### Листинг 6.7. Создание узла-источника и первого узла-обработчика

```
KStream<String, RetailPurchase> retailPurchaseKStream =
    streamsBuilder.stream("transactions",
        Consumed.with(stringSerde, retailPurchaseSerde))
    .mapValues(creditCardMapper);
```

Узел-источник создается вызовом метода `StreamsBuilder.stream`, в который передаются строковый объект `Serde` по умолчанию, пользовательский `Serde` для объектов `RetailPurchase` и имя топика — источника сообщений для потока. В данном случае мы указываем только один топик, но могли бы указать список с несколькими топиками, перечислив их через запятую, или регулярное выражение, соответствующее именам нужных топиков.

В листинге 6.7 мы передали объекты `Serde` с помощью экземпляра класса `Consumed`, но могли и опустить его и передать только имя топика, чтобы задействовать объекты `Serde` по умолчанию, из настроек конфигурации.

Непосредственно за ним следует вызов метода `KStream.mapValues`, который принимает экземпляр `ValueMapper<V, V1>`. `ValueMapper` получает один параметр одного типа (в данном случае объект `RetailPurchase`) и отображает его в новое значение, возможно, другого типа. В этом примере метод `KStream.mapValues` возвращает объект того же типа (`RetailPurchase`), но уже с маскированным номером кредитной карты.

В методе `KStream.mapValues` вы не сможете получить ключ. Если для вычисления нового значения вам нужен ключ и при этом сам ключ не будет изменяться, то используйте интерфейс `ValueMapperWithKey<K, V, VR>`. Если, помимо значения, потребуется изменить еще и ключ, то используйте метод `KStream.map`, который принимает интерфейс `KeyValueMapper<K, V, KeyValue<K1, V1>>`. Рассмотрим пример применения обоих этих методов, начав с `ValueMapperWithKey` (листинг 6.8).

`ValueMapperWithKey` действует так же, как `ValueMapper`, но дополнительно дает доступ к ключу, что может понадобиться для преобразования значения. Здесь мы предполагаем, что ключ — это идентификатор клиента, и если он заканчивается на 333, то это означает, что он принадлежит корпоративному «агенту» и мы можем

удалить данные клиента из покупки. Завершив преобразования, метод возвращает обновленный объект `RetailPurchase`.

#### **Листинг 6.8.** `ValueMapperWithKey`

```
public class RetailValueMapperWithKey implements
    ValueMapperWithKey<String, RetailPurchase, RetailPurchase> {
    @Override
    public RetailPurchase apply(String customerIdKey,
        RetailPurchase value) { ← Метод apply применяет желаемые
            changes to value
        RetailPurchase.builder = value.toBuilder();
        if(customerIdKey != null && customerIdKey.endsWith("333")){
            builder.setCreditCardNumber("0000000000");
            builder.setCustomerId("0000000000");
        }
        return builder.build();
    }
}
```

Процесс использования `KeyValueMapper` выглядит аналогично, но при этом есть возможность изменить ключ, как показано в листинге 6.9.

#### **Листинг 6.9.** `KeyValueMapper`

```
public class RetailKeyValueMapper implements
    KeyValueMapper<String, RetailPurchase,
        KeyValue<String, RetailPurchase>> {
    @Override
    public KeyValue<String, RetailPurchase> apply(String key,
        RetailPurchase value) {
        RetailPurchase.builder = RetailPurchase.newBuilder();
        if(key != null && key.endsWith("333")){ ← Condition for updating
            builder.setCreditCardNumber("0000000000"); ← Condition for updating
            builder.setCustomerId("0000000000");
        }
        return KeyValue.pair(value.getStoreId(), builder.build()); ← Previous key
            is replaced by new one
    }
}
```

Здесь `KeyValueMapper` принимает ключ и значение, так же как `ValueMapperWithKey`, но вместо простого значения возвращает `KeyValue`, позволяя заменить прежний ключ новым. Иногда может понадобиться добавить ключ (что также требует использования интерфейса `KeyValueMapper`) или изменить существующий ключ, например, для выполнения агрегирования, требующего группировки записей по ключу. Агрегирование и влияние изменения ключей на операции с сохранением состояния мы рассмотрим в главе 7.

#### **ПРИМЕЧАНИЕ**

Напомню: Kafka Streams ожидает, что функции не будут иметь побочных эффектов, то есть не будут изменять исходный ключ и/или значение и возвращать новые объекты с изменениями.

## 6.4.2. Добавление узла, извлекающего закономерности при выполнении покупок

Теперь приступим к созданию второго узла-обработчика, отвечающего за извлечение из покупок географической информации, на основе которой ZMart сможет определить паттерны покупок и управлять запасами на складах в различных областях страны. При создании этой части топологии нам придется также решить еще одну задачу. Как заявили бизнес-аналитики ZMart, они хотели бы видеть отдельные записи для каждого товара в покупке и рассматривать покупки, сделанные в региональном масштабе, вместе.

Объект модели данных `RetailPurchase` содержит все товары в покупке, поэтому для каждого из них нужно будет создать новую запись. Кроме того, в качестве ключа покупки нужно будет добавить почтовый индекс. Наконец, вы должны добавить узел-приемник, отвечающий за запись полученных данных в топик Kafka.

В примере узла-обработчика закономерностей можно заметить обработчик `retailPurchaseKStream`, использующий оператор `flatMap`. Метод `KStream.flatMap` принимает `ValueMapper` или `KeyValueMapper`, который, в свою очередь, принимает одну запись и возвращает `Iterable` — любую коллекцию Java с новыми записями, возможно, другого типа. Обработчик `flatMap` «развертывает» `Iterable` в одну или несколько записей, пересылаемых в топологию. На рис. 6.7 показано, как это работает.



**Рис. 6.7.** `FlatMap` создает ноль или более записей из одной входной записи, разворачивая коллекцию, возвращаемую `KeyValueMapper` или `ValueMapper`

`flatMap` — это хорошо известная операция из функционального программирования, на вход которой подается коллекция элементов (часть Map в имени операции), которая разворачивается в последовательность записей. В нашем случае с Kafka Streams розничная покупка пяти товаров разворачивается в пять отдельных объектов `KeyValue` с ключами, соответствующими почтовому индексу и значениям объекта `PurchasedItem`.

Листинг 6.10 содержит код для `KeyValueMapper`.

**Листинг 6.10.** `KeyValueMapper`, возвращающий коллекцию объектов `PurchasedItem`

```

KeyValueMapper<String, RetailPurchase,
Iterable<KeyValue<String, PurchasedItem>>> retailTransactionToPurchases =
    (key, value) -> {
        String zipcode = value.getZipCode();           | Извлекает почтовый индекс из покупки
        return value.getPurchasedItemsList().stream()   |
            .map(purchasedItem ->
                KeyValue.pair(zipcode, purchasedItem))  |
            .collect(Collectors.toList());               | Использует Java-метод
                                                       | stream для создания
                                                       | списка пар KeyValue
    }
  
```

`KeyValueMapper` принимает отдельный объект покупки и возвращает список объектов `KeyValue`. Ключ — это почтовый индекс места, где была совершена покупка, а значение — это товар, включенный в покупку. Теперь поместим новый `KeyValueMapper` в конструируемую в данный момент часть топологии (листинг 6.11).

**Листинг 6.11.** Узел-обработчик, определяющий закономерности, и узел-приемник, выполняющий запись в Kafka

```
KStream<String, Pattern> patternKStream = retailPurchaseKStream
    .flatMap(retailTransactionToPurchases)
    .mapValues(patternObjectMapper);
```

Для каждого товара  
в покупке с помощью  
`flatMap` создается  
новый объект

```
patternKStream.print(Printed.<String, Pattern>toSysOut()
    .withLabel("patterns-stream"));
```

Отображает каждую покупку  
в объект закономерности

```
patternKStream.to("patterns",
    Produced.with(stringSerde, purchasePatternSerde));
```

Выводит запись в консоль

Записывает каждую  
созданную запись в топик  
Kafka с именем "patterns"

В этом примере мы объявили переменную для хранения ссылки на новый экземпляр `KStream`, а зачем это нужно, вы увидите в следующем разделе. Узел — обработчик закономерностей покупок переправляет полученные им записи своему описанному в вызове метода `KStream.to` дочернему узлу, который записывает данные в топик `patterns`. Обратите внимание, как объект `Produced` используется для передачи заранее созданного объекта `Serde`. Я также использовал обработчик `KStream#print`, который выводит пары «ключ — значение» в консоль; подробнее о просмотре записей в потоке мы поговорим в разделе 6.5.

Метод `KStream.to` — полная противоположность методу `KStream.source`. Вместо создания источника данных для топологии метод `KStream.to` задает узел-приемник, записывающий данные из экземпляра `KStream` в топик Kafka. Существуют также перегруженные версии метода `KStream.to`, которые позволяют динамически выбирать топик, и мы скоро это обсудим.

### 6.4.3. Создание узла, определяющего величину вознаграждения

Третим узлом-обработчиком в топологии является накопитель бонусов покупателей, благодаря которому ZMart имеет возможность отслеживать покупки, совершаемые членами клуба постоянных покупателей. Накопитель бонусов отправляет данные в топик, откуда их потребляют приложения в штаб-квартире компании ZMart, вычисляющие бонусы по завершении покупки (листинг 6.12).

**Листинг 6.12.** Третий узел-обработчик и завершающий узел, записывающий данные в Kafka

```
KStream<String, RewardAccumulator> rewardsKStream =
    retailPurchaseKStream.mapValues(rewardObjectMapper);
rewardsKStream.to("rewards", Produced.with(stringSerde, rewardAccumulatorSerde));
```

Для создания узла-обработчика вознаграждений мы воспользовались уже знакомым вам паттерном: создали новый экземпляр `KStream`, отображающий записи с данными о покупках в объекты нового типа. Мы также присоединили к накопителю бонусов узел-приемник, записывающий в топик объекты `KStream` с информацией о поощрениях, чтобы их можно было использовать для определения уровня лояльности покупателей.

Теперь, завершив создание приложения по частям, посмотрим на него целиком (листинг 6.13).

#### Листинг 6.13. Программа KStream для учета покупок клиентами ZMart

```
public class ZMartKafkaStreamsApp {

    @Override
    public Topology topology(Properties streamProperties) {

        StreamsBuilder streamsBuilder = new StreamsBuilder();

        KStream<String, RetailPurchase> retailPurchaseKStream =
            streamsBuilder.stream("transactions",
                Consumed.with(stringSerde, retailPurchaseSerde))
                .mapValues(creditCardMapper); ← Создается узел-источник и первый узел-обрабочик

        KStream<String, Pattern> patternKStream =
            retailPurchaseKStream
                .flatMap(retailTransactionToPurchases)
                .mapValues(patternObjectMapper); ← Создается узел-обрабочик PurchasePattern

        patternKStream.to("patterns",
            Produced.with(stringSerde, purchasePatternSerde));

        KStream<String, RewardAccumulator> rewardsKStream =
            retailPurchaseKStream.mapValues(rewardObjectMapper); ← Создается узел-обрабочик RewardAccumulator

        rewardsKStream.to("rewards",
            Produced.with(stringSerde, rewardAccumulatorSerde));
        retailPurchaseKStream.to("purchases",
            Produced.with(stringSerde, retailPurchaseSerde));

        return streamsBuilder.build(streamProperties);
    }
}
```

#### ПРИМЕЧАНИЕ

Я опустил некоторые подробности в листинге 6.13 для простоты. Учтите, что примеры кода из книги далеко не всегда самодостаточны. Примеры целиком можно найти в каталоге с исходным кодом (`src/main/java/bbejeck/chapter_6/ZMartKafkaStreamsApp.java`).

Как видите этот пример несколько сложнее, чем приложение Yelling, но поток информации в нем имеет похожую структуру. В частности, этапы создания остались прежними.

1. Создание экземпляра `StreamsConfig`.
2. Создание одного или нескольких экземпляров `Serde`.
3. Формирование топологии обработки.
4. Сбор всех компонентов воедино и запуск программы Kafka Streams.

Обратите также внимание на отсутствие логики, отвечающей за создание различных отображений из исходного объекта покупки в новые типы. Это сделано намеренно. Во-первых, код для `KeyValueMapper` или `ValueMapper` будет отличаться в каждом конкретном варианте использования, поэтому нет смысла приводить в книге конкретные реализации.

В этом приложении упоминалось создание объектов `Serde`, но я не объяснял, почему или как их создавать. Давайте потратим еще немного времени и обсудим роль этих объектов в приложениях Kafka Streams.

#### **6.4.4. Использование объектов Serde с сериализаторами и десериализаторами в Kafka Streams**

Как мы узнали в предыдущих главах, брокеры Kafka работают с записями, представленными в виде массивов байтов. За сериализацию при создании записей и десериализацию при потреблении отвечают клиенты. В фреймворке Kafka Streams действуют те же правила, поскольку он использует встроенных потребителей и производителей. Но вместо предоставления определенного десериализатора или сериализатора мы можем настраивать Kafka Streams с помощью объектов `Serde`, содержащих сериализатор и десериализатор.

Некоторые объекты `Serde` предоставляются клиентскими зависимостями «из коробки» (`String`, `Long`, `Integer` и т. д.), но для других объектов вам придется создавать свои объекты `Serde`.

В нашем первом примере, в приложении Yelling, требовался сериализатор/десериализатор лишь для строковых значений, реализации которых предоставляет фабричный метод `Serdes.String()`. В примере ZMart, однако, придется создавать свои экземпляры `Serde` в силу произвольности типов объектов. Мы увидим далее, что нужно, чтобы создать объект `Serde` для класса `Purchase`. Я не стану описывать создание других экземпляров `Serde`, поскольку оно следует тому же паттерну, только с другими типами.

##### **ПРИМЕЧАНИЕ**

Я включил это обсуждение создания объектов `Serde` исключительно для полноты, потому что исходный код уже содержит класс `SerdeUtil`, который предоставляет метод `protobufSerde`. Вы увидите его в примерах, реализующих шаги, описанные в этом разделе.

Для создания объекта Serde необходимо реализовать интерфейсы `Deserializer<T>` и `Serializer<T>`. Мы рассмотрели создание собственных сериализаторов и десериализаторов в разделе 3.7, поэтому я не буду повторять эти детали здесь. Для справки вы можете обратиться к исходному коду с реализацией для `ProtoSerializer` и `ProtoDeserializer` в пакете `bbejeck.serializers` в каталоге с примерами кода для этой книги.

Итак, чтобы создать объект `Serde<T>`, используем фабричный метод `Serdes.serdeFrom`, выполнив следующие шаги:

```
Deserializer<RetailPurchase> purchaseDeserializer = ← Создает десериализатор  
    new ProtoDeserializer<>(); ← для класса RetailPurchase  
Serializer<RetailPurchase> purchaseSerializer = ← Создает сериализатор  
    new ProtoDeserializer<>(); ← для класса RetailPurchase  
Map<String, Class<RetailPurchase>> configs  
    = new HashMap<>();  
    configs.put(false, RetailPurchase.class);  
    deserializer.configure(configs, isKey); ← Настройки  
    ← для десериализатора  
Serde<RetailPurchase> purchaseSerde = ← Создает Protobuf Serde  
    Serdes.serdeFrom(purchaseSerializer, purchaseDeserializer); ← для объектов RetailPurchase
```

Как вы можете видеть, объект `Serde` удобен в качестве контейнера для сериализатора и десериализатора для заданного объекта. Нам понадобилось создать свой объект `Serde` для объектов `Protobuf`, потому что пример не использует `Schema Registry`. Но это вполне возможно. Давайте ненадолго приостановимся и посмотрим, как настроить приложение `Kafka Streams` при использовании его с `Schema Registry`.

## 6.4.5. Kafka Streams и Schema Registry

В главе 4 мы обсудили, почему желательно использовать `Schema Registry` с приложениями на основе `Kafka`. Я кратко повторю эти причины здесь. Предметные объекты в вашем приложении представляют собой неявный контракт между различными пользователями приложения. Например, представьте, что одна команда разработчиков меняет тип поля с `java.util.Date` на `long` и вносит эти изменения в `Kafka`; нижестоящие потребительские приложения потеряют работоспособность из-за неожиданного изменения типа поля.

Используя схему и `Schema Registry` для ее хранения, можно значительно упростить реализацию контракта, обеспечив лучшую координацию и проверку совместимости. Кроме того, проект `Schema Registry` предоставляет (де)сериализаторы и объекты `Serde` с поддержкой `Schema Registry`, избавляя разработчиков от необходимости писать свой код сериализации.

### ПРИМЕЧАНИЕ

`Schema Registry` предоставляет классы `JSONSerde` и `JSONSchemaSerde`, но они не взаимозаменяемы! `JSONSerde` предназначен для объектов `Java`, которые используют `JSON` для описания объекта. `JSONSchemaSerde` предназначен для объектов, которые используют `JSONSchema` как формальное определение объекта.

Итак, что необходимо включить в `ZMartKafkaStreamsApp`, чтобы добавить поддержку Schema Registry? Для этого нужно лишь использовать экземпляры Serde, поддерживающие Schema Registry, выполнив следующие шаги.

1. Создать экземпляр одного из предоставленных классов Serde.
2. Настроить его, указав URL сервера Schema Registry.

Ниже демонстрируются конкретные шаги, которые нужно предпринять:

```
KafkaProtobufSerdePurchase> protobufSerde = ← Создается экземпляр KafkaProtobufSerde
    new KafkaProtobufSerde<>(Purchase.class); ← указанного типа класса, который передается
String url = "https://..."; ← URL экземпляра Schema Registry
Map<String, Object> configMap = new HashMap<>(); ← Помещает URL
configMap.put(← в HashMap
    AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG, url); ← Вызывает метод
    protobufSerde.conserde(configMap, false); ← KafkaProtobufSerde#configure
```

Итак, написав всего несколько строк кода, мы создали объект Serde с поддержкой Schema Registry, который можно использовать в приложении Kafka Streams.

#### ПРИМЕЧАНИЕ

Поскольку Kafka Streams содержит клиенты-потребители и клиенты-производители, те же правила относятся к эволюции схем и совместимости.

Мы далеко продвинулись в разработке приложения Kafka Streams, и нам еще многое предстоит сделать, но давайте остановимся и поговорим о процессе разработки и о том, как можно облегчить себе жизнь при создании приложений Kafka Streams.

## 6.5. ИНТЕРАКТИВНАЯ РАЗРАБОТКА

Мы создали граф для потоковой обработки записей ZMart с тремя узлами-обработчиками, записывающими данные в отдельные топики. Конечно, вы сможете во время разработки использовать для просмотра результатов консольный потребитель. Но вместо использования внешнего инструмента было бы удобнее, чтобы приложение Kafka Streams само выводило или регистрировало данные из интересующей нас точки внутри топологии. Такая визуальная обратная связь непосредственно из приложения могла бы очень повысить эффективность разработки. К счастью, такая возможность есть — методы `KStream.peek()` и `KStream.print()`.

`KStream.peek()` позволяет вам применять операции без сохранения состояния (через интерфейс `ForEachAction`) к каждой записи, проходящей через экземпляр `KStream`. Важно отметить, что применяемые операции не должны изменять ни ключ, ни значение, а лишь журналировать или собрать информацию в произвольных точках топологии. Давайте еще раз вернемся к приложению Yelling и добавим возможность видеть записи до и после их преобразования (некоторые детали опущены для простоты) (листинг 6.14). Полный исходный код вы найдете в папке `bbejeck/chapter_6/KafkaStreamsYellingAppWithPeek`.

**Листинг 6.14.** Вывод записей, протекающих через приложение Yelling

```
ForeachAction<String, String> sysout =  
    (key, value) ->  
        System.out.println("key " + key  
    + " value " + value);  
  
builder.stream("src-topic",  
    Consumed.with(stringSerde, stringSerde))  
    .peek(sysout) ← Выводит записи в консоль по мере  
    .mapValues(value -> value.toUpperCase())  
    .peek(sysout) ← Выводит преобразованные записи  
    .to("out-topic",  
    Produced.with(stringSerde, stringSerde));
```

Мы поместили вызовы метода `peek` для вывода записей в консоль в стратегически важных точках — до и после вызова `mapValues`.

Метод `KStream.print()` специально создавался для вывода записей. Некоторые примеры кода выше демонстрировали его применение, тем не менее рассмотрим его снова (некоторые детали опущены для простоты) (листинг 6.15).

**Листинг 6.15.** Вывод записей с помощью `KStream.print`

```
...  
KStream<...> upperCasedStream = simpleFirstStream.mapValues(...);  
upperCasedStream.print(Printed.toSysOut()); ← Вывод строк после преобразования как пример  
upperCasedStream.to(...);
```

В этом случае мы выводим сообщения сразу после преобразования. Есть ли разница между двумя рассмотренными подходами? Обратите внимание, что в отличие от `KStream.peek()` операция `KStream.print()` завершает цепочку вызовов — за ней не следуют вызовы других методов, потому что `print` — это терминальный метод.

Терминальные методы в Kafka Streams имеют тип возвращаемого значения `void`, поэтому за ним нельзя добавить в цепочку вызов другого метода. В интерфейсе `KStream` есть три терминальных метода: `print`, `foreach` и `to`. Помимо метода `print`, который мы обсудили, вы также будете использовать метод `to` для записи результатов в Kafka. Метод `foreach` можно использовать для обработки каждой записи, когда не нужно записывать результаты в Kafka, например, для вызова микросервиса. Существует еще один терминальный метод — метод `process`, но он считается устаревшим, и поэтому мы не будем его обсуждать. Новый метод `process` (добавленный в Apache Kafka 3.3) позволяет интегрировать DSL с Processor API, но об этом мы поговорим в главе 10.

Для вывода данных можно использовать любой из методов, но лично я предпочитаю `peek`, потому что его можно вставить в существующую цепочку вызовов. Однако это всего лишь мое личное предпочтение, которое может не совпадать с вашим.

К настоящему моменту мы познакомились с некоторыми базовыми возможностями приложений Kafka Streams, но это была лишь верхушка айсберга. Давайте продолжим знакомство и посмотрим, что можно сделать с потоком событий.

## 6.6. ВЫБОР СОБЫТИЙ ДЛЯ ОБРАБОТКИ

Выше мы видели, как применять операции к событиям, протекающим через приложение Kafka Streams. Но при этом все события обрабатывались одинаково. А что, если есть события, которые не должны обрабатываться? Или есть события с некоторым специфическим атрибутом, которые должны обрабатываться иначе?

К счастью, `KStream` имеет методы, обеспечивающие гибкость, необходимую для удовлетворения этих потребностей. Метод `KStream#filter` удаляет записи из потока, которые не соответствуют заданному предикату. `KStream#split` позволяет на основе заданных предикатов разделить поток на ветви для обработки записей по-разному. Чтобы исследовать практическое применение этих новых методов, обновим требования к приложению для ZMart:

- в ZMart изменили программу лояльности, и теперь бонусные баллы начисляются только за покупки на сумму больше десяти долларов. Это изменение исключает мелкие покупки для повседневных нужд из потока вознаграждений;
- компания ZMart расширилась и купила сеть магазинов электроники, а также сеть кофеен. Все данные о покупках в этих новых магазинах также будут проходить через ваше потоковое приложение. Однако вы должны отделить покупки, совершаемые в новых подразделениях, и обрабатывать их по-иному, а все остальные покупки обрабатывать как прежде.

### ПРИМЕЧАНИЕ

Начиная с данного момента я буду ради большей ясности сокращать все примеры кода, оставляя лишь самое важное. Если явно не указано иное, то можете смело предполагать, что код конфигурации и настройки остались прежними. Эти сокращенные примеры сами по себе могут не работать — полные листинги кода для данного примера можно найти в файле `src/main/java/bbejeck/chapter_6/ZMartKafkaStreamsFilteringBranchingApp.java`.

### 6.6.1. Фильтрация покупок

Первое изменение — отмена начисления бонусных баллов за мелкие повседневные покупки. Для этого добавим вызов `KStream.filter()` перед `KStream.mapValues`. Метод `filter` принимает интерфейс `Predicate` (мы будем использовать лямбда-выражение), имеющий один метод `test()`, который принимает два параметра — ключ и значение, — хотя на данном этапе нас интересует только значение.

### ПРИМЕЧАНИЕ

Есть также метод `KStream.filterNot`, но он выполняет обратную фильтрацию, то есть пропускает для дальнейшей обработки только записи, *не соответствующие* предикату.

После внесения этих изменений граф топологии обработки изменится, как показано в листинге 6.16.

Мы успешно обновили поток вознаграждений, исключив начисление бонусных баллов за мелкие покупки.

**Листинг 6.16.** Добавление фильтра, отбрасывающего мелкие повседневные покупки

```

Исходный поток
вознаграждений

KStream<String, RewardAccumulator> rewardsKStream = ←
    retailPurchaseKStream.filter((key, value) -> ←
        value.getPurchasedItemsList().stream()
            .mapToDouble((item -> item.getQuantity() ←
                * item.getPrice())))
                .sum() > 10.00)
        .mapValues(rewardObjectMapper); ←
            Отображает покупку в объект
            RewardAccumulator

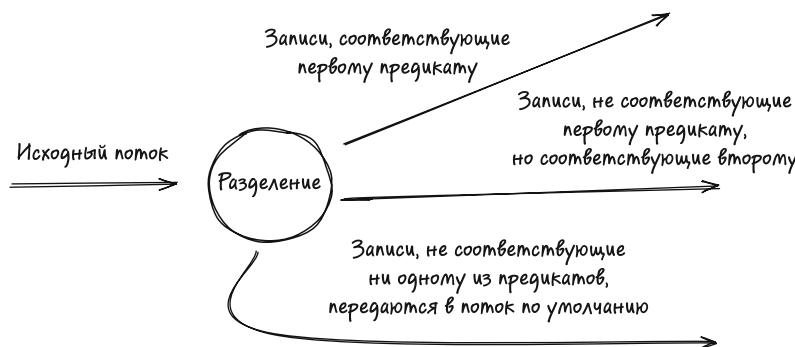
```

Метод KStream.filter принимает предикат Predicate<K,V>, представленный в виде лямбда-выражения

**6.6.2. Разделение/ветвление потока данных**

В поток покупок добавились новые события, которые нужно обрабатывать по-другому. В них все так же нужно маскировать информацию о кредитной карте, но после этого покупки, сделанные в сети кофеен и магазинов электроники, нужно отправить в другие топики. А к прежним событиям должны применяться прежние виды обработки.

Итак, исходный поток нужно разделить на три ветви: две для обработки новых и одну для обработки прежних событий. На первый взгляд разделение потоков выглядит сложным, но Kafka Streams предлагает весьма элегантный способ, как мы сейчас увидим. Схема на рис. 6.8 иллюстрирует концептуальную идею ветвления потока.



**Рис. 6.8.** Создание ветвей для двух специфических видов покупок

Вот основные шаги, которые нужно предпринять для разделения потока на ветви.

1. Использовать метод KStream.split(), который возвращает объект BranchedKStream.
2. Вызвать BranchedKStream.branch() с объектами Predicate и Branched в качестве параметров. Predicate содержит условие, проверяющее запись на соответствие и возвращающее true или false. Объект Branched содержит логику обработки записи. Каждый вызов этого метода создает новую ветвь в потоке.

3. Завершить ветвление вызовом `BranchedKStream.defaultBranch()` или `BranchedKStream.noDefaultBranch()`. Метод `defaultBranch` определяет ветвь по умолчанию, куда Kafka Streams будет помещать записи, не соответствующие предикатам. Метод `noDefaultBranch` просто отбросит несоответствующие записи. При вызове любого из методов, завершающих ветвление, возвращается `Map<String, KStream<K, V>`. Ассоциативный массив `Map` может содержать объекты `KStream` для нового ответвления, в зависимости от того, как построены объекты `Branched`. Позже мы рассмотрим другие варианты ветвления.

Объект `Predicate` — это логический вентиль для сопутствующего ему объекта `Branched`. Если условие возвращает значение `true`, то «вентиль» открывается и запись поступает в логику обработчика этой ветви.

#### ПРИМЕЧАНИЕ

Разделяя потоки `KStream`, нельзя изменять типы ключей или значений, потому что каждая ветвь имеет те же типы, что и родительская или исходная ветвь.

В данном случае мы должны отфильтровать два типа покупок, отправив их в отдельные ветви, а затем создать ветвь по умолчанию, куда будут поступать все остальные покупки. Ветвь по умолчанию — это исходный поток покупок, который будет обрабатывать все записи, не соответствующие ни одному из предикатов. Теперь, познакомившись с концепцией, рассмотрим код, который ее реализует (некоторые детали опущены для простоты) (листинг 6.17). Полный исходный код вы найдете в папке `bvejeck/chap-ter_6/ZMartKafkaStreamsFilteringBranchingApp`.

#### Листинг 6.17. Разделение потока

```
Predicate<String, Purchase> isCoffee =
  (key, purchase) ->
    purchase.getDepartment().equalsIgnoreCase("coffee"); ← Создание
                                                               предикатов,
                                                               определяющих
                                                               ветви

Predicate<String, Purchase> isElectronics =
  (key, purchase) ->
    purchase.getDepartment().equalsIgnoreCase("electronics"); ←

purchaseKStream.split() ← Разделение потока
  .branch(isCoffee, ←
    Branched.withConsumer(coffeeStream -> coffeeStream.to("coffee-topic"))); ← Запись покупок в кофейнях
  .branch(isElectronics, ←
    Branched.withConsumer(electronicStream -> electronicStream.to("electronics"))); ← Запись покупок электронных
  .defaultBranch(Branched.withConsumer(retailStream -> ←
    retailStream.to("purchases"))); ← Устройств в отдельный топик
                                    Ветвь по умолчанию, куда
                                    попадают все остальные покупки
```

В этом примере мы выделили покупки в кофейнях и магазинах электроники в два новых потока. Ветвление позволяет по-разному обрабатывать разные записи в пределах одного потока. В нашем простом примере каждый узел-обработчик всего лишь отправляет записи в разные топики, но вообще ветвление может быть настолько сложным, насколько это необходимо.

## ПРИМЕЧАНИЕ

В этом примере записи отправляются в несколько разных топиков. Хотя существует возможность настроить Kafka для автоматического создания топиков, не стоит полагаться на этот механизм. Если положиться на автоматическое создание топиков, их настройки будут основаны на значениях по умолчанию в файле `server.config`, которые могут отличаться от того, что вам нужно. Поэтому всегда следует заранее обдумывать, какие топики нужны, с каким числом разделов и коэффициентом репликации, и создавать их до запуска приложения Kafka Streams.

В этом примере мы разделили дискретные объекты `KStream`, которые не зависят друг от друга и не взаимодействуют ни с чем другим в приложении. А теперь давайте рассмотрим ситуацию, когда требуется разделить поток событий на отдельные компоненты, а затем объединить новые потоки с существующими в приложении.

Представим, что у нас есть датчики Интернета вещей (Internet of Things, IoT) и на начальном этапе мы объединили показания двух связанных датчиков в один топик, но, добавляя новые датчики, мы решили отправлять их показания в разные топики. Старые датчики прекрасно работают, и было бы слишком хлопотно возвращаться к ним и вносить необходимые изменения для приведения их обработки в соответствие с новой инфраструктурой. Поэтому мы решили написать приложение для разделения устаревшего потока на две ветви и объединения или слияния их с новыми потоками, несущими показания того же типа. Другой недостаток: показания старых датчиков близости выражены в футах, а новых — в метрах. Поэтому, помимо извлечения показаний расстояния в отдельный поток, необходимо преобразовать их в метры.

Теперь рассмотрим пример реализации разделения и слияния, начав с разделения (некоторые детали опущены для простоты) (листинг 6.18).

**Листинг 6.18.** Разделение потока так, чтобы получить доступ к новым потокам

```
KStream<String, Sensor> legacySensorStream =
    builder.stream("combined-sensors", sensorConsumed);
```

Разделяет поток и задает базовое имя для ключей в ассоциативном массиве

```
Map<String, KStream<String, Sensor>> sensorMap =
    legacySensorStream.split(Named.as("sensor-"))
        .branch(isTemperatureSensor, Branched.as("temperature"))
        .branch(isProximitySensor,
            Branched.withFunction(
                ps -> ps.mapValues(feetToMetersMapper), "proximity"))
        .noDefaultBranch();
```

Создает ветвь для показаний температуры и присваивает имя ключу

Создает ветвь для датчика близости вызовом ValueMapper

Указывает, что ветвь по умолчанию отсутствует, потому что, как известно, все записи делятся только на две категории

В целом каждый вызов `branch` приводит к созданию элемента в `Map`. Ключ задается как конкатенация имени, переданного в метод `KStream.split()`, и строки в объекте `Branched`. А роль значения играет экземпляр `KStream`, полученный в результате каждого вызова `branch`.

В первом примере ветвления вызовы `split` и последующие ветвления тоже возвращают `Map`, но пустой. Причина в том, что метод `Branched.withConsumer` (интерфейс `java.util.Consumer`) имеет тип возвращаемого значения `void`, поэтому `Branched` не добавляет элемент в `Map`. Но `Branched.withFunction` (интерфейс `java.util.Function`) принимает объект `KStream<K, V>` и возвращает экземпляр `KStream<K, V>`, поэтому

его результат попадает в Мар. Функция принимает разветвленный объект `KStream` и вызывает `MapValues` для преобразования показаний датчика приближения из футов в метры, потому что показания датчика в обновленном потоке должны быть указаны в метрах.

Хочу обратить ваше внимание на некоторые тонкости в этом примере. Вызов `branch` не получает функцию, но его результат все равно попадает в возвращаемый Мар. Как так получается? Когда передается параметр `Branched` только с именем, он обрабатывается так же, как при использовании экземпляра интерфейса `java.util.Function`, который возвращает предоставленный объект `KStream` и известен как *функция тождественного преобразования*. Но как выбрать между `Branched.withConsumer` и `Branched.withFunction`? Чтобы получить ответ на этот вопрос, пройдемся по следующему блоку кода в нашем примере (листинг 6.19).

#### **Листинг 6.19.** Разделение потока и получение доступа к вновь созданным потокам

```
KStream<String, Sensor> temperatureSensorStream = ← Поток с показаниями новых
    builder.stream("temperature-sensors", sensorConsumed); ← температурных датчиков IoT

KStream<String, Sensor> proximitySensorStream = ← Поток с обновленными
    builder.stream("proximity-sensors", sensorConsumed); ← показаниями датчиков
                                                               приближения IoT

temperatureSensorStream.merge(sensorMap.get("sensor-temperature"))
    .to("temp-reading", Produced.with(stringSerde, sensorSerde)); ← Объединяет старые
                                                               и новые показания
                                                               температуры

proximitySensorStream.merge(sensorMap.get("sensor-proximity"))
    .to("proximity-reading", Produced.with(stringSerde, sensorSerde)); ← Объединяет старые и новые
                                                               показания расстояния
```

Требования к разделению потока заключались в извлечении замеров различных датчиков IoT, размещении их в потоках по типам замеров и преобразовании любых показаний расстояния из футов в метры. Для решения этой задачи мы извлекаем из ассоциативного массива Мар экземпляры `KStream` с соответствующими ключами, созданными в предыдущем блоке кода, который реализует ветвление.

Для объединения старого и нового потоков используется оператор DSL `KStream.merge`, функциональный аналог `KStream.split`. Он объединяет различные объекты `KStream` в один. `KStream.merge` не дает никаких гарантий относительно упорядоченности записей из различных потоков, но относительный порядок в рамках каждого потока сохраняется. Другими словами, порядок обработки между устаревшим и обновленным потоками не гарантировается, но порядок внутри каждого потока сохраняется.

Теперь должно быть понятно, когда следует использовать `Branched.withConsumer` или `Branched.withFunction`. В последнем случае мы получаем доступ к разветвленному потоку `KStream` и можем осуществить интеграцию с внешним приложением, тогда как в первом случае доступ к разветвленному потоку не требуется.

На этом мы завершаем обсуждение ветвления и слияния и переходим к рассмотрению вопросов именования узлов топологии в DSL.

### 6.6.3. Именование узлов топологии

Когда вы описываете топологию с помощью DSL, Kafka Streams создает граф узлов-обработчиков, присваивая каждому из них уникальное имя. Имена узлов генерируются на основе имени функции обработчика и глобального, монотонно увеличивающегося номера. Чтобы убедиться в этом, можно получить описание топологии в виде объекта `TopologyDescription` и вывести его в консоль или в журнал (листинг 6.20).

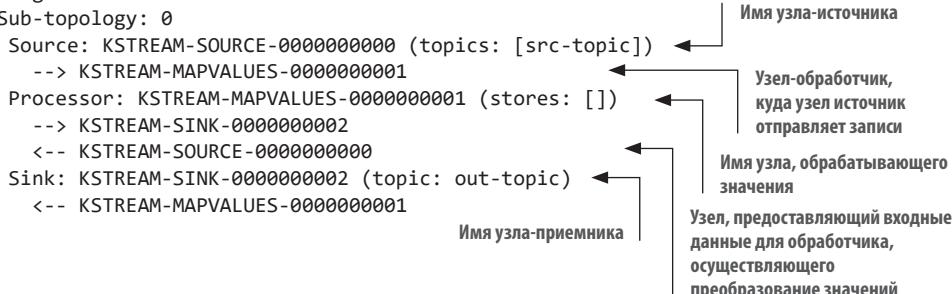
#### Листинг 6.20. Получение описания топологии и его вывод

```
TopologyDescription topologyDescription =
    streamsBuilder.build().describe();
System.out.println(topologyDescription.toString());
```

Этот код выведет в консоль следующие данные (листинг 6.21).

#### Листинг 6.21. Полное описание топологии KafkaStreamsYellingApplication

Topologies:



Как видите, имя первого узла заканчивается на `0`, имя второго узла `KSTREAM-MAPVALUES` заканчивается на `1` и т. д. Список `Sub-topology` описывает часть топологии, содержащую отдельный узел-источник, — все узлы ниже узла-источника являются членами этой части. Если бы мы определили второй поток с другим узлом-источником, то он был бы описан `Sub-topology: 1`. Более подробно о частях топологии мы поговорим позже в этой книге, когда будем рассматривать перераспределение.

Стрелки вправо (`->`) показывают направление передачи записей в топологии. Стрелки влево (`<-`), показывают происхождение потока записей — откуда текущий узел получает данные. Обратите внимание, что узел может пересыпать записи нескольким узлам и получать данные от нескольких узлов.

Описание топологии помогает получить представление о структуре приложения Kafka Streams. Однако в сложных приложениях использование автоматически выбираемых имен с номерами может затруднить понимание. По этой причине в Kafka Streams DSL поддерживается возможность явного указания имен узлов.

Все методы в Streams DSL имеют перегруженные версии, принимающие объект `Named`, с помощью которого можно указать имя узла в топологии. Явное определение имеет большое значение, позволяя с его помощью отразить роль узла в вашем

приложении, а не только то, что он делает. У конфигурационных объектов, таких как `Consumed` и `Produced`, есть метод `withName` для присваивания имени оператору. Давайте вернемся к `KafkaStreamsYellingApplication` и дадим свои имена всем узлам в нем (листинги 6.22, 6.23).

#### Листинг 6.22. Обновленное приложение KafkaStreamsYellingApplication

с именованными узлами

```
builder.stream("src-topic",
    Consumed.with(stringSerde, stringSerde)
        .withName("Application Input")) ← Присвоит имя узлу-источнику
    .mapValues((key, value) -> value.toUpperCase(),
        Named.as("Convert to Yelling")) ← Присвоит имя узлу-обработчику
    .to("out-topic",
        Produced.with(stringSerde, stringSerde)
            .withName("Application Output")) ← Присвоит имя узлу-приемнику
```

#### Листинг 6.23. Полное описание топологии с заданными именами узлов

Topologies:

```
Sub-topology: 0
Source: Application-Input (topics: [src-topic])
    --> Convert-to-Yelling
Processor: Convert-to-Yelling (stores: [])
    --> Application-Output
    <-- Application-Input
Sink: Application-Output (topic: out-topic)
    <-- Convert-to-Yelling
```

Теперь вы знаете, как получить описание топологии и составить представление о роли каждого узла в приложении, а не только о том, что они делают. Именование узлов-обработчиков становится особенно важным, когда заходит речь о состоянии, но мы вернемся к этому позже.

Далее мы посмотрим, как использовать динамическую маршрутизацию в приложениях Kafka Streams.

### 6.6.4. Динамическая маршрутизация сообщений

Представьте, что вам понадобилось различать покупки, сделанные в разных отделах магазина, например в отделе товаров для дома или в обувном отделе. Для этого можно использовать динамическую маршрутизацию записей. Метод `KStream.to()` имеет перегруженную версию, которая принимает экземпляр `TopicNameExtractor`, динамически определяющий название топика Kafka для сохранения записи. Обратите внимание, что топики должны существовать заранее, потому что по умолчанию Kafka Streams не создает топики с полученными названиями.

Итак, вернемся к примеру с ветвлением. У каждого объекта есть поле `department`, поэтому не будем создавать новые ветви, а обработаем события в общем потоке и используем `TopicNameExtractor` для определения имени топика, куда они будут записываться.

`TopicNameExtractor` имеет метод `extract`, в котором мы реализуем логику определения имени топика. Здесь мы проверим отдел, где сделана покупка, и на его основе выберем соответствующий топик. Если для отдела предусмотрен отдельный топик, то метод вернет его имя (в нашем примере оно совпадает с именем отдела). В противном случае будет возвращено имя топика для всех других событий (листинг 6.24).

#### Листинг 6.24. Реализация метода extract для определения имени топика

```
@Override
public String extract(String key,
                      Purchase value,
                      RecordContext recordContext) {
    String department = value.getDepartment();
    if (department.equals("coffee")
        || department.equals("electronics")) { ← Проверить, должно ли
                                                событие покупки помещаться
                                                в отдельный топик
        return department;
    } else { ← Имя топика
        return "purchases"; ← по умолчанию
    }
}
```

#### ПРИМЕЧАНИЕ

Интерфейс `TopicNameExtractor` имеет только один метод для реализации. Я решил использовать конкретный класс, потому что для него можно написать тест.

В этом примере имя топика определяется по значению, но точно так же можно было бы использовать ключ или комбинацию ключа и значения. Третий параметр метода `TopicNameExtract#extract` — это объект `RecordContext`, связанный с записью в Kafka Streams.

Контекст содержит метаданные о записи — отметку времени, смещение, топик, раздел и заголовки `Headers`. Заголовки мы уже обсуждали в главе 4, поэтому не будем рассматривать их повторно. В основном заголовки используются для получения информации о маршрутизации, и Kafka Streams предоставляет их через `ProcessorContext`.

Рассмотрим один из возможных вариантов получения имени топика через `Header`. В этом примере мы извлечем `Headers` из `RecordContext`. Сначала убедимся, что в `Headers` передана не пустая ссылка, а затем получим конкретную информацию о маршрутизации.

#### СОВЕТ

Kafka Streams открывает доступ к `RecordContext` только для экземпляра интерфейса `TopicNameExtractor`. Поэтому, чтобы получить доступ к заголовкам записи, вам нужно использовать Processor API или метод `process`, а затем вызвать метод `Record.headers()`.

После этого, исходя из значения, хранящегося в заголовке, определим и вернем имя топика. Поскольку заголовки являются необязательными и могут отсутствовать или не содержать заголовок, отвечающий за маршрутизацию, мы определили также имя топика по умолчанию (листинг 6.25).

**Листинг 6.25.** Определение имени топика по содержимому Header

```

public String extract(String key,
                      Purchase value,
                      RecordContext recordContext) {
    Headers headers = recordContext.headers(); ← Получает заголовки из RecordContext
    if (headers != null) {
        Iterator<Header> routingHeaderIterator =
            headers.headers("routing").iterator();
        if (routingHeaderIterator.hasNext()) {
            Header routing = routingHeaderIterator.next(); ← Извлекает экземпляр Header, отвечающий за маршрутизацию
            return new String(routing.value(),
                               StandardCharsets.UTF_8); ← Возвращает имя топика из значения в Header
        }
    }
    return defaultTopicName; ← Если информация о маршрутизации не найдена, то возвращается имя топика по умолчанию
}

```

Теперь вы знаете, как использовать Kafka Streams DSL API.

## ИТОГИ ГЛАВЫ

- Kafka Streams — это граф узлов обработки, называемый топологией. Каждый узел в топологии отвечает за выполнение некоторой операции над записями «ключ — значение», проходящими через него. Минимальное приложение Kafka Streams состоит из узла-источника, потребляющего записи из топика, и узла-приемника, возвращающего результаты обратно в топик Kafka. Минимальная конфигурация приложения Kafka Streams включает параметр `application.id` и настройки серверов начальной загрузки. Несколько приложений Kafka Streams с одинаковым значением `application.id` считаются одним логическим приложением.
- Есть возможность использовать функцию `KStream.mapValues` для преобразования значения входящих записей и даже изменять их тип. Эти преобразования не должны изменять исходные объекты. Другой метод, `KStream.map`, выполняет то же действие, но может использоваться для преобразования не только значения, но и ключа.
- Для выборочной обработки записей можно использовать операцию `KStream.filter`, которая отбрасывает записи, не соответствующие предикату. Предикат — это оператор, который принимает объект и возвращает значение `true` или `false`, в зависимости от соответствия этого объекта заданному условию. Метод `KStream.filterNot` выполняет обратную операцию: пересыпает дальше только пары «ключ — значение», не соответствующие предикату.
- Метод `KStream.branch` использует предикаты для разветвления потоков записей и направляет записи, соответствующие заданному предикату, в новый поток. Записи, не соответствующие предикату, отбрасываются. Ветвление — это элегантный способ разделения потока на несколько ветвей, каждая из которых

может обрабатываться независимо. `KStream.merge` выполняет обратную операцию — объединяет два объекта `KStream` в один поток.

- С помощью метода `KStream.selectKey` можно изменить существующий ключ или создать новый.
- Для просмотра содержимого записей в топологии можно использовать `KStream.print` и `KStream.peek` (предоставив экземпляр `ForeachAction`, который выполняет фактический вывод). `KStream.print` — это терминальная операция, то есть вслед за ней нельзя добавить в цепочку вызовы других методов. `KStream.peek` возвращает экземпляр `KStream`, что дает возможность использовать его в середине цепочки.
- Граф, сгенерированный приложением Kafka Streams, можно просмотреть с помощью метода `Topology.describe`. По умолчанию все узлы графа в Kafka Streams получают автоматически сгенерированные имена, что затрудняет изучение графа, особенно в сложных приложениях. Чтобы избежать этой ситуации, можно каждому узлу `KStream` дать свое осмысленное имя, описывающее его роль.
- Записи можно также отправлять в разные топики, передавая параметр `TopicNameExtractor` в метод `KStream.to`. Объект `TopicNameExtractor` может проанализировать ключ, значение или заголовки и на их основе определить имя нужного топика для передачи записи обратно в Kafka. Важно отметить, что все эти топики должны быть созданы заранее.



# Потоки данных и состояние

## В этой главе

- ✓ Операции с сохранением состояния и Kafka Streams.
- ✓ Использование хранилищ в Kafka Streams.
- ✓ Соединение потоков данных для получения дополнительной информации.
- ✓ Отметки времени как движущая сила Kafka Streams.

В предыдущей главе мы с головой окунулись в Kafka Streams DSL и создали топологию обработки для удовлетворения требований к потоковой обработке покупок. Построенная нами топология, хотя и нетривиальная, была одномерной в том смысле, что все преобразования и операции были без сохранения состояния. Мы рассматривали каждую транзакцию изолированно от других, безотносительно к другим событиям, происходящим в тот же момент или в пределах определенного промежутка времени до или после транзакции. Кроме того, мы работали только с отдельными потоками данных, игнорируя возможность получения дополнительной информации за счет их соединения.

В текущей главе мы постараемся извлечь из приложения Kafka Streams максимальный объем информации. Для этого нам придется использовать состояние. Состояние (*state*) — не что иное, как возможность восстанавливать просмотренную ранее информацию и связывать ее с текущей. Состояние можно применять по-разному. Мы рассмотрим пример этого, когда будем изучать операции с сохранением состояния, такие как накопление значений с помощью Kafka Streams DSL.

Еще один пример сохранения состояния, который мы обсудим далее, — соединение потоков данных. Оно очень похоже на операцию соединения в базах данных, например соединение записей из таблиц employee («сотрудник») и department («отдел») для генерации отчета по штату различных отделов компании.

Мы также узнаем, как должно выглядеть состояние и каковы требования к использованию состояния, когда будем обсуждать хранилища состояния в Kafka Streams. Наконец, мы поговорим о важности отметок времени и как они могут пригодиться в операциях с сохранением состояния, например, для обеспечения работы лишь с событиями, происходящими в пределах определенного промежутка времени, или для упрощения работы с поступающими не в том порядке данными.

## 7.1. С СОСТОЯНИЕМ И БЕЗ СОСТОЯНИЯ

Прежде чем перейти к примерам, обсудим разницу между операциями с состоянием и без состояния. В операциях без состояния вся информация, необходимая для обработки события, содержится в нем самом. Операции с состоянием сложнее, поскольку предполагают использование информации из предыдущих событий. Простым примером операции с состоянием может служить агрегирование. Например, взгляните на код в листинге 7.1.

### Листинг 7.1. Пример функции без состояния

```
public boolean numberIsOnePredicate (Widget widget) {  
    return widget.number == 1;  
}
```

Здесь объект `Widget` содержит всю информацию, необходимую для вычисления предиката; нет необходимости искать или хранить данные. А теперь рассмотрим пример функции с состоянием (листинг 7.2).

### Листинг 7.2. Пример функции с состоянием

```
public int count(Widget widget) {  
  
    int widgetCount = hashMap.compute(widget.id,  
        (key, value) -> (value == null) ? 1 : value + 1)  
  
    return widgetCount;  
}
```

Функция `count` в этом примере вычисляет общее количество виджетов с одинаковым значением ID. Чтобы выполнить подсчет, сначала нужно вспомнить значение счетчика для текущего значения ID, увеличить счетчик и сохранить результат. Если счетчик не существует, то он создается и ему присваивается начальное значение 1.

Это довольно тривиальный пример использования состояния, но он хорошо иллюстрирует суть. Мы используем числовые идентификаторы разных объектов в роли ключа в ассоциативном массиве для сохранения и извлечения значения некоторого типа, отражающего требуемое состояние. Мы также используем инициализирующую функцию для создания начального состояния. Эти основные шаги мы будем рассматривать и использовать на протяжении всей главы, хотя они будут реализованы гораздо надежнее, чем позволяет скромный `HashMap`!

## 7.2. ДОБАВЛЕНИЕ ОПЕРАЦИЙ С СОСТОЯНИЕМ В KAFKA STREAMS

Следующий вопрос, который мы рассмотрим: зачем использовать состояние при обработке потока событий? Ответ: чтобы иметь возможность отслеживать информацию или прогресс по связанным событиям. Например, рассмотрим приложение Kafka Streams, определяющее очки, заработанные игроками в онлайн-игре в покер. Участники играют раундами, и их счет в каждом раунде передается на сервер, а затем сбрасывается в начале следующего раунда. Игровой сервер выводит счет игрока в топик.

Поток событий без состояния позволит работать с текущим счетом, накапленным с начала раунда. Но для отслеживания общего количества очков, заработанных игроками, необходимо сохранять все предыдущие результаты.

Это подводит нас к нашему первому примеру операции с состоянием в Kafka Streams. Для этого примера мы используем операцию `reduce`. Операция `reduce`, или в более общем смысле операция `fold`, принимает несколько значений и сворачивает или объединяет их в один результат. Взгляните на рис. 7.1, чтобы понять суть этого процесса.

[17, 17, 12] → [46]

Операция `reduce` принимает список чисел и суммирует их. Так она «свертывает» входные данные в одно значение

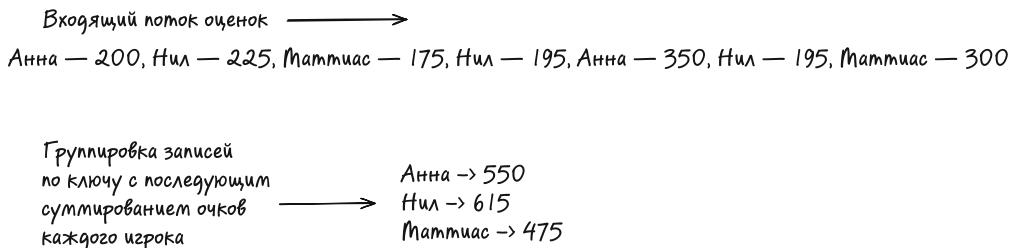
**Рис. 7.1.** Операция `reduce` принимает несколько входных данных и объединяет их в одно значение того же типа

Как показывает иллюстрация, операция `reduce` берет три числа и «свертывает» их в одно результирующее значение. Аналогично приложение Kafka Streams может принимать неограниченный поток очков и суммировать их для каждого игрока. Итак, мы описали саму операцию `reduce`, но в игре участвует еще и некоторая дополнительная информация, которую нужно упомянуть.

Описывая сценарий игры в онлайн-покер, я упомянул, что в ней участвуют отдельные игроки, поэтому логично было бы вести общий счет для каждого отдельного игрока. Но порядок следования очков игроков в потоке не гарантируется, поэтому нам нужна возможность их группировать. Напомню, что Kafka работает с парами «ключ — значение», поэтому предположим, что входящие записи имеют форму «ключ — значение»: `playerId/score`.

Итак, если ключом является `playerId`, то от Kafka Streams требуется только создать корзину или группу для накопления очков по ID, и в результате получатся суммы очков по игрокам. Возможно, будет полезно взглянуть на рис. 7.2, иллюстрирующий эту концепцию.

Группировка по `playerId` гарантирует суммирование очков отдельно для каждого игрока. Эта функция группировки в Kafka Streams похожа на группировку в операциях агрегирования в SQL.



**Рис. 7.2.** Группировка очков по playerId гарантирует суммирование очков отдельно для каждого игрока

### ПРИМЕЧАНИЕ

С этого момента я не буду показывать минимально необходимый код, осуществляющий настройки (то есть создающий экземпляр `StreamBuilder` и объекты `Serdes` для типов записей). В предыдущей главе мы подробно рассматривали эти компоненты, поэтому вы можете вернуться туда, если понадобится освежить память.

Теперь посмотрим на операцию `reduce` в действии (листинг 7.3).

#### Листинг 7.3. Выполнение операции reduce в Kafka Streams

```
KStream<String, Double> pokerScoreStream = builder.stream("poker-game",
    Consumed.with(Serdes.String(), Serdes.Double()));
```

```
pokerScoreStream
    .groupByKey()           ← Группировка по ключам для подсчета
    .reduce(Double::sum,     ← очков по отдельности
            Materialized.with(Serdes.String(), Serdes.Double()))
    .toStream()             ← Преобразование KTable в поток
    .to("total-scores",    ← Запись
        Produced.with(Serdes.String(), Serdes.Double()));      ← результатов
                                                               в топик
```

Это приложение Kafka Streams выводит пары «ключ — значение», например Нил, 650, и это непрерывный поток сумм очков, которые постоянно обновляются.

Просматривая код, можно увидеть, что первым вызывается метод `groupByKey`. Важно отметить, что группировка по ключу является предпосылкой для агрегирования с состоянием в Kafka Streams. А как быть, если ключа нет или нужно выбрать новый? Для случая выбора другого ключа интерфейс `KStream` предоставляет метод `groupBy`. Он принимает параметр `KeyValueMapper`, который можно использовать для выбора нового ключа. Пример выбора нового ключа будет показан в подразделе 7.2.3.

### 7.2.1. Подробности о группировке

Сделаем небольшое отступление и кратко обсудим тип значения, возвращаемого методом группировки, — `KGroupedStream`. Это промежуточный объект, предоставляющий методы `aggregate`, `count` и `reduce`. В большинстве случаев нет необходимости хранить ссылку на `KGroupedStream`, вы просто вызываете нужный вам метод, не задумываясь о существовании объекта.

В каких случаях нужно сохранить ссылку на `KGroupedStream`? Например, всякий раз, когда требуется применить несколько операций агрегирования с применением

группировки по одному и тому же ключу. Один из примеров мы рассмотрим позже, когда будем знакомиться с оконной обработкой. Теперь вернемся к нашей первой операции с состоянием.

Сразу после `groupByKey` мы вызываем `reduce`. Как уже объяснялось, объект `KGroupedStream` существует для нас прозрачно. Метод `reduce` имеет перегруженные версии, принимающие от одного до трех параметров. В данном примере используется версия с двумя параметрами, которая принимает интерфейс `Reducer` и объект конфигурации `Materialized`. В роли `Reducer` используется ссылка на статический метод `Double.sum`, который суммирует предыдущую сумму очков с новым счетом из игры.

Объект `Materialized` предоставляет объекты `Serdes`, которые хранят состояние. Использует для (де)серIALIZации ключей и значений. За кулисами Kafka Streams использует локальное хранилище для поддержки операций с состоянием. Хранилища хранят пары «ключ — значение» как массивы байтов, поэтому нужно передать объекты `Serdes` для сериализации сохраняемых записей и десериализации извлекаемых. Подробнее о хранилищах состояний мы поговорим в следующем разделе.

После `reduce` вызывается `toStream`, поскольку результатом всех операций агрегирования в Kafka Streams является объект `KTable` (который мы еще не рассматривали, но рассмотрим в следующей главе), и для пересылки результатов агрегирования последующим операторам необходимо преобразовать его в `KStream`.

После этого результаты агрегирования можно отправить в выходной топик через узел-приемник, представленный оператором `to`. Но узлы-обработчики с состоянием отличаются своим поведением пересылки от узлов-обрабочиков без состояния, поэтому потратим минуту, чтобы описать это различие.

Kafka Streams предоставляет механизм кэширования результатов операций с состоянием, и только когда Kafka Streams очищает кэш, результаты операций с состоянием пересылаются нижестоящим узлам в топологии. Очистка кэша происходит в двух случаях. Первый — когда кэш заполнится, то есть достигнет размера 10 Мбайт (порог, установленный по умолчанию), второй — когда Kafka Streams фиксирует транзакцию (каждые 30 секунд с настройками по умолчанию). Схема на рис. 7.3 поможет вам закрепить понимание особенностей кэширования в Kafka Streams.

На иллюстрации видно, что кэш находится перед передачей записей, поэтому вы не увидите несколько промежуточных результатов. В момент очистки кэша вы всегда будете видеть последние обновленные значения. Кэширование также уменьшает количество операций записи в хранилище состояний и связанный с ним топик журнализации изменений. Топики журнализации изменений — это внутренние топики, которые Kafka Streams создает для отказоустойчивости хранилищ состояний. Мы рассмотрим эти топики в подразделе 7.4.1.

## СОВЕТ

Если вам понадобится наблюдать каждый результат операции с состоянием, то отключите кэширование, присвоив параметру `StreamsConfig.CACHE_MAX_BYTES_BUFFERING_CONFIG` значение `0`. Но имейте в виду, что этот конфигурационный параметр влияет на все хранилища состояний в топологии. Отменить кэширование для отдельного хранилища можно с помощью объекта `Materialized`, вызовом его метода `Materialized.withCachingDisabled()`. Отключение кэширования лучше всего подходит для среды разработки.

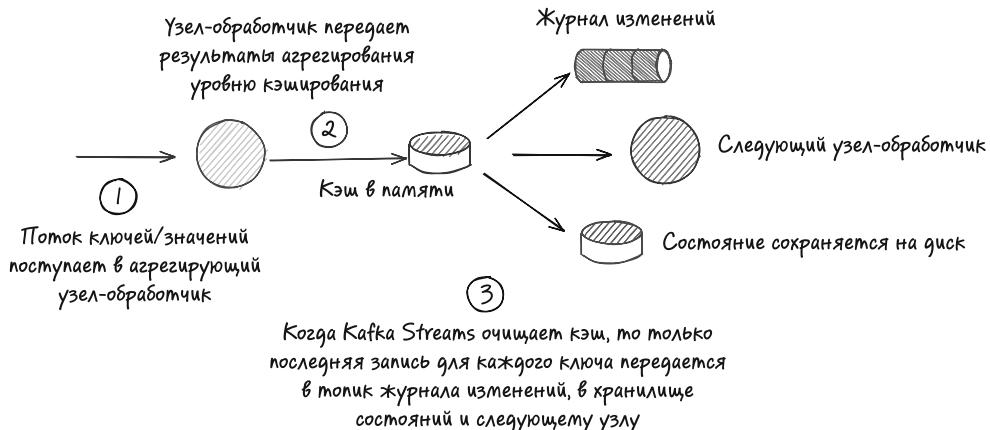


Рис. 7.3. Кэширование промежуточных результатов операции агрегирования

## 7.2.2. Агрегирование и свертка

Вы познакомились с одним оператором с состоянием, но у нас есть еще одна операция с состоянием. Вы могли заметить, что операция `reduce` возвращает тот же тип. Но иногда бывает нужно создать результат другого типа, и это можно сделать с помощью операции `aggregate`. Операция агрегирования имеет похожую концепцию, но с ее помощью можно вернуть тип, отличный от типа значения записи. Рассмотрим пример, чтобы понять, почему в некоторых случаях следует использовать `aggregate` вместо `reduce`.

Допустим, мы работаем в компании ETrade. Нам нужно создать приложение, отслеживающее биржевые транзакции отдельных клиентов, а не крупных институциональных трейдеров. Нам нужно вести подсчет общего объема купленных и проданных акций, объема продаж и покупок в долларовом выражении, а также самую высокую и самую низкую цены, наблюдаемые в любой момент.

Для представления этой информации мы создадим свой объект данных. В ситуациях, когда возникает потребность в использовании своего объекта данных, в игру вступает операция `aggregate`, которая позволяет вернуть результат другого типа, отличающегося от типа входного значения. В данном случае тип входной записи — это единичный объект биржевой транзакции. Результатом агрегирования будет объект другого типа, содержащий требуемую информацию, описанную в предыдущем абзаце.

Поскольку нам понадобится поместить наш объект в хранилище состояний, требующее сериализации, мы создадим схему Protobuf и используем служебные методы для создания Protobuf Serde. Исходя из подробно описанных требований к агрегированию в приложении, реализуем интерфейс `Aggregator<K, V, VR>` как конкретный класс, что позволит нам тестировать его независимо.

Рассмотрим часть реализации агрегатора. Поскольку этот класс содержит некоторую логику, не связанную с изучением Kafka Streams, я покажу только часть

реализации (листинг 7.4). Полную реализацию вы найдете в папке `bbejeck.chapter_7.aggregator.StockAggregator`.

**Листинг 7.4.** Агрегатор, создающий объект с обобщенной информацией о биржевых транзакциях

```
public class StockAggregator implements Aggregator<String,
                                         Transaction,
                                         Aggregate> {

    @Override
    public Aggregate apply(String key,
                           Transaction transaction,
                           Aggregate aggregate) {
        Aggregate.Builder currAggregate =
            aggregate.toBuilder(); ← Реализация метода apply: второй
                                   параметр — входная запись,
                                   третий — текущий агрегат

        double transactionDollars =
            transaction.getNumberShares() ← Для обновления объекта
                                         Protobuf необходимо
                                         использовать построитель
            * transaction.getSharePrice(); ← Объем транзакций
                                             в долларовом выражении

        if (transaction.getIsPurchase()) { ← Если транзакция является
            long currentPurchaseVolume =
                currAggregate.getPurchaseShareVolume();
            currAggregate.setPurchaseShareVolume(
                currentPurchaseVolume
                + transaction.getNumberShares());

            double currentPurchaseDollars =
                currAggregate.getPurchaseDollarAmount();

            currAggregate.setPurchaseDollarAmount(
                currentPurchaseDollars
                + transactionDollars);
        }
    }
}
```

Я не буду подробно описывать экземпляр `Aggregator`, поскольку основная тема этого раздела — как построить агрегирующее приложение Kafka Streams, а особенности реализации агрегирования будут различаться для разных случаев и в данном случае неважны. Но этот код показывает, как создаются данные о транзакциях для указанных акций. Теперь посмотрим, как подключить эту реализацию `Aggregator` к приложению Kafka Streams для сбора информации (листинг 7.5). Исходный код этого примера можно найти в `bbejeck.chapter_7.StreamsStockTransactionAggregations`.

#### ПРИМЕЧАНИЕ

Некоторые детали, такие как вывод записей в консоль, я убрал из исходного кода, представленного в книге. Наши последующие приложения Kafka Streams будут становиться все сложнее, и нам будет проще следовать главной цели — обучению, если я буду показывать только необходимые детали. Но вы всегда можете заглянуть в примеры в репозитории книги, где исходный код приводится без сокращений.

**Листинг 7.5.** Агрегирование в Kafka Streams

```

KStream<String, Transaction> transactionKStream =
    builder.stream("stock-transactions",
        Consumed.with(stringSerde, txnSerde)); ← Создание
                                                экземпляра KStream

transactionKStream.groupBy((key, value) -> value.getSymbol(), ← Группировка по ключу
    Grouped.with(Serdes.String(), txnSerde)) ← и передача функции
    .aggregate(() -> initialAggregate, ← Вызов функции агрегирования
        new StockAggregator(),
        Materialized.with(stringSerde, aggregateSerde))
    .toStream() ← Преобразование результата агрегирования KTable в KStream
    .peek((key, value) -> LOG.info("Aggregation result {}", value))
    .to("stock-aggregations", Produced.with(stringSerde, aggregateSerde)); ← Запись результата
                                                агрегирования в топик

```

Начало приложения уже хорошо знакомо: создается экземпляр `KStream`, выполняется подписка на топик и предоставляется объект `Serdes` для десериализации. Вы уже видели `groupByKey` в примере с `reduce`, но здесь во входных записях роль ключа играет идентификатор клиента, а нам нужно сгруппировать записи по биржевому символу акций. Поэтому, чтобы изменить ключ, мы используем `GroupBy`, который принимает `KeyValueMapper` и лямбда-функцию. В нашем случае лямбда-функция возвращает символ акции из записи и тем самым обеспечивает правильную группировку.

Поскольку топология изменяет ключ, библиотека Kafka Streams должна перераспределить данные. В следующем разделе я расскажу о перераспределении более подробно, а пока достаточно знать, что Kafka Streams позаботится об этом автоматически.

Наконец, мы добрались до сути примера — применения операции агрегирования (листинг 7.6). Операция агрегирования немного отличается от операции `reduce`, требуя начального значения для первого применения. В `reduce` начальным значением служит первое входное значение.

**Листинг 7.6.** Агрегирование в Kafka Streams

```

transactionKStream.groupBy((key, value) -> value.getSymbol(),
    Grouped.with(Serdes.String(), txnSerde))
    .aggregate(() -> initialAggregate, ← Вызов функции агрегирования
        new StockAggregator(),
        Materialized.with(stringSerde, aggregateSerde))
    .toStream() ← Преобразование результата агрегирования KTable в KStream
    .peek((key, value) -> LOG.info("Aggregation result {}", value))
    .to("stock-aggregations", Produced.with(stringSerde, aggregateSerde)); ← Запись результата
                                                агрегирования
                                                в топик

```

Поскольку Kafka Streams не может знать, что именно будет получено в результате агрегирования, мы должны передать начальное значение. В нашем случае это экземпляр объекта `Aggregate` с неинициализированными полями.

Во втором параметре мы передаем реализацию `Aggregator` с нашей логикой агрегирования записей. Необязательный третий параметр — это объект `Materialized`, который здесь используется для передачи объекта `Serdes`, необходимый для работы с хранилищем состояний.

В заключительной части приложения экземпляр `KTable`, полученный в результате агрегирования, преобразуется в `KStream` для дальнейшей передачи в топик. Здесь мы также использовали операцию `peek` перед узлом-приемником, чтобы посмотреть, как выглядят результаты. Обычно оператор `peek` используется только на этапе разработки или отладки.

#### ПРИМЕЧАНИЕ

Напомню, что при выполнении примеров Kafka Streams использует кэширование, поэтому вы не увидите результатов, пока кэш не будет очищен.

Итак, теперь вы познакомились с основными приемами выполнения операций с состоянием в Kafka Streams DSL: `reduce` и `aggregation`. Есть еще одна операция с состоянием, которая заслуживает упоминания, — операция `count`. Это «синтаксический сахар», предназначенный для увеличения счетчика агрегирования. Операцию `count` можно использовать для подсчета общего количества чего-то, например количества раз, когда пользователь выполнял вход на сайт, или количества замеров, полученных от датчика IoT. Я не буду показывать здесь пример применения этой операции, но вы можете увидеть его в примерах исходного кода в папке `bbejeck/chapter_7/StreamsCountingApplication`.

В предыдущем примере, где мы строили агрегаты биржевых транзакций, я упомянул, что изменение ключа для нужд агрегирования требует перераспределения данных. Далее обсудим этот вопрос более подробно.

### 7.2.3. Перераспределение данных

В примере агрегирования мы видели, что изменение ключа потребовало перераспределения данных между разделами. Давайте подробно обсудим, почему Kafka Streams перераспределяет данные и как это перераспределение происходит. Но сначала выясним, почему возникает необходимость перераспределения.

В предыдущей главе вы узнали, что ключ записи Kafka определяет раздел, в который она помещается. Когда изменяется ключ, высока вероятность, что запись должна быть помещена в другой раздел. Поэтому если вы изменили ключ, например, в процессе агрегирования и у вас есть узел-обработчик, зависящий от него, то Kafka Streams перераспределит данные и поместит записи с новыми ключами в правильный раздел. Взгляните на рис. 7.4, который демонстрирует этот процесс.

Как видите, перераспределение — это не что иное, как создание записей в топике с последующим немедленным их потреблением. Когда встроенный производитель Kafka Streams передает записи брокеру, он использует обновленный ключ для выбора нового раздела. За кулисами Kafka Streams вставляет новый узел-приемник для создания записей и новый узел-источник для их потребления. На рис. 7.5 показано состояние до и после обновления топологии Kafka Streams.

Изначально все ключи пустые, поэтому распределение выполняется в циклическом порядке, в результате чего записи с одинаковым идентификатором оказываются в разных разделах

### Исходный топик

#### Раздел 0

```
(null, {"id":"5", "info":"123"} )  
(null, {"id":"4", "info":"abc"} )
```

Для перераспределения поле ID  
выбирается на роль ключа, после чего  
записи отправляются в топик

### Топик для перераспределения

#### Раздел 0

```
("4", {"id":"4", "info":"def"} )  
("4", {"id":"4", "info":"abc"} )
```

#### Раздел 1

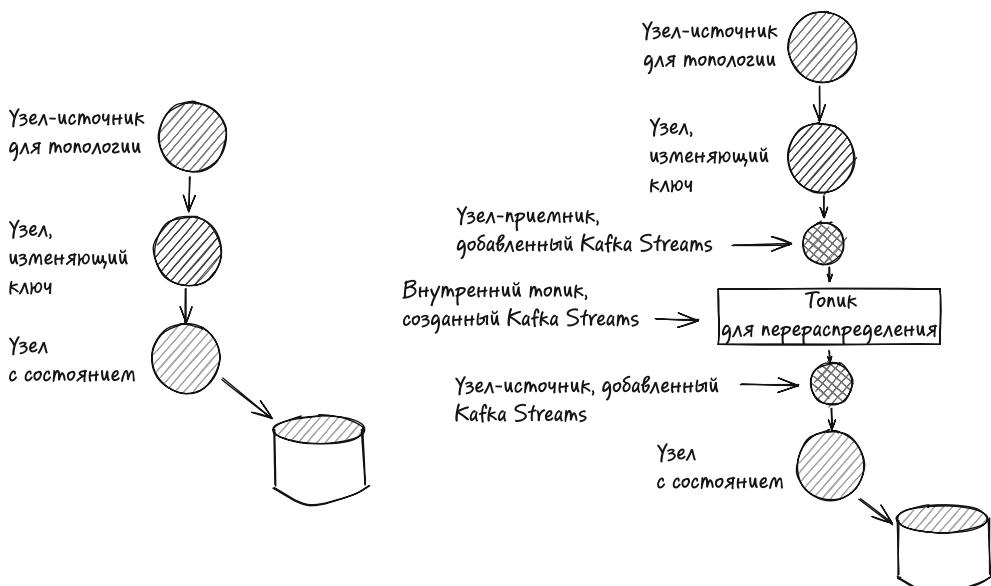
```
null, {"id":"5", "info":"456"} )  
(null, {"id":"4", "info":"def"} )
```

#### Раздел 1

```
("5", {"id":"5", "info":"456"} )  
("5", {"id":"5", "info":"123"} )
```

После заполнения ключей все записи  
с одинаковым ID попадают в один раздел

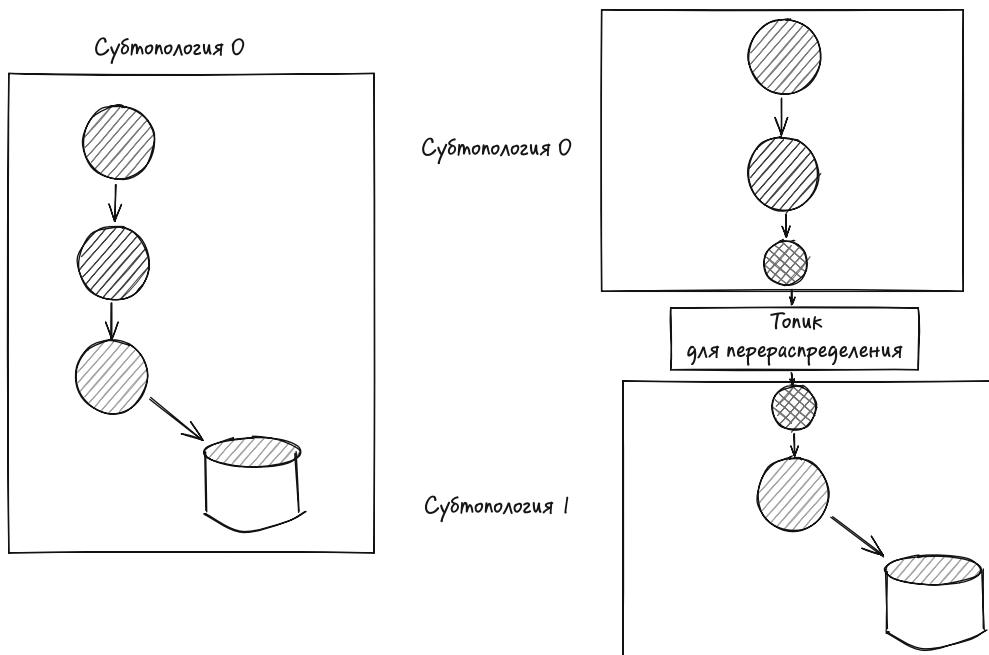
**Рис. 7.4.** Перераспределение: изменение исходного ключа вызывает перемещение записей в другие разделы



**Рис. 7.5.** Обновленная топология, куда Kafka Streams добавила узел-приемник и узел-источник для перераспределения данных

Вновь добавленный узел-источник создает новую субтопологию внутри топологии приложения. Субтопология — это часть общей топологии и повторно использует общий узел-источник. На рис. 7.6 показана обновленная версия топологии перераспределения, демонстрирующая структуру субтопологии. Любые узлы-обработчики, следующие после нового узла-источника, являются частью новой субтопологии.

Что именно заставляет Kafka Streams выполнить перераспределение? Определяющим фактором является наличие операции изменения ключа *и* последующей операции, зависящей от ключа, например `groupByKey`, агрегирование или соединение (соединения мы обсудим чуть ниже). В противном случае, если ни одна из последующих операций не зависит от ключа, Kafka Streams оставит топологию как есть. Рассмотрим пару примеров в листингах 7.7 и 7.8, чтобы прояснить этот момент.



**Рис. 7.6.** Добавление узла-источника и узла-приемника для перераспределения создает новую субтопологию

### Листинг 7.7. Примеры, когда необходимо перераспределение

```
myStream.groupBy(...).reduce(...)... ← Используется groupBy и за ней следует reduce
myStream.map(...).groupByKey().reduce(...)... ← Выполняется map и за ней groupByKey
filteredStream = myStream.selectKey(...).filter(...); ... ← Используется selectKey для выбора
filteredStreaam.groupByKey().aggregate(...)... ← нового ключа; полученный KStream
позднее вызывает groupByKey
```

Эти примеры показывают, что, когда выполняется операция, способная изменить ключ, Kafka Streams устанавливает внутренний логический флаг `repartitionRequired`. Kafka Streams не может знать, изменился ли ключ, поэтому автоматически выполняет перераспределение данных, если обнаруживает операцию, зависящую от ключа, и установленный внутренний флаг. С другой стороны, если ключ меняется, но агрегирование или соединение не выполняется, то топология остается прежней (см. листинг 7.8).

#### Листинг 7.8. Примеры, когда перераспределение не требуется

```
myStream.map(...).peek(...).to(...); ← Используется операция map, но после нее нет операции, зависящей от ключа
myStream.selectKey(...).filter(...).to(...); ← Используется selectKey, но после нее нет операции, зависящей от ключа
```

В этих примерах, даже если ключ изменится, это не повлияет на результаты последующих операций. Например, фильтрация записей зависит от того, что вернет предикат — `true` или `false`. Кроме того, поскольку эти экземпляры `KStream` отправляются в топик, записи с обновленными ключами окажутся в правильных разделах.

Поэтому важно использовать операции, изменяющие ключ (`map`, `flatMap`, `transform`), только когда действительно нужно изменить ключ. В противном случае лучше использовать узлы-обработчики, работающие только со значениями (то есть `mapValues`, `flatMapValues` и т. д.). В таких случаях Kafka Streams не будет перераспределять данные. Существуют перегруженные версии методов `xValues`, предоставляющие доступ к ключу при обновлении значения, но изменение ключа в них приведет к неопределенному поведению.

#### ПРИМЕЧАНИЕ

При группировке записей перед агрегированием используйте `groupByKey`, только когда нужно изменить ключ, в противном случае используйте `groupByKey`.

Прежде чем завершить разговор о перераспределении, обсудим еще одну тему: непреднамеренное создание избыточных узлов перераспределения и способы предотвращения такой ситуации. Представьте, что у вас есть приложение с двумя входными потоками. Вам нужно выполнить агрегирование первого потока и объединить его со вторым. Код мог бы выглядеть примерно так, как в листинге 7.9 (некоторые детали для простоты опущены).

Этот пример кода достаточно прост. Мы берем `originalStreamOne` и меняем ключ, чтобы выполнить агрегирование и соединение. Поэтому мы используем операцию `selectKey`, которая устанавливает флаг `repartitionRequired` в возвращаемом `KStream`. Затем вызываем `count()` и `join` с `inputStreamOne`. В этом примере Kafka Streams создаст за кулисами два топика для перераспределения, один для оператора `groupByKey`, а другой для `join`, хотя в действительности нужно только одно перераспределение.

Чтобы понять происходящее, взгляните на топологию для этого примера, изображенную на рис. 7.7. Обратите внимание, что здесь выполняются два перераспределения, тогда как достаточно только первого, выполняемого после изменения ключа.

**Листинг 7.9.** Изменение ключа с последующим агрегированием и соединением

```
KStream<String, String> originalStreamOne = builder.stream(...);

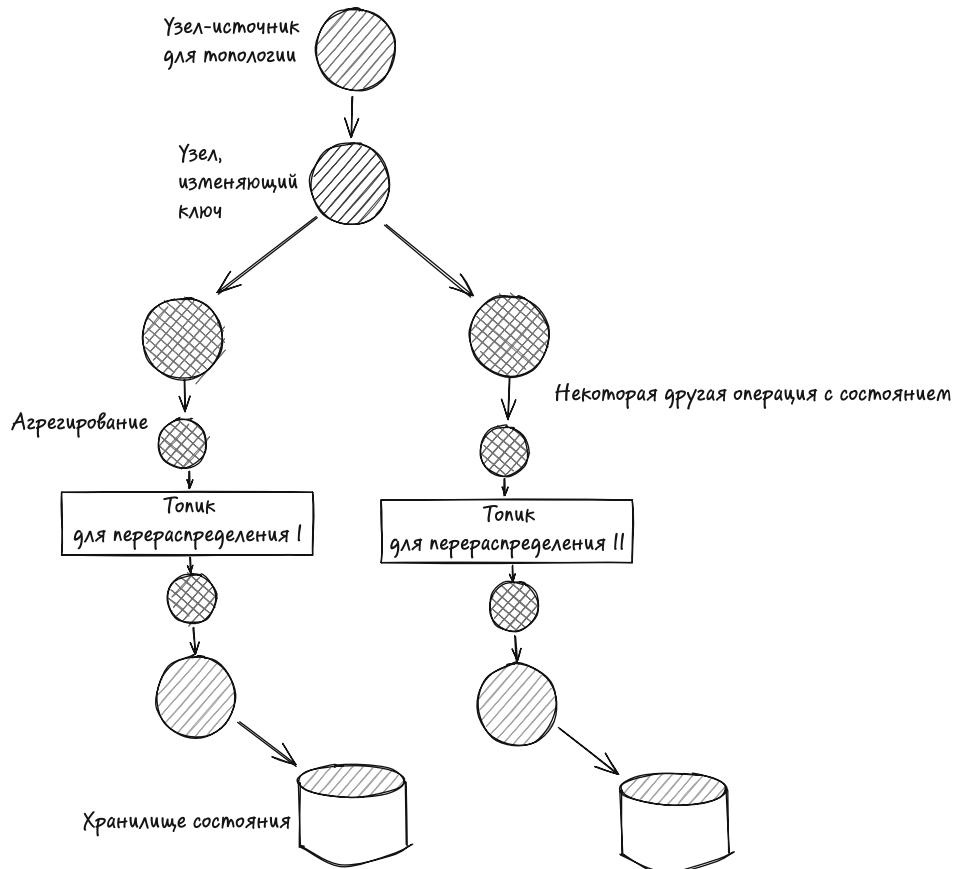
KStream<String, String> inputStreamOne = originalStreamOne.selectKey(...);           | Изменяет ключ исходного потока
                                                                 | и устанавливает флаг needsRepartition

KStream<String, String> inputStreamTwo = builder.stream(...);                         | Второй поток

inputStreamOne.groupByKey().count().toStream().to(...);                                | Выполняет группировку
                                                                 | по ключу, запуская
inputStreamTwo.join(inputStreamOne, (v1, v2)-> v1+"."+v2,                           | одно перераспределение
JoinWindows.ofTimeDifferenceWithNoGrace(...),
StreamJoined.with(...);
```

...  
 Код листинга 7.9 показывает последовательность операций для изменения ключа и последующего соединения двух потоков. Комментарии объясняют каждую строку:

- Изменение ключа и установка флага `needsRepartition` для исходного потока `originalStreamOne`.
- Создание второго потока `inputStreamTwo` из исходного потока `originalStreamOne` с измененным ключом.
- Группировка по ключу и подсчет количества элементов для каждого ключа в первом потоке.
- Соединение потоков `inputStreamOne` и `inputStreamTwo` по ключу, с использованием функции `join` и `StreamJoined.with` для настройки окна времени.



**Рис. 7.7.** Избыточные перераспределения из-за операции изменения ключа, выполненной ранее в топологии

После применения операции изменения ключа к `originalStreamOne` получившийся в результате `KStream`, `inputStreamOne` наследует флаг `repartitionRequired = true`. Соответственно, передача любого `KStream`, полученного из `inputStreamOne`, узлу обработчику, который зависит от ключа, вызовет перераспределение.

Можно ли как-то предотвратить второе перераспределение? Эта проблема имеет два решения. Первое — выполнить перераспределение вручную и тем самым сбросить флаг в `false`, чтобы последующие потоки не вызывали перераспределения. Другой вариант — позволить Kafka Streams самой решить проблему, включив оптимизацию. Давайте сначала рассмотрим ручной подход.

### ПРИМЕЧАНИЕ

Топики для перераспределения занимают некоторое дисковое пространство, однако Kafka Streams активно вычищает записи из них, поэтому вам не придется беспокоиться об их размере на диске. Но операции перераспределения увеличивают задержку обработки, поэтому желательно избегать ненужных перераспределений.

#### 7.2.4. Проактивное перераспределение

Для случаев, когда может понадобиться вручную перераспределить данные, `KStream` предоставляет метод `repartition`. В листинге 7.10 показано, как вручную выполнить перераспределение после изменения ключа (некоторые детали опущены для простоты).

**Листинг 7.10.** Изменение ключа, перераспределение и выполнение агрегирования и соединения

```
KStream<String, String> originalStreamOne = builder.stream(...);
KStream<String, String> inputStreamOne =
    originalStreamOne.selectKey(...);           ← Изменяет ключ, устанавливая флаг
                                                «требуется перераспределение»

KStream<String, String> inputStreamTwo = builder.stream(...);

KStream<String, String> repartitioned =
    inputStreamOne.repartition(Repartitioned
        .with(stringSerde, stringSerde)
        .withName("proactive"));                  ← Вызов метода repartition,
                                                которому передаются объекты Serdes
                                                для (де)серIALIZации ключей и значений,
                                                а также имя топика для перераспределения

repartitioned.groupByKey().count().toStream().to(...);          ← Агрегирование
                                                                перераспределенного потока

KStream<String, String> joinedStream = inputStreamTwo.join(...)

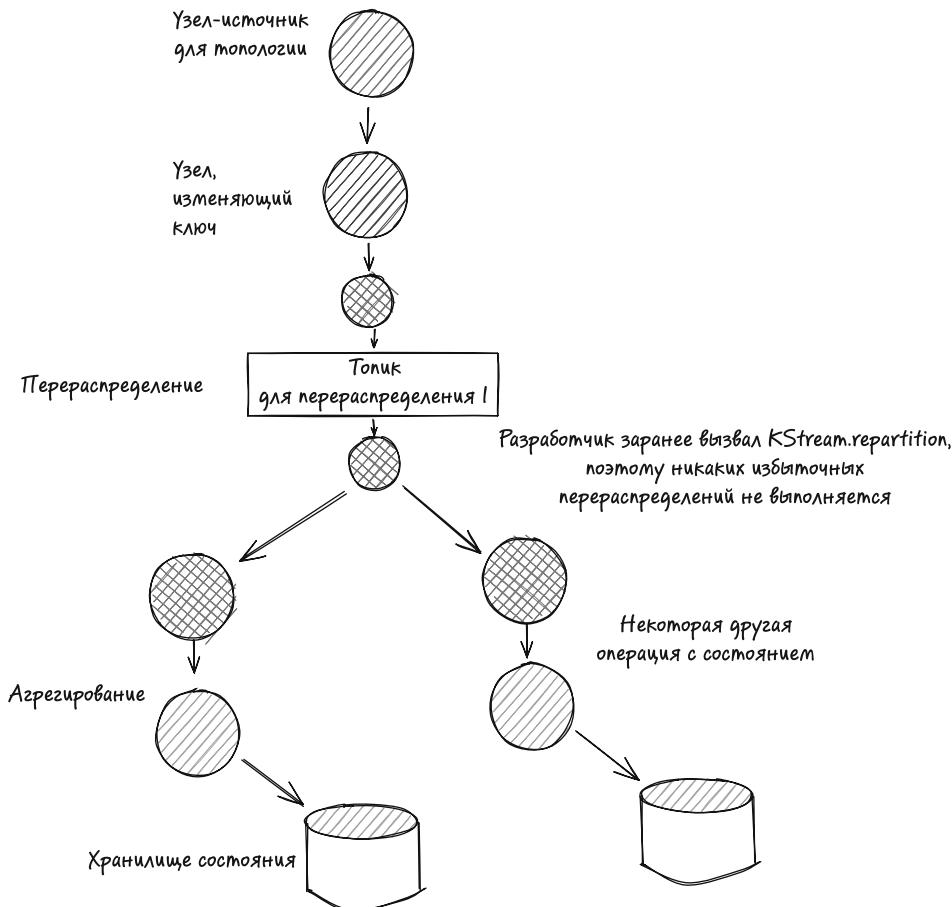
....
```

← Соединение с перераспределенным потоком

Этот код имеет только одно отличие: добавлена операция `repartition` перед вызовом `groupByKey`. В результате Kafka Streams создаст новую комбинацию узлов источника и приемника, породив новую субтопологию. Взгляните, как теперь выглядит топология (рис. 7.8). Разница сразу заметна.

Эта новая субтопология гарантирует сохранение новых ключей в правильные разделы, и, что не менее важно, сброс флага `needsRepartition` в возвращаемом объекте

**KStream.** В результате все последующие операции с состоянием, применяемые к потокам этого объекта **KStream**, не будут запускать перераспределения (если только снова не будет выполнена операция, изменяющая ключ).



**Рис. 7.8.** Благодаря упреждающему перераспределению оно выполняется только один раз, что дает возможность выполнить больше операций с состоянием, не вызывая избыточного перераспределения

Метод **KStream.repartition** принимает один параметр — объект конфигурации **Repartitioned**, позволяющий указать:

- объекты Serdes для ключа и значения;
- базовое имя топика;
- количество разделов для топика;
- экземпляр **StreamPartitioner**, если нужно настроить распределение записей в разделы.

Давайте ненадолго приостановимся и рассмотрим некоторые из этих настроек.

Передача базового имени топика для перераспределения всегда желательна. Я использую термин «базовое имя», потому что Kafka Streams добавляет к этому имени префикс `<application-id>` со значением из конфигурации и суффикс `-repartition`.

Если, к примеру, задан идентификатор приложения `streams-financial` и базовое имя `stock-aggregation`, то будет создан топик для перераспределения с именем `streams-financial-stock-aggregation-repartition`. Есть две причины, почему всегда полезно указывать базовое имя. Во-первых, наличие осмысленного имени топика поможет понять его роль при выводе списка топиков в кластере Kafka. Во-вторых, и это особенно важно, указанное вами имя останется неизменным, даже если вы измените часть топологии, предшествующей перераспределению. Напомню, что если имена узлов-обработчиков не указываются явно, то Kafka Streams генерирует их автоматически и часть этих имен представлена числом с ведущими нулями — значением глобального счетчика.

Иначе говоря, после добавления или удаления операторов в потоке выше операции перераспределения, которым не присваиваются имена явно, их имена изменятся из-за изменений в глобальном счетчике. Это изменение может способствовать появлению проблем при повторном развертывании существующего приложения. Подробнее о важности именования компонентов приложений Kafka Streams с состоянием я расскажу в подразделе 7.4.5.

#### ПРИМЕЧАНИЕ

Объект `Repartitioned` имеет четыре параметра, но совсем не обязательно указывать их все. Можно определить только нужную комбинацию параметров.

Указание количества разделов в топике для перераспределения особенно полезно в двух случаях: для обеспечения совместимого секционирования разделов с обеих сторон соединений и увеличения количества задач для получения более высокой пропускной способности. Обсудим сначала требование к уравниванию числа разделов. При выполнении соединений обе стороны должны иметь одинаковое количество разделов (причины мы обсудим, когда вы начнете знакомиться с операцией соединения в разделе 7.3). Таким образом, с помощью операции `repartition` можно изменить количество разделов, чтобы последующая операция соединения выполнялась без изменения исходного топика, а все изменения сохранялись внутри приложения.

### 7.2.5. Перераспределение для увеличения количества задач

Количество разделов определяет количество задач и в конечном счете количество активных потоков выполнения, которые может иметь приложение. Один из способов увеличить вычислительную мощность — увеличить количество разделов, потому что это приведет к увеличению числа задач и, соответственно, потоков выполнения, обрабатывающих записи. Kafka Streams старается равномерно назначать задачи всем приложениям с одинаковым идентификатором, поэтому такой подход к увеличению

пропускной способности особенно полезен в средах, где можно гибко изменять количество запущенных экземпляров.

Теоретически можно было бы увеличить количество разделов в исходном топике, но на практике такое возможно не всегда. Исходный топик приложения Kafka Streams обычно является общедоступным и его используют другие разработчики и приложения. В большинстве организаций изменения в ресурсах общей инфраструктуры могут вызвать сложные побочные эффекты.

Рассмотрим пример выполнения перераспределения для увеличения количества задач (см. `bbejeck.chapter_7.RepartitionForThroughput`) (листинг 7.11).

#### **Листинг 7.11.** Увеличение количества разделов для увеличения числа задач

```
KStream<String, String> repartitioned =
    initialStream.repartition(Repartitioned
        .with(stringSerde, stringSerde)
        .withName("multiple-aggregation")           ← Увеличит количество
        .withNumberOfPartitions(10));                разделов
```

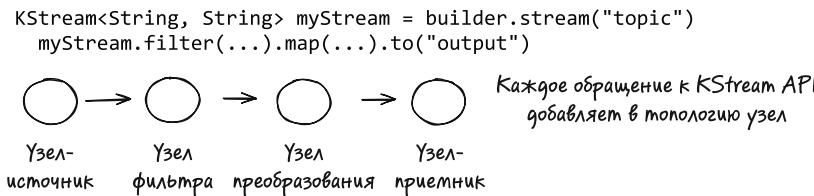
Теперь это приложение будет иметь десять задач, то есть может запускать до десяти потоков выполнения, обрабатывающих записи. Обратите внимание, что число 10 в этом примере выбрано произвольно, потому что его цель — показать, как использовать метод `KStream.repartition`, и не подразумевает выбор обоснованного числа задач для использования в промышленной системе.

Однако помните, что добавление разделов для увеличения пропускной способности дает наибольший эффект, когда ключи распределены относительно равномерно. Например, если 70 % пространства ключей приходится на один раздел, то увеличение количества разделов просто переместит эти ключи в новый раздел. А так как общее распределение ключей не изменилось, то вы не увидите прироста пропускной способности, поскольку основная нагрузка по-прежнему будет ложиться на один раздел, а значит, на одну задачу.

К настоящему моменту мы рассмотрели, как выполнить упреждающее перераспределение при изменении ключа. Но этот подход требует, чтобы вы знали, когда следует выполнить перераспределение, и не забывали это делать. Однако есть более эффективное решение — использовать оптимизации Kafka Stream. В этом случае Kafka Streams автоматически будет устранивать избыточные операции перераспределения данных.

### **7.2.6. Использование оптимизаций Kafka Streams**

Пока вы занимаетесь созданием топологии с помощью различных методов, Kafka Streams за кулисами строит граф — внутреннее представление топологии. Этот граф также можно считать логическим представлением приложения Kafka Streams. Когда вы вызовете метод `StreamBuilder#build`, Kafka Streams просмотрит граф и построит окончательное или физическое представление приложения. Если говорить в общих чертах, то это работает следующим образом: по мере применения каждого метода Kafka Streams добавляет узел в граф, как показано на рис. 7.9.

**Рис. 7.9.** Вызов каждого метода KStream добавляет узел в граф

После вызова очередного метода предыдущий узел становится родителем текущего. Этот процесс продолжается до тех пор, пока не закончится сборка своего приложения.

Попутно Kafka Streams будет записывать метаданные о строящемся графике, например, встречен ли узел перераспределения. Затем, при вызове метода `StreamsBuilder#build` для создания окончательной топологии, Kafka Streams проверит график на наличие избыточных узлов перераспределения и, если обнаружит их, то реорганизует топологию так, чтобы в ней оставались только действительно необходимые узлы перераспределения! Оптимизации являются необязательными и поэтому должны включаться явно, как показано в листинге 7.12.

#### **Листинг 7.12.** Включение оптимизации в Kafka Streams

```
streamProperties.put(StreamsConfig.TOPOLOGY_OPTIMIZATION_CONFIG,
                     StreamsConfig.OPTIMIZE);
builder.build(streamProperties);
```

← Включение оптимизаций  
Передача свойств в StreamBuilder ← В настройках

Итак, чтобы включить оптимизации, сначала нужно определить правильную конфигурацию, поскольку по умолчанию оптимизации отключены. Затем нужно передать объект свойств в метод `StreamBuilder#build`. После этого Kafka Streams перестроит топологию и удалит избыточные узлы перераспределения.

#### ПРИМЕЧАНИЕ

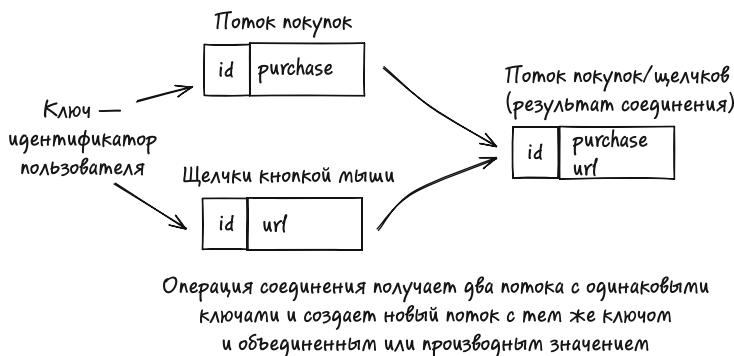
Если в топологии присутствует несколько операций изменения ключа с операциями с состояниями далее в потоке, то эти перераспределения не будут оптимизированы. Устраниены будут только избыточные перераспределения для одного узла, изменяющего ключ.

При включении оптимизации Kafka Streams автоматически обновит топологию, удалит три узла перераспределения, предшествующие агрегированию, и вставит новый одиночный узел перераспределения сразу после операции изменения ключа. В результате будет создана топология, которая выглядит как на рис. 7.8.

Итак, с помощью настройки конфигурации и передачи свойств в `StreamBuilder` можно автоматически удалить любые ненужные перераспределения! Решение о том, какой подход использовать, во многом зависит от личных предпочтений, но имейте в виду, что включение оптимизаций защитит от ситуаций, когда вы что-то упустили из виду. Теперь, рассмотрев перераспределение, перейдем к следующей операции с состоянием — к соединению.

## 7.3. СОЕДИНЕНИЕ ПОТОКОВ

Иногда может потребоваться объединить записи из разных потоков событий, чтобы добиться желаемого результата. Представим, что у нас есть поток покупок с идентификатором клиента в качестве ключа и поток щелчков кнопкой мыши, выполненных пользователем, и нам нужно объединить их, чтобы связать посещенные страницы с покупками. Для этого в Kafka Streams можно использовать соединения. Возможно, вы уже знакомы с концепцией SQL-операции соединения в мире реляционных баз данных, так вот, в основе соединения потоков лежит та же идея. Рассмотрим концептуальную иллюстрацию соединения потоков в Kafka Streams, изображенную на рис. 7.10.



**Рис. 7.10.** Два потока с одинаковыми ключами, но разными значениями

Как показано на рис. 7.10, в двух потоках событий в роли ключа используется один и тот же элемент данных, идентификатор клиента, но значения различаются. В одном потоке значения — это покупки, а в другом — ссылки на страницы, откуда пользователь перешел на сайт для совершения покупки.

### ПРИМЕЧАНИЕ

Поскольку операция соединения требует, чтобы разные топики имели идентичные ключи, размещающиеся в одном разделе, к ней применяются те же правила, что и к операции изменения ключа. Если экземпляр `KStream` содержит `repartitionRequired=true`, то Kafka Streams выполнит перераспределение перед операцией соединения, чтобы обеспечить совместимость секционирования. Таким образом, все, что говорилось о перераспределении выше в этой главе, применимо и к операции соединения.

В этом разделе мы объединим события из двух потоков с одним и тем же ключом, чтобы получить новое событие. Лучший способ познакомиться с соединением потоков — рассмотреть конкретный пример, поэтому вернемся в мир розничной торговли. Представьте крупного розничного продавца, который продает все, что только можно представить. Чтобы привлечь больше клиентов в свои магазины, продавец сотрудничает с национальной сетью кофеен и в каждом магазине выделяет место для кофейни.

Чтобы побудить клиентов зайти в его магазин, продавец запустил программу лояльности, участники которой, покупая кофе в кофейне внутри магазина и покупая что-либо в этом же магазине (в любом порядке), автоматически получают бонусные баллы после совершения второй покупки. Клиенты могут обменять эти баллы на товары в любом магазине. Главное правило: для начисления бонусных баллов покупки должны быть сделаны с интервалом не более 30 минут.

Поскольку сети магазинов и кофеен используют разные вычислительные инфраструктуры, записи о покупках помещаются в два разных потока событий, но это не проблема, потому что в обоих в качестве ключа используется идентификатор участника программы лояльности. Следовательно, для решения этой задачи можно использовать операцию соединения потоков.

### 7.3.1. Реализация соединения потоков

Следующий шаг — выполнение фактического соединения. Рассмотрим код в листинге 7.13, реализующий соединение (некоторые детали опущены для простоты). Поскольку в соединении участвуют несколько компонентов, я расскажу о них подробнее далее в этом разделе. Полный исходный код этого примера вы найдете в `src/main/java/bbejeck/chapter_7/KafkaStreamsJoinsApp.java`.

**Листинг 7.13.** Использование метода `join()` для объединения двух потоков в один

```
KStream<String, CoffeePurchase>
    coffeePurchaseKStream = builder.stream(...)

KStream<String, RetailPurchase>
    retailPurchaseKStream = builder.stream(...)

ValueJoiner<CoffeePurchase,
    RetailPurchase,
    Promotion> purchaseJoiner = new PromotionJoiner(); ←
                                                                | Экземпляр ValueJoiner,
                                                                | производящий
                                                                | объединенное значение

JoinWindows thirtyMinuteWindow =
    JoinWindows.ofTimeDifferenceWithNoGrace(Duration.minutes(30)); ←
                                                                | JoinWindow
                                                                | определяет
                                                                | максимальный
                                                                | временной
                                                                | интервал между
                                                                | записями,
                                                                | участвующими
                                                                | в соединении

KStream<String, Promotion> joinedKStream =
    coffeePurchaseKStream.join(retailPurchaseKStream, ←
        purchaseJoiner,
        thirtyMinuteWindow,
        StreamJoined.with(stringSerde, ←
            coffeeSerde,
            storeSerde)
        .withName("purchase-join")
        .withStoreName("join-stores")); ←
                                                                | Конструирование
                                                                | соединения
                                                                | StreamJoined — объект
                                                                | с настройками
```

Методу `KStream.join` необходимо предоставить четыре параметра:

- `retailPurchaseKStream` — поток покупок, с которым выполняется соединение;
- `purchaseJoiner` — реализация интерфейса `ValueJoiner<V1, V2, R>`, которая принимает два значения (не обязательно одного типа). `ValueJoiner.apply` принимает оба значения для данного ключа из соединяемых потоков, выполняет реализован-

ную в нем логику и возвращает (возможно, новый) объект типа R. В этом примере `purchaseJoiner` извлекает некоторую информацию из обоих объектов `Purchase` и возвращает объект `PromotionProto`;

- `thirtyMinuteWindow` — экземпляр `JoinWindows`. Метод `JoinWindows.ofTimeDifferenceWithNoGrace` возвращает максимальную разницу во времени между двумя значениями для включения их в соединение. В частности, отметка времени во втором потоке `retailPurchaseKStream` может отстоять не далее чем на 30 минут вперед или назад от отметки времени в записи из `coffeePurchaseKStream` с тем же ключом;
- экземпляр `StreamJoined`, предоставляющий дополнительные параметры, необходимые для выполнения соединения. В данном случае это ключ, значение Serde для вызывающего потока и значение Serde для второго потока. При объединении записей имеется только один ключ, Serde, потому что ключи должны быть одного типа. Метод `withName` предоставляет имя узла в топологии и базовое имя для топика перераспределения (если требуется). `withStoreName` — это базовое имя для хранилищ состояний, используемых в процессе соединения. Об использовании хранилищ состояний при соединении я расскажу в разделе 7.4.

#### **ПРИМЕЧАНИЕ**

Объекты Serde нужны в операциях соединения, потому что Kafka Streams материализует стороны соединения в хранилищах оконных состояний. Для ключа предоставляется только один объект Serde, потому что обе стороны соединения должны иметь ключ одного и того же типа.

Соединение — одна из самых мощных операций в Kafka Streams и одна из самых сложных для понимания. Давайте потратим немного времени на обзор особенностей ее работы.

### **7.3.2. Особенности работы соединений**

За кулисами KStream DSL API выполняет массу работы, чтобы получить соединение. И вам будет полезно узнать, как фактически работают соединения. Kafka Streams создает узел-обработчик соединения с хранилищем состояний для каждой стороны, участвующей в соединении. На рис. 7.11 показано, как это выглядит с концептуальной точки зрения.

При создании обработчика соединения каждой из сторон Kafka Streams передает имя хранилища состояний противоположной стороны соединения — левая сторона получает имя хранилища правой стороны, а правая сторона — имя левого хранилища. Зачем каждой из сторон имя хранилища противоположной стороны? Причина заключается в особенностях работы соединений в Kafka Streams. Чтобы было понятнее, рассмотрим еще одну иллюстрацию на рис. 7.12.

Когда поступает новая запись (узел-обработчик слева используется для `coffeePurchaseKStream`), обработчик помещает запись в свое хранилище, а затем ищет совпадение в хранилище с правой стороны (в хранилище для `retailPurchaseKStream`), извлекая записи с тем же ключом и в пределах временного диапазона, указанного в `JoinWindows`.

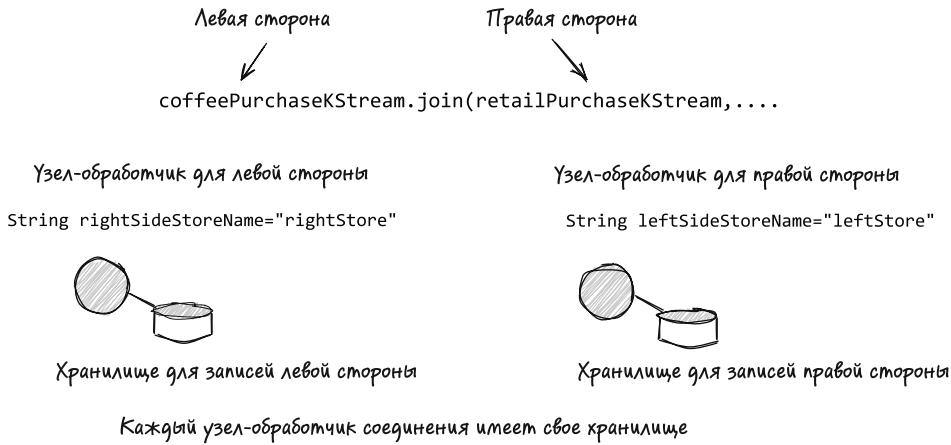


Рис. 7.11. В соединении потоков обе стороны получают узел-обработчик и хранилище состояний

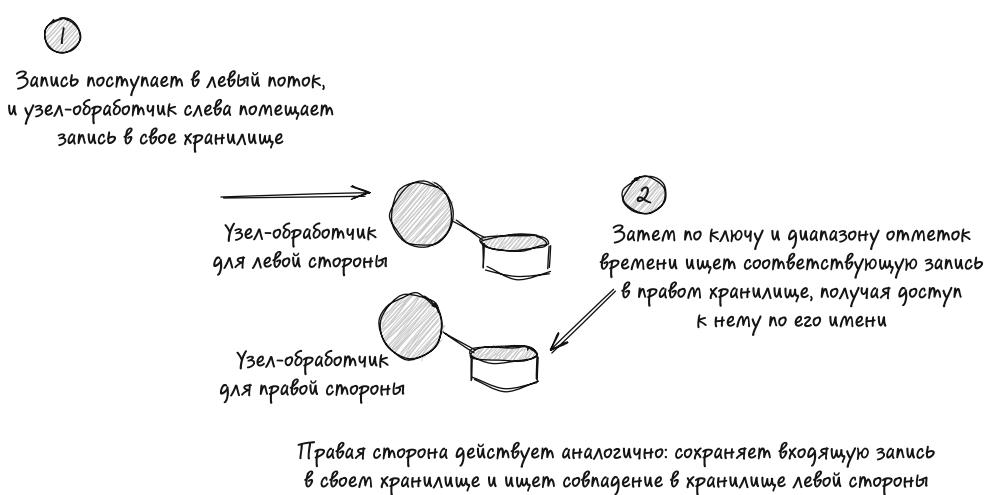
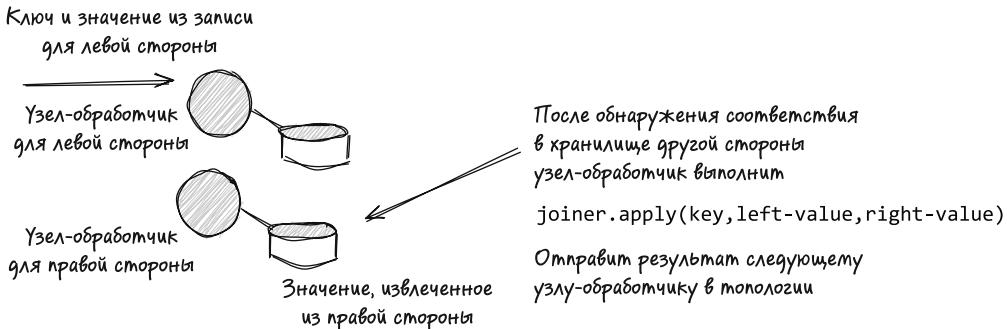


Рис. 7.12. При поступлении новой записи узлы-обработчики операции соединения ищут совпадения в хранилище состояний другой стороны

Наконец, нам нужно посмотреть, что произойдет, если будет найдено совпадение. Взгляните еще на одну иллюстрацию на рис. 7.13.

Итак, после того, как для входящей записи будет найдено совпадение в хранилище другой стороны соединения, узел-обработчик этого соединения (в нашей иллюстрации — `coffeePurchaseKStream`) берет ключ и значение из своей входной записи и значение каждой записи, извлеченной из хранилища другой стороны, и вызывает метод `ValueJoiner.apply`, который создаст запись соединения согласно предоставленной вами реализации. После этого узел-обработчик отправит ключ и результат соединения узлам-обработчикам, находящимся далее в потоке.



**Рис. 7.13.** Обнаружив совпадение, узел-обработчик вызывает метод apply экземпляра ValueJoiner с ключом, значением своей записи и значением записи с другой стороны

Теперь, познакомившись с внутренней механикой работы операции соединения, обсудим подробнее некоторые из параметров настройки соединений.

### 7.3.3. ValueJoiner

Чтобы получить результат соединения, нужно создать экземпляр ValueJoiner<V1, V2, R>. ValueJoiner принимает два объекта одного или разных типов и возвращает один объект, возможно, третьего типа. В нашем случае ValueJoiner принимает объекты CoffeePurchase и RetailPurchase и возвращает объект Promotion. Взглянем на код (вы найдете его в src/main/java/bbejeck/chapter\_7/joiner/PromotionJoiner.java) (листинг 7.14).

#### Листинг 7.14. Реализация ValueJoiner

```
public class PromotionJoiner
    implements ValueJoiner<CoffeePurchase,
                           RetailPurchase,
                           Promotion> {

    @Override
    public Promotion apply(
        CoffeePurchase coffeePurchase,
        RetailPurchase retailPurchase) {
        double coffeeSpend = coffeePurchase.getPrice();           ← Извлекает сумму, потраченную в кофейне
        double storeSpend = retailPurchase.getPurchaseList()     ← Суммирует стоимость всех покупок
            .stream()
            .mapToDouble(pi -> pi.getPrice() * pi.getQuantity()).sum();
        double promotionPoints = coffeeSpend + storeSpend;      ← Вычисляет количество бонусных баллов
        if (storeSpend > 50.00) {
            promotionPoints += 50.00;
        }
        return Promotion.newBuilder()                                ← Конструирует и возвращает новый объект Promotion
            .setCustomerId(retailPurchase.getCustomerId())
            .setDrink(coffeePurchase.getDrink())
            .setItemsPurchased(retailPurchase.getPurchaseCount())
            .setPoints(promotionPoints).build();
    }
}
```

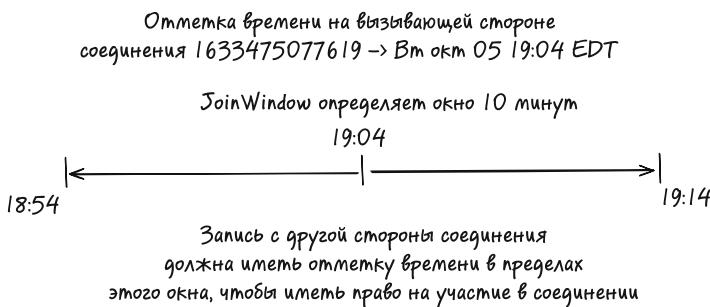
Чтобы создать объект `Promotion`, мы извлекаем суммы, потраченные с обеих сторон соединения, и вычисляем количество бонусных баллов для вознаграждения клиента. Интерфейс `ValueJoiner` имеет только один метод — `apply`, поэтому для представления соединения можем использовать лямбда-выражение. Но в данном случае мы создали конкретную реализацию, чтобы получить возможность написать отдельный модульный тест для `ValueJoiner`. Мы обсудим это решение в главе 14.

#### ПРИМЕЧАНИЕ

Kafka Streams также предоставляет интерфейс `ValueJoinerWithKey`, позволяющий получить доступ к ключу для вычисления результата соединения. Однако ключ считается доступным только для чтения, и его изменение в реализации `ValueJoinerWithKey` приведет к неопределенному поведению.

### 7.3.4. JoinWindows

Конфигурационный объект `JoinWindows` играет важную роль в процессе выполнения соединения: он определяет разность между отметками времени записей из двух потоков, которые могут участвовать в соединении. Обратимся к рис. 7.14, чтобы понять роль `JoinWindows`.



**Рис. 7.14.** Конфигурационный объект `JoinWindows` определяет максимальную разницу между отметками времени в записях двух сторон, которые могут участвовать в соединении

Точнее говоря, параметр `JoinWindows` — это максимальная удаленность во времени (до или после) отметки времени в записи вторичной (другой) стороны от отметки времени в записи первичной (вызывающей) стороны, чтобы записи можно было считать пригодными для соединения. В нашем примере окно соединения имеет протяженность 30 минут. Допустим, запись из `coffeeStream` имеет отметку времени 12:00, тогда записи в `storeStream`, которые могут участвовать в соединении, должны иметь отметку времени между 11:30 и 12:30.

В `JoinWindows()` доступны два дополнительных метода: `after` и `before`. Их можно использовать для уточнения времени и, возможно, порядка событий, участвующих в соединении. Допустим, что мы установили начальное окно соединения

равным 30 минутам, но нам нужно, чтобы с замыкающей стороны окно было короче и составляло, скажем, 5 минут. Например, если заданная запись имеет отметку времени 12:00, то соединение с другой записью возможно, только если отметка времени в той другой записи находится в диапазоне от 11:30 до 12:05. Чтобы реализовать это, следует использовать метод `JoinWindows.after` (продолжаем пример из листинга 7.13) (листинг 7.15).

**Листинг 7.15.** Использование метода `JoinWindows.after` для изменения протяженности окна с замыкающей стороны

```
coffeeStream.join(storeStream,...,
    thirtyMinuteWindow.after(Duration.ofMinutes(5))....
```

Здесь протяженность окна с начальной стороны остается прежней, запись в `storeStream` может иметь отметку времени со значением не менее 11:30, но с замыкающей стороны протяженность окна соединения короче; отметка времени в записи в `storeStream` теперь не должна превышать 12:05.

Метод `JoinWindows.before` работает аналогично в обратном направлении. Допустим, нам нужно сократить протяженность окна с начальной стороны, в таком случае можно использовать следующий код (листинг 7.16).

**Листинг 7.16.** Использование метода `JoinWindows.before` для изменения протяженности окна с начальной стороны

```
coffeeStream.join(storeStream,...,
    thirtyMinuteWindow.before(Duration.ofMinutes(5))....
```

Теперь ситуация изменилась и отметка времени в записи в `storeStream` не должна быть более чем на 5 минут меньше отметки времени в записи в `coffeeStream`. Таким образом, допустимыми для соединения будут считаться записи (в `storeStream`) с отметками времени от 11:55 до 12:30. Методы `JoinWindows.before` и `JoinWindows.after` также можно использовать для управления порядком поступления записей, участвующих в соединении.

Например, чтобы разрешить соединение записей, только если покупка в магазине происходит *после* покупки в кофейне в течение 30 минут, следует использовать вызов `JoinWindows.of(Duration.ofMinutes(0).after(Duration.ofMinutes(30))`. А чтобы учитывать только покупки в магазине, произведенные *перед* покупкой в кофейне, следует использовать вызов `JoinWindows.of(Duration.ofMinutes(0).before(Duration.ofMinutes(30)))`.

### 7.3.5. Совместимое секционирование

Для выполнения операции соединения в Kafka Streams необходимо, чтобы обе стороны имели одинаковое число разделов, то есть они должны иметь одинаковое число разделов и ключи одного типа. Чтобы добиться совместимого секционирования, все производители Kafka должны использовать один и тот же класс секционирования при создании исходных топиков Kafka Streams. Аналогично необходимо использовать идентичные экземпляры `StreamPartitioner` для любых операций, записывающих данные в выходные топики Kafka Streams с помощью метода `KStream.to()`. Если вы

придерживаетесь стратегий распределения по умолчанию, то вам не придется беспокоиться о совместимости секционирования.

Как видите, класс `JoinWindows` дает множество способов для управления соединением двух потоков. Важно помнить, что отметки времени в записях управляют поведением операции соединения. Они могут устанавливаться Kafka (брокером или производителем) или встраиваться в полезную нагрузку записи. Чтобы использовать отметку времени, встроенную в запись, нужно предоставить свою реализацию `TimestampExtractor`, и я расскажу об этом и о семантике отметок времени в главе 9.

### 7.3.6. StreamJoined

Последний параметр, который мы обсудим, — конфигурационный объект `StreamJoined`. С помощью `StreamJoined` можно передать объекты Serdes для ключа и значений, участвующих в соединении. Обратите внимание, что при использовании Schema Registry должны передаваться Serdes, которые поддерживают Schema Registry. Желательно всегда явно передавать объекты Serdes для записей соединения, потому что они могут отличаться от заданных в конфигурации на уровне приложения. Можно также дать имя узлу-обработчику, выполняющему соединение, и хранилищу состояний, используемому для хранения найденных записей, участвующих в соединении. Подробнее об именовании хранилищ я расскажу в подразделе 7.4.5.

Прежде чем уйти от соединений, давайте обсудим еще несколько доступных их вариантов.

### 7.3.7. Другие варианты соединений

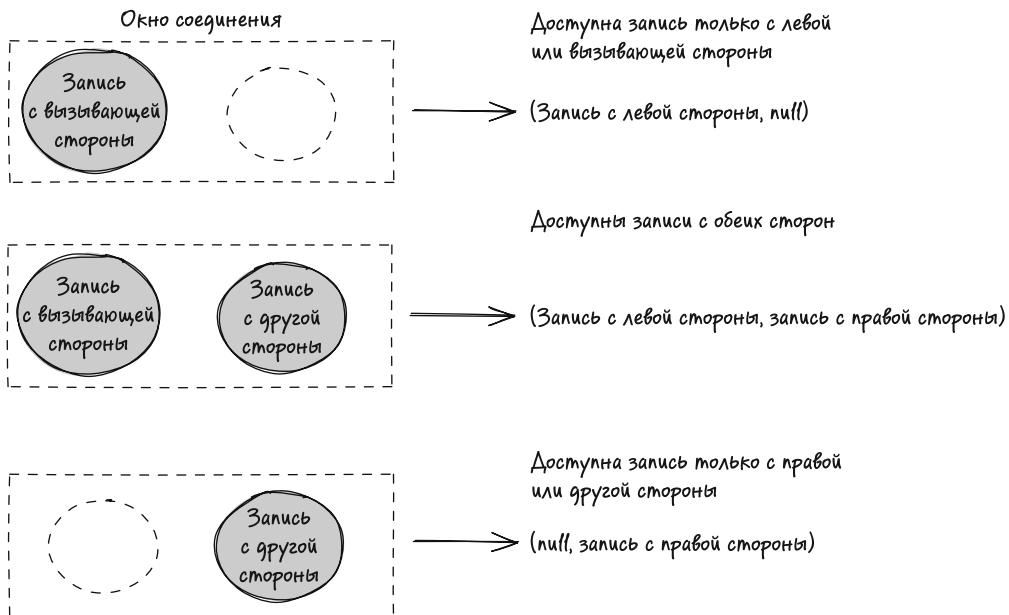
В текущем примере было продемонстрировано *внутреннее соединение* (inner join). Во внутреннем соединении в отсутствие записи с любой из сторон соединение не выполняется и объект `Promotion` не создается. Но другие варианты соединений не требуют обеих записей. Kafka Streams создаст результат, даже если другая сторона соединения отсутствует. Эти варианты соединений можно использовать в ситуациях, когда нужна информация, даже если одна из записей недоступна.

### 7.3.8. Внешние соединения

Внешние соединения (outer joins) всегда возвращают результат, но он может не включать обе стороны соединения. Внешнее соединение можно использовать, когда требуется получить результат, независимо от того, было соединение успешным или нет. Если бы в нашем примере потребовалось выполнить внешнее соединение, то мы реализовали бы его так:

```
coffeePurchaseKStream.outerJoin(retailPurchaseKStream...)
```

Внешнее соединение отправляет результат, содержащий записи с любой стороны или с обеих. Например, результат соединения может иметь вид `left+right`, `left+null` или `null+right`, в зависимости от того, что присутствует. Эти три возможных результата показаны на рис. 7.15.



**Рис. 7.15.** Внешнее соединение может дать три результата: только с событием из вызывающего потока, с событиями из обоих потоков и только с событием из другого потока

### 7.3.9. Внешнее левое соединение

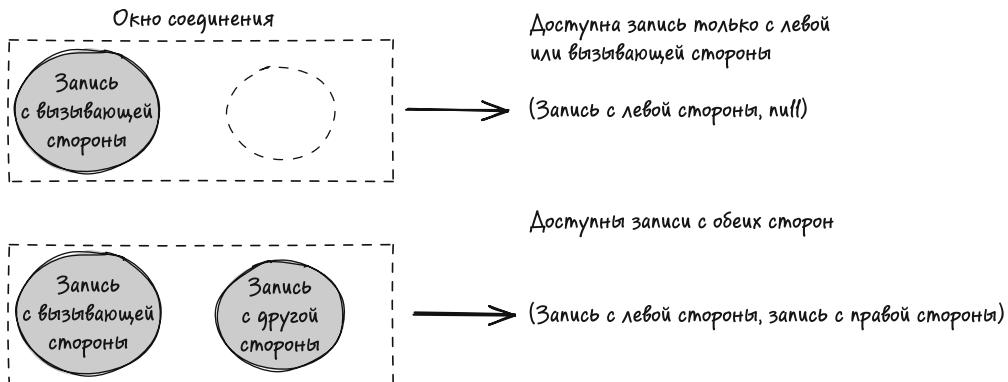
Внешнее левое соединение (left-outer join) всегда дает результат. Отличие от внешнего соединения в том, что левая (она же вызывающая) сторона соединения всегда присутствует в результате, имеющем вид `left+right` или `left+null`. Внешнее левое соединение можно использовать, когда записи из левого (или вызывающего) потока играют важную роль в бизнес-логике. Если бы в листинге 7.13 потребовалось использовать внешнее левое соединение, то мы реализовали бы его так:

```
coffeePurchaseKStream.leftJoin(retailPurchaseKStream...)
```

Результат внешнего левого соединения показан на рис. 7.16.

Мы исследовали разные типы соединений, но пока не совсем ясно, в каких случаях их использовать. Давайте поговорим об этом и начнем с текущего примера соединения. Для данного примера имеет смысл только внутреннее соединение, потому что мы определяем величину поощрения (количество бонусных баллов) на основе двух покупок, каждая из которых поступает в приложение в своем потоке. Если с другой стороны нет соответствующей покупки, то ничего возвращать не надо.

В случаях, когда одна из сторон соединения критически важна, а другая полезна, но допускается ее отсутствие, хорошим выбором будет внешнее левое соединение, когда критически важный поток находится на левой (вызывающей) стороне. Пример применения этого соединения я покажу в главе 8, когда мы будем обсуждать потоково-табличные соединения.



**Рис. 7.16.** Внешнее левое соединение может дать два результата: с записями с обеих сторон или только с записью с левой стороны

Наконец, когда каждый из двух потоков, участвующих в соединении, важен сам по себе, для таких случаев как нельзя лучше подойдет внешнее соединение. Рассмотрим IoT, где есть два потока измерений от взаимосвязанных датчиков. Соединение информации от датчиков дает более полную картину, но вообще достаточно информации с любой из сторон, если она доступна.

В следующем разделе мы подробно рассмотрим рабочую лошадку операций с состоянием — хранилище состояний.

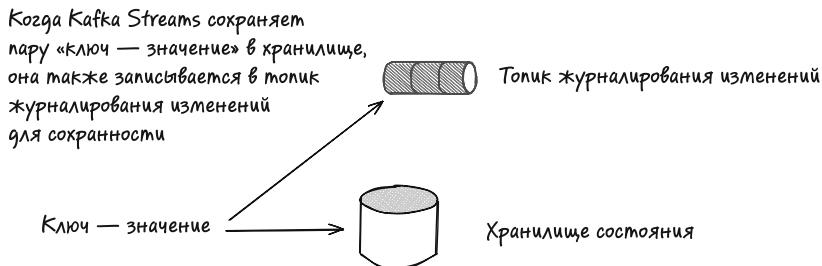
## 7.4. ХРАНИЛИЩА СОСТОЯНИЙ В KAFKA STREAMS

К настоящему моменту мы рассмотрели операции с состоянием, доступные в Kafka Streams DSL API, но обошли стороной механизмы хранения, лежащие в их основе. В этом разделе мы рассмотрим основы использования хранилищ состояний в Kafka Streams и критические факторы, связанные с использованием состояния в потоковых приложениях в целом. В будущем это поможет вам делать обоснованный выбор при использовании состояния в своих приложениях Kafka Streams.

Но прежде, чем углубляться в детали, рассмотрим общую информацию. На верхнем уровне хранилища состояний в Kafka Streams являются хранилищами типа «ключ – значение» и делятся на две категории: постоянные и оперативные. И те и другие долговечны, потому что для поддержки хранилищ Kafka Streams использует топики журнализации изменений.

Постоянные хранилища хранят записи на локальном диске и восстанавливают свое содержимое после перезапуска. Оперативные хранилища хранят записи в оперативной памяти, поэтому их необходимо явно восстанавливать после перезапуска. Для восстановления своего содержимого все хранилища состояний используют топики журнализации изменений. Чтобы понять, как хранилище состояний использует топик журнализации изменений для восстановления, рассмотрим, как они реализованы в Kafka Streams.

В DSL, когда к топологии применяется операция с состоянием, Kafka Streams создает хранилище состояний для узла-обработчика (по умолчанию постоянного типа). Вместе с хранилищем Kafka Streams также создает топик журнализирования изменений, поддерживающий хранилище. Когда Kafka Streams записывает записи в хранилище, они также записываются в журнал изменений. Этот процесс показан на рис. 7.17.



**Рис. 7.17.** При сохранении записей типа «ключ — значение» в хранилище они также записываются в топик журнализирования изменений для обеспечения сохранности информации

Итак, когда Kafka Streams помещает запись в хранилище состояний, то она также отправляется в топик Kafka, лежащий в основе этого хранилища. Выше в этой главе я упоминал, что вы увидите результаты агрегирования не сразу, потому что для хранения результатов Kafka Streams использует кэш. И только когда кэш очищается при фиксации или при заполнении, агрегированные записи передаются следующим узлам-обработчикам. В этот момент Kafka Streams создаст записи в топике журнализирования изменений.

#### ПРИМЕЧАНИЕ

Если отключить кэширование, то в хранилище состояний будет отправляться каждая запись, это означает, что каждая запись попадет в топик журнализирования изменений.

### 7.4.1. Восстановление хранилища состояний с помощью топика журнализирования изменений

Итак, как Kafka Stream использует топик журнализирования изменений? Для начала рассмотрим хранилище в оперативной памяти. Поскольку содержимое оперативного хранилища не сохраняется после перезапуска, то такие хранилища будут восстанавливать свое содержимое, начиная с первой записи в топике журнализирования изменений. Иначе говоря, даже потеряв все свое содержимое в момент перезапуска, оперативное хранилище сможет возобновить работу с того же места.

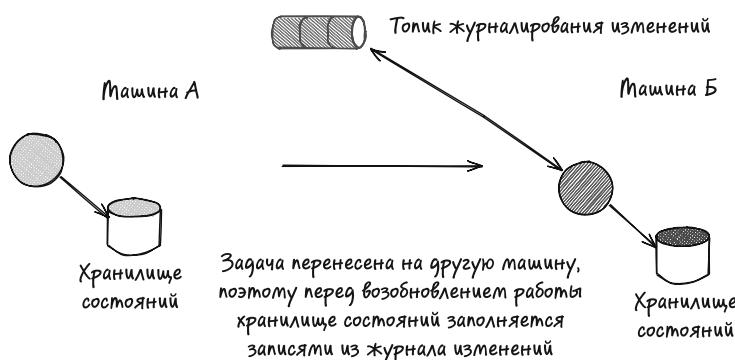
Постоянным хранилищам полное восстановление требуется только в случае потери всех локальных состояний или при обнаружении повреждения данных. Для постоянных хранилищ Kafka Streams поддерживает файл контрольной точки и вместо восстановления с нуля использует смещение в файле для начала восстановления. Если смещение станет недействительным, то Kafka Streams удалит файл

контрольной точки и выполнит восстановление с самого начала топика. Kafka Streams немного по-разному использует файлы контрольных точек в режимах `exact_once` и `exact_once_v2`. После открытия хранилища состояний в режиме EOS (Exactly-Once Semantic — семантика «точно один раз») файл контрольной точки удаляется. А после штатного выключения Kafka Streams генерируется заново. Выполнение этого процесса в режиме EOS гарантирует, что полное восстановление будет выполняться только после выключения из-за ошибки.

Это различие в шаблонах восстановления вносит интересный поворот в споры о компромиссах постоянных и оперативных хранилищ. Оперативное хранилище обеспечит более быстрый поиск, поскольку ему не нужно обращаться к диску для извлечения, а постоянное хранилище в условиях штатной работы, как правило, быстрее возобновит работу после перезапуска, потому что ему не придется восстанавливать много записей.

Другая ситуация, которую следует рассмотреть, — это состав запущенного приложения Kafka Streams. Как рассказывалось выше, вы можете динамически изменять количество запущенных приложений, увеличивая или уменьшая его. Kafka Streams автоматически назначает задачи из существующих приложений новым членам или переносит задачи из приложений, выпавших из группы, в те, что все еще работают. Задача, отвечающая за операцию с состоянием, будет иметь свое хранилище состояний (о хранилищах состояний и задачах я расскажу далее).

Рассмотрим случай, когда приложение Kafka Streams теряет один из своих членов. Напомню, что вы можете запустить несколько экземпляров приложения Kafka Streams на разных машинах, и те из них, что имеют одинаковый идентификатор приложения, будут считаться частью одного логического приложения. При выходе из строя одного из членов Kafka Streams выполнит перебалансировку и переназначит задачи, выполнявшиеся сбойным приложением, другим членам. Для любой переназначенной операции с состоянием Kafka Streams создаст новое пустое хранилище и выполнит его восстановление, прочитав все записи с начала топика журнализирования изменений. Схема на рис. 7.18 иллюстрирует эту ситуацию.



**Рис. 7.18.** Когда задача с состоянием перемещается на другую машину, Kafka Streams восстанавливает ее хранилище состояний чтением всех записей с самого начала топика журнализирования изменений

Таким образом, использование топика журнализирования изменений Kafka Streams обеспечивает сохранность данных в приложениях даже в случае сбоев, правда, при этом обработка задерживается на некоторое время, необходимое для восстановления хранилища. К счастью, Kafka Streams предлагает средство для этой ситуации — резервные задачи.

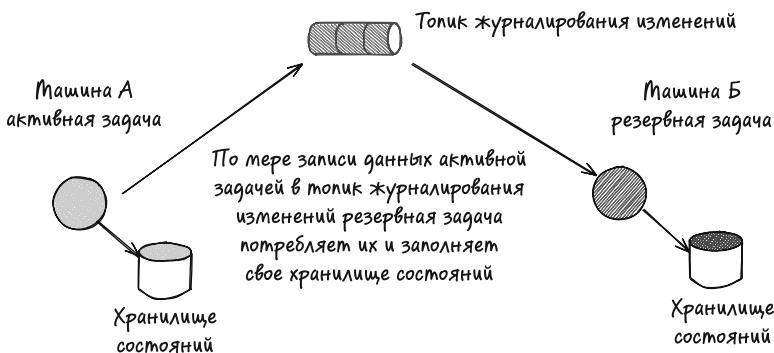
### 7.4.2. Резервные задачи

Kafka Streams предоставляет поддержку резервных задач для быстрого переключения в случае отказа экземпляра приложения. Резервная задача действует как «тень» активной задачи, потребляя данные из топика журнализирования изменений в локальное хранилище состояний для резервирования. Если активная задача вдруг терпит сбой и выпадает из группы, то резервная задача становится новой активной задачей. Но, так как она потребляла данные из топика журнализирования изменений, ей потребуется минимальное время, чтобы включиться в работу.

#### ПРИМЕЧАНИЕ

Чтобы создать резервные задачи, нужно присвоить конфигурационному параметру `num.standby.replicas` значение больше 0 и развернуть  $N + 1$  экземпляров Kafka Streams (где  $N$  равно количеству желаемых реплик). В идеале эти экземпляры Kafka Streams должны развертываться на отдельных машинах.

Несмотря на простоту концепции, рассмотрим процесс работы резервной задачи, изображенный на рис. 7.19.



**Рис. 7.19.** Резервная задача копирует данные активной задачи, потребляя их из топика журнализирования изменений, и постоянно синхронизирует свое локальное хранилище состояний с хранилищем активной задачи

Итак, согласно иллюстрации, резервная задача потребляет записи из топика журнализирования изменений и помещает их в свое локальное хранилище состояний. Резервная задача не обрабатывает никаких записей. Ее единственная цель — синхронизировать свое хранилище состояний с хранилищем состояний активной задачи. Подобно любым стандартным приложениям производителей и потребителей, активные и резервные задачи никак не координируют свои действия.

Поскольку резервная задача имеет состояние, полностью синхронизированное с состоянием активной задачи, или в крайнем случае будет отставать совсем немного, когда Kafka Streams объявит резервную задачу активной, она возобновит обработку с минимальной задержкой. Резервные задачи имеют свои компромиссы, которые следует учитывать. Так или иначе, резервные задачи дублируют данные, но, учитывая преимущество почти мгновенного переключения из резервного состояния в активное, с этим можно смириться.

### ПРИМЕЧАНИЕ

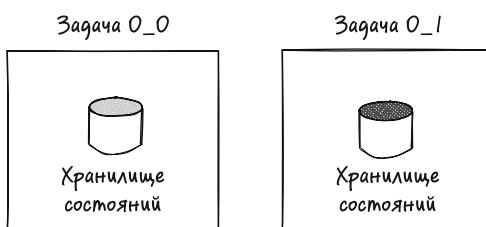
Для улучшения масштабирования производительности Kafka Streams в соответствии с Kafka KIP-441 (<http://mng.bz/0GmI>) была проделана огромная работа. Поэтому если вы включили поддержку резервных задач и резервный экземпляр в какой-то момент стал активным, то позднее Kafka Streams может определить, что было бы предпочтительнее назначить выполнение задачи с состоянием другому экземпляру, и осуществит ее перенос.

К настоящему моменту вы узнали, как хранилища состояний поддерживают операции с состоянием, как обеспечивается их надежность с помощью топиков журнализирования изменений и как резервные задачи помогают организовать быстрое переключение при сбое. Но есть еще кое-что, что мы должны охватить. Сначала мы рассмотрим, как назначаются хранилища состояний, а затем обсудим их настройки, включая выбор типа хранилища и настройку топика журнализирования изменений, если это необходимо.

### 7.4.3. Назначение хранилищ состояний в Kafka Streams

В предыдущей главе мы обсудили роль задач в Kafka Streams. Я хочу повторить еще раз, что задачи работают в архитектуре без общего доступа и только в одном потоке выполнения. Приложение Kafka Streams может иметь несколько потоков выполнения, а каждый поток может выполнять несколько задач, но они не имеют никаких общих данных. Еще раз подчеркну, что задачи работают в архитектуре без общего доступа, это означает, что когда задача имеет состояние, то только она будет иметь доступ к хранилищу с этим состоянием состояний. При такой организации сами собой отпадают проблемы с блокировками, свойственные параллельной обработке.

Вернемся к примеру агрегирования акций в подразделе 7.2.2 и предположим, что входной топик имеет два раздела, то есть две задачи. Взгляните на рис. 7.20, иллюстрирующий назначение задач с хранилищами состояний в этом примере.



Каждая задача является единственным владельцем назначенного ей хранилища, и только она имеет право на чтение и запись в него

**Рис. 7.20.** Задачам с состоянием назначается хранилище состояний

Глядя на эту иллюстрацию, можно заметить, что задача, связанная с хранилищем состояния, — единственная, которая имеет к нему доступ. Теперь давайте обсудим, как Kafka Streams размещает хранилища состояний в файловой системе.

#### **7.4.4. Местоположение хранилищ состояний в файловой системе**

В приложении с состоянием при первом запуске Kafka Streams создает для всех хранилищ состояний корневой каталог с именем из конфигурационного параметра `StreamsConfig.STATE_DIR_CONFIG`. Если параметр `STATE_DIR_CONFIG` не определен, то по умолчанию используется имя временного каталога для виртуальной машины Java (JVM), за которым следует системно-зависимый разделитель и затем `kafka-streams`. Например, у меня в MacOS корневой каталог по умолчанию для хранилищ состояний имеет имя `/var/folders/1k/d_9_qr558zd6ghbqwt0zc80000gn/T/kafka-streams`.

##### **ПРИМЕЧАНИЕ**

Значение параметра `STATE_DIR_CONFIG` должно быть уникальным для каждого экземпляра Kafka Streams, использующего одну и ту же файловую систему.

К имени корневого каталога Kafka Streams добавляет идентификатор приложения, который должен быть указан в настройках. Опять же у меня на ноутбуке путь выглядит так: `/var/folders/1k/d_9_qr558zd6ghbqwt0zc80000gn/T/kafka-streams/test-application/`.

##### **СОВЕТ**

Чтобы узнать, где в вашей системе находится временный каталог, запустите в окне терминала командную оболочку Java, выполнив команду `jshell`, а затем введите `System.getProperty("java.io.tmpdir")` и нажмите клавишу Return. Эта команда выведет путь к временному каталогу.

Дерево каталогов разветвляется на уникальные каталоги для каждой задачи. При этом для каждой задачи с состоянием Kafka Streams создает каталог с именем, скомпонованным из идентификатора субтопологии и раздела (через подчеркивание). Например, для задачи с состоянием из первой субтопологии, назначенной разделу 0, будет создан каталог с именем `0_0`.

Следующий каталог получает имя `rocksdb` по реализации хранилища. То есть на данном этапе путь будет выглядеть как `/var/folders/1k/d_9_qr558zd6ghbqwt0zc80000gn/T/kafka-streams/test-application/0_0/rocksdb`. Именно в нем находится конечный каталог из узла-обработчика (если только он не предоставлен объектом `Materialized`, о котором я расскажу ниже). Рассмотрим код в листинге 7.17 из приложения Kafka Streams с состоянием и сгенерированные имена топологии, чтобы понять, как конечный каталог получает свое имя.

##### **Листинг 7.17.** Простое приложение Kafka Streams с состоянием

```
builder.stream("input")
    .groupByKey()
    .count()
    .toStream()
    .to("output")
```

Это приложение имеет топологию с соответствующим названием.

Topologies:

```
Sub-topology: 0
  Source: KSTREAM-SOURCE-0000000000 (topics: [input])
    --> KSTREAM-AGGREGATE-0000000002
  Processor: KSTREAM-AGGREGATE-0000000002
    (stores: [KSTREAM-AGGREGATE-STATE-STORE-0000000001])
      --> KTABLE-TOSTREAM-0000000003
      <-- KSTREAM-SOURCE-0000000000
  Processor: KTABLE-TOSTREAM-0000000003 (stores: [])
    --> KSTREAM-SINK-0000000004
    <-- KSTREAM-AGGREGATE-0000000002
  Sink: KSTREAM-SINK-0000000004 (topic: output)
    <-- KTABLE-TOSTREAM-0000000003
```

The diagram illustrates the data flow in the topology. It starts with a 'Source' node (KSTREAM-SOURCE-0000000000) receiving data from a topic 'input'. This data is processed by a 'Processor' node (KSTREAM-AGGREGATE-0000000002). The output of this processor is stored in a 'store' (KSTREAM-AGGREGATE-STATE-STORE-0000000001), which is then converted into a 'KTABLE' (KTABLE-TOSTREAM-0000000003) by another 'Processor' node. Finally, the data is sent to a 'Sink' node (KSTREAM-SINK-0000000004) with a topic 'output'. Annotations on the right side of the diagram identify the 'KSTREAM-AGGREGATE-0000000002' node as the 'Имя агрегирующего узла-обработчика' (Name of the aggregating processing node) and the 'KSTREAM-AGGREGATE-STATE-STORE-0000000001' store as the 'Имя хранилища, назначенного обработчику' (Name of the store assigned to the processor).

В этом примере топологии для метода `count()` сгенерировано имя `KSTREAM-AGGREGATE-0000000002`. Обратите внимание, что ему назначено хранилище с именем `KSTREAM-AGGREGATE-STATE-STORE-0000000001`. То есть Kafka Streams берет базовое имя узла-обработчика с состоянием и добавляет `STATE-STORE` и число, сгенерированное глобальным счетчиком. Теперь посмотрим, как выглядит полный путь к этому хранилищу состояний: `/var/folders/1k/d_9_qr558zd6ghbqwt0zc80000gn/T/kafka-streams/test-application/0_0/rocksdb/KSTREAM-AGGREGATE-STATE-STORE-0000000001`.

Итак, конечный каталог `KSTREAM-AGGREGATE-STATE-STORE-0000000001` в этом пути содержит файлы RocksDB нашего хранилища. Теперь, если вывести список топиков в брокере после запуска приложения Kafka Streams, то можно увидеть имя `test-application-KSTREAM-AGGREGATE-STATE-STORE-0000000001-changelog`. Этот топик является журналом изменений для хранилища состояний. Обратите внимание, что при создании имени для топика Kafka Streams следует соглашению: `<application-id>-<state store name>-changelog`.

#### 7.4.5. Именование операций с состоянием

Подход к именованию операций с состоянием поднимает интересный вопрос: что произойдет, если добавить операцию перед `count()`? Допустим, мы решили добавить фильтр, чтобы исключить определенные записи из подсчета. Топология обновится, как показано в листинге 7.18.

##### Листинг 7.18. Обновленная топология с фильтром

```
builder.stream("input")
  .filter((key, value) -> !key.equals("bad"))
  .groupByKey()
  .count()
  .toStream()
  .to("output")
```

Напомню, что при создании имен узлов-обработчиков Kafka Streams использует глобальный счетчик, поэтому после добавления операции все узлы-обработчики,

находящиеся далее в потоке, получат новые имена, потому что их номера увеличиваются на 1. Новая топология будет выглядеть так, как показано в листинге 7.19.

### Листинг 7.19. Обновленные имена в топологии

Topologies:

```

Sub-topology: 0
  Source: KSTREAM-SOURCE-0000000000 (topics: [input])
    --> KSTREAM-FILTER-0000000001
  Processor: KSTREAM-FILTER-0000000001 (stores: [])
    --> KSTREAM-AGGREGATE-0000000003
    <-- KSTREAM-SOURCE-0000000000
  Processor: KSTREAM-AGGREGATE-0000000003
    (stores: [KSTREAM-AGGREGATE-STATE-STORE-0000000002]) ← Новое имя для операции
    --> KTABLE-TOSTREAM-0000000004
    <-- KSTREAM-FILTER-0000000001
  Processor: KTABLE-TOSTREAM-0000000004 (stores: [])
    --> KSTREAM-SINK-0000000005
    <-- KSTREAM-AGGREGATE-0000000003
  Sink: KSTREAM-SINK-0000000005 (topic: output)
    <-- KTABLE-TOSTREAM-0000000004

```

Обратите внимание, как изменилось имя хранилища состояния. Это изменение означает, что появился новый каталог с именем KSTREAM-AGGREGATE-STATE-STORE-0000000002, а соответствующий топик журнализирования изменений теперь называется `test-application-KSTREAM-AGGREGATE-STATE-STORE0000000002-changelog`.

### ПРИМЕЧАНИЕ

Любые изменения до операции с состоянием могут привести к сдвигу в именах, то есть удаление операторов будет иметь тот же эффект сдвига.

Что это значит для нас? При повторном развертывании этого приложения Kafka Streams каталог будет содержать только некоторые основные файлы RocksDB, но не прежнее содержимое. Оно находится в прежнем каталоге хранилища состояния. Обычно пустой каталог хранилища состояния не представляет проблемы, так как Kafka Streams восстановит его из топика журнализирования изменений. За исключением этого случая, топик журнализирования изменений тоже будет создан заново, поэтому он также пустой. Таким образом, несмотря на то что данные все еще хранятся в Kafka, приложение Kafka Streams начнет работу заново с пустым хранилищем состояний из-за изменения имени.

В принципе, можно вернуть прежние смещения и снова обработать данные, но лучше вообще избегать смены имен и явно задавать имя хранилища состояния, не полагаясь на автоматически сгенерированное. В предыдущей главе я предлагал явно задавать имена узлов-обработчиков, чтобы получить осмысленное представление топологии. Но в данном случае явное именование служит не только этой цели, но также делает приложение устойчивым к изменению топологии.

Вернемся к простому примеру с операцией `count()` и обновим приложение, передав в операцию объект `Materialized` (листинг 7.20).

**Листинг 7.20.** Явное задание имени хранилища состояния с помощью объекта Materialized

```
builder.stream("input")
    .groupByKey()
    .count(Materialized.as("counting-store"))
    .toStream()
    .to("output")
```

После того как мы явно указали имя хранилища состояния, Kafka Streams даст каталогу на диске имя `counting-store`, а топику журнализирования изменений — имя `test-application-counting-store-changelog`, и оба этих имени будут зафиксированы. Они останутся неизменными независимо от любых изменений в топологии. Важно отметить, что имена хранилищ состояний в топологии должны быть уникальными. Иначе вы получите исключение `TopologyException`.

### ПРИМЕЧАНИЕ

На сдвиг имен влияют только операции с состоянием. А поскольку операции без состояния не хранят никакого состояния, изменения в именах узлов-обработчиков не будут иметь никакого эффекта.

Суть в том, чтобы *всегда* явно давать имена хранилищам состояний и перераспределять разделы топиков с помощью соответствующего конфигурационного объекта. Давая имена компонентам приложения, которые хранят состояние, можно гарантировать, что изменения в топологии не нарушают совместимость. В табл. 7.1 приводится список конфигурационных объектов и указывается, к каким операциям они применяются.

**Таблица 7.1.** Конфигурационные объекты Kafka Streams, управляющие именованием хранилищ состояний и топиков для перераспределения

Конфигурационный объект	Какие имена определяет	Где используется
Materialized	Хранилище состояния, топик журнализирования изменений	Агрегирование
Repartitioned	Топик для перераспределения	Перераспределение (вручную пользователем)
Grouped	Топик для перераспределения	groupBy (автоматическое перераспределение)
StreamJoined	Хранилище состояния, топик журнализирования изменений, топик для перераспределения	Соединения (автоматическое перераспределение)

Именование хранилищ состояний дает дополнительное преимущество — позволяет запрашивать их во время работы приложения Kafka Streams, предоставляя интерактивные, материализованные представления потоков. Об интерактивных запросах я расскажу в следующей главе.

К данному моменту мы узнали, как Kafka Streams использует хранилища состояний для поддержки операций с состоянием. Мы также узнали, что по умолчанию

Kafka Streams использует постоянные хранилища, но, кроме них, доступны оперативные хранилища, размещаемые в оперативной памяти. В следующем разделе я расскажу, как настраивать типы хранилищ и определять конфигурационные параметры для топиков журнализирования изменений.

#### 7.4.6. Настройка типа хранилища

Все примеры в этой главе используют постоянные хранилища состояний, но выше я говорил, что также можно использовать оперативные хранилища, размещаемые в памяти. Соответственно, возникает вопрос: как использовать хранилище в памяти? До сих пор мы использовали конфигурационный объект `Materialized` для выбора объектов `Serdes` и имени хранилища, но его также можно использовать для предоставления пользовательского экземпляра `StateStore`. Kafka Streams упрощает применение разных версий доступных типов хранилищ в памяти (до сих пор я рассматривал только «классические» хранилища типа «ключ — значение», но в следующей главе я расскажу о хранилищах сеансов, оконных хранилищах и хранилищах с отметками времени).

Лучший способ узнать, как задействовать другой тип хранилища, — изменить один из существующих примеров. Давайте вернемся к первому примеру, в котором операция с состоянием используется для отслеживания результатов в онлайн-игре в покер (листинг 7.21).

**Листинг 7.21.** Выполнение свертки в Kafka Streams с использованием хранилищ в памяти

```
KStream<String, Double> pokerScoreStream = builder.stream("poker-game",
    Consumed.with(Serdes.String(), Serdes.Double()));

pokerScoreStream
    .groupByKey()
    .reduce(Double::sum,
        Materialized.<String, Double>as(
            Stores.inMemoryKeyValueStore("memory-poker-score-store"))
                .withKeySerde(Serdes.String())
                .withValueSerde(Serdes.Double()))
    .toStream()
    .to("total-scores",
        Produced.with(Serdes.String(), Serdes.Double()));
```

Итак, с помощью перегруженного метода `Materialized.as` мы передаем `StoreSupplier`, используя один из фабричных методов, доступных в классе `Stores`. Обратите внимание, что, как и прежде, мы передаем экземпляры `Serde`, необходимые для работы с хранилищем. Вот и все, что нужно для переключения типа хранилища с постоянного на оперативный.

#### ПРИМЕЧАНИЕ

Переключение на другой тип хранилища не вызывает сложностей, поэтому я привел только один пример.

Чем выгодно хранилище в оперативной памяти? Оно обеспечит более быстрый доступ к хранимым значениям, потому что ему не нужно обращаться к диску для их

извлечения. Таким образом, топология, использующая хранилища в оперативной памяти, будет иметь более высокую пропускную способность, чем топология, использующая постоянные хранилища. Но есть нюансы, которые следует учитывать.

Во-первых, объем хранилища в памяти ограничен, и как только хранилище заполнится, оно может вызвать исключение `OutOfMemoryError`. Обратите внимание, что проблемы с исчерпанием памяти можно избежать, использовав метод `Stores.1ruMap`, который будет вытеснять редко используемые записи при достижении максимального настроенного объема. Во-вторых, в случае штатной остановки и повторного запуска приложения Kafka Streams с постоянными хранилищами быстрее возобновляют обработку потоков, потому что хранилище состояний уже будет готово, а вот на восстановление содержимого хранилищ в памяти из топика журналирования изменений требуется время.

Kafka Streams предоставляет фабричный класс `Stores` с методами для создания экземпляров `StoreSupplier` или `StoreBuilder`. Выбор между ними зависит от Kafka Streams API. При использовании DSL вы будете создавать экземпляры `StoreSupplier` и включать их в объект `Materialized`. А при использовании Processor API вы будете создавать экземпляры `StoreBuilder` и вручную добавлять их в топологию. Processor API мы рассмотрим в главе 10.

#### СОВЕТ

Чтобы увидеть все типы хранилищ, можно создать представление JavaDoc для класса `Stores` (<http://mng.bz/eEez>).

Теперь, узнав, как задать тип хранилища, рассмотрим еще один вопрос, связанный с хранилищами состояний: как настроить топик журналирования изменений.

### 7.4.7. Настройка топиков журналирования изменений

В топиках журналирования изменений нет ничего необычного. Для их настройки можно использовать любые доступные конфигурационные параметры. Но многие предпочитают ограничиваться настройками по умолчанию и прибегают к явному определению настроек, только когда в этом есть необходимость.

#### ПРИМЕЧАНИЕ

Журналы изменений хранилищ состояний — это сжатые топики, обсуждавшиеся в главе 2. Как вы помните, семантика удаления требует указать в ключе значение `null`, поэтому для удаления записи из хранилища состояний навсегда нужно выполнить операцию `put(key, null)`.

Вернемся к предыдущему примеру, где мы присвоили хранилищу состояний свое имя. Данные, обрабатываемые этим приложением, имеют много уникальных ключей. Журналы изменений в Kafka Streams представлены сжатыми топиками, которые используют иной подход к очистке старых записей.

Вместо удаления сегментов журнала времени от времени сегменты журналов сжимаются: для каждого ключа оставляется только последняя запись, а более старые записи с тем же ключом удаляются. Но, поскольку пространство ключей велико, сжатия может оказаться недостаточно, так как размер сегмента журнала будет продолжать

растя. Существует простое решение этой проблемы — достаточно задать стратегию очистки `delete` и `compact` (листинг 7.22).

#### Листинг 7.22. Задание стратегии очистки

```
Map<String, String> changeLogConfigs = new HashMap<>();
changeLogConfigs.put("cleanup.policy", "compact,delete");

builder.stream("input")
    .groupByKey()
    .count(Materialized.as("counting-store"))
        .withLoggingEnabled(changeLogConfigs) ← Использует метод withLoggingEnabled
    .toStream()                                для задания конфигурации
    .to("output")
```

Вы можете скорректировать конфигурацию для этой конкретного топика журнализации изменений. Выше я упоминал, что для отключения кэширования, которое Kafka Streams использует для операций с состоянием, нужно присвоить параметру `StreamsConfig.CACHE_MAX_BYTES_BUFFERING_CONFIG` значение `0`. Но он находится в конфигурации приложения и поэтому применяется глобально ко всем операциям с состоянием. Если вам нужно отключить кэширование только для определенной операции, вызовите метод `Materialized.withCachingDisabled()` при передаче объекта `Materialized`.

#### ВНИМАНИЕ

Объект `Materialized` также предоставляет метод для отключения журналирования. Его вызов приведет к тому, что хранилище состояний лишится топика журнализации изменений и в случае перезапуска приложения не сможет восстановить прежнее состояние. Используйте этот метод только в случае крайней необходимости. Работая с Kafka Streams, я еще не встречал веской причины для использования этого метода.

## ИТОГИ ГЛАВЫ

- Потоковая обработка нередко требует хранения состояния. В простых случаях возможна обработка без сохранения состояния, но для принятия сложных решений без операций с состоянием не обойтись.
- Kafka Streams предоставляет операции с состоянием, которые свертывают, агрегируют и объединяют данные. Хранилище состояний для таких операций создается автоматически, и по умолчанию используются постоянные хранилища.
- Для поддержки любых операций с состоянием можно использовать хранилища в оперативной памяти, передав в конфигурационный объект `Materialized` экземпляр `StoreSupplier`, созданный с помощью фабричного класса `Stores`.
- Для выполнения операций записи должны иметь действительные ключи. Если записи не имеют ключа или требуется сгруппировать или объединить записи по другому ключу, то ключ можно изменить, а Kafka Streams автоматически выполнит перераспределение данных.
- Важно всегда явно указывать имена хранилищ состояний и топиков для перераспределения, чтобы обеспечить отказоустойчивость приложения при внесении изменений в топологию.

# 8

## *KTable API*

### **В этой главе**

- ✓ Потоки изменений, KTable и GlobalKTable.
- ✓ Агрегирование записей с помощью KTable.
- ✓ Обогащение потоков событий с помощью соединений.
- ✓ Соединение KTable с другим KTable.

Эта глава познакомит вас с новым API в Kafka Streams: `Ktable`. KTable — это поток обновлений или изменений. Мы уже использовали KTable, выполняя операции агрегирования, потому что любые такие операции в Kafka Streams выполняются с привлечением KTable. KTable — это важная абстракция, предназначенная для работы с записями с одинаковыми ключами. В KStream записи с одинаковыми ключами являются независимыми событиями. Но в KTable каждая последующая запись обновляет предыдущую с тем же ключом.

Зачем изучать потоки обновлений? Чтобы уметь использовать их в случаях, когда в некотором фрагменте данных интерес представляет только последняя запись. Например, рассмотрим профиль пользователя. Когда кто-то обновляет свой профиль, только самая новая запись является правильной. Все предыдущие версии профиля не имеют значения. По сравнению с реляционной базой данных поток событий (KStream) можно рассматривать как серию вставок, где первичный ключ — это автоматически увеличивающееся число. Каждая вновь вставленная запись не имеет отношения к предыдущим. Но в KTable роль первичного ключа играет ключ в паре «ключ — значение». Поэтому вместо вставки новой записи происходит обновление существующей, если она имеется.

Вы узнаете, как производить агрегирование с помощью **KTable**. Агрегирование в **KTable** работает иначе, чем в **KStream**, из-за отсутствия необходимости выполнять группировку по первичному ключу, чтобы получить одну запись. Вместо этого вам придется подумать, как сгруппировать записи для расчета результата агрегирования.

**KTable** также можно использовать в качестве таблицы поиска и обогащать записи в потоке событий дополнительными сведениями за счет объединения их с записями в таблице. Кроме того, можно даже соединить две таблицы, используя внешний ключ. Помимо этого, вы узнаете об уникальной конструкции **GlobalKTable**. **KTable** делится на разделы, и это означает, что каждый экземпляр содержит данные только для одного раздела. Но **GlobalKTable** включает все записи из своего исходного топика по всем экземплярам приложения.

## 8.1. KTABLE: ПОТОК ОБНОВЛЕНИЙ

Чтобы усвоить концепцию потока обновлений, полезно сравнить его с потоком событий и рассмотреть различия. Разберем конкретный пример отслеживания обновлений цен на акции (рис. 8.1).

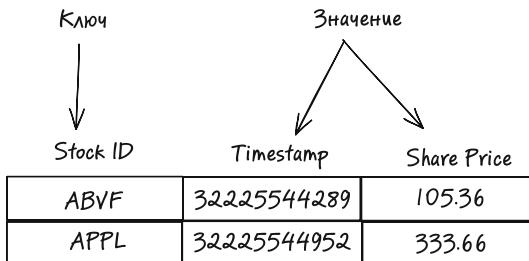


**Рис. 8.1.** Диаграмма неограниченного потока котировок акций

Как видите, каждая котировка акций представляет собой дискретное событие, они не связаны друг с другом. Даже если за несколько котировок отвечает одна компания, они все равно рассматриваются по отдельности. Такое представление событий соответствует потоку данных событий, описываемому **KStream**.

Взглянем теперь, как эта концепция соотносится с таблицами базы данных. Каждая запись представляет результат операции вставки в таблицу, в которой роль первичного ключа играет число, монотонно увеличивающееся в каждой операции вставки, как показано в таблице котировок акций на рис. 8.2.

Далее обратимся снова к потоку записей. Поскольку записи автономны, данный поток соответствует операциям вставки в таблицу. На рис. 8.3 эти две концепции объединены в целях иллюстрации.

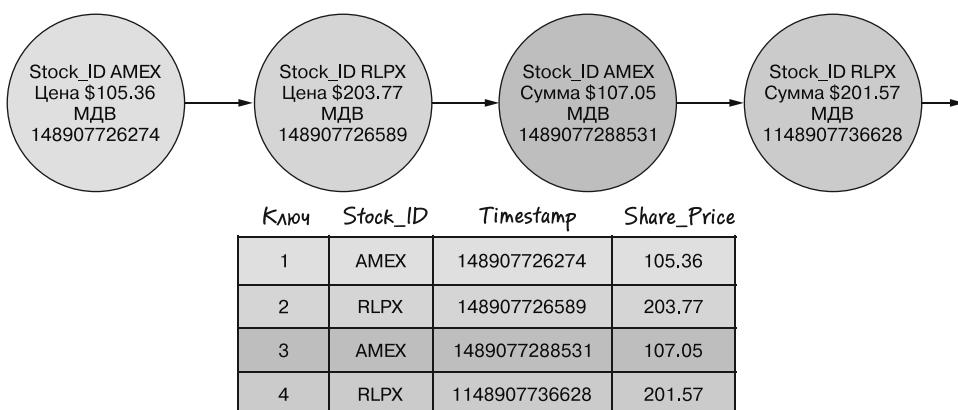


Строки из таблицы можно представить в виде пар «ключ — значение».

Например, первую строку можно преобразовать в следующую пару «ключ — значение»

```
{
    key: { stock_id:ABVF },
    value: { ts: 32225544289, price: 105.36 }
}
```

**Рис. 8.2.** Простая таблица базы данных с курсом акций различных компаний. В ней есть столбец с ключом, а также другие столбцы со значениями. Ее строки можно рассматривать как пары «ключ — значение», если «свалить» все остальные столбцы в контейнер «значение»



Этот рисунок демонстрирует взаимосвязь между событиями и операциями вставки в таблицу. Хотя здесь приведены курсы акций только для двух компаний, событий — четыре, потому что мы рассматриваем каждый элемент в потоке как отдельное событие.

В результате каждое событие представляет операцию вставки, которая увеличивает значение ключа (key) на единицу.

С учетом этого каждое событие представляет новую независимую запись (операцию вставки в таблицу базы данных)

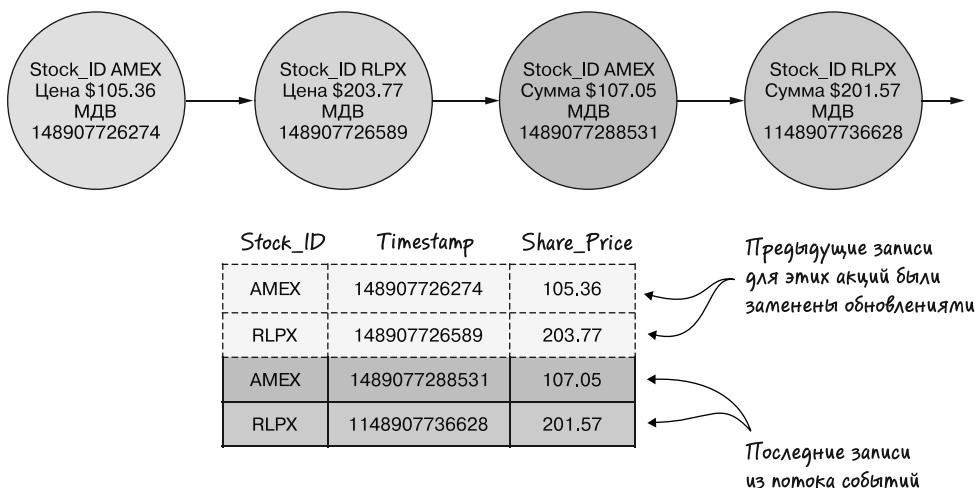
**Рис. 8.3.** Поток индивидуальных событий по сравнению со вставками в таблицу базы данных. Аналогично можно представить построчную потоковую обработку таблицы

Самое главное здесь — возможность рассматривать поток событий аналогично последовательности операций вставки в таблицу, благодаря чему можно лучше разобраться в использовании потоков данных для работы с событиями. Следующий этап — изучение случая, когда события в потоке взаимосвязаны.

### 8.1.1. Обновления записей (журнал изменений)

Допустим, вы решили отслеживать поведение покупателей, для чего получаете поток транзакций покупателя, но теперь отслеживаете их действия в различные моменты времени. Если добавить ключ — идентификатор покупателя, то можно связать события покупки друг с другом и получить поток обновлений вместо потока событий.

Если поток событий мы сравнивали с журналом, то поток обновлений можно сравнить с журналом изменений. Рисунок 8.4 иллюстрирует эту концепцию.



*При использовании идентификатора акции в качестве первичного ключа в журнале изменений последующие события с тем же ключом рассматриваются как обновления. В данном случае записей будет только две, по одной на компанию. Хотя в будущем могут поступить новые записи, относящиеся к этим же компаниям, они не будут накапливаться*

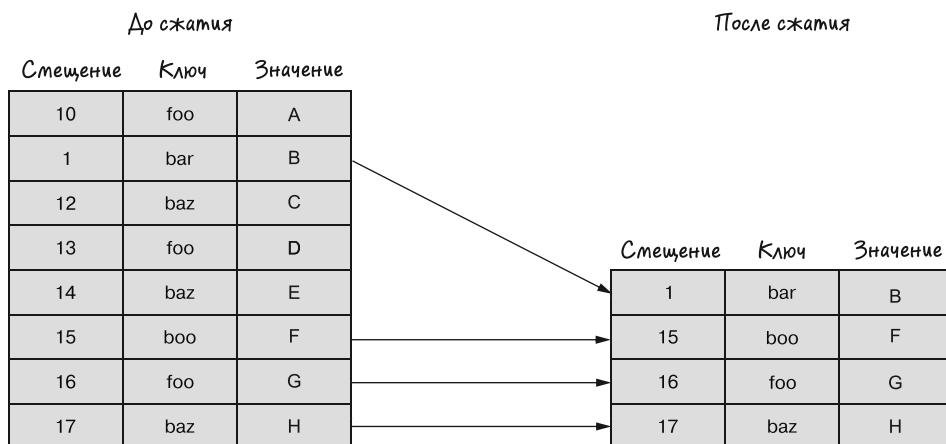
**Рис. 8.4.** В журнале изменений каждая входящая запись заменяет предыдущую запись с тем же ключом (если таковая есть). В потоке записей у нас имеется четыре события, а в потоке обновлений (в журнале изменений) их только два

На этом рисунке видна взаимосвязь между потоком обновлений и таблицей базы данных. Как журнал (log), так и журнал изменений (changelog) отражают добавляемые в конец файла входящие записи. В журнале видны все записи, а в журнале изменений — только последняя запись для каждого ключа.

## ПРИМЕЧАНИЕ

Как в журнале, так и в журнале изменений записи при поступлении добавляются в конец файла. Различие в том, что журнал используется, когда нужно видеть *все* записи, а журнал изменений — только когда нужно видеть *последнюю* запись для каждого ключа.

Для сокращения журнала с сохранением последних записей для всех ключей можно воспользоваться сжатием журналов, обсуждавшимся в главе 2. Результат сжатия журнала показан на рис. 8.5. Раз нас интересуют только последние значения, то можно удалить более старые пары «ключ — значение».



**Рис. 8.5.** Слева показан журнал до сжатия, в котором можно заметить повторяющиеся ключи с разными значениями — обновления. Справа показан журнал после сжатия — для каждого ключа оставлено только последнее значение, и размер журнала за счет этого уменьшился

## ПРИМЕЧАНИЕ

Информация для этого раздела взята из статей Джая Крепса (Jay Kreps) *Introducing Kafka Streams: Stream Processing Made Simple* (<http://mng.bz/49HO>) и *The Log: What Every Software Engineer Should Know About Real-time Data's Unifying Abstraction* (<http://mng.bz/eE3w>).

Вы уже знакомы с потоками событий по работе с **KStream**. Следующий этап, после того как мы разобрались во взаимосвязи между потоками и таблицами, — сравнение потока событий с потоком обновлений. Для представления журналов изменений (потоков обновлений) мы будем использовать абстракцию **KTable**.

### 8.1.2. KStream и KTable в действии

Давайте сравним **KStream** и **KTable**. Для этого запустим простое приложение слежения за котировками акций. **KStream** и **KTable** будут читать и выводить записи в консоль с помощью метода `print()`. Приложение слежения за котировками выполнит три итерации и создаст девять записей.

## ПРИМЕЧАНИЕ

В `KTable` нет таких методов, как `print()` и `peek()`, поэтому для вывода записей нужно преобразовать `KTable` из потока обновлений в поток событий, используя метод `toStream()`.

В листинге 8.1 показан пример программы, которая выводит события изменения котировок акций в консоль (находится в `src/main/java/bbejeck/chapter_8/KStreamVsKTableExample.java`). Исходный код можно найти по адресу <https://github.com/bbejeck/KafkaStreamsInAction 2ndEdition>.

### Листинг 8.1. Вывод в консоль KTable и KStream

```
KTable<String, StockTickerData> stockTickerTable =
    builder.table(STOCK_TICKER_TABLE_TOPIC);   ← Создание экземпляра KTable
KStream<String, StockTickerData> stockTickerStream =
    builder.stream(STOCK_TICKER_STREAM_TOPIC);   ← Создание экземпляра KStream

stockTickerTable.toStream()
    .print(Printed.<String, StockTickerData>toSysOut()
        .withLabel("Stocks-KTable"));   ← KTable выводит результаты в консоль

stockTickerStream
    .print(Printed.<String, StockTickerData>toSysOut()
        .withLabel("Stocks-KStream"));   ← KStream выводит
                                         результаты в консоль
```

## ИСПОЛЬЗОВАНИЕ ОБЪЕКТОВ SERDE ПО УМОЛЧАНИЮ

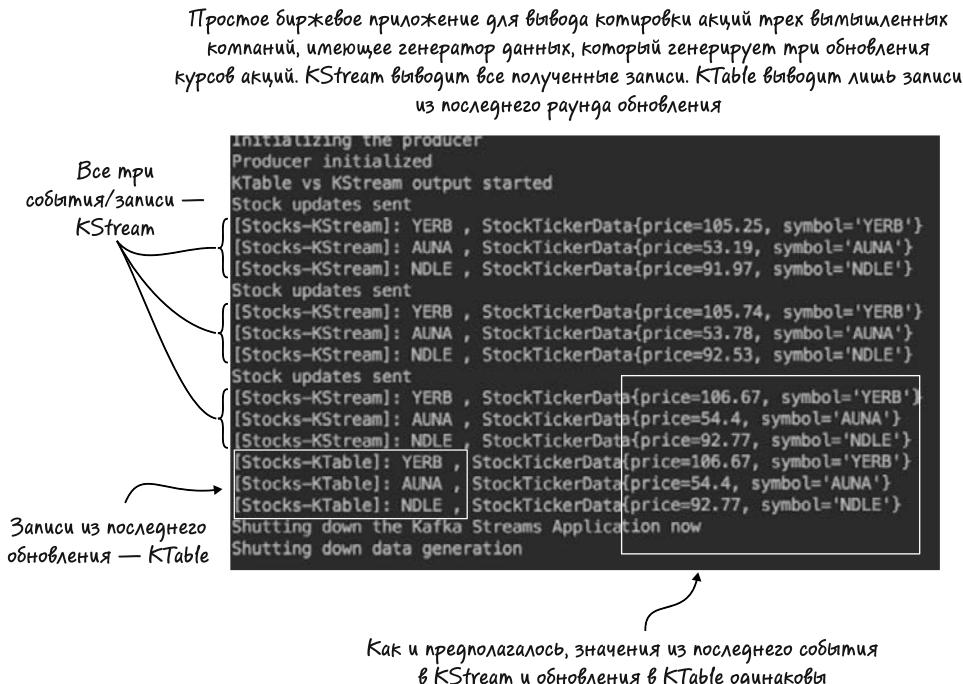
При создании `KTable` и `KStream` мы не задавали никаких объектов `Serde`. А равно и при обоих вызовах метода `print()`. А все благодаря тому, что мы внесли в конфигурацию объекты `Serde` для использования по умолчанию, примерно вот так:

```
props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
    Serdes.String().getClass().getName());
props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
    StreamsSerdes.StockTickerSerde().getClass().getName());
```

Если же вы хотели бы использовать другие типы данных, то для чтения или внесения записей необходимо передать объекты `Serde` через перегруженные методы.

С точки зрения объекта `KTable` он получил не девять отдельных записей, а три исходных записи и две порции обновлений и вывел в консоль только последнюю из порции. Обратите внимание, что записи в `KTable` совпадают с последними тремя записями, опубликованными `KStream`. Мы обсудим механизмы, с помощью которых `KTable` выводит только обновления, в следующем разделе.

Главный вывод: записи с одинаковыми ключами в потоке данных по своей сути обновления, а не самостоятельные новые записи (рис. 8.6). Именно понятие потока обновлений лежит в основе интерфейса `KTable`.



**Рис. 8.6.** Вывод записей с одинаковыми ключами: KTable по сравнению с KStream

## 8.2. ОБЪЕКТЫ KTABLE ИМЕЮТ ХРАНИМОЕ СОСТОЯНИЕ

В предыдущем примере, когда мы создали таблицу с помощью `builder.table()`, библиотека Kafka Streams также создала хранилище `StateStore` для отслеживания состояния, которое по умолчанию является постоянным. Поскольку хранилища состояний работают только с массивами байтов, нам нужно передать экземпляры `Serde`, чтобы хранилище могло (де)сериализовать ключи и значения. Конкретные экземпляры `Serde` мы передаем потоку событий с помощью конфигурационного объекта `Consumed`, и точно так же можно поступить при создании `KTable`:

```
builder.table(STOCK_TICKER_TABLE_TOPIC,
    Consumed.with(Serdes.String(),
        StockTradeSerde()));
```

Теперь хранилище состояний будет использовать объекты `Serde`, которые мы передали в объекте `Consumed`. Перегруженная версия `StreamsBuilder.table` также принимает экземпляр `Materialized`, что позволяет настроить тип хранилища и задать его имя, чтобы обеспечить доступность для запросов. Интерактивные запросы мы обсудим в главе 13.

Можно также создать `KTable` напрямую с помощью метода `KStream.toTable`. Этот метод меняет интерпретацию записей, после вызова они будут рассматриваться не как события, а как обновления. Можно также использовать метод `KTable.toStream` для

преобразования потока обновлений в поток событий. Последнее преобразование мы обсудим в следующем разделе, когда будем рассматривать KTable. Главное, что следует запомнить: KTable создается непосредственно из топика, а Kafka Streams автоматически добавляет хранилище состояний, поддерживающее KTable.

Выше я рассказал, как KTable обрабатывает операции вставки и обновления, а как удалить запись? Чтобы удалить запись из KTable, нужно отправить пару «ключ — значение» со значением null. Значение null действует как маркер, который на сленгге Kafka Streams называют «надгробием», и в конечном счете удаляется из хранилища состояний и топика журнализирования изменений, а следовательно, и из таблицы.

Как и KStream, KTable распределяется по задачам, число которых определяется количеством разделов в исходном топике. Это распределение по задачам означает, что записи в таблице могут находиться в разных экземплярах приложения.

### 8.3. KTABLE

В KTable есть методы, похожие на методы в KStream: filter, filterNot, mapValues и transformValues. Они тоже поддерживают возможность объединения в цепочки, возвращая новый экземпляр KTable.

Функционально эти методы очень похожи на аналогичные методы в KStream, но имеют некоторые отличия: к парам «ключ — значение», в которых значение равно null, они применяют семантику удаления.

Вот как семантика удаления влияет на работу KTable.

- Если входная пара «ключ — значение» содержит значение null, то узел-обработчик не выполняет с ней никаких действий, а просто пересыпает ее в новую таблицу как маркер «надгробия».
- В методах filter и filterNot, если запись не соответствует предикату, она заменяется маркером «надгробия», который пересыпается в новую таблицу.

Пример вы найдете в KTableFilterExample в пакете bvejeck.chapter\_8. Он применяет KTable.filter к потоку записей, часть из которых имеет значения null, и отфильтровывает некоторые значения, не равные null. Но мы уже обсуждали фильтрацию ранее, поэтому я не буду рассматривать этот пример здесь и дам вам возможность сделать это самостоятельно.

Теперь обсудим агрегирование и соединение с помощью KTable.

### 8.4. АГРЕГИРОВАНИЕ С ПОМОЩЬЮ KTABLE

Агрегирование в KTable действует иначе, чем в KStream. Давайте исследуем имеющиеся различия на примере. Представьте, что мы создаем приложение для отслеживания цен на акции. Для любой акции нас интересует только последняя цена, поэтому имеет смысл использовать KTable. Кроме того, мы хотели бы отслеживать динамику работы различных сегментов рынка. Например, мы могли бы сгруппировать акции Google, Apple и Confluent в сегмент рынка технологий. Поэтому нам понадобится выполнить агрегирование и сгруппировать различные акции по сегменту рынка. Реализация агрегирования с помощью KTable будет выглядеть так, как показано в листинге 8.2.

**Листинг 8.2.** Агрегирование с помощью KTable

```
KTable<String, StockAlert> stockTable =
    builder.table("stock-alert",
        Consumed.with(stringSerde, stockAlertSerde));
```

Создается исходный экземпляр KTable

```
stockTable.groupBy((key, value) ->
    KeyValue.pair(value.getMarketSegment(), value),
    Grouped.with(stringSerde, stockAlertSerde))
    .aggregate(segmentInitializer, ← Создается агрегат
        adderAggregator, ← Передается суммирующий агрегатор
        subtractorAggregator, ← Передается вычитающий агрегатор
        Materialized.with(stringSerde, segmentSerde))
    .toStream()
```

Настраивается группировка по сегменту рынка, и через конфигурационный объект Grouped задаются экземпляры Serde, необходимые для перераспределения

```
.to("stock-alert-aggregate",
    Produced.with(stringSerde, segmentSerde));
```

Мы создаем KTable, выполняем groupBy и изменяем ключ, используя сегмент рынка, что приведет к перераспределению. Исходный ключ — это символ акции, поэтому разные акции из данного сегмента рынка могут оказаться в разных разделах.

Но это требование скрывается за тем фактом, что при агрегировании с помощью KTable всегда нужно выполнять операцию groupBy. Почему? Напомню, что в KTable входной ключ считается первичным ключом. Как и в реляционной базе данных, группировка по первичному ключу всегда приводит к получению единственной записи. Поэтому записи нужно группировать по другому полю — комбинирование первичного ключа и группирующих полей даст результаты, подходящие для агрегирования. Как и в KStream, вызов метода KTable.groupBy возвращает промежуточную таблицу — KGroupedTable, которая используется для вызова метода aggregate.

Есть еще одно отличие. Метод aggregate в KTable, как и в KStream, принимает в первом параметре экземпляр Initializer, задающий значение по умолчанию для первой операции агрегирования. Но далее он принимает два агрегатора, один из которых добавляет новое значение в агрегат, а другой вычитает из агрегата старое значение предыдущей записи с тем же ключом. Взгляните на рис. 8.7, чтобы понять, что происходит.

```
(key, newValue, aggr) -> {
    aggr.add(newValue);
    return aggr;
}
```

Суммирующий агрегатор добавляет новое значение для данного ключа в агрегат

```
(key, previousValue, aggr) -> {
    aggr.subtract(previousValue);
    return aggr;
}
```

Вычитающий агрегатор вычитает предыдущее значение для данного ключа из агрегата

**Рис. 8.7.** Операция агрегирования в KTable использует два агрегатора: суммирующий и вычитающий

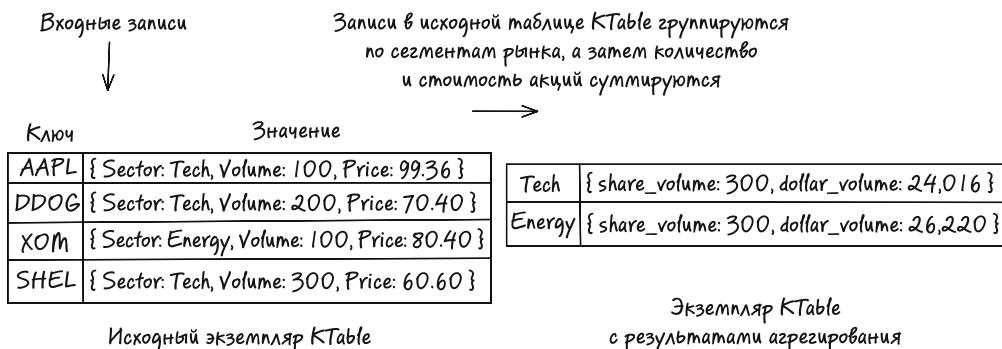
Вот еще один взгляд на агрегирование. Выполняя то же самое в реляционной таблице и суммируя значения в записях, отобранных группировкой, мы бы получили только единственное последнее значение на запись. Например, SQL-эквивалент показанного выше агрегирования в KTable мог бы выглядеть как в листинге 8.3.

### Листинг 8.3. SQL-эквивалент агрегирования в KTable

```
SELECT market_segment,
       sum(share_volume) as total_shares,
       sum(share_price * share_volume) as dollar_volume
  FROM stock_alerts
 GROUP BY market_segment;
```

Когда поступает новая запись, первым делом обновляется таблица оповещений. Затем запускается агрегирующий запрос, чтобы извлечь обновленную информацию. Это в точности описывает работу KTable. Новая входная запись обновляет таблицу `stock_alerts`, и она передается для агрегирования. В обновлении для каждой акции может быть только одна запись, поэтому производится добавление в агрегат значения из новой записи и удаление значения из предыдущей записи.

Кому-то процесс может показаться сложным для полного понимания, поэтому поясню его несколькими иллюстрациями. Представим, что были получены некоторые записи и запустился расчет некоторых агрегатов (рис. 8.8).



**Рис. 8.8.** Записи поступают в объект KTable, который вычисляет агрегат для каждой из них

Операция агрегирования суммирует количество акций в каждой транзакции и сумму торгов в долларах, которая вычисляется путем умножения цены акции на количество. Затем происходит новая сделка с акциями (рис. 8.9).

Информация о сделке, включающей акции CFLT (компании Confluent), поступает в исходный топик и передается в KTable. Поскольку для CFLT нет предыдущей записи, выполняется только вставка записи в исходную таблицу. Так как компания Confluent относится к технологическому сегменту, необходимо обновить этот агрегат (рис. 8.10).

Для обновления агрегата в поле `share_volume` добавляется количество проданных акций и увеличивается поле `dollar_volume` путем умножения количества акций на значение в поле со стоимостью одной акции. Ситуация становится интереснее, когда поступает информация еще об одной сделке с акциями CFLT. Взгляните на рис. 8.11.

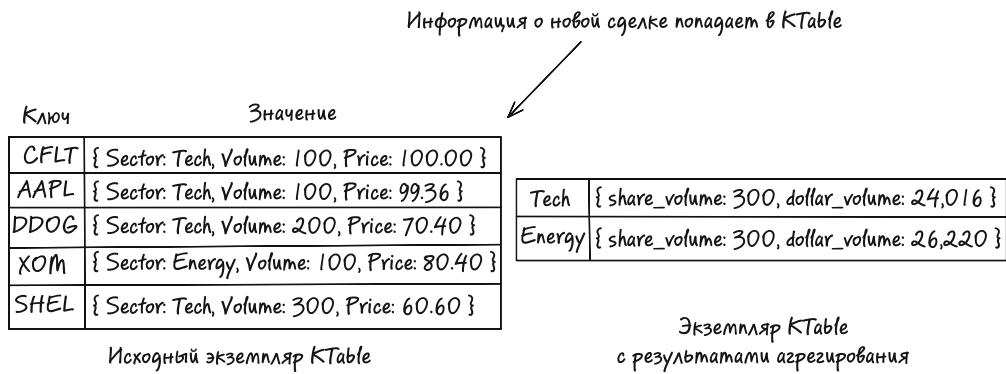


Рис. 8.9. Появляется информация о новой сделке, и обновляется исходный объект KTable

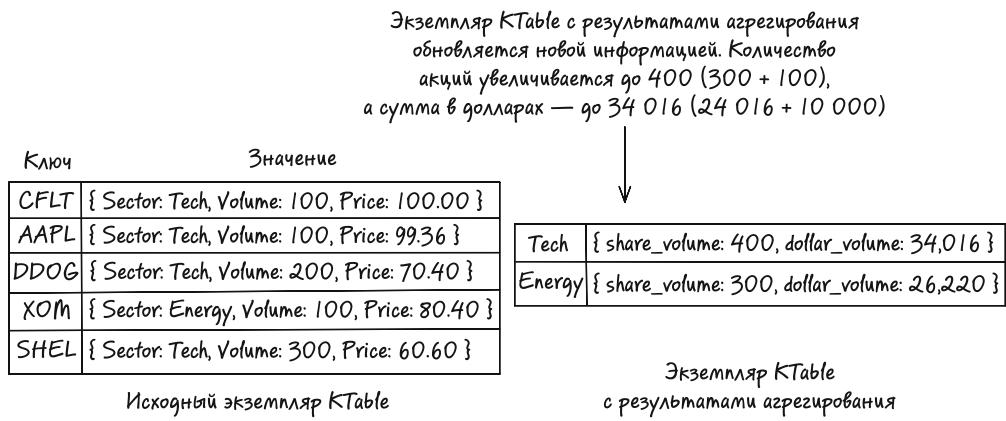


Рис. 8.10. С вновь полученной записью обновляется агрегат

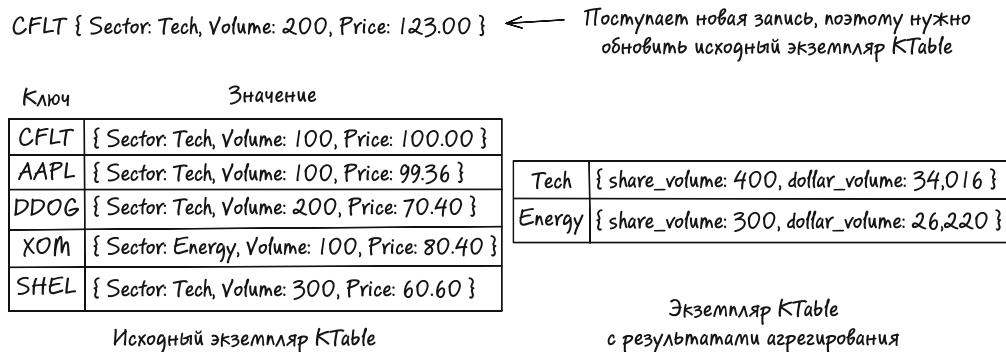


Рис. 8.11. Появляется информация еще об одной сделке с акциями CFLT, и запускается серия событий обновления

Поскольку в KTable уже есть запись для акций CFLT, естественной мыслью является обновление ключа CFLT новым значением. Но сначала библиотека Kafka Streams должна получить предыдущую запись из хранилища состояний и сохранить ее в переменной, а уже потом добавлять новое значение в таблицу. Мы должны сохранить последнее значение, потому что оно необходимо для обновления агрегата далее в потоке.

В каждом сегменте рынка в результатах агрегирования каждой акции соответствует одна запись. Когда в сегмент поступает новая запись, выполняется двухэтапный процесс обновления. Во-первых, из агрегата вычитается предыдущая и прибавляется новая запись. Дополнительно из-за того, что ключ, возвращаемый функцией groupBy, мог измениться, Kafka Streams будет пересыпать старые и новые значения отдельно. Обобщенная схема всего процесса обновления агрегата в KTable показана на рис. 8.12.

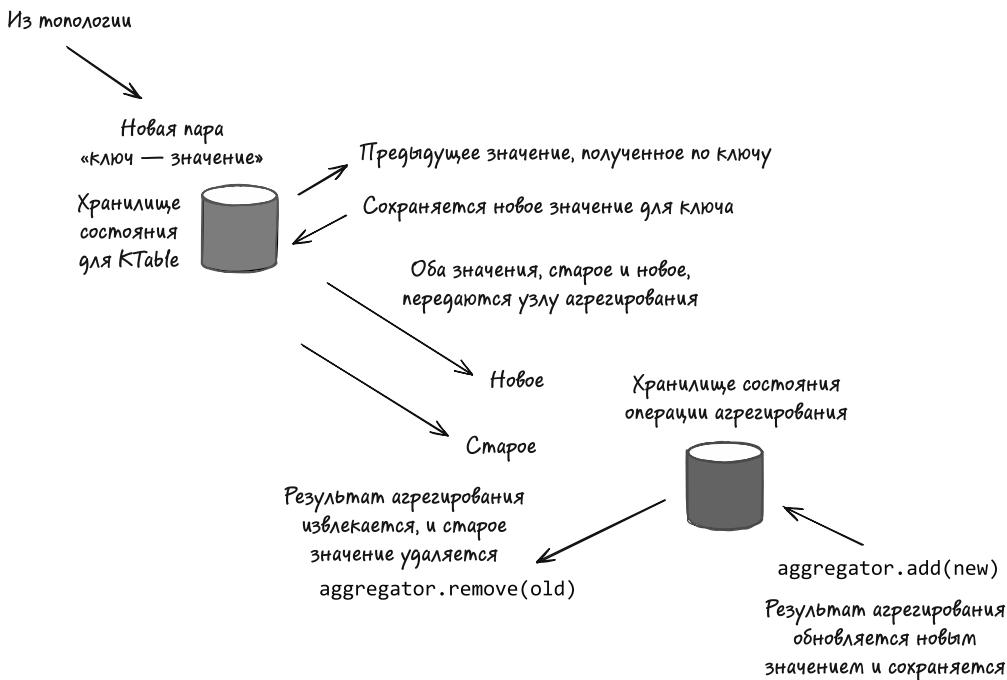


Рис. 8.12. Обобщенная схема всего процесса обновления агрегата в KTable

Теперь, увидев, как работает агрегирование в KTable, рассмотрим экземпляры Aggregator. Но, так как мы уже рассматривали их в главе 7, исследуем только логику суммирования и вычитания. Имейте в виду, что, несмотря на простоту примера, основные принципы, которые он демонстрирует, применимы к любой операции агрегирования в KTable. Начнем с суммирования (некоторые детали опущены для простоты) (листинг 8.4).

**Листинг 8.4.** Логика суммирования в Aggregator

```
final Aggregator<String, StockAlert, SegmentAggregate> adderAggregator =  
    (key, newStockAlert, currentAgg) -> {  
  
    long currentShareVolume =  
        newStockAlert.getShareVolume(); ← Из текущего события StockAlert  
    double currentDollarVolume =  
        newStockAlert.getShareVolume() * newStockAlert.getSharePrice(); ← Для текущего  
    } ← события StockAlert вычисляется сумма сделки в долларах  
  
    aggBuilder.setShareVolume(currentAgg.getShareVolume() +  
        currentShareVolume); ← К текущему общему  
    aggBuilder.setDollarVolume(currentAgg.getDollarVolume() +  
        currentDollarVolume); ← количеству акций  
    } ← прибавляется количество акций из последнего  
    ← событии StockAlert  
    ← К текущей общей сумме продаж  
    ← прибавляется вычисленная  
    ← сумма новой сделки
```

Как видите, суммирование имеет простую логику работы: из последнего события `StockAlert` извлекается количество акций и прибавляется к текущему агрегату. То же самое делается с суммой торгов в долларах (после вычисления путем умножения количества акций на цену одной акции).

**ПРИМЕЧАНИЕ**

Объекты Protobuf неизменяемы, поэтому при обновлении значений для каждого объекта необходимо создавать новые экземпляры, используя сгенерированный конструктор.

Теперь перейдем к логике вычитания. Как вы наверняка догадались, она выполняет обратную операцию — вычитает те же значения и результаты вычислений для предыдущей записи с информацией об акциях той же компании в данном сегменте рынка. Поскольку сигнатура не изменилась, я покажу только вычисления (некоторые детали опущены для простоты) (листинг 8.5).

**Листинг 8.5.** Логика вычитания в Aggregator

```
long prevShareVolume = prevStockAlert.getShareVolume();  
double prevDollarVolume =  
    prevStockAlert.getShareVolume() * prevStockAlert.getSharePrice();  
  
aggBuilder.setShareVolume(currentAgg.getShareVolume()  
    - prevShareVolume); ← Вычитается количество акций,  
aggBuilder.setDollarVolume(currentAgg.getDollarVolume())  
    - prevDollarVolume); ← полученное в предыдущем  
    ← событии StockAlert  
    ← Вычитается сумма сделки,  
    ← соответствующая предыдущему  
    ← событию StockAlert
```

Логика проста: мы вычитаем значения, полученные в `StockAlert`, которые Kafka Streams заменила в агрегате.

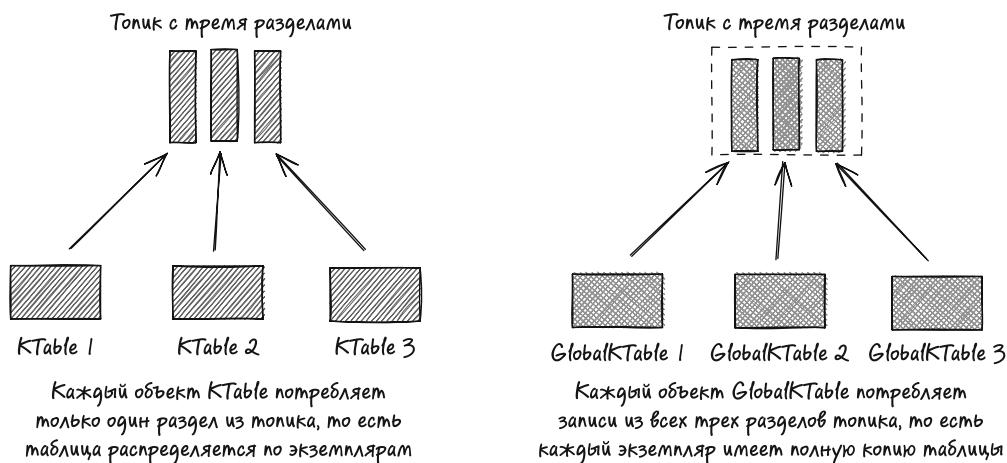
Итак, как мы выяснили, операция агрегирования в `KTable` сохраняет только последнее значение для каждой уникальной комбинации исходного ключа `KTable` и ключа, используемого для группировки. Здесь стоит отметить, что `KTable` также предоставляет методы `reduce` и `count`, при работе с которыми вы будете выполнять аналогичные шаги: сначала вызывать `groupBy`, а в вызов `reduce` передавать реализацию `Reducer` для суммирующего и вычитающего агрегатора. Я не буду рассматривать их

здесь, чтобы не повторяться, но в исходном коде примеров для книги вы сможете увидеть, как используются оба этих метода, `reduce` и `count`.

На этом мы завершаем знакомство с KTable. Но прежде, чем перейти к более сложным операциям, я хотел бы рассмотреть еще одну абстракцию таблиц — GlobalKTable.

## 8.5. GLOBALKTABLE

Я уже упоминал GlobalKTable выше в этой главе, когда говорил, что таблица KTable делится на разделы и, следовательно, распределена между экземплярами приложения Kafka Streams (конечно, с тем же идентификатором приложения). Другими словами, KTable содержит только записи из одного раздела топика. Уникальность GlobalKTable в том, что она потребляет все данные из исходного топика. Это означает, что в таблице содержится полная копия всех записей из всех экземпляров приложения. Взгляните на рис. 8.13, иллюстрирующий это.



**Рис. 8.13.** GlobalKTable содержит все записи в топике из всех экземпляров приложения

Как видите, исходный топик для KTable имеет три раздела, и в каждом из трех экземпляров приложения объект KTable потребляет записи только из одного раздела. А объект GlobalKTable, напротив, имеет полную копию содержимого всех трех разделов исходного топика в каждом экземпляре. Kafka Streams материализует GlobalKTable на локальном диске в KeyValueStore, но не создает топик журнализации изменений, потому что исходный топик служит резервной копией для восстановления. Код в листинге 8.6 показывает, как создать объект GlobalKTable.

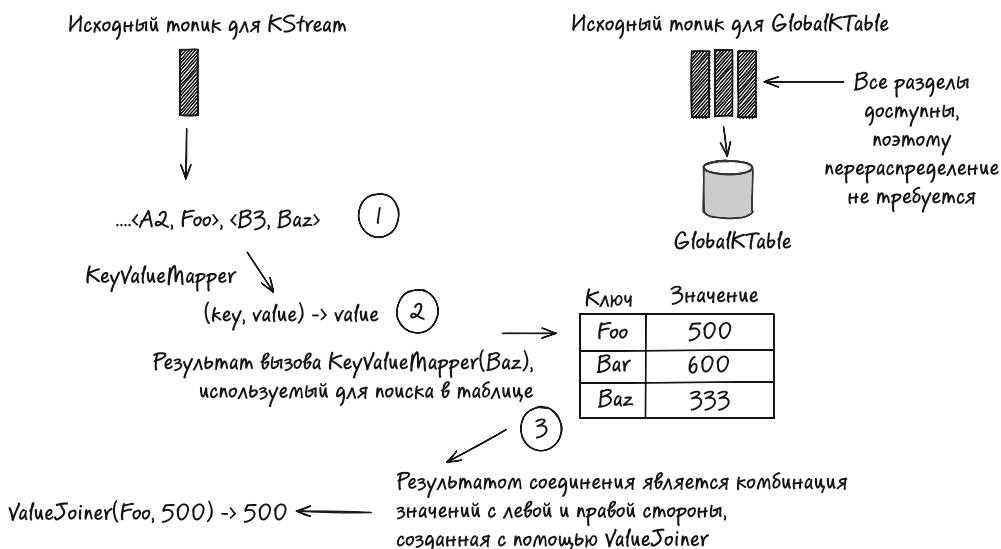
### Листинг 8.6. Создание GlobalKTable

```
StreamsBuilder builder = new StreamsBuilder();
GlobalKTable<String, String> globalTable =
    builder.globalTable("topic",
        Consumed.with(Serdes.String(),
                      Serdes.String()));
```

`GlobalKTable` не предлагает API. Поэтому возникает естественный вопрос: когда использовать `GlobalKTable`, а когда `KTable`? `GlobalKTable` особенно удобно использовать для распространения информации по всем экземплярам Kafka Streams с целью использования ее в соединениях. Например, представьте, что у нас есть поток покупок с идентификатором пользователя. С помощью последовательности символов и цифр, представляющих человека, совершившего покупку, можно извлечь кое-какие сведения.

Но если у нас будет возможность добавить имя, адрес, возраст, род занятий и т. д., то мы получим больше информации о событиях. Поскольку информация о пользователях меняется нечасто (то есть люди не меняют работу или адрес еженедельно), `GlobalKTable` хорошо подходит для распространения достаточно статичных данных. Каждая таблица имеет полную копию данных, поэтому она лучше всего подходит для обогащения потока событий.

Еще одно преимущество `GlobalKTable`, обусловленное потреблением всех разделов исходного топика, заключается в том, что при выполнении соединения с `KStream` ключи не обязательно должны совпадать. Для создания соединения можно использовать значение из потока. Взгляните на рис. 8.14, чтобы понять, как это работает.

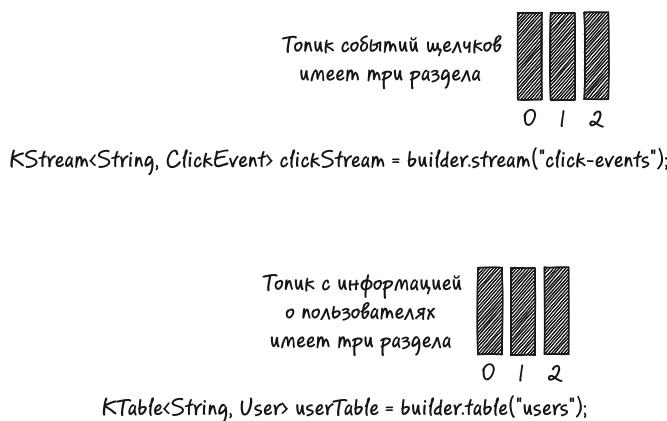


**Рис. 8.14.** Поскольку `GlobalKTable` материализует все разделы исходного топика, для создания соединения можно использовать значение из потока

Поскольку поток содержит значения, соответствующие ключам таблицы, а таблица содержит данные из всех разделов, вы можете извлечь информацию, необходимую для создания соединения, из значения таблицы. Как это реализовать и обогатить поток, я расскажу в подразделе 8.6.3, где мы рассмотрим соединение `KStream–GlobalKTable`.

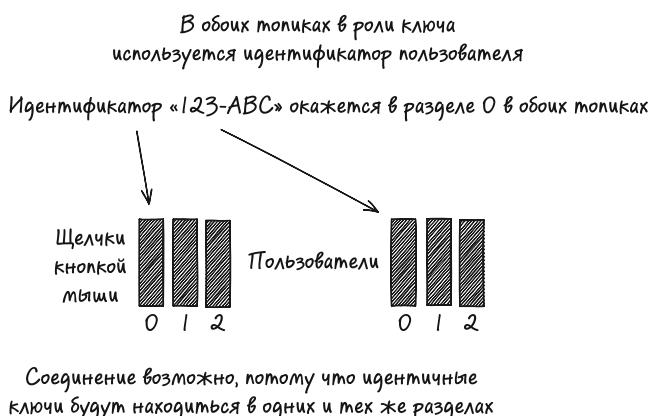
## 8.6. СОЕДИНЕНИЕ С ТАБЛИЦАМИ

В предыдущей главе мы познакомились с операцией соединения двух объектов KStream, однако Kafka Streams поддерживает также соединения KStream — KTable, KStream — GlobalKTable и KTable — KTable. Но зачем может понадобиться вычислять соединение между потоком и таблицей? Соединения «поток — таблица» дают прекрасную возможность обогащения событий дополнительной информацией. Чтобы иметь возможность соединения «поток — таблица» и «таблица — таблица», обе стороны должны иметь совместимое секционирование, то есть базовые исходные топики должны иметь одинаковое количество разделов. Схема на рис. 8.15 поможет вам понять, как выглядит совместимое секционирование на уровне топиков.



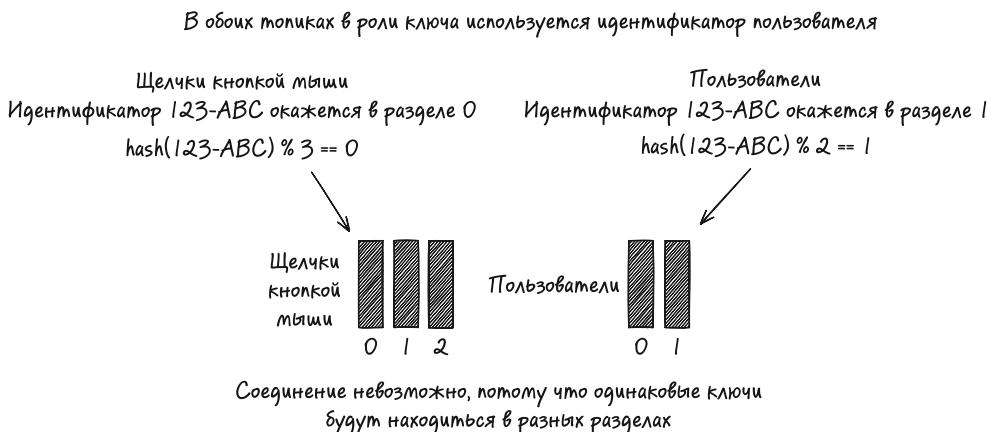
**Рис. 8.15.** Топики с совместимым секционированием имеют одинаковое количество разделов, поэтому экземпляры KStream и KTable будут иметь задачи, работающие с одним и тем же разделом

Как видите, это совершенно разные топики, просто имеющие одинаковое количество разделов. Теперь рассмотрим пару иллюстраций, чтобы понять, зачем может понадобиться выполнять соединение. Начнем с положительного примера на рис. 8.16.



**Рис. 8.16.** Топики с разным количеством разделов размещают одни и те же ключи в разных разделах

Судя по иллюстрации, оба ключа, по которым осуществляется соединение, идентичны. В результате оба попадут на раздел 0, поэтому в данном случае соединение возможно. Далее рассмотрим отрицательный случай на рис. 8.17.



**Рис. 8.17.** Топики с разным количеством разделов размещают одни и те же ключи в разных разделах

Даже притом что ключи идентичны, из-за разного количества разделов записи с одинаковыми ключами окажутся в разных разделах, то есть операция соединения окажется невозможной. Однако это не означает, что ничего нельзя сделать, чтобы обеспечить возможность соединения. Если у вас есть экземпляры `KStream` и `KTable`, для которых нужно выполнить соединение, но они имеют несовместимое секционирование, то вам нужно выполнить операцию перераспределения. Мы рассмотрим пример, как это сделать, в подразделе 8.6.1. Обратите внимание, что, поскольку `GlobalKTable` имеет полную копию записей, для вычисления соединения между потоком и глобальной таблицей не требуется иметь совместимое секционирование.

Познакомившись с требованиями к секционированию, рассмотрим конкретный пример, почему может понадобиться выполнить соединение потока с таблицей. Представим, что у нас есть поток событий щелчков кнопкой мыши, выполненных пользователем на сайте, а также таблица текущих пользователей, выполнивших вход. Объект события щелчка кнопкой мыши содержит только идентификатор пользователя и ссылку на страницу, куда выполнялся переход, но нам хотелось бы иметь дополнительную информацию. Для этого можно выполнить соединение потока щелчков кнопкой мыши с таблицей пользователей, и мы сможем в режиме реального времени получить гораздо больше полезной информации о закономерностях в действиях пользователей нашего сайта. Пример реализации такого соединения показан в листинге 8.7.

**Листинг 8.7.** Соединение «поток — таблица» для обогащения потока событий дополнительной информацией

```
KStream<String, ClickEvent> clickEventKStream =
    builder.stream("click-events",
        Consumed.with(stringSerde, clickEventSerde));

KTable<String, User> userTable =
    builder.table("users",
        Consumed.with(stringSerde, userSerde));

clickEventKStream.join(userTable, clickEventJoiner)
    .peek(PrintKV("stream-table-join"))
    .to("stream-table-join",
        Produced.with(stringSerde, stringSerde));
```

Код в этом примере создает поток событий щелчков кнопкой мыши и таблицу пользователей, выполнивших вход. В этом примере предполагается, что в качестве ключа в потоке и первичного ключа в таблице используется идентификатор пользователя, поэтому мы можем выполнить соединение между ними без лишних ухищрений. Затем вызывается метод `join` потока, которому в первом параметре передается таблица.

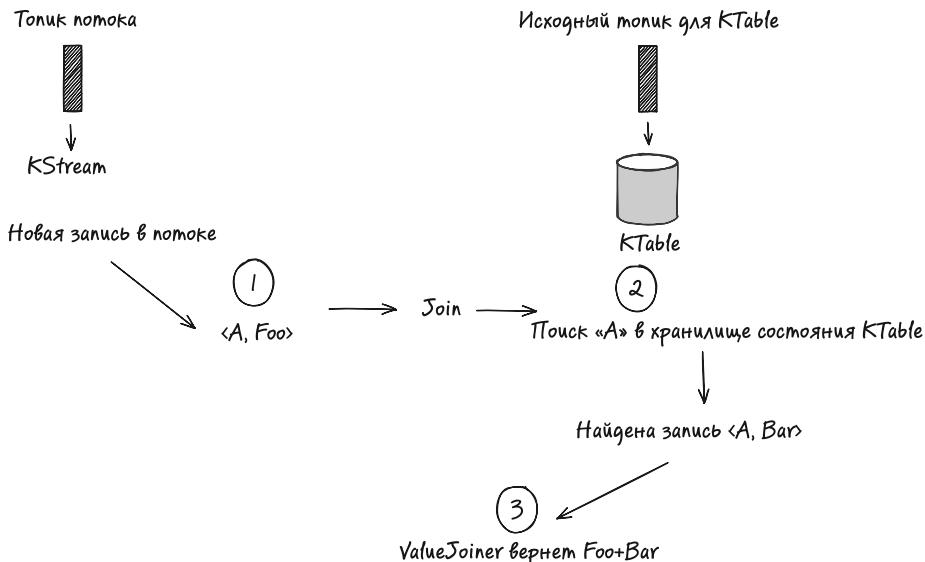
### 8.6.1. Соединение потоков и таблиц

Теперь я хотел бы представить некоторые различия между соединениями «поток — таблица» и «поток — поток», с последним из которых вы познакомились в предыдущей главе. Соединения «поток — таблица» не являются симметричными: поток всегда должен находиться слева, называющей стороне, а таблица — всегда справа.

Соединения потоков и таблиц не являются оконными. Когда на стороне потока появляется новая запись, Kafka Streams выполняет поиск ключа в таблице справа. Проверка отметок времени для обеих сторон не выполняется, если только не используются хранилища состояний с версиями, о которых мы поговорим в следующем разделе.

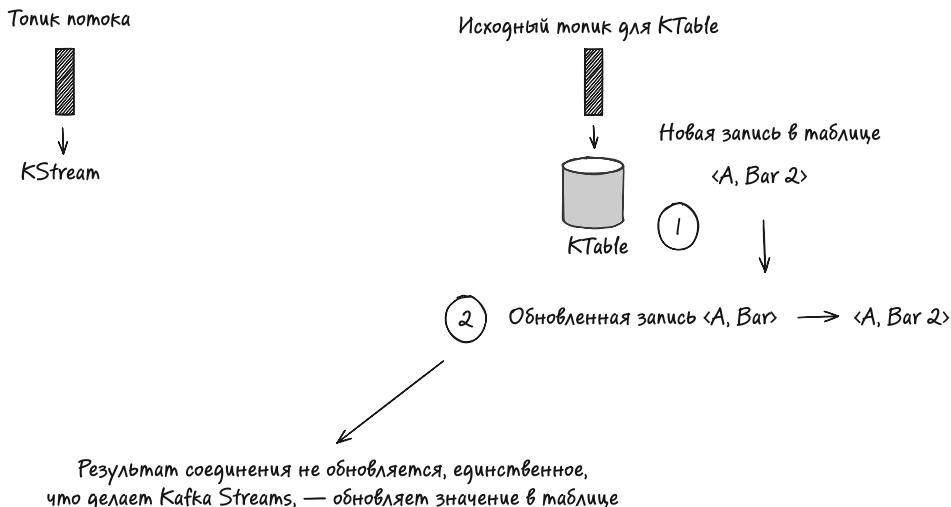
Чтобы захватить результат соединения, необходимо передать объект `ValueJoiner`, который принимает значения с обеих сторон и создает новое значение, которое может быть того же типа, что и значения с любой из сторон, или совершенно нового. Соединения «поток — таблица» могут быть внутренними (как показано в этом примере) или внешним левым.

Соединение вычисляется только для вновь поступающих записей в потоке, обновления записей в таблице просто обновляют значения ключей, но не выдают новых результатов соединения. Рассмотрим пару иллюстраций, которые помогут понять, что это значит. Схема на рис. 8.18 иллюстрирует появление новой записи в `KStream`.



**Рис. 8.18.** Соединение «поток — таблица» генерирует результат, только когда обновление случается на стороне потока

Когда поступит новая запись, Kafka Streams выполнит поиск ключа в KTable и применит логику ValueJoiner, чтобы получить результат соединения. Теперь рассмотрим случай, когда обновление получает KTable (рис. 8.19).

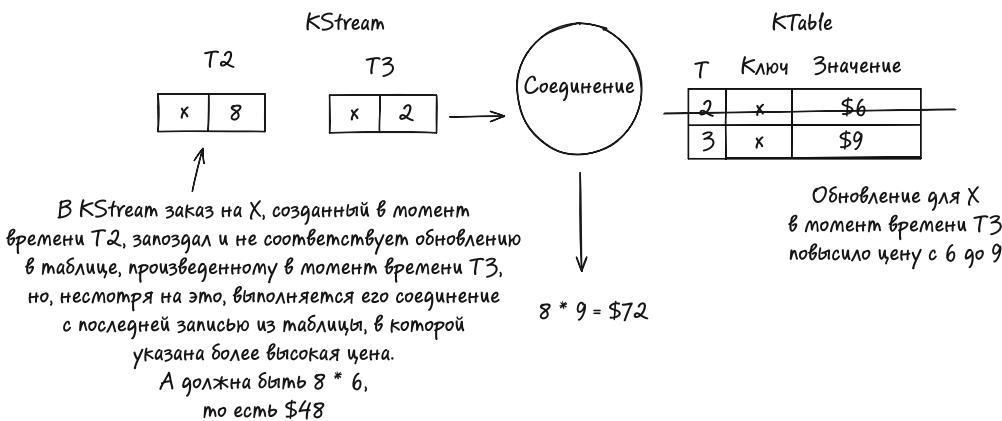


**Рис. 8.19.** При обновлении таблицы результат соединения не обновляется, обновляется только сама таблица

Как показано здесь, когда объект `KTable` получает новую запись, он просто обновляет соответствующий ключ, но не вызывает никаких действий по вычислению соединения. В этом примере соединения «поток — таблица» время не учитывается, так как `KTable` хранит относительно статичные данные о пользователях. Но в ситуациях, когда время или времененная семантика имеют большое значение, этот аспект нужно учитывать.

## 8.6.2. Версионированные таблицы KTable

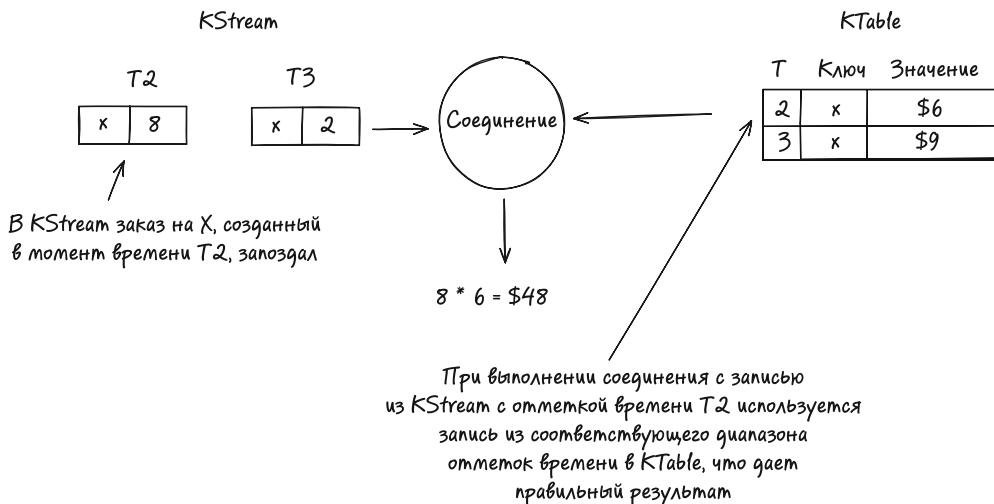
На рис. 8.20 показан случай реализации соединений «поток — таблица» с ограничением по времени.



**Рис. 8.20.** Обновление KTable и соединение с запоздавшей записью KStream приводит к ошибочному результату

На рис. 8.20 показан `KStream` с заказом или товаром, стоимость которого постоянно меняется, и таблица `KTable`, содержащая цены. Когда пользователь размещал заказ в момент времени `T2`, цена была установлена на уровне `$6`, а в момент времени `T3` цена была увеличена до `$9`. Но запись в `KStream`, размещенная в момент времени `T2`, запоздала. В результате происходит соединение с заказом,енным в момент времени `T2`, с ценой, обновившейся в момент времени `T3`, то есть клиент будет вынужден заплатить  $8 * \$9 = 72$  вместо ожидаемых  $8 * \$6 = 48$ .

Такой результат обусловлен тем, что, когда в `KTable` поступает новая запись, Kafka Streams автоматически применяет обновление к таблице и любое последующее соединение с записью в `KStream` будет использовать текущую соответствующую запись в таблице. Нам нужно предотвратить проблему ошибочного определения стоимости таких «запоздавших» заказов, добавив в соединения поддержку семантики времени, чтобы обеспечить учет отметки времени в записи на стороне потока, и сохранив в `KTable` значение, соответствующее предыдущей отметке времени. На рис. 8.21 проиллюстрирована эта идея.



**Рис. 8.21.** Таблица с историческими записями, содержащими отметки времени, позволяет выполнять корректные во времени соединения с записями в KStream

Здесь показано, что, даже если заказ запаздывает, соединение все равно будет использовать правильную цену из таблицы, соответствующую моменту времени **T2**, что даст ожидаемый и верный результат. Но как добавить поддержку учета времени обновлений записей в **KTable**? Для этого достаточно использовать *версионированное* хранилище состояний, но перед этим нужно создать версионированный экземпляр **StoreSupplier**, как показано в листинге 8.8.

#### Листинг 8.8. Создание версионированного хранилища состояния

```
KeyValueBytesStoreSupplier versionedStoreSupplier =
    Stores.persistentVersionedKeyValueStore(
        "user-details-table",
        Duration.ofSeconds(30));
```

Имя хранилища состояния

Продолжительность времени, в течение которого старые записи доступны для объединения

Обратите внимание на второй параметр. Это объект **Duration**, который определяет, как долго должны оставаться доступными старые записи. Существует перегруженная версия метода, которая принимает объект **Duration** и определяет размеры сегментов для хранения старых записей.

Далее нужно подключить **StoreSupplier** к **KTable**, как показано в листинге 8.9.

#### Листинг 8.9. Подключение StoreSupplier к KTable

```
KTable<String, User> userTable =
    builder.table(rightInputTableTopic,           ← Метод StreamBuilder.table создает таблицу
        Consumed.with(stringSerde, userSerde),
        Materialized.as(versionedStoreSupplier));  ← Объект Consumed для передачи экземпляров Serde
                                                    ← Подключение версионированного экземпляра StoreSupplier к KTable
```

Этими двумя шагами мы подключили версионированное хранилище состояния к KTable, что позволит выполнять соединения, корректные во времени.

### 8.6.3. Соединение потоков и глобальных таблиц

Главная особенность примера в предыдущем разделе: наличие ключей в потоке событий щелчков кнопкой мыши. Но, как вы наверняка помните, ключи в топике могут быть пустыми, соответственно, они будут пустыми и в потоке **KStream**. Как в таких случаях выполнить соединение с таблицей? Конечно, можно провести перераспределение, как было показано в предыдущих главах, но, может быть, есть более простой способ? Для решения этой проблемы продолжим исследование соединения **KStream — GlobalKTable**.

Во-первых, это вторая разновидность соединений в Kafka Streams (другим считается соединение внешнего ключа **KTable**). Оно не требует совместимости секционирования. Напомню, что **GlobalKTable** не секционируется как **KTable**, а содержит все данные из своего исходного топика. В результате глобальные таблицы поддерживают соединения с потоками, ключи в которых не совпадают с ключом глобальной таблицы или вообще не существуют. Теперь посмотрим, как использовать эту информацию в практическом приложении.

Представьте, что мы работаем на производстве, где для управления процессами используются датчики IoT, измеряющие температуру и расстояние. Информация с датчиков собирается и выводится в топик Kafka. В период первоначальной наладки сервиса инженеры по работе с данными не использовали никаких ключей. Позднее они создали дополнительный топик с метаданными для различных датчиков. Роль ключей в этом топике играют идентификаторы датчиков, а роль значений — метаданные.

Нам нужно включить метаданные в показания каждого датчика, обработанные нашим приложением Kafka Streams. Такая ситуация, когда записи с показаниями датчиков не имеют ключей, будто специально создана для соединения потока и глобальной таблицы. Сначала рассмотрим настройку соединения потока и глобальной таблицы в Kafka Streams, а затем подробно исследуем каждый компонент (листинг 8.10).

#### Листинг 8.10. Пример соединения **KStream — GlobalKTable**

```
sensorKStream.join(sensorInfoGlobalKTable,           ← Таблица GlobalKTable, участвующая
                    sensorIdExtractor,          ← в соединении
                    sensorValueJoiner);       ← Селектор ключа для соединения
                                            ← Экземпляр ValueJoiner,
                                            ← вычисляющий результат
```

В случае соединения **KStream — GlobalKTable** во втором параметре методу **join** передается **KeyValueMapper**, который принимает ключ и значение потока и создает ключ, используемый для соединения с глобальной таблицей (что очень похоже на соединение внешнего ключа **KTable**). Результат соединения получит ключ потока (который может быть пустым) независимо от ключа **GlobalKTable** или того, что возвращает предоставленная функция. На рис. 8.22 показан селектор ключа, объясняющий его роль.



**Рис. 8.22.** Селектор ключа выбирает нужное поле и возвращает его значение для использования в качестве ключа в соединении

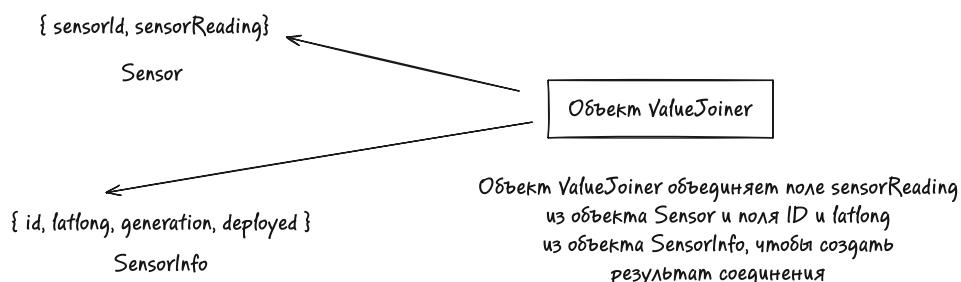
Как показывает эта иллюстрация, селектор ключа знает, какое поле из значения нужно извлечь, чтобы использовать его в качестве ключа для соединения. Но имейте в виду, что для создания ключа селектор может использовать всю информацию, доступную в записи, а не только одно какое-то поле. Теперь взглянем на код (листинг 8.11).

#### Листинг 8.11. Интерфейс KeyValueMapper

```
KeyValueMapper<String, Sensor, String> sensorIdExtractor = // Реализация возвращает идентификатор датчика
    (key, value) -> value.getId(); // Объявляет KeyValueMapper как функциональный интерфейс
```

На всякий случай напомню, что `KeyValueMapper` принимает два параметра, ключ и значение, и возвращает один объект, в данном случае строку с идентификатором датчика. Kafka Streams использует этот идентификатор для поиска соответствующей записи в глобальной таблице.

Теперь перейдем к `ValueJoiner`. Сначала взгляните на рис. 8.23, иллюстрирующий основную идею.



**Рис. 8.23.** ValueJoiner использует объекты, участвующие в соединении, чтобы создать новый объект с результатом

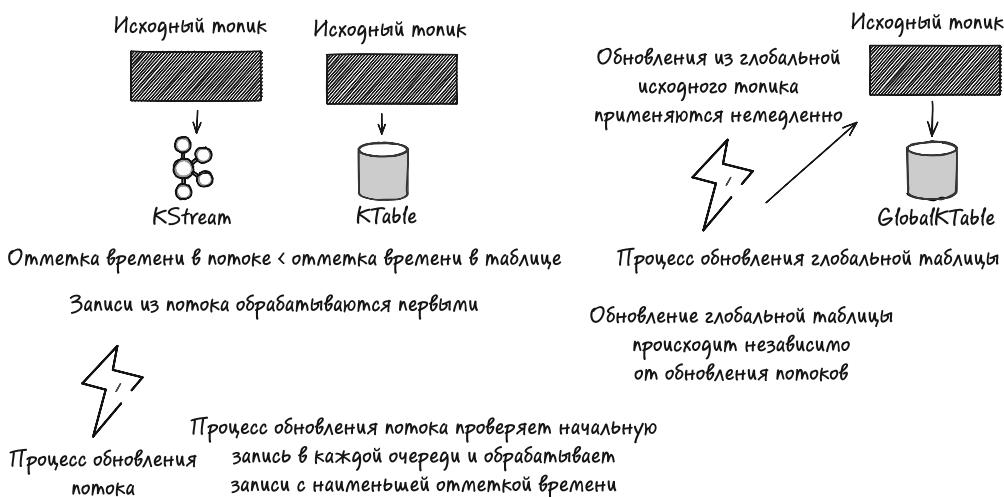
Как показано на этой иллюстрации, объект `ValueJoiner` принимает две записи с одним и тем же ключом и объединяет их в третий объект — результат соединения. В листинге 8.12 показана реализация `ValueJoiner`, возвращающая результат соединения с информацией о датчике.

**Листинг 8.12.** Реализация ValueJoiner

```
ValueJoiner<Sensor, SensorInfo, String> sensorValueJoiner = Объявление ValueJoiner  
в форме лямбда-функции
    (sensor, sensorInfo) -> String.format("Sensor %s  
located at %s  
had reading %s",  
        sensorInfo.getId(),  
        sensorInfo.getLatitude(),  
        sensor.getReading()); Создает строку с показанием  
и идентификатором датчика
```

Здесь можно увидеть, какую роль играет `ValueJoiner`: он объединяет идентификатор датчика, его местоположение и показание. Итак, мы благополучно реализовали соединение записей без ключа из `KStream` с таблицей `GlobalKTable` и в результате дополнели показания датчика нужной нам информацией.

Семантика соединений с глобальной таблицей имеет свои отличия. Kafka Streams обрабатывает входные записи для `KTable` вместе с другими входными записями с учетом их отметок времени, благодаря чему при соединении потока и таблицы происходит синхронизация по времени. Но когда соединение выполняется с `GlobalKTable`, Kafka Streams применяет обновления в глобальной таблице, как только они становятся доступны (рис. 8.24).



**Рис. 8.24.** В Kafka Streams имеется отдельный поток выполнения, обновляющий таблицы `GlobalKTable`. Обновления применяются за пределами процесса обработки входных записей

Обновление `GlobalKTable` производится независимо от других компонентов приложения Kafka Streams с использованием отдельного потока выполнения, специально предназначенного для обновления любых глобальных хранилищ или таблиц. По этой причине входные записи для `GlobalKTable` применяются немедленно, без учета отметок времени в них.

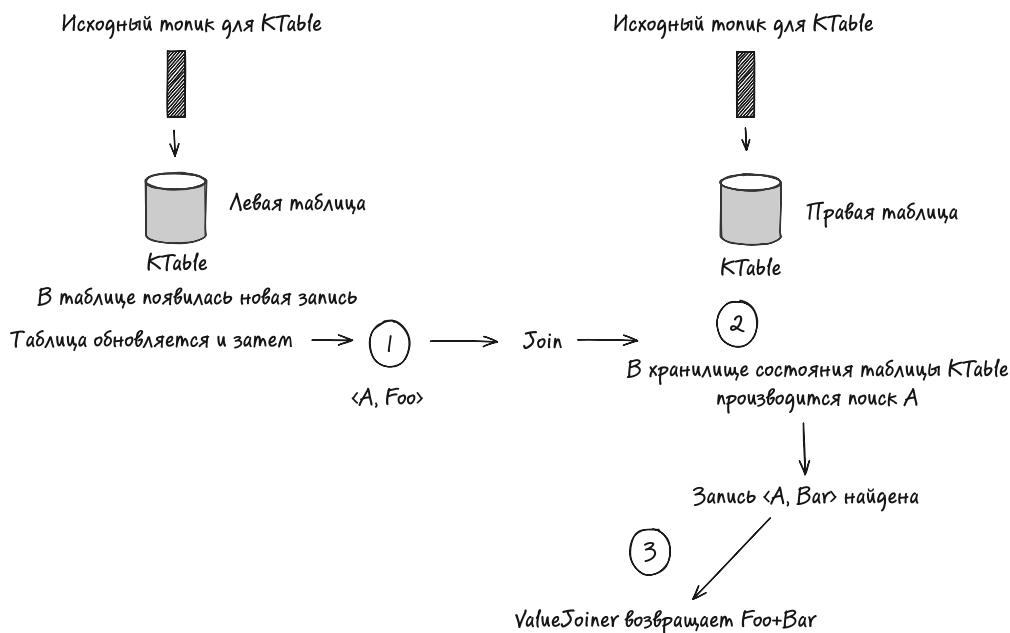
Конечно, каждое решение подразумевает некоторый компромисс. Использование `GlobalKTable` означает потребность в большем объеме дискового пространства и оказывает более существенную нагрузку на брокер, потому что данные

не секционируются. Соединение потока и глобальной таблицы не является симметричной операцией: `KStream` всегда находится на вызывающей или левой стороне соединения. Кроме того, результаты соединения создаются только для обновлений в потоке, появление записи для `GlobalKTable` просто обновляет внутреннее состояние таблицы. Наконец, доступны либо внутренние, либо внешние левые соединения.

Итак, какие соединения предпочтительнее: `KStream` с `KTable` или `KStream` с `GlobalKTable`? Это сложный вопрос, так как правила дпускают значительную гибкость. Но в общем случае рекомендуется использовать `GlobalKTable` в случаях, когда данные, присоединяемые к записям из потока, относительно статичны. Если присоединяемые табличные данные имеют большой объем, то лучше использовать `KTable`, чтобы библиотека Kafka Streams могла распределить их по нескольким экземплярам.

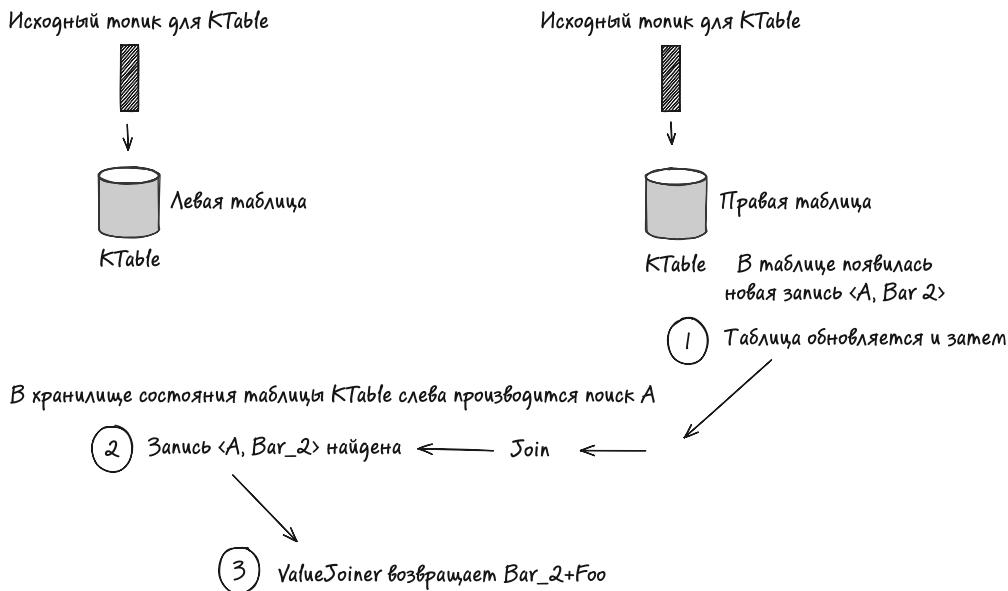
#### 8.6.4. Соединение двух таблиц

Далее поговорим о соединениях «таблица — таблица». Соединения двух таблиц похожи на соединения с участием потоков. Чтобы осуществить такое соединение, нужно предоставить экземпляр `ValueJoiner`, вычисляющий результаты соединения. Но дополнительно к соединениям двух таблиц применяется ограничение: исходные топики для обеих сторон должны иметь одинаковое количество разделов. Соединение двух таблиц похоже на соединение «поток — поток», за исключением того, что не поддерживает оконную обработку и вычисление результатов соединения будет выполнено при обновлении любой из сторон. Таким образом, когда таблица слева получит новую запись, события будут развиваться примерно так, как показано на рис. 8.25.



**Рис. 8.25.** Таблица слева обновляется, и запускается расчет нового результата соединения

Поступление новой записи в таблицу слева приводит к ее обновлению и запуску расчета соединения. Точно такой же процесс происходит, когда обновление получает таблица справа (рис. 8.26) — когда правая таблица, участвующая в соединении, обновляется новой записью, выполняются те же самые шаги.



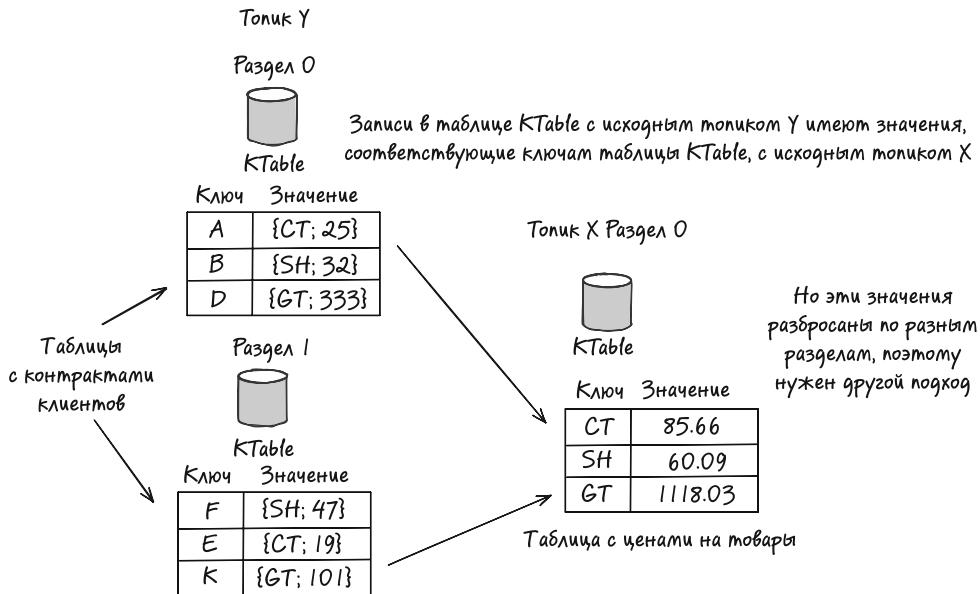
**Рис. 8.26.** Изменения в таблице справа тоже могут инициировать вычисление результатов соединения

Но соединения «таблица — таблица» имеют свои ограничения, которые мы рассмотрим еще на одном примере. Представьте, что мы работаем в фирме, торгующей товарами, и используем **KTable**, которая отслеживает последние предложения по контрактам для данного клиента. В другой таблице **KTable** содержатся последние цены на данный набор товаров. Нам нужно соединить эти две таблицы, чтобы получить запись с информацией о текущем контракте с клиентом и ценой товара.

Но мы должны решить одну проблему: в качестве первичного ключа в таблице клиентских контрактов используется идентификатор клиента, а в таблице товаров — код товара. То есть прямое соединение таблиц с разными первичными ключами невозможно (рис. 8.27).

Как видите, значения в одной таблице имеют поле с ключом в другой таблице. Но сложность в том, что контракты разбросаны по разным разделам, а коды товаров, первичный ключ для таблицы, отображаются точно в один раздел.

До сих пор все примеры соединений, которые вы видели, включали таблицы с одинаковым первичным ключом. Однако если значение одной таблицы содержит поле, соответствующее первичному ключу другой таблицы, то соединение вполне возможно. Такие соединения называются соединениями по внешнему ключу.



**Рис. 8.27.** В качестве первичного ключа в таблице клиентских контрактов используется идентификатор клиента, а код товара, соответствующий ключу в другой таблице, содержится в значении

Это именно наш случай, потому что значение таблицы контрактов клиента содержит коды товаров. KTable поддерживает возможность соединения по внешнему ключу для таких таблиц, как наши таблицы контрактов и товаров. Реализация соединения по внешнему ключу выглядит просто: нужно вызвать метод `KTable.join`, как показано в листинге 8.13.

#### Листинг 8.13. Соединение по внешнему ключу двух таблиц KTable

```
clientContractTable.join(commodityPriceTable,           ← Другая таблица или правая сторона соединения
                        foreignKeyExtractor,          ← Функция извлечения внешнего ключа
                        joiner);                  ← Параметр ValueJoiner
```

Соединение по внешнему ключу реализуется так же, как любое другое соединение «таблица — таблица», за исключением того, что нужно передать дополнительный параметр — объект `java.util.Function`, который извлекает ключ, необходимый для вычисления соединения. В частности, функция извлекает ключ из значения слева, после чего этот ключ сопоставляется с ключом таблицы справа.

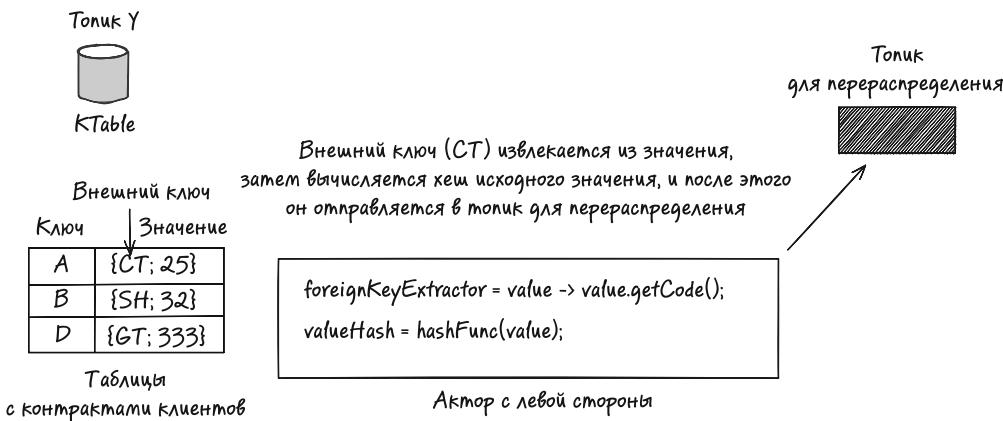
Если функция возвращает `null`, то соединение не происходит. Дополнительное замечание о соединениях по внешнему ключу: поскольку соединение выполняется со значением таблицы справа, задействованные таблицы не должны иметь совместимое секционирование.

Использование внешнего ключа поддерживают и внутренние, и внешние левые соединения. Как и в случае соединений по первичному ключу, обновление любой

из сторон вызовет повторный расчет соединения. Но есть некоторые дополнительные тонкости, которые следует обсудить, чтобы понять, как работает соединение по внешнему ключу, поэтому рассмотрим несколько иллюстраций, которые помогут нам понять происходящее.

Операции с контрактами производятся быстро и часто, и иногда в таблицу контрактов вносятся обновления, поэтому нам также необходим механизм, гарантирующий, что не будут появляться соединения с ошибочными результатами.

Наконец, мы должны учесть природу отношения «многие к одному» из таблицы справа к таблице слева. Значению записи в левой таблице соответствует ровно одна запись справа, но ключу с правой стороны будет соответствовать несколько значений слева, поскольку он отображается в значения, а не в первичный ключ. Рассмотрим первый шаг — извлечение ключа из значения таблицы слева (рис. 8.28).



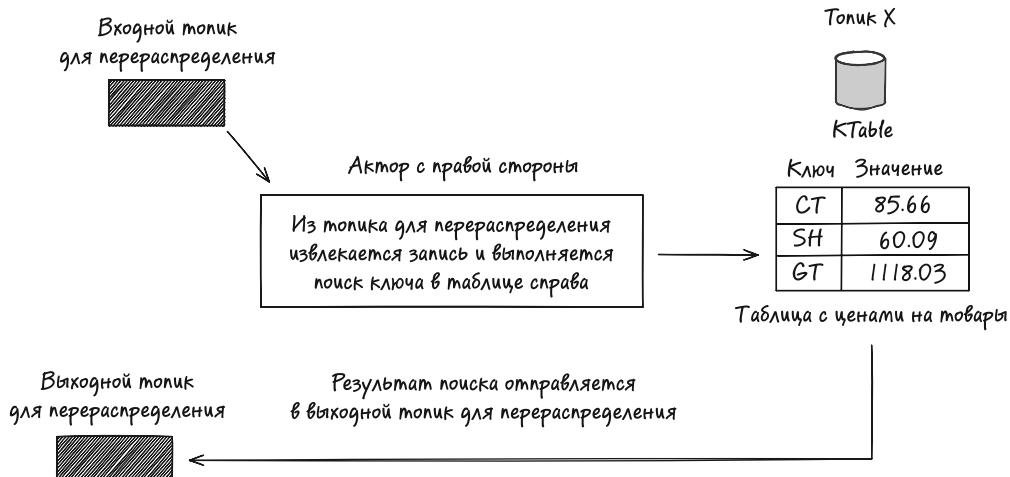
**Рис. 8.28.** Извлечение внешнего ключа из значения, вычисление хеша исходного значения и отправка их в топик для перераспределения

Метод `join` извлекает ключ из значения в левой таблице с помощью переданной ему функции. Затем он вычисляет хеш значения, который мы скоро увидим, когда он вступит в игру.

Далее в топик для перераспределения передается запись, в которой ключом служит результат вызова предоставленной функции, а значение содержит ключ таблицы слева и хеш значения. Этот топик для перераспределения имеет секционирование, совместимое с правой таблицей.

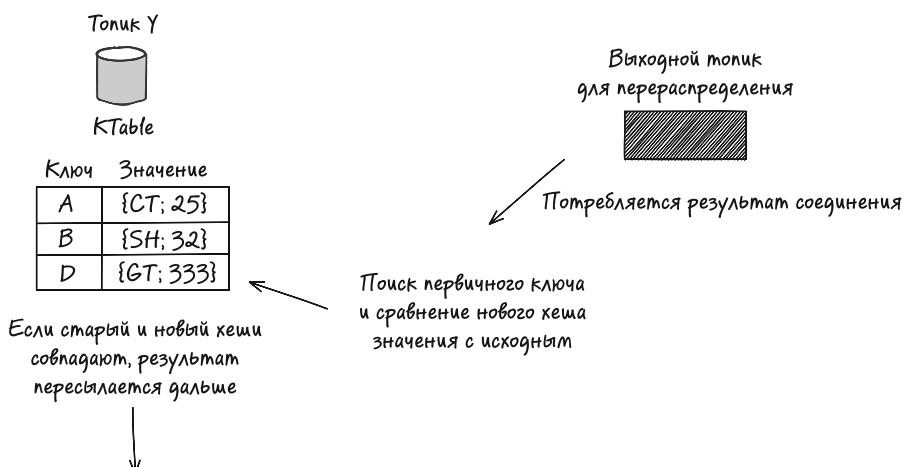
Теперь посмотрим, что происходит после того, как левая сторона отправляет запись в топик для перераспределения (рис. 8.29).

Код, действующий с правой стороны соединения, можно представить как актора или агента. Этот актор потребляет записи из входного топика для перераспределения и использует результат для выполнения поиска по ключу в правой таблице. Он также материализует ключ и хеш в хранилище состояний для использования при обновлении правой таблицы на следующем шаге.



**Рис. 8.29.** Правая сторона соединения использует извлеченный ключ и отыскивает соответствующий ключ в правой таблице, после чего отправляет результат обратно в топик для перераспределения

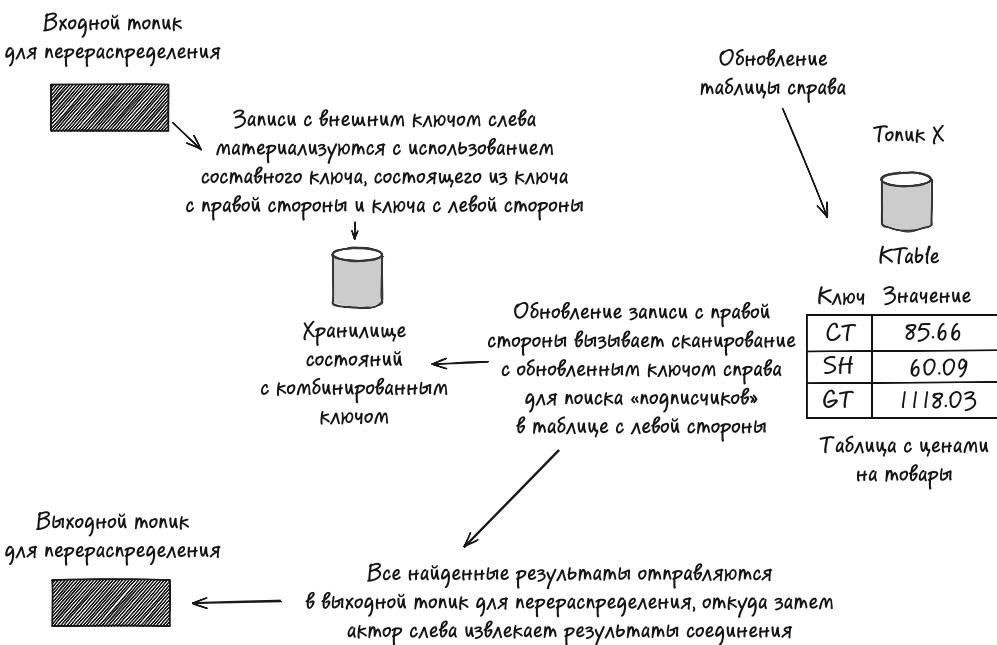
После получения результата поиска (и отбрасывания пустых результатов) агент справа публикует еще в одном топике для перераспределения запись, включающую хеш исходного значения. Обратите внимание, что ключ для исходящих результатов — это исходный ключ в левой таблице. По аналогии с правой стороной код, работающий с левой стороной соединения, тоже можно рассматривать как агент (рис. 8.30).



**Рис. 8.30.** Агент с левой стороны потребляет данные из второго топика для перераспределения, выполняет поиск по исходному ключу в таблице слева и сравнивает хеш исходного значения с оригиналом

Актор с левой стороны потребляет результат поиска из выходного топика для перераспределения, отыскивает соответствующий ключ в таблице слева и вычисляет новый хеш значения найденной записи. Если он совпадает с исходным хешем, то результат соединения пересыпается дальше, а если не совпадает, то результат просто отбрасывается, потому что несовпадение означает, что значение изменилось и мог измениться внешний ключ, а значит, результат недействительный.

К настоящему моменту мы рассмотрели соединение по внешнему ключу, когда появление новой или обновление существующей записи происходит в таблице слева. Теперь рассмотрим процесс, когда обновление происходит в таблице справа. Выполняемые шаги отличаются, потому что нужно учесть тот факт, что одна запись в таблице справа может соответствовать нескольким записям слева (рис. 8.31).



**Рис. 8.31.** После обновления правой стороны актор сканирует хранилище состояний с составными ключами и отправляет все найденные совпадения в выходной топик для перераспределения

Когда появление новой или обновление существующей записи происходит в таблице справа, производится сканирование материализованной таблицы с сопоставлением любых ранее отправленных внешних ключей при обновлении таблицы с левой стороны. Затем актор с правой стороны отправляет все результаты в выходной топик для перераспределения результатов, а актор с левой стороны использует их, следуя тому же алгоритму обработки результатов одиночного соединения с таблицей справа.

### ПРИМЕЧАНИЕ

В `KStream` нет возможности выполнить похожее соединение по внешнему ключу, и это сделано намеренно. В `KStream` имеются методы `map` и `selectKey`, благодаря которым легко можно изменить ключ потока и упростить реализацию соединения.

К данному моменту мы рассмотрели различные виды соединений, доступные в `KTable` и `GlobalKTable`. Однако на этом знакомство с таблицами не заканчивается. Нам еще предстоит рассмотреть возможность исследования содержимого таблиц с помощью интерактивных запросов и подавления вывода из таблицы (только для `KTable`) с целью получения единственного конечного результата. Интерактивные запросы мы рассмотрим в главе 13, а о подавлении вывода поговорим в следующей главе, когда будем обсуждать оконные операции.

## ИТОГИ ГЛАВЫ

- `KTable` — это поток обновлений, имитирующий таблицу базы данных, в которой роль первичного ключа играет ключ в парах «ключ — значение» в потоке. Все последующие записи с одинаковым ключом считаются обновлениями предшествующих им записей с тем же ключом. Агрегирование в `KTable` действует аналогично SQL-запросу `Select ... GroupBy` к таблице в реляционной базе данных.
- Выполнение соединений с `KStream` с `KTable` — отличный способ обогатить поток событий. `KStream` содержит данные событий, а `KTable` — факты или характеристики данных.
- Имеется возможность выполнить соединение двух таблиц `KTable`, а также соединение двух таблиц `KTable` по внешнему ключу.
- `GlobalKTable` содержит все записи из входного топика и не разбивает их на разделы. Благодаря этому каждый экземпляр приложения имеет все записи, что позволяет использовать глобальную таблицу в качестве справочника. Соединения с `GlobalKTable` не требуют совместимого секционирования с `KStream` и поддерживают передачу функции, которая вычисляет ключ для соединения.



# *Оконные операции и отметки времени*

## **В этой главе**

- ✓ Роль оконных операций и различных типов.
- ✓ Обработка записей, следующих не по порядку.
- ✓ Подавление промежуточных результатов.
- ✓ Важность отметок времени.

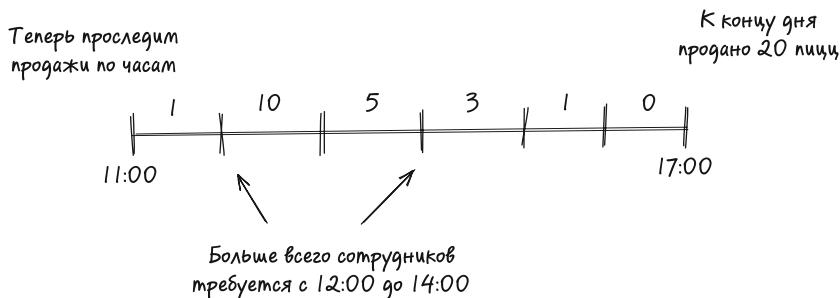
В предыдущих главах вы узнали, как выполнять операции агрегирования с `KStream` и `KTable`. В этой главе, опираясь на ранее полученные знания, вы будете учиться применять эти операции для уточнения ответов, возвращаемых операциями агрегирования, используя для этого окна. Окна, или оконные операции, позволяют вычислять агрегированные данные в рамках дискретных временных интервалов. В этой главе вы научитесь применять оконные операции в конкретных ситуациях для достижения конкретных целей.

Оконные операции чрезвычайно важны для практического применения, потому что в противном случае агрегаты будут продолжать расти с течением времени и извлечение полезной информации станет затруднительным, особенно если в нашем распоряжении имеется лишь гигантский набор фактов без особого контекста. В качестве простого примера представим, что мы отвечаем за комплектование персонала пиццерии в студенческом городке нашего университета (рис. 9.1). Пиццерия работает с 11:00 до 17:00. Общий объем продаж обычно составляет 20 пиц (это наш агрегат).



**Рис. 9.1.** Простой взгляд на большую совокупность данных не дает полной картины

Определить оптимальное число сотрудников можно, только имея дополнительную информацию. Знания только общего количества пицц, проданных за день, явно недостаточно, и нам остается только догадываться, что при объеме продаж 20 пицц за 6 часов мы продаем примерно три пиццы в час (рис. 9.2), с чем легко справляются два студента-работника, но так ли это на самом деле?

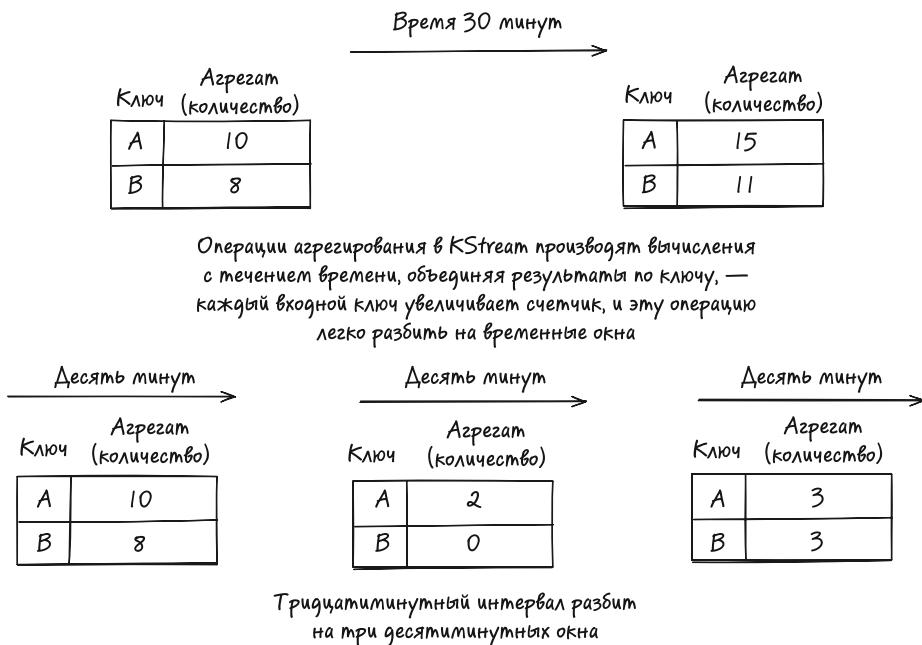


**Рис. 9.2.** Анализ продаж по часам (с разбиением на окна вручную) дает более глубокое понимание ситуации

По этим грубым прикидкам мы можем сказать, что от половины до трех четвертей продаж (от 10 до 15 пицц) приходятся на период между 12:00 и 13:00, то есть наблюдается классический обеденный час пик. Рассматривая продажи по часам (этакая форма оконной обработки!), мы более явственно видим потребности в сотрудниках. Это вымышленный пример, но он позволяет увидеть, как разбиение на окна может помочь получить более полное понимание поведения, стоящего за агрегированием.

Теперь, прежде чем продолжить, я должен рассказать о некоторых особенностях оконной обработки. Как мы знаем, агрегирование поддерживает `KStream`, и `KTable`, но разбиение на окна доступно только в `KStream`. Это обусловлено природой обеих абстракций.

Напомню, что `KStream` — это поток событий, в котором записи с одинаковым ключом считаются независимыми событиями. Рисунок 9.3 дает представление об агрегировании потоков событий.



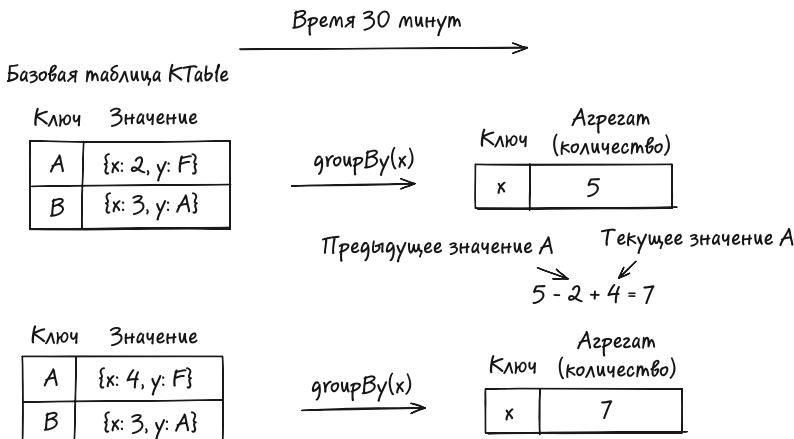
**Рис. 9.3.** Операции агрегирования в KStream выполняют группировку данных по времени, что отлично подходит для оконного агрегирования

Судя по этой иллюстрации, агрегирование входных записей KStream с тем же ключом Kafka Streams продолжит добавлять их, конструируя более крупное целое. Это группировка по временной шкале, и она отлично подходит для оконного агрегирования. Как показывает иллюстрация, мы можем взять агрегат, собранный за 30 минут, разбить его на три десятиминутных окна и получить более подробные сведения об активности за каждые десять минут.

В KTable агрегирование работает немного иначе. Взгляните на рис. 9.4, чтобы понять причину.

Напомню, что при работе с KTable сначала вызывается метод `groupByKey` с полем, отличным от первичного ключа. И по мере поступления новых записей в базовую таблицу Kafka Streams обновляет предыдущие записи (с тем же ключом), удаляет старые записи из агрегата и добавляет новые. Соответственно, каждый агрегат в KTable является комбинацией уникальных значений, каждое из которых представляет некоторый момент времени, поэтому нет смысла использовать разбиение на окна.

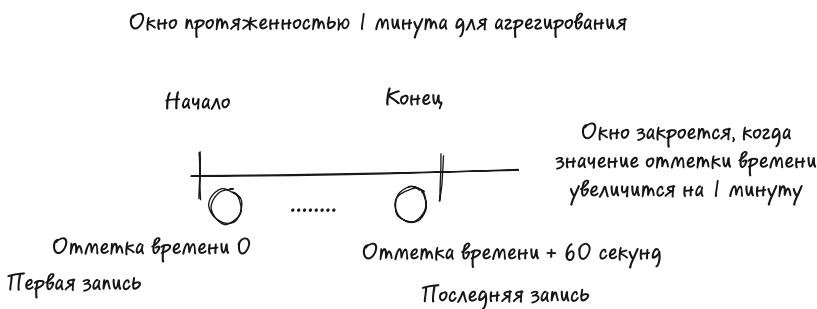
В этой главе мы также познакомимся с разными типами поддерживаемых окон. Некоторые окна имеют фиксированные временные размеры, другие гибко изменяют их, подстраиваясь, в зависимости от записей, содержащихся в потоке событий. Помимо этого, мы рассмотрим разные варианты использования и узнаем, в каких случаях каждый тип окна имеет наибольшую ценность.



Каждый агрегат представляет момент времени, а не группировку во времени

**Рис. 9.4.** Операции агрегирования в KTable выполняют группировку данных в определенный момент времени, что не позволяет выполнять оконное агрегирование

Наконец, в этой главе мы поговорим о важности отметок времени, которые управляют перемещением окон. Kafka Streams — это приложение, управляемое событиями, то есть отметки времени в записях управляют оконными операциями. Взгляните на рис. 9.5, иллюстрирующий сказанное.



Отметки времени событий управляют перемещением окна

**Рис. 9.5.** Отметки времени являются ключом и движущей силой в управлении окнами

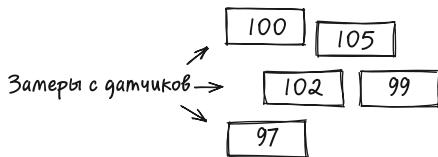
Оконные операции в Kafka Streams помнят самое большое значение отметки времени, встреченной к данному моменту, идвигают время вперед, только когда встретится отметка времени с большим значением. Такое запоминание отметок времени является ключом к открытию и закрытию окна. Сделав отметку времени центром действия, можете увидеть, как разворачивается поток событий по мере их возникновения.

## 9.1. РОЛЬ И ТИПЫ ОКОН

В этом разделе мы рассмотрим пример, который поможет понять, как применять оконные операции для получения полезной информации. Представьте, что мы работаем в промышленной компании, производящей конденсаторы потока<sup>1</sup>. Спрос на них значительно вырос после того, как доктор Браун успешно создал машину для путешествий во времени. Под давлением этого спроса в компании хотели бы выпускать как можно больше конденсаторов, но есть одна загвоздка.

При повышении температуры до некоторого критического уровня производственная линия останавливается, что приводит к задержкам. Поэтому перед нами поставили задачу — следить за температурой, и если она становится слишком высокой, то уменьшать скорость линии, чтобы избежать дорогостоящих остановок производства. Для этого мы установили температурные датчики IoT и организовали передачу их показаний в топик Kafka. Далее нам нужно разработать приложение Kafka Streams для обработки этих показаний с агрегированием, чтобы не пропустить момент, когда температура становится слишком высокой. Схема на рис. 9.6 демонстрирует то, к чему мы стремимся.

*Нам нужно приложение, следящее за показаниями температуры.  
В данном случае имеется три замера с температурой больше 100 градусов*



**Рис. 9.6.** Подсчет количества событий с высокой температурой для информирования о необходимости замедления линии

Как видите, идея довольно простая: создать агрегат, усредняющий замеры температуры и записывающий самые высокие. Мы уже рассматривали операции агрегирования в KStream в главе 7, тем не менее вспомним, как они реализуются (листинг 9.1).

### Листинг 9.1. Агрегирование для слежения за температурой, измеряемой датчиками IoT

```

KStream<String,Double> iotHeatSensorStream =
    builder.stream("heat-sensor-input",
        Consumed.with(stringSerde, doubleSerde));
iotHeatSensorStream.groupByKey()           ← Для агрегирования необходима группировка по ключу
    .aggregate(() -> new IoTSensorAggregation(tempThreshold),      ← Агрегирование
              aggregator,
              Materialized.with(stringSerde, aggregationSerde))
    .toStream()
    .to("sensor-agg-output", Produced.with(
        stringSerde, aggregationSerde));                                ← Вывод результата
                                                                    агрегирования в топик
  
```

<sup>1</sup> Вымышленное устройство. В фильме «Назад в будущее» конденсатор потока был важной частью машины времени. — Примеч. пер.

Эта простая топология Kafka Stream, выполняющая агрегирование, дает нам представление об изменении температуры в процессе производства потоковых конденсаторов. Мы уже видели реализации интерфейса Aggregator раньше, тем не менее давайте кратко рассмотрим работу этой реализации (листинг 9.2).

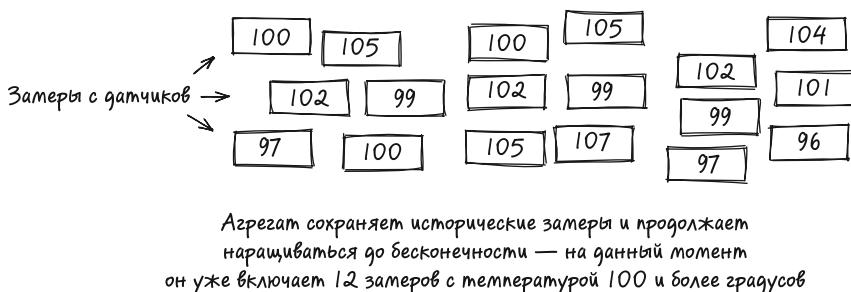
#### Листинг 9.2. Реализация Aggregator для слежения за температурой

```
public class IoTStreamingAggregator implements
    Aggregator<String, Double, IoTSensorAggregation> {
    @Override
    public IoTSensorAggregation apply(String key,
                                       Double reading,
                                       IoTSensorAggregation aggregate) {

        aggregate.temperatureSum += reading;
        aggregate.numberReadings += 1;
        if (aggregate.highestSeen < reading) {
            aggregate.highestSeen = reading;
        }
        if (reading >= aggregate.readingThreshold) {
            aggregate.tempThresholdExceededCount += 1;
        }
        aggregate.averageReading =
            aggregate.temperatureSum / aggregate.numberReadings;

        return aggregate;
    }
}
```

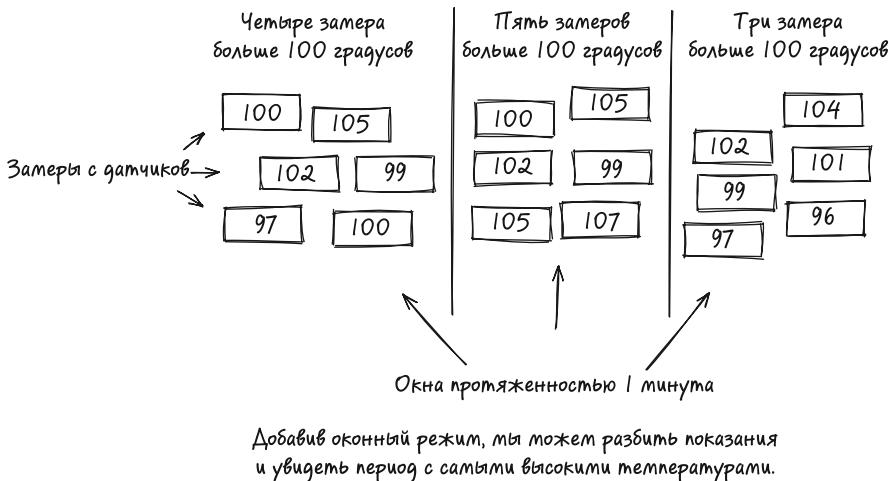
Эта реализация работает просто. Она подсчитывает текущее количество и сумму замеров для вычисления среднего значения и количества превышений установленного порога. Однако, потратив немного времени на исследование, вы быстро заметите ограничение текущего решения (рис. 9.7): агрегат продолжает наращиваться до бесконечности.



**Рис. 9.7.** Агрегат продолжает наращиваться с течением времени, сохраняя всю историческую информацию

Как видите, с течением времени становится все сложнее определить, что мы видим: скачок температуры в данный момент или последствия скачка в прошлом.

Мы могли бы усложнить код агрегирования, чтобы решить проблему, но есть способ получше: добавить в агрегирование оконный режим (рис. 9.8).



**Рис. 9.8.** Используя оконный режим, можно разбить агрегат на отдельные блоки времени

### 9.1.1. Прыгающие окна

Добавив оконный режим в агрегирование, можно группировать результаты в отдельные слоты или окна времени. Посмотрим, как добавить поддержку оконного режима (агрегирование мы уже рассмотрели, поэтому я опишу только новые действия) (листинг 9.3).

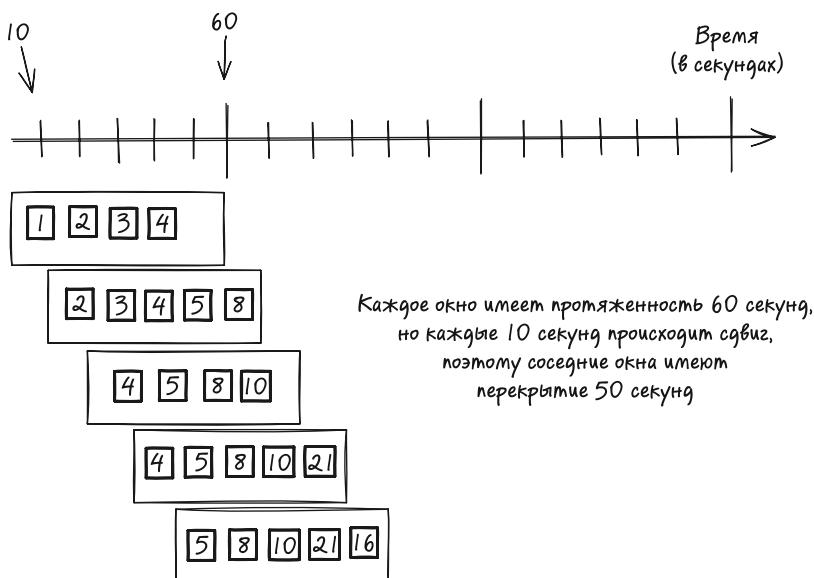
#### Листинг 9.3. Разделение агрегата с замерами температуры на слоты

```
KStream<String,Double> iotHeatSensorStream =
    builder.stream("heat-sensor-input",
        Consumed.with(stringSerde, doubleSerde));
    iotHeatSensorStream.groupByKey()           ← Группировка по ключу
        .windowedBy(TimeWindows.ofSizeWithNoGrace(
            Duration.ofMinutes(1)))          ← Мы добавили
        .advanceBy(Duration.ofSeconds(10)))   ← поддержку окон
        .aggregate((() -> new IoTSensorAggregation(tempThreshold),
            aggregator,
            Materialized.with(stringSerde, aggregationSerde))
        .toStream().to("sensor-agg-output",
            Produced.with(serdeString,
                sensorAggregationSerde))      ← Агрегирование
                                                ← Отправка результата
                                                ← агрегирования в топик
```

Здесь мы добавили вызов `windowedBy` сразу после `groupByKey`. `GroupByKey` возвращает `KGroupedStream`, имеющий все необходимое для добавления поддержки окон. В вызов `windowedBy` нужно передать единственный параметр — экземпляр окна для агрегирования. В данном случае мы используем класс `TimeWindows`, содержащий статические фабричные методы для создания окна.

Теперь, организовав агрегирование, нужно решить, в каком виде мы хотели бы получать данные? Наша команда решила, что лучше было бы получать средние температуры (плюс самые высокие из наблюдаемых) за последнюю минуту с обновлением каждые 10 секунд. Другими словами, каждые 10 секунд мы будем получать средние температуры за последнюю минуту. Иллюстрация на рис. 9.9 наглядно показывает, как это будет выглядеть.

Окна этого типа называются прыгающими. Прыгающее окно имеет фиксированный размер, в данном случае 1 минута, но каждые 10 секунд оно сдвигается (прыгает) на 10 секунд вперед.



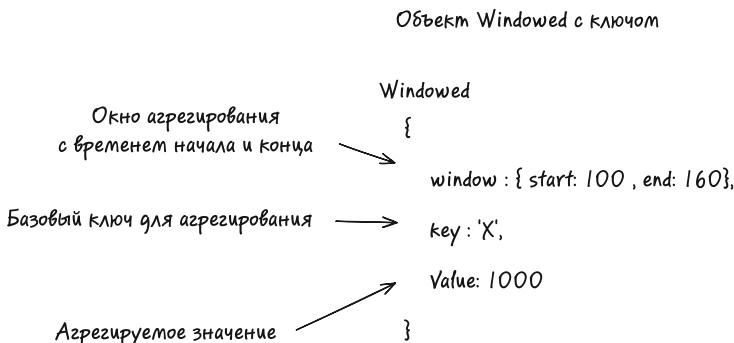
**Рис. 9.9.** Каждые 10 секунд в топик помещается средняя температура за последнюю минуту

Шаг смещения окна меньше его размера, поэтому соседние окна имеют перекрытие, то есть некоторые замеры из предыдущего окна могут быть включены в следующее. В этом случае перекрытие необходимо для целей сравнения.

Чтобы получить прыгающее окно, мы вызвали `TimeWindow.ofSizeWithNoGrace(Duration.ofMinutes(1))`. Этот вызов задает окно размером 1 минута. Я отложу объяснение части `WithNoGrace` в имени метода, пока мы не познакомимся со всеми типами окон, потому что концепция Grace в равной степени применима к ним ко всем. Затем, чтобы установить размер шага, равным 10 секундам, мы добавили вызов `advanceBy(Duration.ofSeconds(10))` сразу после определения окна.

Сразу после обновления кода среда разработки сообщает об ошибке в части топологии `to("sensor-agg-output", Produced(..))` (или, если вы не используете IDE, такую как IntelliJ, то получите сообщение об ошибке при попытке скомпилировать приложение в командной строке). Эта ошибка обусловлена тем, что при выполнении

оконного агрегирования Kafka Streams заключает ключ в экземпляр `Windowed`, который содержит ключ и экземпляр `Window`, как показано на рис. 9.10.



**Рис. 9.10.** Оконные операции агрегирования заключают ключ в класс `Windowed` вместе с объектом `Window` окна агрегирования

В результате изменения типа ключа он перестал соответствовать типу значения, который ожидается получить от объекта `Serde`, указанного в конфигурационном объекте `Produced`. Чтобы исправить эту ошибку, можно извлечь ключ из экземпляра `Windowed` или изменить тип `Serde` для работы с типом `Windowed`.

Но прежде, чем показать оба решения, попробую ответить на вопрос: какое решение выбрать? Ответ прост: выбор полностью зависит от ваших предпочтений и потребностей. Как я уже говорил, ключ `Windowed` содержит не только базовый ключ, но и экземпляр `Window` окна агрегирования и, как следствие, время начала и конца окна, что является ценной информацией, необходимой для оценки эффекта. Поэтому при прочих равных я рекомендую сохранять информацию о границах окна, содержащуюся в ключе `Windowed`.

Рассмотрим сначала первое решение — с извлечением базового ключа (листинг 9.4).

#### Листинг 9.4. Извлечение базового ключа из оконного агрегата

```

KStream<String,Double> iotHeatSensorStream =
    builder.stream("heat-sensor-input",
        Consumed.with(stringSerde, doubleSerde));
iotHeatSensorStream.groupByKey()
    .windowedBy(TimeWindows.ofSizeWithNoGrace(Duration.ofMinutes(1))
        .advanceBy(Duration.ofSeconds(10)))
    .aggregate(() -> new IoTSensorAggregation(tempThreshold),
        aggregator,
        Materialized.with(stringSerde, aggregationSerde))
        .toStream()
        .map((windowedKey, value) -> KeyValue.pair(windowedKey.key(),
            value))
        .to("sensor-agg-output",
            Produced.with(serdeString, sensorAggregationSerde))
    
```

← Для извлечения базового ключа используется метод map

Чтобы извлечь базовый ключ, нужно добавить вызов метода `map`, создать новый экземпляр `KeyValue` и затем вызвать метод `Windowed.key()`. Второе решение, сохраняющее ключ `Windowed`, показано в листинге 9.5.

**Листинг 9.5.** Использование ключа `Windowed` и изменение Serde для передачи данных в топик

```
Serde<Windowed<String>> windowedSerdes = ← Создание объекта Serde,
    WindowedSerdes.timeWindowedSerdeFrom(String.class, ← поддерживаящего ключ Windowed
        60_000L ← Параметр, представляющий
    ); ← Размер окна
    KStream<String,Double> iotHeatSensorStream = ←
        builder.stream("heat-sensor-input",
            Consumed.with(stringSerde, doubleSerde));
    iotHeatSensorStream.groupByKey()
        .windowedBy(TimeWindows.ofSizeWithNoGrace(Duration.ofMinutes(1)))
        .advanceBy(Duration.ofSeconds(10))
        aggregate(() -> new IoTSensorAggregation(tempThreshold),
            aggregator,
            Materialized.with(stringSerde, aggregationSerde))
            .toStream()
            .to("sensor-agg-output",
                Produced.with(windowedSerdes, sensorAggregationSerde)) ← Передача windowedSerdes
                                                                для сериализации
                                                                ключа перед
                                                                передачей в топик
```

Первый шаг — использование класса `WindowedSerdes`, предоставляемого библиотекой Kafka Streams, для создания объекта `Serde`, способного сериализовать объект `Window` и исходный ключ агрегирования. При создании `Serde` нужно указать два параметра: класс исходного ключа и размер окна в миллисекундах.

Затем новый объект `Serde(windowedSerdes)` передается в вызов метода `Produced.with` в первом параметре, поскольку он предназначен для сериализации ключа. С этого момента я могу использовать любой из подходов в своих примерах, но мой совет сохранять в результатах информацию об окне остается в силе.

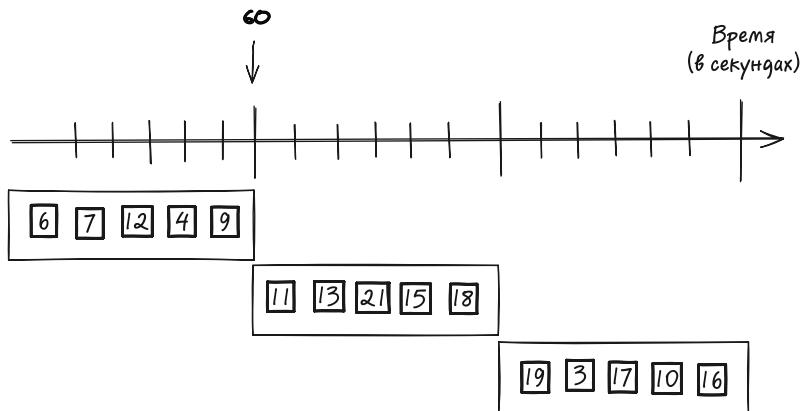
Теперь, решив проблему и начав запускать новое приложение с поддержкой оконного агрегирования, мы замечаем, что перекрытие результатов не совсем соответствует нашим аналитическим потребностям. Нам нужно видеть, какие датчики сообщают о высоких температурах за уникальный период. Другими словами, нам нужны неперекрывающиеся результаты, получаемые с помощью кувыркающегося (*tumbling*) окна.

## 9.1.2. Кувыркающиеся окна

Сделав шаг перемещения окна равным его размеру, мы гарантированно избавимся от перекрытия результатов. Каждое окно будет сообщать об уникальных событиях. На рис. 9.11 показано, как работает кувыркающееся окно.

Поскольку шаг перемещения окна равен его размеру, каждое окно содержит уникальные результаты и не имеет перекрытий с предыдущими окнами. Okna, шаг перемещения которых совпадает с их размером, называют кувыркающимися. Кувыркающиеся окна являются частным случаем прыгающих окон, в которых шаг перемещения совпадает с размером окна.

Чтобы получить такое окно, достаточно удалить вызов `advanceBy` из операции создания окна. Если для окна не указана величина шага, то по умолчанию Kafka Streams использует шаг, равный размеру окна. В листинге 9.6 показана обновленная реализация, использующая кувыркающиеся окна.



Каждое окно имеет промягженность 60 секунд и перемещается вперед на 60 секунд, поэтому оно не пересекается с предыдущим окном

**Рис. 9.11.** Получение окон с уникальными результатами за счет перемещения окна на его размер — кувыркающееся окно

**Листинг 9.6.** Переход на использование кувыркающегося окна путем удаления вызова `advanceBy`

```
Serde<Windowed<String>> windowedSerdes =
    WindowedSerdes.timeWindowedSerdeFrom(String.class,
                                            60_000L
                                         );
```

```
KStream<String,Double> iotHeatSensorStream =
    builder.stream("heat-sensor-input",
                   Consumed.with(stringSerde, doubleSerde));
    iotHeatSensorStream.groupByKey()
        .windowedBy(TimeWindows.ofSizeWithNoGrace(Duration.ofMinutes(1)))
        aggregate(() -> new IoTSensorAggregation(tempThreshold),
                  aggregator,
                  Materialized.with(stringSerde, aggregationSerde))
        .toStream()
        .to("sensor-agg-output",
            Produced.with(windowedSerdes, sensorAggregationSerde))
```

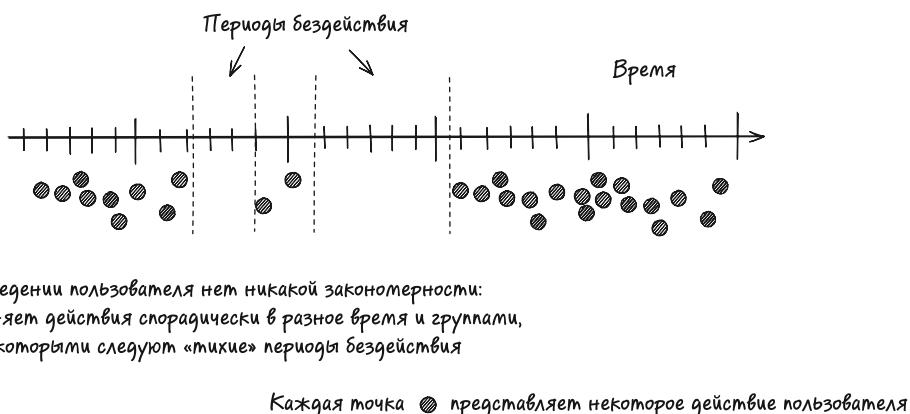
Теперь, после удаления  
вызыва `advanceBy`,  
окно определяется  
как кувыркающееся

Текущий код выглядит удивительно похожим на предыдущий, однако так и должно быть, потому что мы удалили лишь вызов `advanceBy`, а все остальное осталось прежним. Конечно, можно оставить вызов `advanceBy` и передать ему значение размера

окна, но я советую не использовать его при создании кувыркающегося окна, чтобы исключить любую двусмысленность.

Теперь, удовлетворенные разработанным подходом к отслеживанию замеров с датчиков IoT, перейдем к новой проблеме. Спрос на потоковые конденсаторы продолжает расти. Чтобы помочь бизнесу повысить эффективность работы с клиентами, отдел маркетинга решил начать отслеживать просмотры страниц и события щелчков кнопкой мыши на веб-сайте компании. Окрыленные предыдущим успехом, мы решили взять на себя помочь в управлении этими новыми событиями.

Но беглого знакомства с задачей оказалось достаточно, чтобы понять, что в этом случае полезнее был бы другой подход к управлению окнами. Датчики IoT выдают замеры с постоянной скоростью, и поэтому результаты четко вписывались в прыгающее или в кувыркающееся окно. Однако просмотры страниц и события щелчков отличаются большей спорадичностью, как показано на рис. 9.12.



**Рис. 9.12.** В поведении пользователя нет никакой закономерности

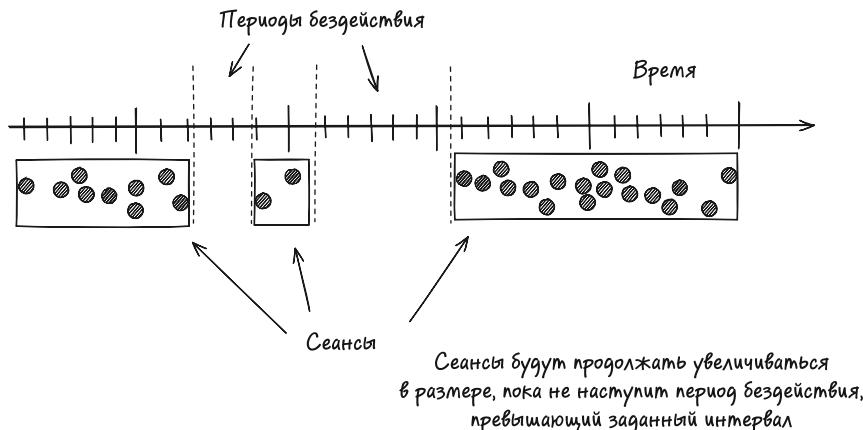
Как видите, поведение пользователя практически непредсказуемо и действия могут разделяться большими или малыми промежутками бездействия. Для отслеживания таких событий требуется окно другого типа, которое может непрерывно увеличиваться, пока продолжается действие. Kafka Streams предоставляет то, что нам нужно, — сеансовое окно.

### 9.1.3. Сеансовые окна

Сеансовые окна, в отличие от других типов окон, продолжают увеличиваться в размерах по мере поступления событий, но только до определенного момента. Когда в течение определенного промежутка времени не выполняется никаких действий, сеанс закрывается и для любых последующих действий после перерыва запускается новый сеанс. Взгляните на рис. 9.13, иллюстрирующий эту идею.

Как видите, здесь непрерывно поступающие события увеличивают размер окна. Однако, когда период бездействия превышает заданный интервал, для всех вновь поступающих записей запускается новый сеанс. Промежуток бездействия — это

отличительная черта сеансовых окон. Вместо создания окна фиксированного размера мы указываем, как долго ждать нового действия, прежде чем сеанс будет считаться закрытым. В противном случае окно продолжит увеличиваться.



**Рис. 9.13.** Сеансовые окна продолжают увеличиваться, пока не наступит перерыв в активности, после чего начинается новый сеанс

Посмотрим, что нам нужно для реализации решения с сеансовым окном. Мы будем использовать агрегирование, похожее на то, что использовалось с кувыркающимися и прыгающими окнами.

Для реализации агрегирования с окном этого типа нужно использовать `SessionWindow`. Вашему агрегатору (экземпляру `Aggregator`) также нужно передать экземпляр функционального интерфейса `Merger`, который знает, как объединить два сеанса. Будет разумнее, если сначала мы рассмотрим общий код, а затем я объясню, как настроить сеансовое окно, и расскажу о слиянии сеансов. В листинге 9.7 показан код, который мы будем использовать.

#### Листинг 9.7. Использование SessionWindow для отслеживания просмотров страниц клиентами

```

Создание объекта Serde
поддержкой сеансовых окон
Serde<Windowed<String>> sessionWindowSerde =
    WindowedSerdes.sessionWindowSerdeFrom(String.class); ←

KStream<String, String> pageViewStream = builder.stream("page-view",
    Consumed.with(serdeString, serdeString));
pageViewStream.groupByKey()
    .windowedBy(SessionWindow.ofInactivityGapWithNoGrace(
        Duration.ofMinutes(2))) ← Настраойка сеансового окна для агрегирования
    .aggregate(HashMap::new, ← Использование дескриптора метода
        sessionAggregator, ← для создания начального экземпляра
        sessionMerger) ←
    .toStream() ← Добавление в агрегатор поддержки
    .to("page-view-session-aggregates", ← слияния сеансов в виде SessionMerger
        Produced.with(sessionWindowSerdes, pageViewAggregationSerde))

```

Эта реализация агрегирования похожа на предыдущие (конечно, имена топиков различаются, как и используемые объекты Serdes, но это лишь детали реализации). Обратите внимание: чтобы создать правильный экземпляр Serde для сеансового окна `SessionWindow`, нужно использовать `WindowedSerdes.sessionWindowedSerdeFrom`. Еще одно важное отличие отмечено второй аннотацией и заключается в настройке `SessionWindow`.

Чтобы использовать сеансы для агрегирования, мы вызвали метод `SessionWindow.ofInactivityGapWithNoGrace`, определяющий тип окна. Передаваемый ему параметр определяет не размер окна, а продолжительность периода бездействия перед закрытием. Оконная операция будет определять превышение этого периода по отметкам времени в событиях. А теперь углубимся в детали агрегирования.

Прежде чем исследовать реализацию `Merger`, кратко рассмотрим код агрегирования, который в этом примере совершенно новый (листинг 9.8).

**Листинг 9.8.** Агрегирование просмотров страниц и подсчет количества посещений

```
public class PageViewAggregator
    implements Aggregator<String, String, Map<String, Integer>> {

    @Override
    public Map<String, Integer> apply(String userId,
                                      String url,
                                      Map<String, Integer> aggregate) {

        aggregate.compute(url, (key, count)
                          -> (count == null) ? 1 : count + 1); ←
        return aggregate;
    }                                              Метод Map.compute подсчитывает количество
                                                    посещений страницы пользователем
}
```

Реализация агрегирования использует `HashMap` и метод `Map.compute` для отслеживания просмотренных страниц и подсчета количества посещений пользователем каждой из них в течение данного сеанса.

Теперь перейдем к объекту `Merger`. Интерфейс `Merger` определяет один абстрактный метод с тремя параметрами: ключом агрегирования и двумя агрегатами, текущим и следующим, которые можно объединить. Когда Kafka Streams выполняет метод `Merger.apply`, он берет два агрегата и объединяет их в новый общий агрегат. Листинг 9.9 содержит код с реализацией слияния сеансов отслеживания просмотров страниц.

**Листинг 9.9.** Слияние агрегатов двух сеансов в один

```
public class PageViewSessionMerger
    implements Merger<String, Map<String, Integer>> {

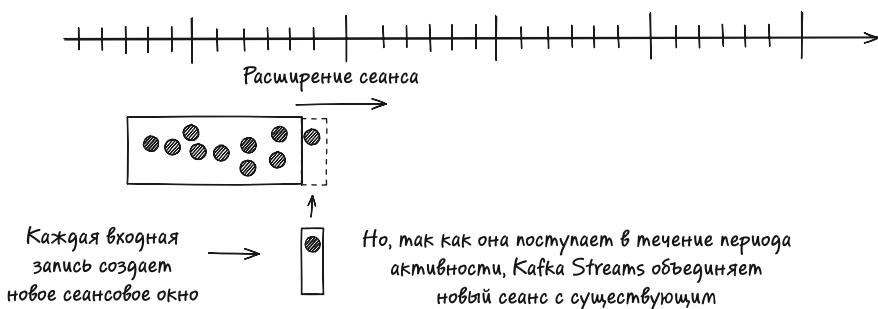
    @Override
    public Map<String, Integer> apply(String aggKey,
                                      Map<String, Integer> mapOne,
                                      Map<String, Integer> mapTwo) {
```

```

        mapTwo.forEach((key, value)->
            mapOne.compute(key, (k,v) -> (v == null) ? value : v + value
        )));
        return mapOne;
    }
}
}

```

Операция слияния проста по своей сути: она объединяет содержимое двух массивов `HashMap` с результатами отслеживания просмотров страниц в один. Необходимость в слиянии сеансовых окон может возникнуть, если вдруг обнаружится какая-то запоздавшая запись, которая может соединить два старых сеанса в один более крупный. Процесс слияния сеансов интересен и заслуживает дополнительного пояснения. Его иллюстрирует рис. 9.14.



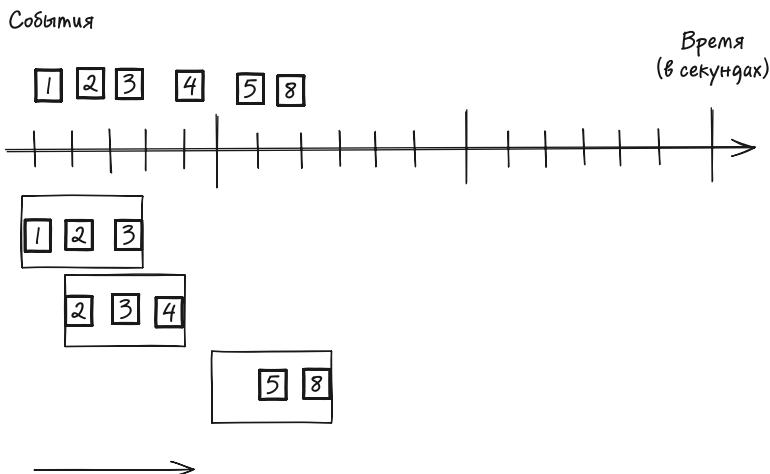
**Рис. 9.14.** Поддержка слияния сеансовых окон позволяет объединить два сеанса в один

Как видите, каждая входная запись создает новый сеанс (`SessionWindow`) с временем начала и конца, соответствующими отметке времени в этой записи. Затем Kafka Streams отыскивает все сеансовые окна для данного ключа, отстоящие во времени от текущей записи не далее, чем `window.inactivityGap`. В результате этого поиска, скорее всего, будет найдено одно сеансовое окно, поскольку предыдущие записи в пределах промежутка бездействия уже объединили все предыдущие окна.

Важно отметить, что поскольку Kafka Streams извлекает предыдущие сеансы по времени, тем самым гарантируется, что слияние применяется в порядке поступления, то есть объединенные агрегаты будут вычисляться во временном порядке.

#### 9.1.4. Скользящие окна

Теперь мы можем отслеживать поведение пользователя с помощью `SessionWindow`, но мы решили использовать еще один вид анализа — просматривать события щелчка кнопкой мыши на расстоянии 10 секунд друг от друга в одноминутном окне, потому что для нас важно знать, какие страницы сайта посещают пользователи перед покупкой. В Kafka Streams это можно сделать с помощью `SlidingWindow`. Скользящее окно сочетает в себе характеристики `TimeWindows`, имея фиксированный размер, и `SessionWindows`, определяя время начала и конца по отметкам времени в записях. Но нам нужно, чтобы поток записей анализировался непрерывно, как показано на рис. 9.15.



Окно непрерывно скользит по событиям и включает только события, происходящие в интервале, заданном фиксированным размером окна

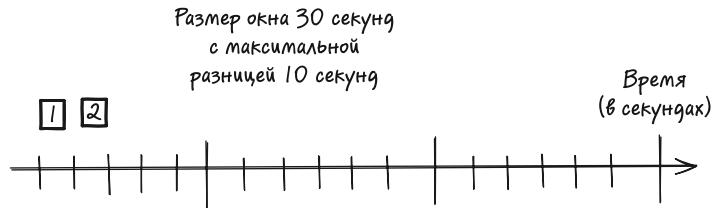
**Рис. 9.15.** Скользящие окна обеспечивают непрерывный обзор изменения событий

Начало и конец окна определяются только отметками времени в записях, но размер окна остается фиксированным и составляет 30 секунд. В принципе, можно сымитировать скользящее окно, создав прыгающее окно протяженностью 30 секунд и с шагом 1 миллисекунда, но это будет очень неэффективно. Причины я объясню после того, как мы рассмотрим реализацию скользящего окна.

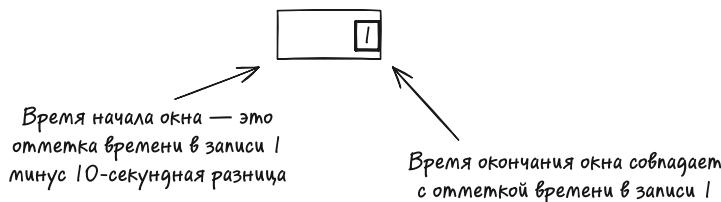
Для создания скользящего окна используется метод `SlidingWindows.ofTimeDifferenceWithNoGrace`. Он принимает один параметр — максимальную разницу во времени между записями, выраженную в виде объекта `Duration`.

В скользящих окнах время начала и конца включается в окно, в отличие от прыгающих и кувыркающихся окон, которые включают в окно только время начала, но исключают время конца. Другое отличие: скользящие окна «оглядываются назад» в поисках событий, чтобы включить в себя и их. Рассмотрим еще одну схему на рис. 9.16, поясняющую только что сказанное.

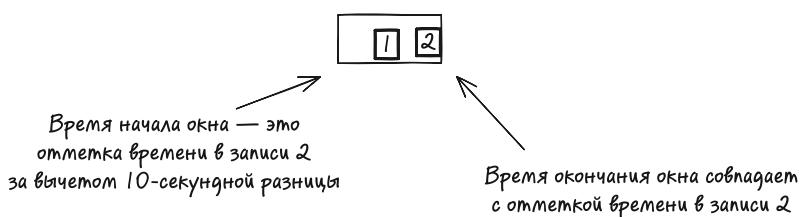
Здесь запись 1 поступает в поток и создает новое окно с временем окончания, совпадающим с отметкой времени в записи за вычетом максимальной разницы записей. В предшествующий 10-секундный период не происходило никаких событий, поэтому запись остается сама по себе. Затем поступает запись 2 и создается новое окно, но на этот раз, поскольку отметка времени в записи 1 находится в пределах установленной разницы во времени, она включается в это новое окно. Таким образом, в сценарии со скользящим окном каждая входная запись приводит к созданию нового окна, и оно будет «оглядываться назад» на другие события с отметками времени в пределах разницы во времени для их включения в себя. В остальном код реализации ничем не будет отличаться от другого кода, написанного для других операций агрегирования. В листинге 9.10 приводится полный пример использования скользящего окна (некоторые детали опущены для простоты).



Запись 1 поступает в поток событий и создает новое окно, но в предшествующий 10-секундный период нет других записей



Поступает запись 2, и создается новое окно, но в окно также включается запись 1, поскольку ее отмечка времени отстоит в прошлом не далее заданной разницы во времени



**Рис. 9.16.** Скользящие окна включают время начала и окончания в свои границы и охватывают события, произошедшие в течение определенной разницы во времени

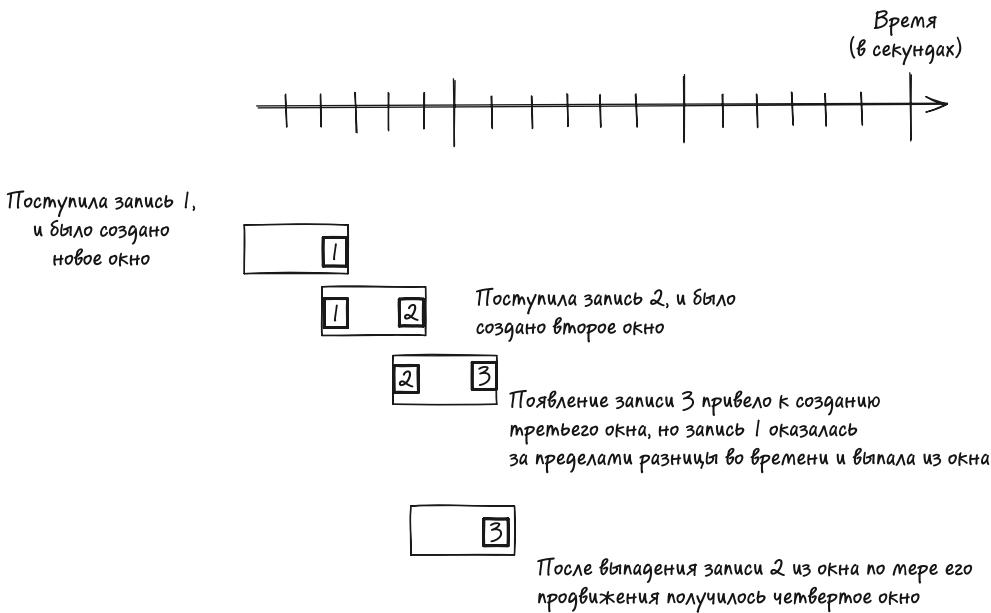
**Листинг 9.10.** Агрегирование просмотров страниц с помощью скользящего окна

```
KStream pageViewStream = builder.stream("page-view",
                                         Consumed.with(serdeString,pageViewSerde))
pageViewStream.groupByKey()
    .windowedBy(SlidingWindows.ofTimeDifferenceWithNoGrace(
        Duration.ofSeconds(30)) ← Настойка скользящего окна для агрегирования
        .aggregate(HashMap::new,
pageViewAggregator) ← Добавление экземпляра Aggregator
        .toStream()
        .to("page-view-sliding-aggregates",
Produced.with(windowedSerdes, pageViewAggregationSerde))
```

`SlidingWindows.ofTimeDifferenceWithNoGrace(Duration.ofSeconds(30))` — это центральная часть кода, описанного в предыдущем абзаце. Здесь задается тип окна.

Поскольку Aggregator тот же самый, что и в примере с сеансовым окном, я не буду рассматривать его снова.

О скользящих окнах важно помнить следующее: они могут перекрываться (каждое следующее окно может содержать записи, включенные в предшествующие окна), но вообще Kafka Streams создает новое окно, только когда поступает новая запись или когда она выпадает, и каждое окно будет иметь уникальный набор результатов, как показано на рис. 9.17.



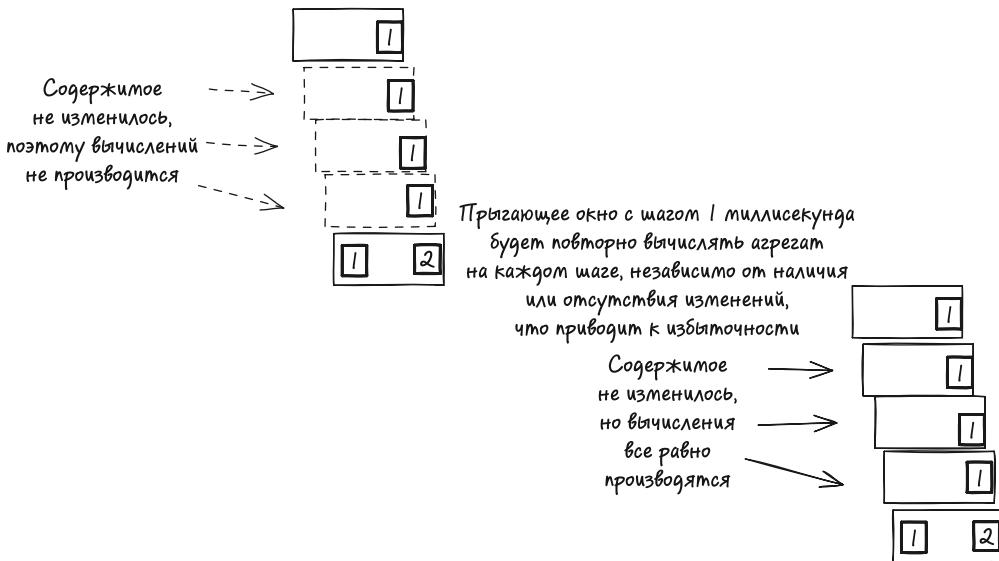
**Рис. 9.17.** Скользящие окна создаются, только когда поступают новые записи или когда они выпадают

Как показано на этой иллюстрации, поступление первой записи приводит к созданию нового окна. Поступление второй записи тоже создает новое окно, в которое также включается первая запись. Далее поступает третья запись и создается третье окно. Но, поскольку запись 1 оказывается за пределами разницы во времени, в это последнее окно включается только запись 2. Затем по мере продвижения во времени вторая запись выпадает из окна и создается четвертое окно.

Выше в этом разделе я упоминал, что предпочтительнее использовать скользящее окно, чем прыгающее с минимальным размером шага. Иллюстрация на рис. 9.18 объясняет причину.

Скользящее окно вычисляет агрегат, только когда в него входит новая запись или из него выпадает ранее существовавшая запись (именно поэтому каждое скользящее окно содержит уникальные результаты). Таким образом, даже если скользящее окно продвигается с шагом 1 миллисекунда, агрегат будет вычисляться только при изменении содержимого окна.

Скользящее окно сдвигается на 1 миллисекунду, но выполняет вычисления только при изменении своего содержимого — соответственно, есть перекрытие, но нет избыточности



**Рис. 9.18.** Скользящие окна вычисляют свои агрегаты, только когда изменяется их содержимое, а прыгающие окна выполняют вычисления на каждом шаге

Прыгающее окно всегда будет вычислять агрегат. Рисунок 9.18 как раз и показывает, что скользящее окно выполняет вычисления, только когда поступает новая запись или существующая покидает окно, а эквивалентное прыгающее окно выполнит вычисления (агрегирование)  $N$  раз, даже притом что содержимое изменилось только один раз, что приводит к избыточным вычислениям.

На этом мы завершаем обзор типов окон, доступных в Kafka Streams. Но прежде, чем двинуться дальше, я хотел бы подвести итоги обсуждения и упомянуть дополнительные детали, которые помогут вам создавать оконные приложения.

### 9.1.5. Выравнивание окон по времени

Для начала обсудим, как разные окна выравниваются по времени, то есть как Kafka Streams определяет начальное и конечное время. Любое окно, созданное с помощью класса `TimeWindows`, выравнивается по эпохе. Рассмотрим две иллюстрации, чтобы понять, что это значит. Схема на рис. 9.19 показывает, что означает выравнивание по эпохе.

Выравнивание по эпохе означает, что первое окно охватывает интервал  $[0, \text{windowSize})$ , второе размещается вплотную за первым и так до тех пор, пока не будет достигнуто текущее время. Важно понимать, что окна — это логические сущности; в Kafka Streams не так уж много экземпляров окон. Вот что подразумевается под выравниванием окон по эпохе. Время разбивается на интервалы с протяженностью,

равной размеру окна, начиная с начала эпохи Unix ( полночь 1 января 1970 года). Теперь взгляните на рис. 9.20, который показывает, как все это работает на практике.

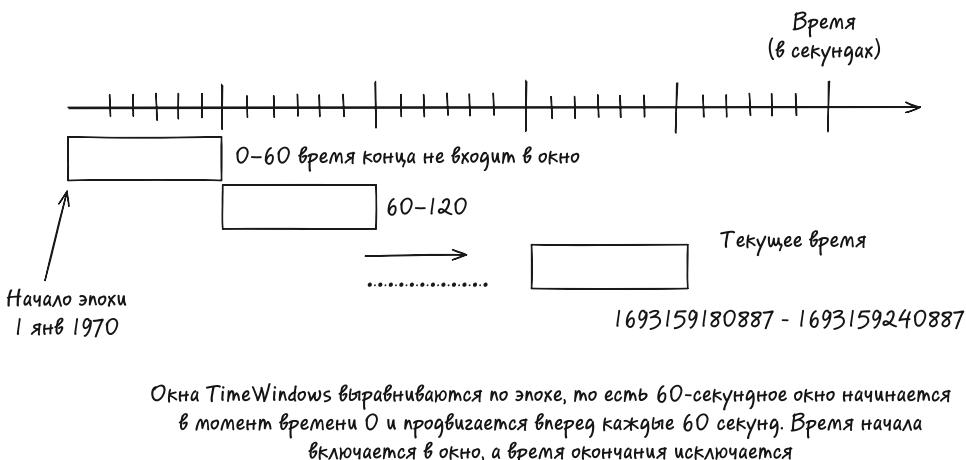


Рис. 9.19. TimeWindows выравнивается по эпохе

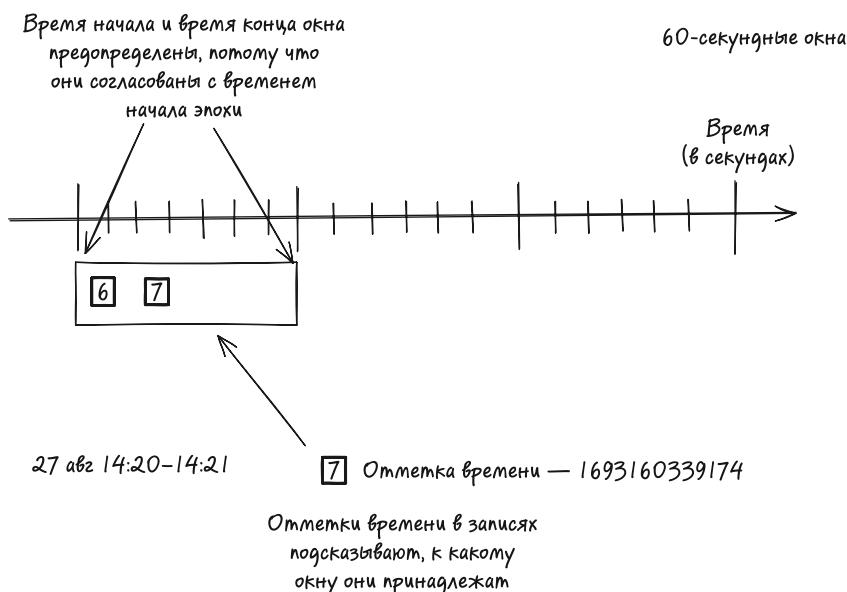
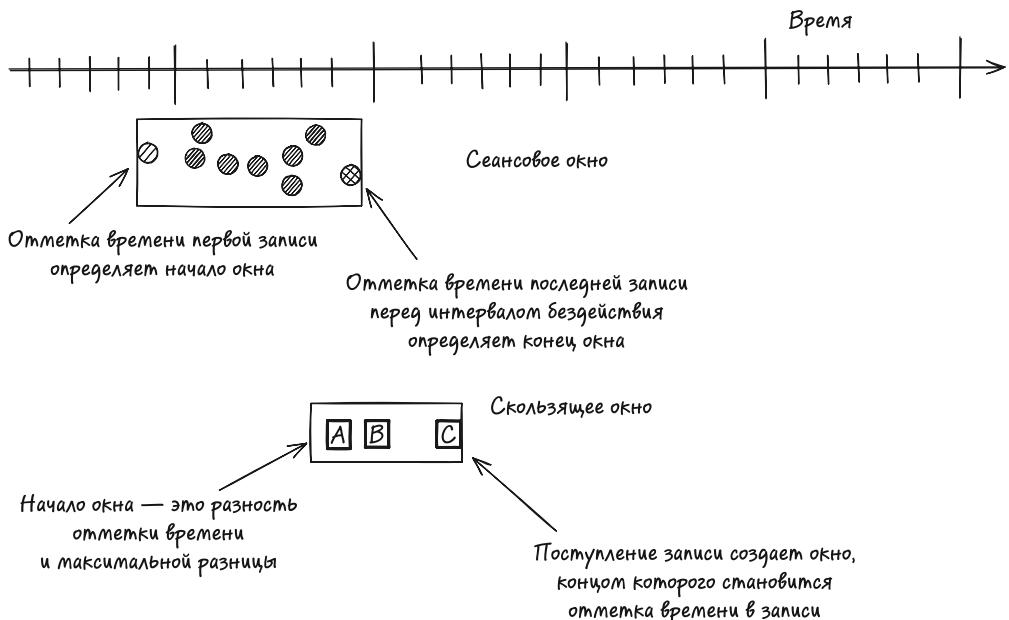


Рис. 9.20. Время начала эпохи определяет границы окон TimeWindows

Итак, отметки времени событий в записях в кувыркающихся или прыгающих окнах определяют не начало или конец окна, а то, к какому окну они принадлежат.

Теперь рассмотрим скользящие и сеансовые окна, изображенные на рис. 9.21, который поможет нам прояснить идею.



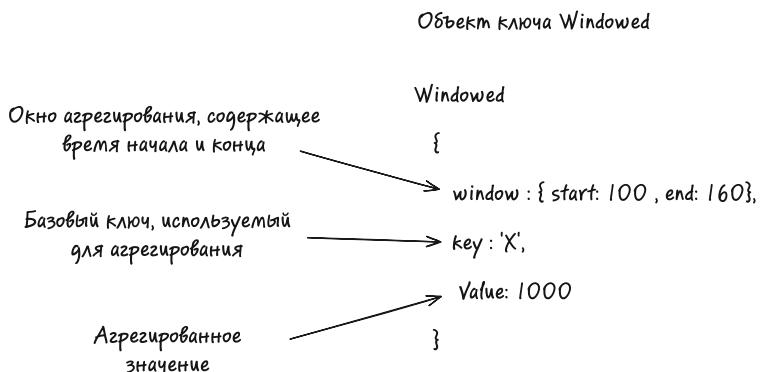
**Рис. 9.21.** Начало и конец скользящих и сеансовых окон определяются отметками времени событий

Как видите, отметки времени событий определяют границы сеансового и скользящего окна. Несмотря на то что `SlidingWindows` имеет фиксированный размер, начало окна выравнивается по отметке времени события. То же относится и к `SessionWindows`. Отметка времени в первой записи устанавливает время начала окна. Конечное время скользящего окна определяется как сумма заданной максимальной разницы во времени и отметки времени в записи, создавшей окно. Конечное время сеансового окна определяется как отметка времени последней записи, поступившей до периода бездействия.

### 9.1.6. Получение результатов в окне для анализа

Теперь уделим немного времени вычислению оконных агрегатов. Я уже упоминал об этом выше в этой главе, но считаю нужным повторить еще раз. Выполняя оконное агрегирование, Kafka Streams заключает ключ в экземпляр `Windowed`, который, помимо ключа, содержит также объект `Window` с начальным и конечным временем окна. Взгляните на рис. 9.22, демонстрирующий оконный ключ `Windowed`.

Поскольку в ключе `Windowed` вместе с агрегированным значением хранится также время начала и конца окна, мы имеем все, что нужно для анализа агрегата в заданном временном интервале. Теперь я покажу некоторые основные шаги, которые можно предпринять для просмотра результатов в `Windowed`. Используем для этого пример `TumblingWindow`. Сначала мы рассмотрим код агрегирования в листинге 9.11, где показана только сама операция агрегирования (некоторые детали опущены для простоты).



**Рис. 9.22.** Оконные операции агрегирования заключают ключ в экземпляр класса Windowed, содержащий ключ и окно агрегирования

### Листинг 9.11. Агрегирование с помощью TumblingWindow

```
iotHeatSensorStream.groupByKey()
    .windowedBy(TimeWindows.ofSizeWithNoGrace(Duration.ofMinutes(1)))
    .aggregate(() -> new IoTSensorAggregation(tempThreshold),
              aggregator,
              Materialized.with(stringSerde, aggregationSerde))
```

Напомню, что операции агрегирования возвращают KTable. В этом есть определенный смысл, поскольку нам нужно, чтобы новый результат агрегирования для заданного ключа заменял предыдущий (общий обзор вы найдете в главе 8). Но KTable не предлагает никаких способов получить его содержимое, поэтому прежде всего нужно преобразовать KTable в KStream с помощью метода `toStream()` (некоторые детали опущены для простоты) (листинг 9.12).

### Листинг 9.12. Преобразование результата агрегирования из типа KTable в тип KStream

```
iotHeatSensorStream.groupByKey()
    .windowedBy(TimeWindows.ofSizeWithNoGrace(Duration.ofMinutes(1)))
    .aggregate(() -> new IoTSensorAggregation(tempThreshold),
              aggregator,
              Materialized.with(stringSerde, aggregationSerde))
    .toStream()   ──────────| Преобразование результата агрегирования
                           | из типа KTable в тип KStream
```

Теперь у нас есть объект KStream, предлагающий несколько способов доступа к результатам. Следующий шаг — вызвать метод `peek` и передать ему лямбда-функцию в параметре с типом интерфейса `ForEachAction` (листинг 9.13).

Здесь мы просто объявили пару «ключ — значение», переданные в качестве параметров методу `peek`, но я хотел бы отметить важность использования осмысленных имен для ключа и значения. Здесь мы дали ключу имя `windowedKey`, которое точно описывает объект ключа как экземпляр класса `Windowed`. Значению мы

дали имя `aggregation`, поскольку оно точно отражает его суть. Хотя именование является второстепенным моментом, тем не менее выбор говорящих имён может облегчить чтение кода другим (или вам самим, когда вы вернетесь к нему по прошествии некоторого времени!) и быстро понять, что представляют собой ключ и значение.

#### Листинг 9.13. Использование метода `peek` для вывода времени начала и конца окна

```
iotHeatSensorStream.groupByKey()
    .windowedBy(TimeWindows.ofSizeWithNoGrace(Duration.ofMinutes(1)))
    .aggregate(() -> new IotSensorAggregation(tempThreshold),
        aggregator,
        Materialized.with(stringSerde, aggregationSerde))
    .toStream()
    .peek((windowedKey, aggregation) -> { ← Вызов метода peek потока KStream}
```

Далее нужно получить объект `Window` из ключа и извлечь время начала и конца окна. Для этого сначала добавим код для извлечения объекта `Window`. Затем получим начальное и конечное время окна, как показано в листинге 9.14 (некоторые детали опущены для простоты).

#### Листинг 9.14. Вывод результатов оконного агрегирования для аналитических целей

```
iotHeatSensorStream.groupByKey()
    .windowedBy(TimeWindows.ofSizeWithNoGrace(Duration.ofMinutes(1)))
    .aggregate(() -> new IotSensorAggregation(tempThreshold),
        aggregator,
        Materialized.with(stringSerde, aggregationSerde))
    .toStream()
    .peek((windowedKey, aggregation) -> {
        Window window = windowedKey.window(); ← Извлечение объекта
        Instant start =
            window.startTime() ← Время начала окна в виде экземпляра Instant
            .truncatedTo(ChronoUnit.SECONDS);
        Instant end =
            window.endTime() ← Время конца окна в виде экземпляра Instant
            .truncatedTo(ChronoUnit.SECONDS);
    })
})
```

Мы извлекли объект `Window` из ключа и получили время начала вызовом метода `Window.startTime()`, который возвращает объект `java.time.Instant`. Мы также округлили время, отбросив миллисекунды с помощью метода `Instant.truncatedTo`. Теперь завершим код и выведем время начала и конца окна, а также результат агрегирования (листинг 9.15).

Этот последний шаг добавляет вывод времени начала и конца окна вместе со значением агрегирования. Этот простой пример с `KStream.peek` выводит начало и конца окна в удобном для человека формате вместе с результатом агрегирования. Это хорошая отправная точка для вашего воображения и создания вашей аналитической функции.

**Листинг 9.15.** Добавление инструкции вывода

```
iotHeatSensorStream.groupByKey()
    .windowedBy(TimeWindows.ofSizeWithNoGrace(Duration.ofMinutes(1)))
    .aggregate(() -> new IoTSensorAggregation(tempThreshold),
        aggregator,
        Materialized.with(stringSerde, aggregationSerde))
    .toStream()
    .peek((windowedKey, aggregation) -> {
        Window window = windowedKey.window();
        Instant start =
            window.startTime()
                .truncatedTo(ChronoUnit.SECONDS);
        Instant end =
            window.endTime()
                .truncatedTo(ChronoUnit.SECONDS);
        LOG.info("Window started {} ended {} with value {}", ←
            start,
            end, aggregation); })})
```

Инструкция для вывода времени  
начала и конца, а также значения

Одним из следствий сохранения ключа агрегирования в исходном формате является необходимость предоставить доступ к Serde любым приложениям, которые пожелают использовать данные агрегирования. Кроме того, наличие времени начала и конца окна в ключе изменит раздел, назначаемый записи. Это означает, что записи с одним и тем же ключом окажутся в разных разделах из-за разного времени начала и конца окна в ключе `Windowed`. Хорошая идея — распределять исходящие записи по базовому ключу. Один из возможных подходов — реализовать `StreamPartitioner`, который определит раздел для сохранения результата агрегирования по базовому ключу. Листинг 9.16 показывает реализацию этого решения.

**Листинг 9.16.** Определение корректного раздела по базовому ключу

```
@Override
public Optional<Set<Integer>> partitions(String topic,
    Windowed<K> windowedKey,
    V value,
    int numPartitions) {
    if(windowedKey == null) {
        return Optional.empty();
    }
    byte[] keyBytes = keySerializer.serialize(topic,
        windowedKey.key());
    if (keyBytes == null) {
        return Optional.empty();
    }
    Integer partition =
        Utils.toPositive(Utils.murmur2(keyBytes)) % numPartitions;
    return Optional.of(Collections.singleton(partition));
}
```

Я опустил некоторые детали в листинге 9.16, но вы можете увидеть полный пример этой реализации `StreamPartitioner` в примерах исходного кода для книги в `bbejeck.chapter_9.bbejeck.chapter_9.partitionner.WindowedStreamsPartitioner.java`. Чтобы использовать пользовательскую реализацию секционирования, необходимо добавить ее в конфигурационный объект `Produced`, как показано в листинге 9.17.

#### Листинг 9.17. Добавление своей реализации StreamPartitioner в Produced

```
WindowedStreamsPartitioner<String, IotSensorAggregation>
➥ windowedStreamsPartitioner =
➥ new WindowedStreamsPartitioner<>(stringSerde.serializer());

...
.to(outputTopic, Produced.with(
    windowedSerdes, aggregationSerde)
➥ .withStreamPartitioner(windowedStreamsPartitioner));
```

И снова я опустил некоторые детали. Полный пример вы найдете в файле `bbejeck.chapter_9.tumbling.IotStreamingAggregationStreamPartitionerTumblingWindows.java`, где демонстрируется использование `StreamPartitioner` для секционирования оконных агрегатов по исходному ключу.

Однако подход с использованием пользовательской реализации секционирования по-прежнему оставляет информацию об окне в ключе, а результат агрегирования — в значении. Он порождает «утечку абстракции», поскольку потребляющие приложения должны знать, что ключ имеет тип `Windowed`. Вы можете попробовать отобразить время начала и конца окна с результатом агрегирования в один объект, и тогда вся информация будет доступна в этом одном объекте. Для этого нужно сначала расширить объект `IotSensorAggregation`, добавив в него два новых поля типа `long`, например, с именами `windowStart` и `windowEnd`. Затем предоставить реализацию `KeyValueMapper`, которая извлечет информацию о времени из ключа, преобразует ее в значение и вернет объект `KeyValue` с исходным ключом и обновленным агрегированным значением (листинг 9.18).

#### Листинг 9.18. Включение времени окна в объект с результатом агрегирования

```
public class WindowTimeToAggregateMapper implements
➥ KeyValueMapper<Windowed<String>, IotSensorAggregation,
➥ KeyValue<String, IotSensorAggregation> {
@Override
public KeyValue<String, IotSensorAggregation>
➥ apply(Windowed<String> windowed,
    IotSensorAggregation iotSensorAggregation) {
    long start = windowed.window().start(); ←
    long end = windowed.window().end(); ←
    iotSensorAggregation.setWindowStart(start); ←
    iotSensorAggregation.setWindowEnd(end); ←
    return KeyValue.pair(windowed.key(), iotSensorAggregation);
}
```

По завершении реализации `KeyValueMapper` остается только добавить его в топологию с помощью оператора `KStream.map`, как показано в листинге 9.19 (некоторые детали опущены для простоты).

#### Листинг 9.19. Добавление реализации KeyValueMapper

```
KeyValueMapper<Windowed<String>, IoTSensorAggregation,
→ KeyValue<String, IoTSensorAggregation>> windowTimeMapper =
    → new WindowTimeToAggregateMapper();

iotHeatSensorStream.groupByKey()
    .windowedBy(TimeWindows.ofSizeWithNoGrace(Duration.ofMinutes(1)))
    .aggregate(() -> new IoTSensorAggregation(tempThreshold),
        aggregator,
        Materialized.with(stringSerde, aggregationSerde))
    .toStream()
    .map(windowTimeMapper)           ← | Объединение времени начала и конца окна
    .to(outputTopic, Produced.with( | с результатом агрегирования и замена
        stringSerde, aggregationSerde)); | ключа Windowed исходным
```

Теперь исходящие записи с результатами агрегирования будут содержать простой ключ и время начала и конца окна вместе с результатом агрегирования в значении.

Прежде чем двинуться дальше, взгляните на табл. 9.1, где обобщается все, что вы узнали о различных типах окон.

**Таблица 9.1.** Обобщение

Тип окна	Выравнивание окна	Фиксированный размер	Назначение
Прыгающее	По эпохе	Да	Измеряет каждое изменение $x$ по сравнению с прошлым $y$
Кувыркающееся	По эпохе	Да	Суммирует события за период времени
Скользящее	По отметкам времени в событиях	Да	Фиксирует изменения в непрерывно скользящем окне/скользящие средние значения
Сеансовое	По отметкам времени в событиях	Нет	Фиксирует события, происходящие в пределах заданного максимального периода бездействия друг от друга

Теперь окунемся в область, связанную с окнами всех типов: обработкой неупорядоченных данных и выдачей только одного результата в конце окна.

## 9.2. ОБРАБОТКА НЕУПОРЯДОЧЕННЫХ ДАННЫХ С ОТСРОЧКОЙ

В главе 4 вы узнали, что KafkaProducer устанавливает отметку времени в записи при ее производстве. В идеальном мире отметки времени в записях в Kafka Streams должны всегда увеличиваться. Но реальность распределенных систем такова, что может произойти все что угодно. Производители Kafka взаимодействуют с брокерами через сетевые соединения, что делает их восприимчивыми к сетевым сбоям, нарушающим порядок следования запросов, производящих записи. Рассмотрим ситуацию на рис. 9.23.

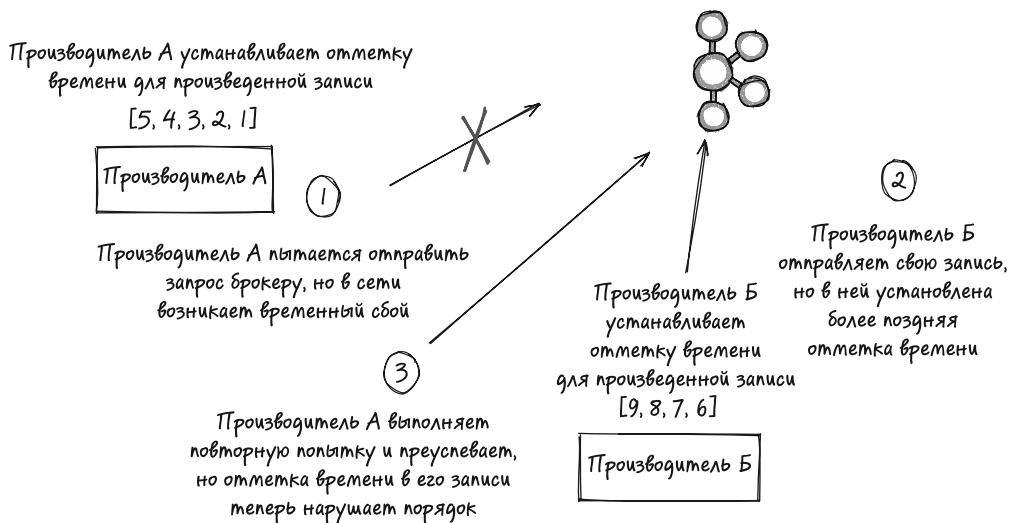


Рис. 9.23. Сбой в сети, приводящий к нарушению порядка следования запросов, посылаемых двумя производителями

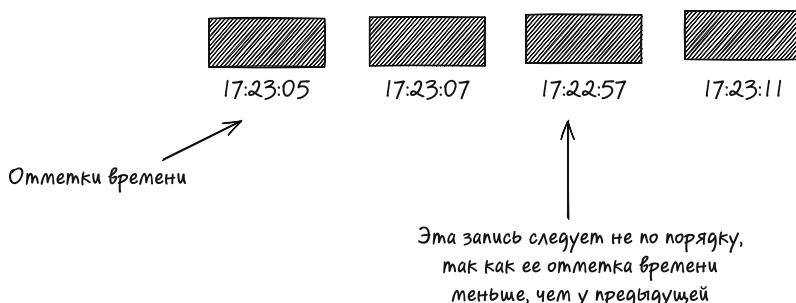
Как здесь показано, производитель А собирает запись для отправки брокеру, немного опережая производитель Б, действующий на другом хосте, но когда производитель А пытается отправить запись, возникает сетевой сбой и запрос не отправляется. У производителя Б, находящегося на другом хосте, не возникает проблем с подключением, и его запрос успешно отправляется брокеру.

Сетевой сбой на стороне производителя А длится около 10 секунд (что не является проблемой, поскольку производитель будет продолжать пытаться отправить запрос до истечения времени `delivery.timeout`, которое по умолчанию составляет 2 минуты). В конечном счете запрос благополучно отправляется. Но теперь отметки времени на стороне брокера следуют не по порядку.

Производители устанавливают отметку времени события, когда принимают `ProducerRecord` в методе `Producer.send`, поэтому любая существенная задержка в отправке может привести к нарушению порядка отметок времени на стороне брокера. Поскольку Kafka обрабатывает записи в порядке их смещений, то потребитель может получить записи с более ранними отметками времени позже.

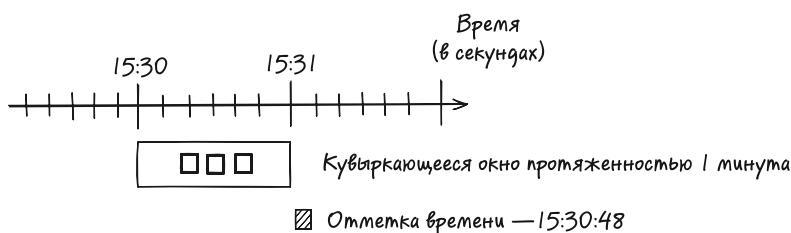
В других случаях отметки времени могут встраиваться в полезную нагрузку значения записи. В таком сценарии вы в принципе не сможете гарантировать упорядоченность отметок времени, потому что производитель не контролирует их. Конечно, ситуация со сбоем сети, описанная в предыдущем абзаце, применима и здесь.

Взгляните на рис. 9.24, который наглядно демонстрирует идею неупорядоченных данных.



**Рис. 9.24.** Неупорядоченные записи поступают с нарушением порядка

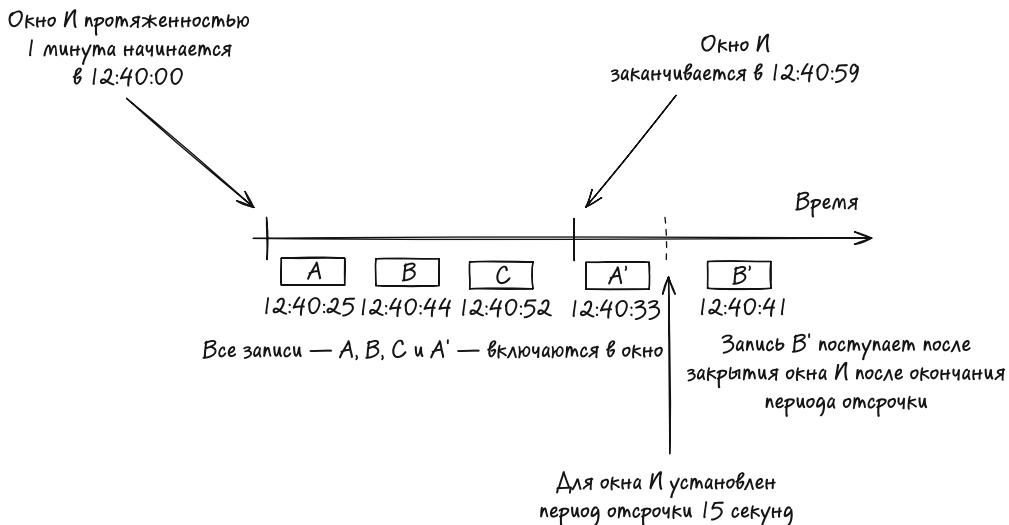
Запись, следующая не по порядку, — это такая запись, в которой отметка времени меньше (раньше), чем у предыдущей. Вернувшись к оконному агрегированию, легко заметить, как такие неупорядоченные данные могут повлиять на результаты их обработки. На рис. 9.25 показаны неупорядоченные данные и как они соотносятся с окнами.



**Рис. 9.25.** Неупорядоченные данные могут не попасть в окно, в которое они могли бы попасть, если бы поступали организованно, в установленном порядке

Нарушение порядка означает, что Kafka Streams исключит записи из анализа, которые должны были попасть в окно, но не попали, потому что к моменту их прибытия окно уже закрылось.

Но не будем переживать, потому что в Kafka Streams есть механизм учета неупорядоченных данных, который называется Grace. Выше в этой главе мы видели методы с именами `TimeWindows.ofSizeWithNoGrace` или `TimeWindows.ofSizeAndGrace`, и именно так применяется (или не применяется) механизм Grace к управлению окнами. Grace — это отсрочка, промежуток времени после закрытия окна, в течение которого разрешается добавить неупорядоченную запись в агрегат. Схема на рис. 9.26 иллюстрирует эту концепцию.



**Рис. 9.26.** Grace (отсрочка) — это период времени, в течение которого разрешается добавлять неупорядоченные записи в окно после его закрытия

Глядя на иллюстрацию, можно заметить, что Grace позволяет добавлять записи в агрегаты, в которые они были бы включены, если бы поступили вовремя, и обеспечивает более высокую точность расчетов. После истечения этого периода любые неупорядоченные записи считаются просроченными и удаляются. В случаях, когда желательно исключить из обработки любые неупорядоченные данные, следует использовать варианты `WithNoGrace` конструкторов окон.

Не существует никаких общих рекомендаций по выбору периода отсрочки и вообще его использования, вы должны рассматривать необходимость применения отсрочки в каждом конкретном случае. Но помните, что отсрочка помогает гарантировать, что вы получите самые точные результаты за счет включения в обработку записей, поступающих не по порядку.

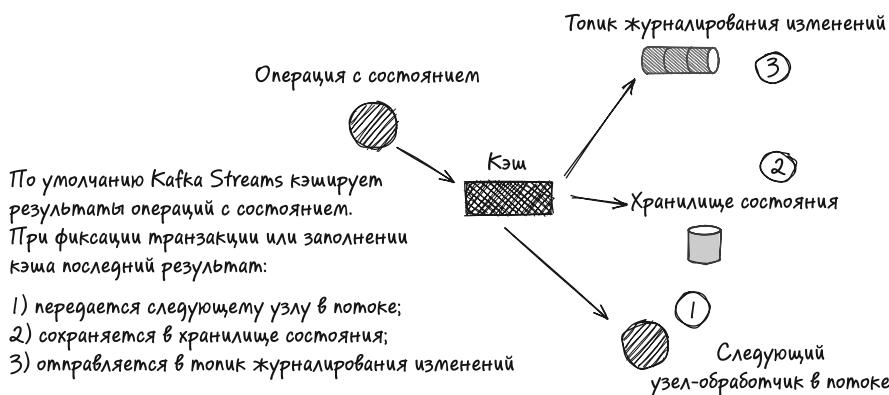
Теперь у нас налажено оконное агрегирование, и мы чувствуем прилив воодушевления от информации, которую предоставляет наше приложение. Но, немного

поразмыслив, мы обнаружили, что было бы проще анализировать поведение пользователей на сайте производителя потоковых конденсаторов и получать более надежные результаты, если бы получали их после закрытия сеанса.

Как рассказывалось в главе 7 при обсуждении операций агрегирования в KStream, Kafka Streams кэширует результаты и выдает обновления при фиксации (каждые 30 секунд) или при очистке кэша. Такое поведение применимо и к операциям оконного агрегирования. Но, учитывая, что окна имеют определенное начало и конец, у нас может появиться желание получить один окончательный результат после закрытия окна, и мы рассмотрим такую возможность в следующем разделе.

## 9.3. ОКОНЧАТЕЛЬНЫЕ РЕЗУЛЬТАТЫ ОКОННОГО АГРЕГИРОВАНИЯ

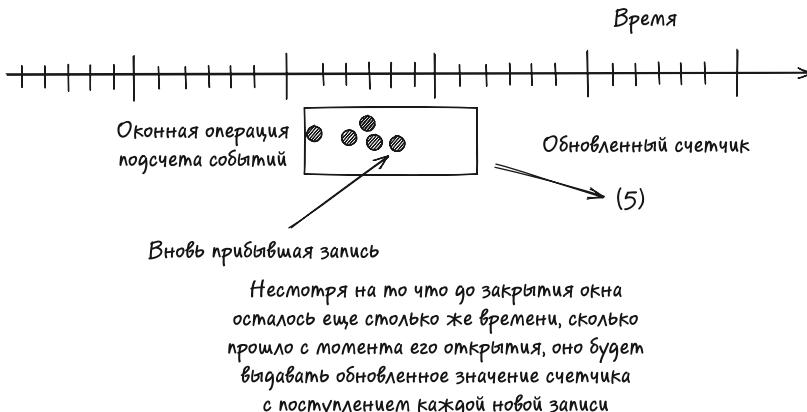
Как вы уже знаете, Kafka Streams не пересыпает автоматически результаты операций с состоянием, таких как агрегирование, после каждого обновления, а кэширует их. И только когда транзакция фиксируется или кэш заполняется, Kafka Streams отправляет последние результаты операции с состоянием и сохраняет записи в хранилище состояний. Рассмотрим схему на рис. 9.27, иллюстрирующую этот процесс.



**Рис. 9.27.** Kafka Streams кэширует результаты операций с состоянием и пересыпает их при фиксации транзакции или заполнении кэша

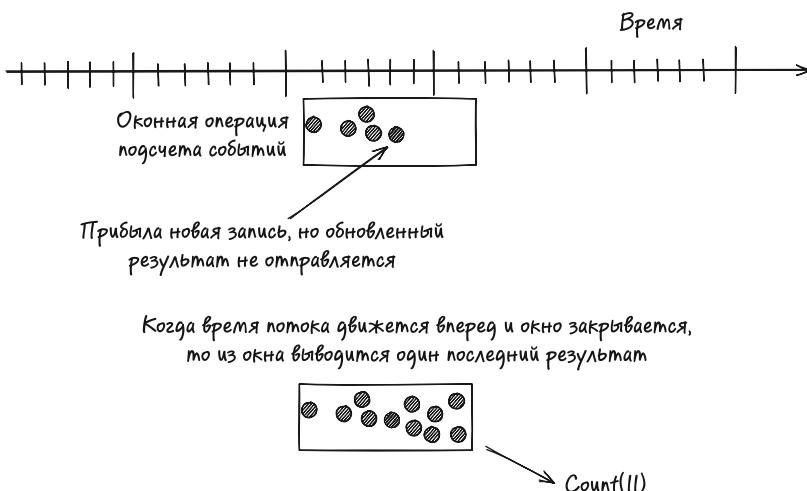
Результаты агрегирования продолжат накапливаться в кэше и в какой-то момент будут переданы следующим узлам-обработчикам. В этот момент Kafka Streams тоже сохраняет записи в хранилище состояний и в топике журнализации изменений.

Этому рабочему процессу следуют все операции с состоянием, включая оконные. Соответственно, узлы в потоке получат промежуточные результаты оконной операции, как показано на рис. 9.28.



**Рис. 9.28.** Оконные операции будут выдавать промежуточные результаты, даже если окно все еще открыто

Даже если окно еще не закрылось, оно может выдать обновленный результат при поступлении новой записи. Но иногда желательно, чтобы результат, окончательный результат, отправлялся только после завершения оконной операции. Схема на рис. 9.29 наглядно показывает, что это означает.



**Рис. 9.29.** Когда окно закрывается, в поток выдается только один результат

На этой схеме показан именно окончательный результат: операция не выдает никаких обновлений, пока окно не будет закрыто. На рисунке я упомянул время потока, и нам еще предстоит разобраться с этим термином в разделе 9.6, а пока просто имейте в виду, что Kafka Streams внутри ведет свой отсчет времени.

## ПРИМЕЧАНИЕ

Окончательные результаты доступны только для оконных операций. В потоковых приложениях обработки событий, таких как Kafka Streams, новые записи продолжают и продолжают появляться до бесконечности, поэтому никогда не наступит такой момент, который можно было бы назвать окончательным. Но оконное агрегирование представляет собой дискретный период, и последнюю запись, созданную к моменту закрытия окна, можно рассматривать как окончательный результат.

В Kafka Streams есть два варианта получения окончательного результата оконных операций. Один из них — использовать сравнительно недавно внедренную стратегию `EmitStrategy` или операцию `KTable.suppress`. Подход с `EmitStrategy` проще, поэтому я расскажу о нем в первую очередь.

В примерах оконных операций мы видели, что для выполнения агрегирования в окне нужно добавить вызов `windowedBy(..)` сразу после метода `KStream.groupByKey()`, как показано в листинге 9.20.

### Листинг 9.20. Заключение операции агрегирования в окно

```
iotHeatSensorStream.groupByKey()
    .windowedBy(TimeWindows.ofSizeWithNoGrace(
        Duration.ofMinutes(1)))
```

Этот небольшой фрагмент кода взят из предыдущего примера кувыркающегося окна в листинге 9.6. Если нам не нужны промежуточные результаты, то мы можем активировать стратегию `EmitStrategy` для окна, чтобы оно выдавало результат только после закрытия, как показано в листинге 9.21 (некоторые детали опущены для простоты).

### Листинг 9.21. Установка стратегии вывода результата в момент закрытия окна

```
iotHeatSensorStream.groupByKey()
    .windowedBy(TimeWindows.ofSizeWithNoGrace(
        Duration.ofMinutes(1)))
    .emitStrategy(EmitStrategy.onWindowClose()) ←
    .aggregate(...)
```

Устанавливает стратегию  
вывода результата  
в момент закрытия окна

С помощью единственной строки кода (`.emitStrategy(EmitStrategy.onWindowClose())`) мы настроили кувыркающееся окно на отправку результата только после закрытия окна.

Есть также стратегия `EmitStrategy.onWindowUpdate()`, но ее не нужно устанавливать явно, потому что она выбирается по умолчанию.

Теперь перейдем к другой форме получения окончательных результатов из оконных операций: подавлению. Пользоваться операцией `KTable.suppress` тоже просто, поскольку для этого также достаточно добавить одну строку кода (листинг 9.22).

### Листинг 9.22. Оператор suppression

```
.suppress(Suppressed.untilWindowCloses(
    StrictBufferConfig.unbounded())) ← Применение оператора suppress
```

Я опущу некоторые детали о `suppress` и расскажу о них позже, а пока рассмотрим, как применить этот оператор в приложении, выполняющем подсчет посещений страниц в сеансовом окне (некоторые детали опущены для простоты) (листинг 9.23).

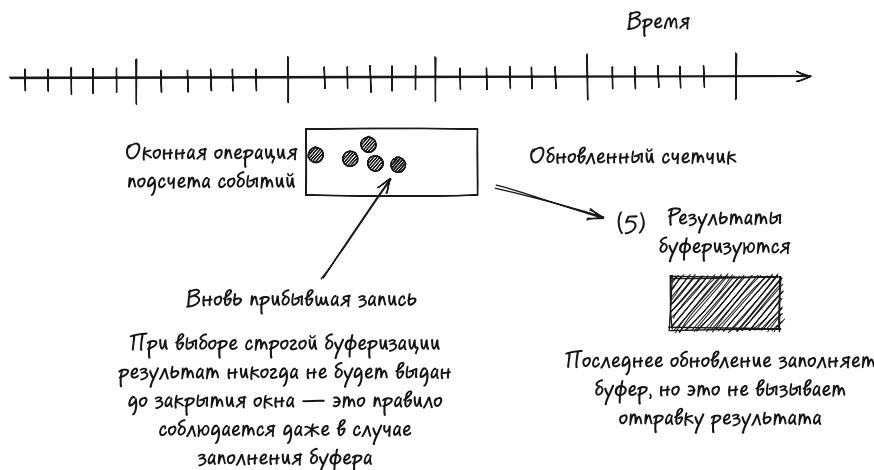
**Листинг 9.23.** Добавление подавления в операцию подсчета посещений страниц в пределах сеансового окна

```
KStream pageViewStream = builder.stream("page-view",
    Consumed.with(serdeString,pageViewSerde))
pageViewStream.groupByKey()
    .windowedBy(SessionWindow
        .ofInactivityGapWithNoGrace(
            Duration.ofMinutes(2))) ←
    .aggregate(HashMap::new,
        sessionAggregator,           Использовать сеансовое окно
        sessionMerger)              для операции агрегирования
    .suppress(Suppressed.untilWindowCloses(unbounded())) ←
    .toStream()
    .to("page-view-session-aggregates",
        Produced.with(windowedSerdes, pageViewAggregationSerde)) ←
        Добавление подавления для получения
        из сеансового окна только
        окончательного результата
```

Обратите внимание, что оператор `suppress` принимает параметр с конфигурационным объектом `Suppressed`, который сам получает параметр `BufferConfig`. Теперь вы знаете, как можно подавлять промежуточные результаты оконной операции. А теперь, когда я подкинул вам немножко новых сведений, давайте углубимся в особенности настройки и детали работы подавления.

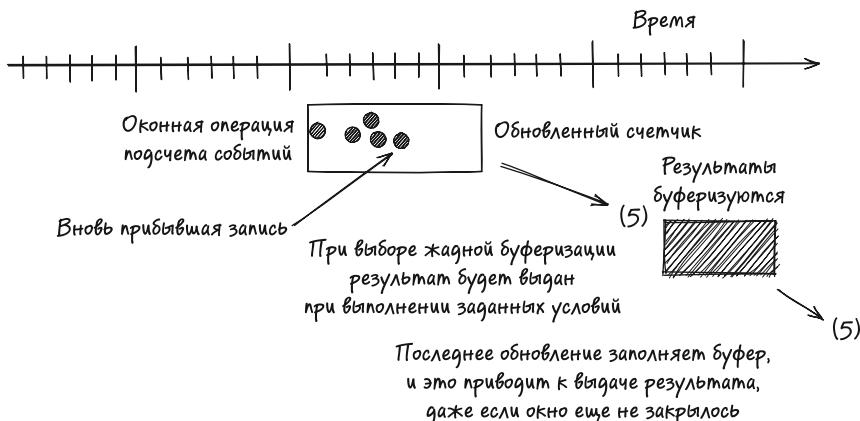
Механизм подавления в Kafka Streams буферизует промежуточные результаты по ключам до закрытия окна и появления окончательного результата. Буферизация выполняется в памяти, поэтому сразу же можно увидеть проблему: риск исчерпать всю свободную память, если сохранять все записи для всех ключей до момента закрытия окна. С другой стороны, некоторые промежуточные результаты можно отбрасывать, чтобы освободить память.

С учетом этих двух сценариев были реализованы два основных варианта подавления в Kafka Streams: со строгой и жадной буферизацией. Обсудим оба варианта, начав со строгого. Схема на рис. 9.30 поможет вам понять работу этого варианта.



**Рис. 9.30.** Строгая буферизация сохраняет все результаты и никогда не выдает их до закрытия окна

При выборе строгой буферизации результаты буферизуются по времени и гарантируется, что результат никогда не будет выдан раньше времени. Теперь посмотрим, что такое жадная буферизация и ее наглядное описание на рис. 9.31.



**Рис. 9.31.** Жадная буферизация сохраняет результаты до достижения указанного количества записей или объема в байтах и может выдавать результаты до закрытия окна

При выборе жадной буферизации результаты буферизуются по размеру (количеству байтов) или по количеству записей. В случае выполнения заданных условий оконная операция отправит результаты следующему узлу в потоке. Это способствует уменьшению количества записей, передаваемых через поток, но не гарантирует, что они представляют окончательный результат.

Строгий подход гарантирует отправку только окончательного результата и отсутствие дубликатов. Жадная буферизация тоже может дать окончательный результат, но не устраняет вероятность выдачи нескольких промежуточных результатов. Компромисс выбора между ними можно представить так: при строгой буферизации размер буфера не ограничен и существует возможность получить исключение `OutOfMemory` (OOM). При жадной буферизации, напротив, исключение OOM никогда не возникнет, но можно получить несколько результатов. Вероятность OOM может показаться пугающей, но если вы чувствуете, что буфер никогда не переполнится, то вполне можно использовать строгую буферизацию.

#### ПРИМЕЧАНИЕ

Вероятность исключения OOM в действительности намного ниже, чем кажется на первый взгляд. Все приложения на Java, использующие структуры данных в памяти — `List`, `Set` или `Map`, — потенциально могут вызвать OOM, если в эти структуры постоянно что-то добавляется. Для их эффективного использования требуется знать объем входных данных и объем имеющейся кучи.

Что касается рекомендаций по выбору варианта для использования, могу лишь сказать, что выбор зависит от конкретных требований. Я не могу предложить никаких

рекомендаций, но могу продемонстрировать некоторые сценарии, знакомство с которыми может помочь вам сделать выбор в ваших приложениях. Начнем с варианта строгой буферизации.

### 9.3.1. Строгая буферизация

Результаты подавления окон с неограниченным буфером мы уже видели выше. Эту конфигурацию подавления мы использовали в примере в подразделе 9.1.3 во всех сеансовых окнах, тем не менее рассмотрим ее еще раз. Для простоты я покажу не весь код агрегирования, а только часть, касающуюся подавления (с использованием статического импорта), однако в репозитории книги вы найдете полный код примера с различными конфигурациями подавления (листинг 9.24).

**Листинг 9.24.** Подавление всех обновлений до закрытия окна с неограниченным буфером

```
.suppress(untilWindowCloses(unbounded())) ←
    | Подавляет отправку любых результатов
    | до закрытия окна с неограниченным буфером
```

В этой конфигурации не накладывается никаких ограничений на буферизацию. Вывод результатов агрегирования продолжит подавляться до тех пор, пока окно не закроется, что гарантирует получение только окончательного результата.

Далее показана настройка варианта строгой буферизации с дополнительным параметром, определяющим размер буфера. Здесь мы указываем максимальный размер буфера и то, что в случае превышения заданного предела приложение должно быть остановлено.

Мы можем ограничить размер буфера, указав максимальное количество записей или байтов. В листинге 9.25 ограничивается максимальное количество записей в буфере.

**Листинг 9.25.** Настройка подавления для получения окончательного результата с завершением приложения при превышении заданного предела

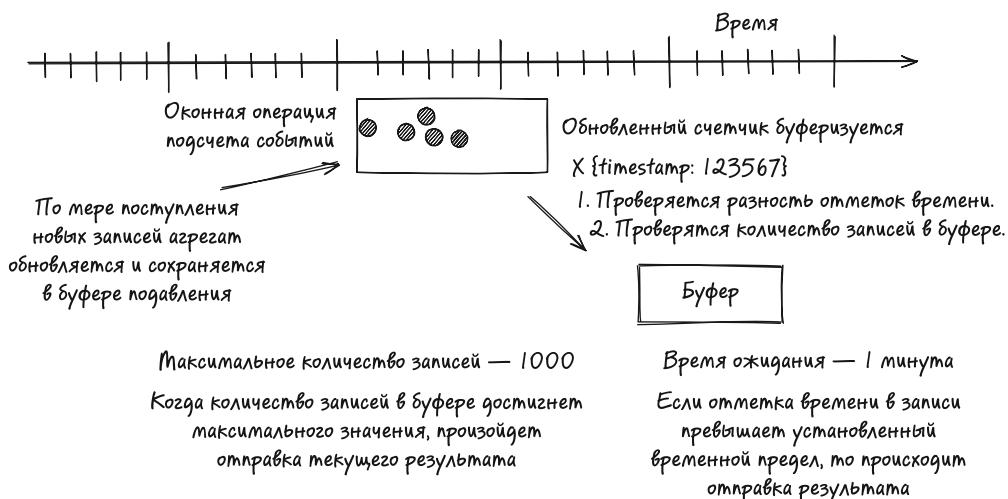
```
.suppress(untilWindowCloses(maxRecords(10_000))
        .shutDownWhenFull()) ←
    | Устанавливает максимальное
    | число записей, равное 10 000
    |
    | Указывает, что при превышении предела
    | приложение должно быть остановлено
```

Здесь мы указали максимальное количество записей, равное 10 000 и то, что, если число буферизованных записей превысит этот порог, приложение будет корректно закрыто. Параметр `BufferConfig.maxBytes` работает аналогично, за исключением того, что в нем нужно указать максимальное количество байтов вместо максимального количества записей.

Но остановка приложения, даже контролируемая, — спорный выбор. Далее мы рассмотрим жадную буферизацию, которую можно использовать в случаях, когда предпочтительнее получить промежуточный результат (то есть могут быть дубликаты) вместо остановки приложения.

### 9.3.2. Жадная буферизация

Выбирая жадную буферизацию, мы оказываемся на другой стороне медали. Пере-  
полнение буфера не вызовет остановку приложения, но операция агрегирования  
может выдать промежуточный результат. В этом примере демонстрируется несколько  
разных концепций, поэтому давайте сначала рассмотрим рис. 9.32, чтобы понять, как  
работает жадная буферизация.



**Рис. 9.32.** Выбирая жадную буферизацию, мы должны указать, как долго следует ждать поступления новой записи, прежде чем выдать результат

Отличие этого примера состоит в том, что здесь мы указываем, как долго ждать появление новой записи перед выдачей результата. Когда поступает новая запись, результат обновляется, но таймер не сбрасывается. Мы также указываем максимальный размер буфера, поэтому если он заполнится до истечения заданного времени ожидания, то оператор подавления отправит текущий результат. Теперь рассмотрим код, реализующий это поведение (листинг 9.26).

**Листинг 9.26.** Использование подавления с жадной буферизацией

```
.suppress(untilTimeLimit(Duration.ofMinutes(1)),
         maxRecords(1000)
         .emitEarlyWhenFull())
```

Задается время ожидания 1 минута перед отправкой результата

Устанавливает максимальный размер буфера равным 1000 записей

Определяет действие, которое должно быть выполнено при заполнении буфера

Одна из интересных особенностей жадной буферизации — возможность установить ограничение по времени, соответствующее размеру окна (плюс продолжительность отсрочки), поэтому, в зависимости от количества записей, есть шанс получить

окончательный результат, только когда окно закроется. Но если количество записей превысит размер буфера, то узел-обработчик отправит текущий результат независимо от того, достигнуто ли ограничение по времени. А если вы хотите, чтобы ограничение по времени соответствовало закрытию окна, то нужно включить период отсрочки, если таковой предусматривается, в ограничение по времени.

На этом мы завершаем обсуждение подавления вывода результатов из операций агрегирования в Kafka Streams. Примеры в этом разделе продемонстрировали только использование `KStream` и оконного агрегирования, но тот же принцип можно применить и к неоконным операциям агрегирования в `KTable`, используя оператор подавления с ограничением времени.

Теперь перейдем к последнему разделу этой главы — к отметкам времени в Kafka Streams.

## 9.4. ОТМЕТКИ ВРЕМЕНИ В KAFKA STREAMS

В главе 4 мы обсудили отметки времени в записях Kafka, а в этом разделе обсудим использование отметок времени в Kafka Streams. Отметки времени играют важную роль в следующих областях Kafka Streams:

- в соединении потоков;
- обновлении журнала изменений (`KTable API`);
- определении момента срабатывания метода `Processor.punctuate()` (`Processor API`);
- поведении окон.

Отметки времени в потоковой обработке можно разделить на три категории:

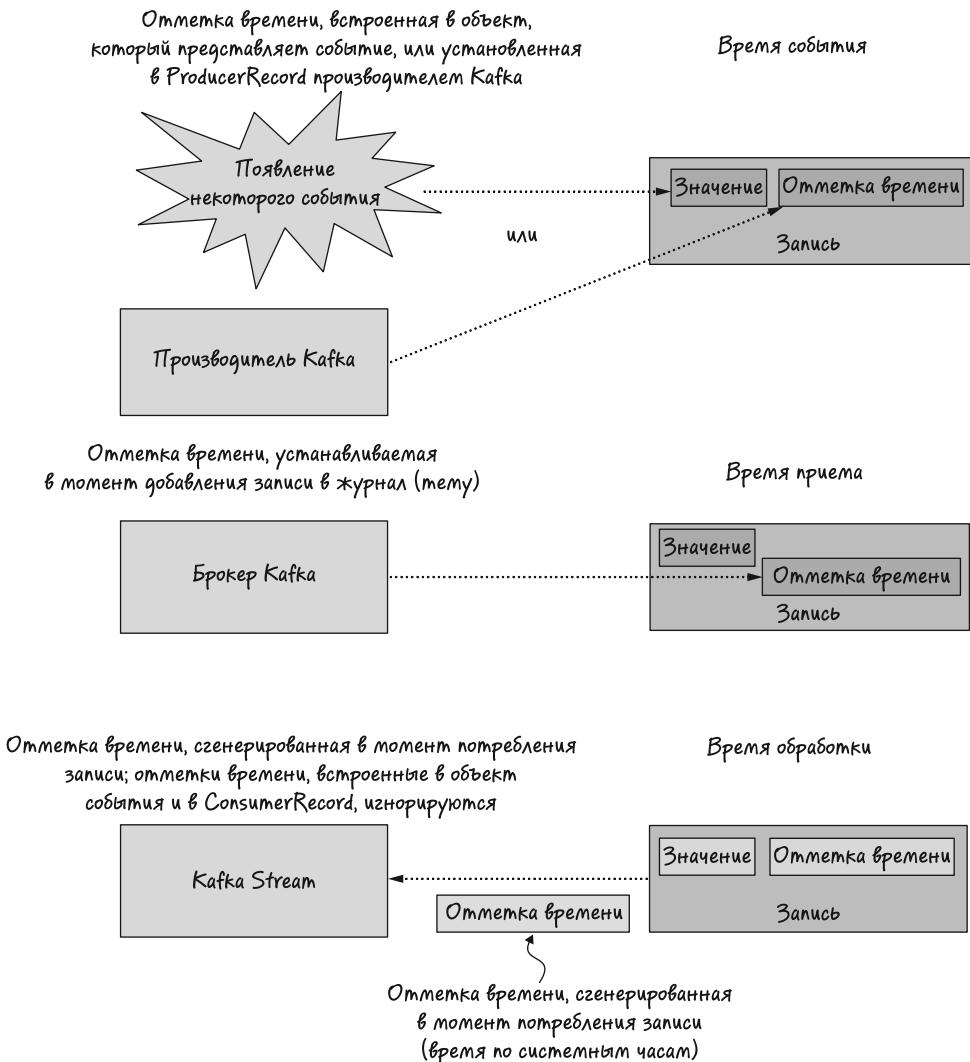
- *время события* — отметки времени, которые устанавливаются при возникновении события, обычно внутри объектов, представляющих события. К этой категории относятся также отметки времени, устанавливаемые при создании `ProducerRecord`;
- *время приема* — отметки времени, которые устанавливаются в момент поступления событий в конвейер обработки данных. К таковым относятся, например, отметки времени, устанавливаемые брокером Kafka (если задан конфигурационный параметр `LogAppendTime`);
- *время обработки* — отметки времени, которые устанавливаются, когда событие начинает проходить через конвейер обработки.

Все эти категории показаны на рис. 9.33.

В этом разделе вы увидите, как Kafka Streams позволяет выбрать ту или иную семантику отметок времени.

### ПРИМЕЧАНИЕ

До сих пор мы неявно предполагали, что клиенты и брокеры находятся в одном часовом поясе, но так бывает не всегда. При использовании отметок времени безопаснее всего нормализовать время приведением к часовому поясу UTC и тем самым устраниć путаницу с часовыми поясами, в которых работают брокеры и клиенты.



**Рис. 9.33.** Отметки времени в Kafka Streams делятся на три категории: время события, время приема и время обработки

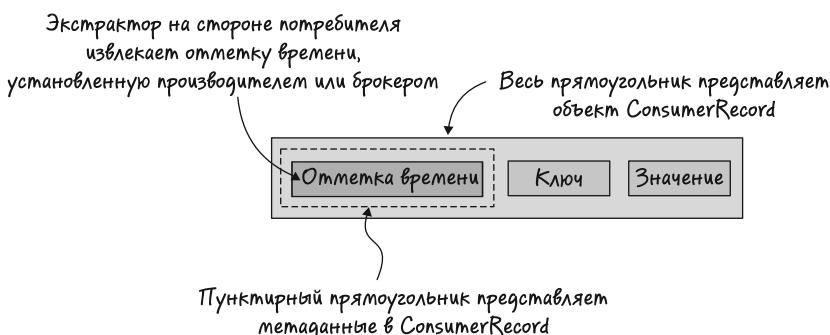
В большинстве случаев, когда используется семантика времени события, достаточно отметить времени, помещенной в метаданные `ProducerRecord`. Но иногда потребности могут отличаться. Вот некоторые примеры:

- сообщения, отправляемые в Kafka и имеющие отметки времени внутри объектов сообщений, становятся доступными производителю Kafka с некоторой задержкой, поэтому иногда бывает желательно учитывать только встроенную отметку времени;
- при обработке записей приложением Kafka Streams необходимо учитывать системное время, а не отметки времени в записях.

## 9.5. TIMESTAMPEXTRACTOR

Kafka Stream предоставляет интерфейс `TimestampExtractor` с одной абстрактной и четырьмя конкретными реализациями, позволяющий применять различные семантики обработки. Когда требуется использовать отметки времени, встроенные в значения в записях, можно создать свою реализацию `TimestampExtractor`. Давайте кратко рассмотрим встроенные экстракторы отметок времени и затем реализуем свой собственный.

Почти все готовые реализации `TimestampExtractor` работают с отметками времени, которые устанавливаются производителем или брокером в метаданных сообщения, тем самым обеспечивая использование семантики времени события (отметки времени, установленной производителем) или времени обработки (отметки времени, установленной брокером). На рис. 9.34 показано извлечение отметки времени из объекта `ConsumerRecord`.



**Рис. 9.34.** Отметка времени в объекте `ConsumerRecord`: устанавливается производителем или брокером, в зависимости от конфигурации

Обычно предполагается, что по умолчанию к отметкам времени применяется семантика времени события, которая задается настройкой `CreateTime`, но если использована настройка `LogAppendTime`, то будет возвращаться значение отметки времени, созданной брокером Kafka при добавлении записи в журнал (семантика времени приема). `ExtractRecordMetadataTimestamp` — это абстрактный класс с базовой функциональностью для извлечения отметки времени из метаданных в `ConsumerRecord`. Большинство конкретных реализаций расширяют этот класс. Реализации переопределяют абстрактный метод `ExtractRecordMetadataTimestamp.onInvalidTimestamp` для обработки недействительных отметок времени (когда отметка времени меньше 0).

Вот список классов, расширяющих класс `ExtractRecordMetadataTimestamp`:

- `FailOnInvalidTimestamp` — генерирует исключение для недопустимой отметки времени;
- `LogAndSkipOnInvalidTimestamp` — возвращает недопустимую отметку времени и выводит предупреждающее сообщение о том, что Kafka Streams отклонит запись из-за недопустимого значения отметки времени;
- `UsePartitionTimeOnInvalidTimestamp` — если отметка времени недопустима, то создается новая отметка времени на основе текущего времени потока.

Мы познакомились со встроенными экстракторами отметок времени событий, но есть еще один экстрактор, который мы рассмотрим далее.

### 9.5.1. Метод

#### **WallclockTimestampExtractorSystem.currentTimeMillis()**

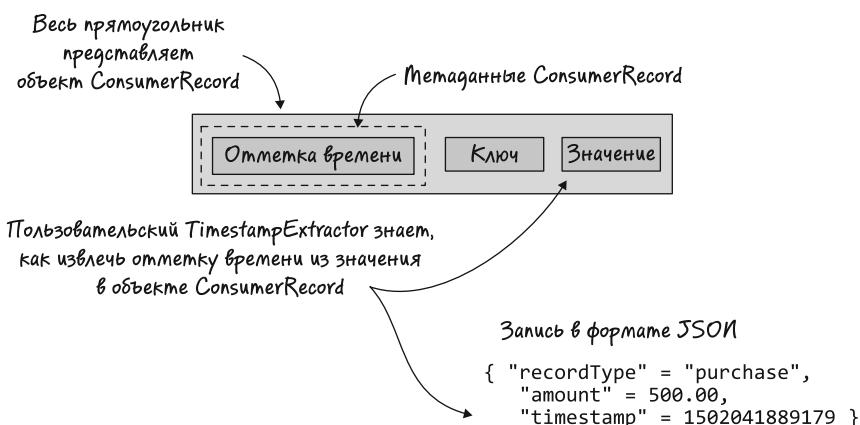
`WallclockTimestampExtractor` реализует семантику времени обработки. Он не извлекает никаких отметок времени и просто возвращает текущее время в миллисекундах, вызывая метод `System.currentTimeMillis()`. Класс `WallclockTimestampExtractor` можно использовать, когда нужна семантика времени обработки.

На этом мы закончили знакомство с готовыми экстракторами отметок времени и теперь посмотрим, как создать свою версию.

### 9.5.2. Пользовательский экстрактор отметок времени

Для работы с отметками времени (или их вычисления) в значении объекта из `ConsumerRecord` необходим пользовательский экстрактор, реализующий интерфейс `TimestampExtractor`. Например, предположим, что мы работаем с датчиками IoT и часть информации — это точное время замера, выполненного датчиком. Нашим вычислениям нужна точная отметка времени, а для этого необходимо использовать ту, что встроена в запись, отправленную в Kafka, а не ту, которую установил производитель.

На рис. 9.35 показан пример использования отметки времени, встроенной в значение в объекте события, вместо отметки, установленной Kafka (производителем или брокером).



**Рис. 9.35.** Пользовательский `TimestampExtractor` возвращает отметку времени, полученную на основе значения, содержащегося в `ConsumerRecord`. Эта отметка времени может быть существующим значением или вычисляться из свойств объекта события

В листинге 9.27 показан пример реализации `TimestampExtractor` (находится в `src/main/java/bbejeck/chapter_9/timestamp_extractor/TransactionTimestampExtractor.java`).

**Листинг 9.27.** Пользовательская реализация TimestampExtractor

```

public class TransactionTimestampExtractor implements TimestampExtractor {
    @Override
    public long extract(ConsumerRecord<Object, Object>
                        consumerRecord,
                        long partitionTime) {
        PurchaseTransaction purchaseTransaction =
            (PurchaseTransaction) consumerRecord.value(); ←
        return purchaseTransaction.transactionTime(); ←
    }
}

```

Извлекает объект  
PurchaseTransaction из пары  
«ключ — значение»,  
отправленной в Kafka

Возвращает отметку  
времени, зафиксированную  
в момент продажи

В примере соединения мы использовали свою реализацию `TimestampExtractor` для получения отметок времени фактического времени покупки. Такой подход позволяет соединять записи даже в случае задержек доставки записей.

### 9.5.3. Выбор `TimestampExtractor` для использования

Теперь, обсудив работу экстракторов отметок времени, настроим приложение и укажем ему, какой из экстракторов использовать. Задать экстрактор отметок времени можно двумя способами.

Первый — задать глобальный экстрактор, указав его в параметрах настройки приложения Kafka Streams. Если глобальный параметр не задан, то по умолчанию будет использоваться `FailOnInvalidTimestamp.class`. Например, следующий код настраивает использование `TransactionTimestampExtractor` с помощью глобального свойства приложения:

```
props.put(StreamsConfig.DEFAULT_TIMESTAMP_EXTRACTOR_CLASS_CONFIG,
         TransactionTimestampExtractor.class);
```

Второй вариант — передать экземпляр `TimestampExtractor` через объект `Consumed`:

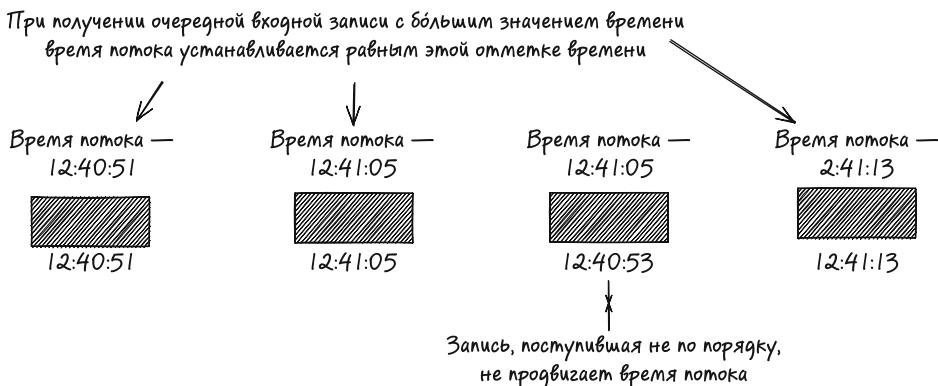
```
Consumed.with(Serdes.String(), purchaseSerde)
        .withTimestampExtractor(new TransactionTimestampExtractor())
```

Второй способ позволяет применять разные реализации `TimestampExtractor` для разных источников данных, тогда как первый определяет, какой экземпляр `TimestampExtractor` будет использоваться во всем приложении.

## 9.6. ВРЕМЯ ПОТОКА

Прежде чем закончить эту главу, я хочу обсудить слежение за временем в Kafka Streams во время обработки, то есть использование времени потока. Время потока — это не категория отметок времени, а текущее время в узле-обработчике Kafka Streams. Поскольку Kafka Streams выбирает следующую запись для обработки по отметке времени, значения будут последовательно увеличиваться. Время потока — это самая важная отметка времени, которую видит узел-обработчик и которая позволяет тому узнать текущее время. Поскольку приложение Kafka Streams разбивается на задачи, а задача отвечает за обработку записей из закрепленного за ней раздела, значение времени потока не является единым для всего приложения Kafka Streams, оно уникально на уровне задачи (то есть раздела).

Время потока движется только вперед. Записи, поступающие не по порядку, всегда обрабатываются, за исключением случаев превышения времени отсрочки в оконных операциях, но их отметки времени не влияют на время потока. На рис. 9.36 показано, как работает время потока в приложении Kafka Streams.



**Рис. 9.36.** Время потока — это наибольшая отметка времени, зафиксированная на данный момент, и она же является текущим временем приложения

Как показано на иллюстрации, текущее время приложения течет вперед по мере прохождения записей по топологии, при этом записи, следующие не по порядку, продолжают проходить через приложение, но не изменяют время потока.

Время потока имеет решающее значение для корректности оконных операций, поскольку окно продвигается и закрывается только по мере течения времени потока. Если в топиках, служащих источниками данных для приложения, записи появляются от случая к случаю, то можно столкнуться с ситуацией отсутствия наблюдаемых оконных результатов. Взгляните на рис. 9.37, чтобы понять, что происходит.



**Рис. 9.37.** Время потока определяет поведение окон, поэтому при низкой активности можно не заметить обновления окон

Такое очевидное отсутствие обработки объясняется отсутствием достаточного количества входных записей,двигающих время потока вперед и обеспечивающих выполнение вычислений в окне.

Тестируя приложения, важно помнить о влиянии отметок времени на работу Kafka Streams и о том, что ручная настройка значений отметок времени может помочь проверить интересующие вас черты поведения. Мы затронем вопрос использования отметок времени для тестирования в главе 14, посвященной тестированию. Время потока также играет важную роль при использовании пунктуаторов, о которых рассказывается в следующей главе, посвященной Processor API.

## ИТОГИ ГЛАВЫ

- Оконная обработка — это способ вычисления агрегированных значений за заданный период времени. Как и все другие операции в Kafka Streams, новые входные записи порождают обновления, которые передаются далее по потоку. С другой стороны, оконные функции могут использовать подавление, чтобы выдать единственный окончательный результат в момент закрытия окна. Оконные операторы Kafka Streams также предлагают метод `emitStrategy`, позволяющий задавать стратегию выдачи окончательных результатов. Этот метод проще в применении, чем механизм подавления.
- Существует четыре типа окон: прыгающие, кувыркающиеся, скользящие и сеансовые. Прыгающие и кувыркающиеся окна охватывают фиксированный временной интервал. Скользящие окна могут менять свой размер, в зависимости от поведения записей, включаемых в окно. Сеансовые окна полностью управляются поведением записей, и окно может продолжать расти, пока интервал между входными записями не превысит установленный период бездействия.
- Отметки времени влияют на поведение приложения Kafka Streams, и это влияние особенно заметно в оконных операциях, где отметки времени в записях управляют открытием и закрытием операций. Время потока — это наибольшая отметка времени, наблюдаемая приложением Kafka Streams во время обработки.
- Kafka Streams предоставляет различные экземпляры `TimestampExtractor`, что позволяет использовать в приложениях различную семантику отметок времени — время события, время добавления в журнал или время обработки.

# 10

## API узлов-обработчиков

### В этой главе

- ✓ Что лучше: абстракции более высокого уровня или больше возможностей контроля.
- ✓ Создание топологии с использованием источников, узлов-обработчиков и приемников.
- ✓ Углубляемся в API узлов-обработчиков на примере узла финансовой аналитики.
- ✓ Создаем узел совместной группировки.
- ✓ Интеграция API узлов-обработчиков и Kafka Streams API.

До сих пор мы работали в этой книге с высокоуровневым API Kafka Streams. Но именно DSL дает возможность разработчикам создавать надежные приложения с помощью минимального количества кода. Возможность быстрой компоновки топологий обработки — важная особенность DSL Kafka Streams. Благодаря этой возможности можно быстро воплощать в жизнь свои идеи по обработке данных, не увязая в запутанных настройках вроде тех, что бывают у других фреймворков.

В какой-то момент даже при использовании самых лучших инструментов вы натолкнетесь на какой-нибудь нестандартный случай: задачу, которая требует отхода от привычного пути. Как бы то ни было, вам придется перейти на уровень ниже и создать код, который просто невозможно написать с использованием абстракций более высокого уровня.

Классический пример компромисса между повышением уровня абстракции и расширением возможностей контроля — использование фреймворков объектно-

реляционного отображения (object-relational mapping, ORM). Хороший фреймворк ORM отображает объекты предметной области в таблицы базы данных и создает нужные SQL-запросы во время выполнения программы. Когда приложение использует SQL-операции не выше среднего уровня сложности (простые операторы `SELECT` или `JOIN`), фреймворки ORM экономят массу времени. Но, как бы хорош ORM-фреймворк ни был, всегда найдется хотя бы несколько запросов (очень сложные соединения, операторы `SELECT` со вложенными подзапросами), которые просто не работают так, как вам нужно. Вам придется самим написать код SQL для получения информации из базы данных в нужном формате. Здесь и возникает необходимость компромисса между повышением уровня абстракции и расширением возможностей контроля. Зачастую можно сочетать свой код SQL с высокоуровневыми отображениями, реализуемыми фреймворком.

Эта глава посвящена тем случаям, когда требуется такая потоковая обработка, которую использование DSL Kafka Streams только усложняет. Например, как вы видели при работе с `CTable`, фреймворк управляет временем отправки записей далее по конвейеру. Но вам может понадобиться явно управлять отправкой записей. Например, при отслеживании сделок на Уолл-стрит вам нужно отправлять записи далее только в случае превышения ценой акции определенного порогового значения. Для подобного контроля можно задействовать API узлов-обработчиков (Processor API). Хотя API узлов-обработчиков повышает сложность разработки, он предоставляет более широкие возможности. С его помощью можно создавать пользовательские узлы-обработчики, которые выполняют практически все, что только может вам понадобиться.

Из этой главы вы узнаете, как с помощью API узлов-обработчиков решать задачи, непосильные Kafka Streams DSL:

- планировать выполнение каких-либо действий через равные промежутки времени (на основе меток даты/времени записей или системного времени);
- получить полный контроль над тем, когда записи отправляются далее по конвейеру;
- отправлять записи конкретным дочерним узлам;
- создавать отсутствующую в Kafka Streams API функциональность (я покажу пример при создании узла совместной группировки).

Сначала посмотрим на использование API узлов-обработчиков на примере создания топологии.

## 10.1. СОЗДАНИЕ ТОПОЛОГИИ С ИСПОЛЬЗОВАНИЕМ ИСТОЧНИКОВ, УЗЛОВ-ОБРАБОТЧИКОВ И СТОКОВ

Допустим, вы владелец успешной пивоварни (Pops Hops) с несколькими точками продаж. Недавно вы начали принимать онлайн-заказы от оптовых торговцев, включая сбыт за рубежом, в Европе. Вам нужно организовать маршрутизацию заказов внутри компании, в зависимости от того, местный заказ или международный, с преобразованием валюты любых продаж в рамках ЕС из фунтов стерлингов в евро или доллары США.

Упрощенная схема такого технологического процесса будет выглядеть примерно так, как показано на рис. 10.1. При реализации этого примера вы увидите, насколько

гибко позволяет API узлов-обработчиков выбирать конкретные дочерние узлы для пересылки записей. Начнем с создания узла-источника.

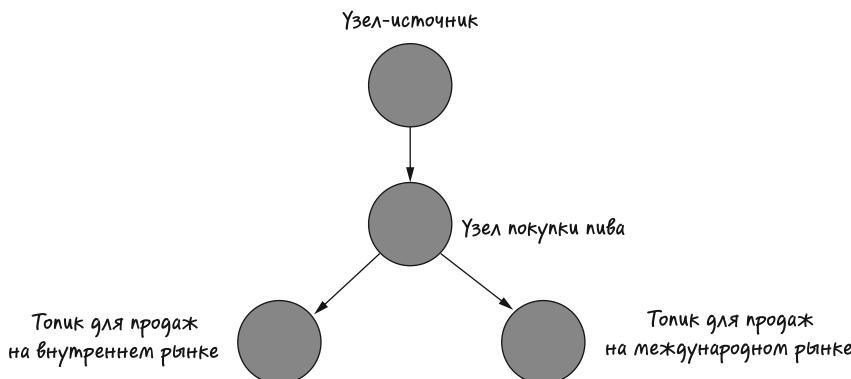


Рис. 10.1. Конвейер сбыта пива

### 10.1.1. Добавление узла-источника

Первый шаг формирования топологии — создание узлов-источников. В листинге 10.1 (код которого можно найти в файле `src/main/java/bbejeck/chapter_10/PopsHopsApplication.java`) создается узел-источник для нашей новой топологии.

**Листинг 10.1.** Создание узла-источника нашего «пивного» приложения

```

Topology topology = new Topology();
topology.addSource(LATEST,           ← Задает режим сброса
                    purchaseSourceNodeName,   ← Задает имя узла
                    new UsePartitionTimeOnInvalidTimestamp(),   ← Задает TimestampExtractor
                    stringDeserializer,      ← Задает десериализатор ключей
                    beerPurchaseDeserializer,   ← Задает десериализатор
                    INPUT_TOPIC)             ← Задает имя топика, откуда
                                            будут потребляться данные
  
```

Метод `Topology.addSource()` принимает несколько параметров, которые не использовались нами в DSL. Во-первых, мы указываем имя узла-источника. В Kafka Streams DSL передавать имя не требовалось, потому что имя для узла генерировал экземпляр `KStream`. Но при использовании API узлов-обработчиков необходимо явно указывать имена узлов топологии. Имена узлов применяются для привязки дочерних узлов к родительским. Кроме того, в DSL необязательные конфигурационные параметры узла-источника передаются через объект `Consumed`, а в API узлов обработчиков они передаются напрямую с помощью различных перегруженных версий метода `Topology.addSource`.

Далее мы указываем экстрактор отметок времени для использования с этим источником. В данном примере используется класс `UsePartitionTimeOnInvalidTimestamp`, который извлекает отметку времени, анализирует ее и, если она недействительная (имеет отрицательное значение), возвращает максимальное значение отметки

времени, полученной к данному моменту (время потока). Но в большинстве случаев можно ожидать, что отметки времени будут действительными.

#### ПРИМЕЧАНИЕ

При создании узла-источника с помощью API узлов-обработчиков требуется указать только два обязательных параметра: имя узла-обработчика и имя или шаблон топика.

Затем мы указали десериализаторы ключей и значений — еще одно отличие от Kafka Streams DSL. В DSL при создании узлов-источников и узлов-приемников мы передавали экземпляры Serde. Объект Serde сам содержит сериализатор и десериализатор, а Kafka Streams DSL использует нужный в зависимости от того, требуется ли преобразование объекта в массив байтов или обратно. А поскольку API узлов-обработчиков — абстракция более низкого уровня, необходимо явно указывать десериализатор при создании узла-источника и сериализатор при создании узла-приемника. Наконец, мы указали имя входного топика.

Теперь посмотрим, как добавить узлы-обработчики, выполняющие некоторые операции с входными записями.

### 10.1.2. Добавление узла-обработчика

Сейчас мы добавим узел для обработки записей, поступающих из узла-источника (соответствующий код можно найти в файле `src/main/java/bbejeck/chapter_10/PopsHopsApplication.java`) (листинг 10.2). Но прежде, чем обсуждать функциональность узла-обработчика, посмотрим, как подключать их.

#### Листинг 10.2. Добавление узла-обработчика

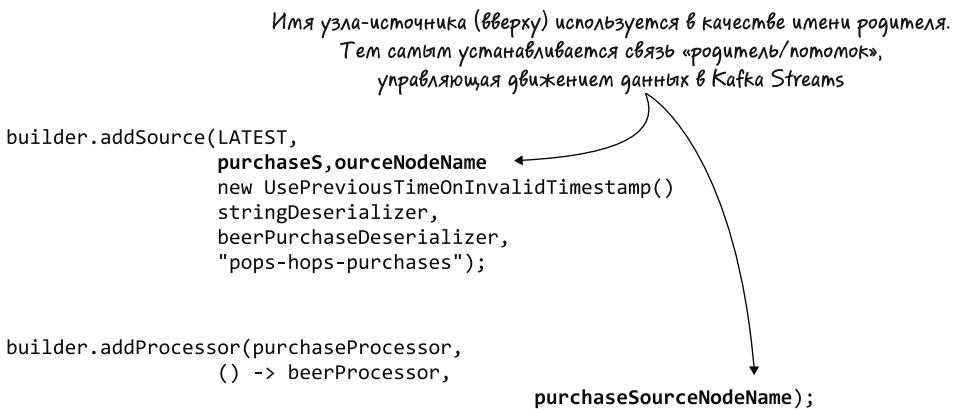
```
Map<String, Double> conversionRates = Map.of("EURO", 1.1, "POUND", 1.31);

topology.addSource(LATEST,
    purchaseSourceNodeName,
    new UsePreviousTimeOnInvalidTimestamp(),
    stringDeserializer,
    beerPurchaseDeserializer,
    INPUT_TOPIC)
    .addProcessor(purchaseProcessor,           ← Присваивает
        () -> new BeerPurchaseProcessor(      ← имя узлу-обработчику
            domesticSalesSink,                ← Аргументы конструктора
            internationalSalesSink,           ← узла-обработчика
            conversionRates),
            purchaseSourceNodeName)           ← Имя родительского узла
```

В этом коде для формирования топологии используется паттерн «текущего» интерфейса. Отличие от Kafka Streams API заключается в возвращаемом типе. В Kafka Streams API любой вызов оператора `KStream` возвращает новый экземпляр `KStream` или `KTable`. В API узлов-обработчиков же любое обращение к `Topology` возвращает тот же экземпляр `Topology`.

В предыдущем коде передается `ProcessorSupplier`. Метод `Topology.addProcessor` принимает во втором параметре экземпляр интерфейса `ProcessorSupplier`, но, поскольку `ProcessorSupplier` — интерфейс с одним методом, его можно заменить лямбда-выражением. Важнее всего в этом разделе то, что третий параметр,

`purchaseSourceNodeName`, метода `addProcessor()` совпадает со вторым параметром метода `addSource()`, как показано на рис. 10.2.



**Рис. 10.2.** Связываем родительский и дочерний узлы в API узлов-обработчиков

Благодаря этому между узлами устанавливается связь типа «родитель/потомок», которая, в свою очередь, определяет путь перемещения записей от одного узла-обработчика к другому в приложении Kafka Streams.

Теперь взгляните на рис. 10.3, где показано, что мы уже создали.



**Рис. 10.3.** Топология API узлов-обработчиков на текущий момент, включая имена узлов и родителей

Уделим немного времени обсуждению функций узла-обработчика `BeerPurchaseProcessor`, созданного в листинге 10.2. У него есть две задачи:

- преобразование сумм продаж на внешнем рынке (в евро) в доллары США;
- маршрутизация записи к соответствующему узлу-приемнику в зависимости от типа продажи (за рубежом или на внутреннем рынке).

Все эти действия происходят в методе `process()`. Подытожим вкратце, какие действия он производит:

- проверяет тип валюты. Если сумма указана не в долларах — переводит в доллары;
- если речь идет о продаже за рубежом — направляет обновленную запись в топик `international-sales`;
- в противном случае отправляет запись непосредственно в топик `domestic-sales`.

В листинге 10.3 показан код для этого узла-обработчика (его можно найти в файле `src/main/java/bbejeck/chapter_10/processor/BearPurchaseProcessor.java`).

### Листинг 10.3. Узел покупки пива (BeerPurchaseProcessor)

```
public class BeerPurchaseProcessor extends
    ContextualProcessor<String, BeerPurchase, String, BeerPurchase> {

    private final String domesticSalesNode;
    private final String internationalSalesNode;
    private final Map<String, Double> conversionRates;

    public BeerPurchaseProcessor(String domesticSalesNode,
                                String internationalSalesNode,
                                Map<String, Double> conversionRates) {
        this.domesticSalesNode = domesticSalesNode;
        this.internationalSalesNode = internationalSalesNode; | Задает имена
        this.conversionRates = conversionRates; | различных узлов,
    } | которым будут
        | передаваться записи

    @Override
    public void process(
        Record<String, BeerPurchase> beerPurchaseRecord) { | Метод process(), где происходят
            | основные действия

        BeerPurchase beerPurchase = beerPurchaseRecord.value();
        String key = beerPurchaseRecord.key();
        BeerPurchase.Currency transactionCurrency =
            beerPurchase.getCurrency();

        if (transactionCurrency != BeerPurchase.Currency.DOLLAR) {
            BeerPurchase.Builder builder =
                BeerPurchase.newBuilder(beerPurchase);
            double internationalSaleAmount = beerPurchase.getTotalSale();
            String pattern = "###.##";
            DecimalFormat decimalFormat = new DecimalFormat(pattern);
            builder.setCurrency(BeerPurchase.Currency.DOLLAR);
            builder.setTotalSale(Double.parseDouble(decimalFormat.format(
                convertToDollars(transactionCurrency.name(),
                    internationalSaleAmount)))); | Преобразует суммы продаж
            | за рубежом в доллары США
            Record<String, BeerPurchase> convertedBeerPurchaseRecord =
                new Record<>(key, builder.build(),
                    beerPurchaseRecord.timestamp());
            context().forward(convertedBeerPurchaseRecord, | С помощью объекта
                | ProcessorContext
                | (возвращаемого методом
                | context()) посыпает
                | записи узлу-приемнику
                | для международного рынка
                internationalSalesNode);
        } else {
            context().forward(
                beerPurchaseRecord,
                domesticSalesNode); | Отправляет записи с информацией о продажах
            | на внутреннем рынке в узел-приемник
            | для внутреннего рынка
        }
    }
}
```

Этот пример расширяет `ContextualProcessor` — класс, переопределяющий методы интерфейса `Processor`, за исключением метода `process()`. Именно в методе `Processor.process()` мы производим действия над проходящими через топологию записями.

## ПРИМЕЧАНИЕ

В интерфейсе `Processor` имеются методы `init()`, `process()`, `punctuate()` и `close()`. `Processor` — основная движущая сила логики любого потокового приложения, работающего с записями. В примерах мы в основном будем использовать класс `ContextualProcessor`, так что переопределять станем только необходимые нам методы. Класс `ContextualProcessor` сам инициализирует объект `ProcessorContext`, так что переопределять метод `init()` не нужно, разве что вам понадобится выполнить какие-либо дополнительные настройки.

Последние несколько строк в листинге 10.3 демонстрируют главное в этом примере — способность направлять записи конкретным дочерним узлам. Метод `context()` в данных строках извлекает ссылку на объект `ProcessorContext` для нужного узла-обработчика. Все узлы-обработчики топологии получают ссылки на `ProcessorContext` с помощью метода `init()`, выполняемого потоковой задачей (объектом `StreamTask`) при инициализации топологии.

Вы узнали, как обрабатывать записи, теперь нам нужно подключиться к узлу-приемнику (топику), чтобы передать записи обратно в Kafka.

### 10.1.3. Добавление узла-приемника

Сейчас вы уже, наверное, хорошо представляете себе схему применения API узлов-обработчиков. Для добавления источника мы воспользовались методом `addSource`, а для добавления узла-обработчика — методом `addProcessor`. Как нетрудно догадаться, для подключения узла-приемника (топика) к узлу-обработчику используется метод `addSink()`. На рис. 10.4 показана обновленная топология.



**Рис. 10.4.** Завершаем топологию, добавляя узлы-приемники

Теперь мы можем обновить нашу топологию, добавив код для узлов-приемников (его можно найти в файле `src/main/java/bbejeck/chapter_10/PopsHopsApplication.java`) (листинг 10.4).

#### Листинг 10.4. Добавление узла-стока

```
String domesticSalesSink = "domestic-beer-sales";
String internationalSalesSink = "international-beer-sales";

topology.addSource(LATEST,
    purchaseSourceNodeName,
    new UsePreviousTimeOnInvalidTimestamp(),
    stringDeserializer,
    beerPurchaseDeserializer,
    INPUT_TOPIC)
.addProcessor(purchaseProcessor,
    () -> new BeerPurchaseProcessor(
        domesticSalesSink,
        internationalSalesSink,
        conversionRates),
    purchaseSourceNodeName)
.addSink(internationalSalesSink, ← Имя узла-приемника
    "international-sales", ← Топик, который этот узел-приемник представляет
    stringSerializer, ← Сериализатор для ключей
    beerPurchaseSerializer, ← Сериализатор для значений
    purchaseProcessor) ← Родитель для этого узла-приемника

.addSink(domesticSalesSink, ← Имя узла-приемника
    "domestic-sales", ← Топик, который этот узел-приемник представляет
    stringSerializer, ← Сериализатор для ключей
    beerPurchaseSerializer, ← Сериализатор для значений
    purchaseProcessor); ← Родитель для этого
                           | узла-приемника
```

В листинге 10.4 мы добавили два узла-приемника, один — для информации о продажах на внутреннем рынке и второй — на международном. Записи будут заноситься в топик, соответствующий валюте транзакции.

Важно отметить, что имя родительского узла у обоих добавляемых узлов-приемников совпадает. Благодаря этому мы связали оба узла-приемника с одним нашим узлом-обработчиком (как показано на рис. 10.4).

В этом первом примере было показано, как создаются приложения Kafka Streams с использованием API узлов-обработчиков. Здесь мы рассмотрели конкретный пример, однако общие принципы добавления узла (-ов)-источника (-ов), одного или нескольких узлов-обработчиков и, наконец, узла (-ов)-приемника (-ов) остаются неизменными для любого приложения. Хотя API узлов-обработчиков требует несколько большего объема кода, чем Kafka Streams API, создание топологий с его помощью все равно остается несложной задачей. В следующем примере я покажу, насколько гибок API узлов-обработчиков.

## 10.2. УГЛУБЛЯЕМСЯ В API УЗЛОВ-ОБРАБОТЧИКОВ НА ПРИМЕРЕ УЗЛА БИРЖЕВОЙ АНАЛИТИКИ

Вернемся к сфере финансовых операций и примерим шляпу внутридневного трейдера. Внутридневному трейдеру необходимо анализировать изменения курсов акций, чтобы выбрать оптимальные моменты для покупки и продажи. Его задача — извлечь выгоду из колебаний рынка и сорвать быструю прибыль. Мы будем учитывать несколько ключевых показателей в надежде, что они укажут, когда вам имеет смысл начинать действовать.

Вот список требований:

- отображать текущий курс акций;
- показывать, повышается или понижается цена акции;
- отображать общее количество проданных на текущий момент акций, а также тренд изменения цены (понижение/повышение);
- отправлять записи далее по конвейеру только для акций с 2%-ным трендом (понижение/повышение);
- собирать не менее 20 выборок заданных акций, прежде чем производить с ними какие-либо вычисления.

Рассмотрим, как осуществить этот анализ вручную. На рис. 10.5 показан пример дерева решений, которое понадобится нам для принятия подобных решений.

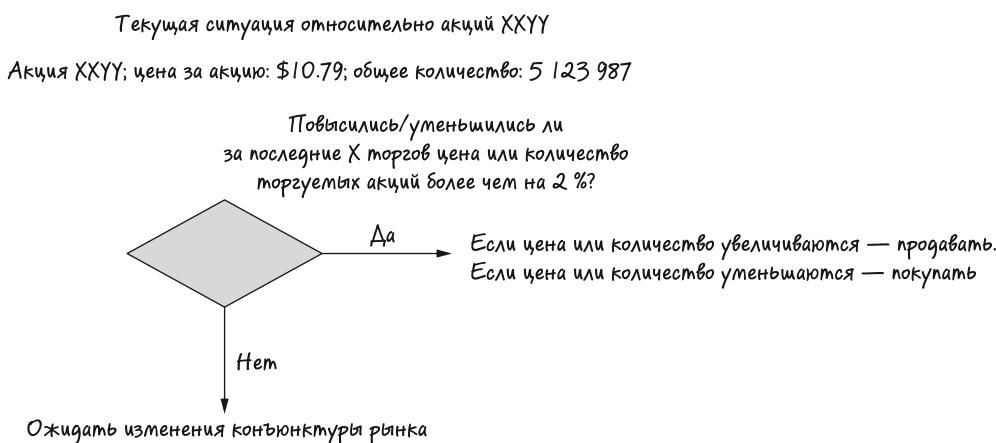


Рис. 10.5. Обновления тренда курса акций

Для нашего анализа необходимо провести несколько расчетов. Кроме того, мы воспользуемся результатами этих расчетов для определения того, нужно ли (и если да, то когда) отправлять записи далее по конвейеру.

Данное ограничение на отправку записей означает, что мы не можем положиться на стандартный механизм фиксации времени или сброса кэша на диск для управления их движением, что исключает использование Kafka Streams API. Само собой

разумеется, что нам придется сохранять состояние, чтобы отслеживать изменения в динамике. Нам необходима возможность создания пользовательских узлов-обработчиков. Посмотрим, как решить эту задачу.

### ТОЛЬКО ДЛЯ ДЕМОНСТРАЦИОННЫХ ЦЕЛЕЙ

Я совершенно уверен, что это и так понятно, но все равно констатирую очевидное: настоящие оценки курсов акций приводятся только для демонстрационных целей. Не делайте никаких рыночных прогнозов на основе этого примера. Данная модель абсолютно не похожа на реальные и представлена только для демонстрации более сложного сценария обработки. Я отнюдь не внутридневной трейдер!

## 10.2.1. Узел-обработчик показателей акций

В листинге 10.5 приведена реализация приложения для обработки показателей акций (его можно найти в файле `src/main/java/bbejeck/chapter_10/StockPerformanceApplication.java`).

**Листинг 10.5.** Приложение для обработки показателей акций с пользовательским узлом-обработчиком

```
Topology topology = new topology();
String stocksStateStore = "stock-performance-store";
double differentialThreshold = 0.02;           ← Задает порог изменения показателей
                                                акций, по достижении которого
                                                информация будет отправляться дальше

KeyValueBytesStoreSupplier storeSupplier =      ← Создается хранилище состояния в памяти
Stores.inMemoryKeyValueStore(stocksStateStore); ← в виде пар «ключ — значение»

StoreBuilder<KeyValueStore<String, StockPerformance>> storeBuilder
= Stores.keyValueStoreBuilder(
storeSupplier, Serdes.String(), stockPerformanceSerde); ← Создается объект StoreBuilder
                                                для вставки в топологию

topology.addSource("stocks-source",
                    stringDeserializer,
                    stockTransactionDeserializer,
                    "stock-transactions")
    .addProcessor("stocks-processor",          ← Добавляет узел-обработчик в топологию
                                                спомощью ProcessorSupplier
                    new StockPerformanceProcessorSupplier(storeBuilder),
                    "stocks-source")
    .addSink("stocks-sink",
                    "stock-performance",
                    stringSerializer,
                    stockPerformanceSerializer,           ← Добавляет узел-приемник
                                                для вывода результатов
                    "stocks-processor");
```

В этом примере использована конкретная реализация `ProcessorSupplier` вместо лямбда-выражения. Это связано с тем, что интерфейс `ProcessorSupplier` предоставляет метод `stores`, который автоматически подключит узел-обработчик к любым предоставленным экземплярам `StoreBuilder`. Листинг 10.6 содержит исходный код `StockPerformanceProcessorSupplier`.

**Листинг 10.6.** Реализация ProcessorSupplier

```

public class StockPerformanceProcessorSupplier
    implements ProcessorSupplier<String, Transaction,
                           String, StockPerformance> {
    StoreBuilder<?> storeBuilder;

    public StockPerformanceProcessorSupplier(StoreBuilder<?> storeBuilder) {
        this.storeBuilder = storeBuilder;
    }

    @Override
    public Processor<String, Transaction, String, StockPerformance> get() {
        return new StockPerformanceProcessor(storeBuilder.name()); ←
    }
}

@Override
public Set<StoreBuilder<?>> stores() {
    return Collections.singleton(storeBuilder); ←
}
}

```

Возвращает новый экземпляр  
StockPerformanceProcessor

Возвращает экземпляры StoreBuilder  
для подключения к узлу-обработчику

С помощью метода `ProcessorSupplier.stores` можно автоматически подключать экземпляры `StateStore` к узлам-обработчикам и тем самым упростить построение топологии, потому что в этом случае отпадает необходимость вызывать `Topology.addStateStore` с именами узлов, которым нужен доступ к хранилищу.

Технологический процесс в этой топологии такой же, как и в предыдущем примере, так что сосредоточим внимание на новых элементах узла-обработчика. В предыдущем примере не нужны никакие настройки, так что можно было положиться на инициализацию объекта `ProcessorContext` методом `ContextualProcessor.init`. В этом же примере, однако, мы собираемся использовать хранилище состояния, а также запланировать время отправки записей вместо пересылки их сразу при получении.

Рассмотрим сначала метод `init()` узла-обработчика (его можно найти в файле `src/main/java/bbejeck/chapter_10/processor/StockPerformanceProcessor.java`) (листинг 10.7).

**Листинг 10.7.** Задачи метода init()

```

@Override
public void init(ProcessorContext<String, StockPerformance> context) {
    super.init(processorContext);
    keyValueStore = context().getStateStore(stateStoreName); ←
    StockPerformancePunctuator punctuator =
        new StockPerformancePunctuator(differentialThreshold,
                                        context(),
                                        keyValueStore); ←
    context().schedule(10000,
                      PunctuationType.STREAM_TIME,
                      punctuator); ←
}

```

Инициализирует ProcessorContext через  
вызов суперкласса AbstractProcessor

Получает хранилище  
состояния, созданное  
при формировании топологии

Инициализирует объект Punctuator,  
отвечающий за выполнение обработки  
по расписанию

Планирует вызов Punctuator.  
punctuate() каждые 10 секунд

Во-первых, нам нужно инициализировать `ContextualProcessor` с помощью `ProcessorContext`, для чего вызывается метод `init()` суперкласса. Далее мы получаем ссылку на созданное в топологии хранилище состояния. Все, что от нас требуется, — сохранить ее в переменной для дальнейшего использования в узле-обработчике. В листинге 10.7 также впервые в этой книге использован `Punctuator` — интерфейс, представляющий собой обратный вызов для выполнения логики узла-обработчика в соответствии с расписанием, инкапсулированный в методе `Punctuator.punctuate()`.

#### **СОВЕТ**

Метод `ProcessorContext.schedule(long, PunctuationType, Punctuator)` возвращает тип `Cancellable`, что позволяет отменить выполнение пунктуации и реализовать более сложные сценарии, подобные тем, что перечислены в статье *Punctuate Use Cases*, <http://mng.bz/YSKF>. Я не стану приводить примеры в тексте или обсуждать здесь этот вопрос, но некоторые примеры вы можете найти в каталоге `src/main/java/bbejeck/chapter_10/cancellation`.

В последней строке в листинге 10.7 мы используем `ProcessorContext` (получая его вызовом метода `context()`), чтобы запланировать выполнение `Punctuator` каждые 10 секунд. Второй параметр, `PunctuationType.STREAM_TIME`, указывает, что метод `Punctuator.punctuate()` должен вызываться каждые 10 секунд, согласно отметкам времени в данных. Можно также выбрать вариант `PunctuationType.WALL_CLOCK_TIME`, и тогда `Punctuator.punctuate()` тоже будет вызываться каждые 10 секунд, но в соответствии с системным временем в окружении Kafka Streams. Выясним, в чем различие между этими двумя настройками `PunctuationType`.

### **10.2.2. Семантика пунктуации**

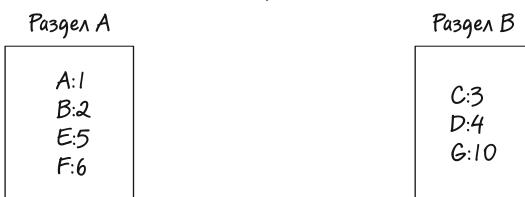
Начнем наш разговор о семантике пунктуации со `STREAM_TIME`, поскольку этот вариант требует некоторых пояснений. Рисунок 10.6 иллюстрирует понятие времени потока (stream time). Обратите внимание, что некоторые внутренние механизмы Kafka Streams здесь не показаны.

Обсудим некоторые детали, чтобы лучше понять, как происходит выполнение по расписанию.

- `StreamTask` извлекает из `PartitionGroup` наименьшую отметку времени. `PartitionGroup` — это набор разделов, назначенных `StreamThread`, который содержит всю информацию об отметках времени для всех разделов группы.
- Во время обработки записей `StreamThread` выполняет итерации по своему объекту `StreamTask`, при этом каждая задача в конце концов вызывает метод `punctuate` для каждого из своих подходящих для пунктуации узлов-обработчиков. Напомню, что нужно собрать не менее 20 сделок с конкретным видом акций, прежде чем оценивать его показатели.
- Если отметка времени с прошлого вызова `punctuate` (плюс запланированный промежуток времени) меньше извлеченной из `PartitionGroup` или равна ей, то Kafka Streams вызывает метод `punctuate` данного узла-обработчика.

В разделах A и B буквы означают записи, а числа — отметки времени.

В данном примере предполагается, что метод `punctuate` должен выполняться каждые 5 секунд



Первым выбирается раздел A:

- 1) процесс вызывается для записи A;
- 2) процесс вызывается для записи B

Теперь наименьшая отметка времени в разделе B:

- 3) процесс вызывается для записи C;
- 4) процесс вызывается для записи D

Возвращаемся к разделу A, потому что теперь наименьшая отметка времени находится в нем:

- 5) процесс вызывается для записи E;
- 6) вызывается метод `punctuate`, потому что, судя по отметкам времени, прошло 5 секунд;
- 7) процесс вызывается для записи E

Наконец возвращаемся к разделу B:

- 8) процесс вызывается для записи G;
- 9) опять вызывается метод `punctuate`, потому что, согласно отметкам времени, прошло еще 5 секунд

**Рис. 10.6.** Расписание пунктуации при использовании настройки STREAM\_TIME

Важно отметить, что в этом случае приложение увеличивает отметки времени с помощью `TimestampExtractor`, соответственно, вызовы метода `punctuate()` будут производиться регулярно, только если данные поступают с постоянной скоростью. Если же данные поступают от случая к случаю, то метод `punctuate()` не будет вызываться регулярно через заданные интервалы времени.

При применении варианта `PunctuationType.WALL_CLOCK_TIME`, напротив, метод `Punctuator.punctuate` будет вызываться более предсказуемо благодаря использованию системного времени. Отмечу, что семантика системного времени не дает гарантий точности — системное время меняется в пределах интервала опроса, и шаг изменения зависит от того, сколько времени нужно для выполнения цикла опроса. Поэтому в примере из листинга 10.7 можно ожидать, что пунктуация будет производиться через интервалы, близкие к десятисекундным, вне зависимости от темпов поступления данных.

Какой подход использовать — зависит исключительно от ваших потребностей. Если вам нужно регулярно выполнять какие-либо действия независимо от движения данных, то лучше использовать системное время. А если вычисления производятся только над входными данными и небольшой разрыв во времени между вызовами допустим, то попробуйте использовать семантику времени потока.

#### ПРИМЕЧАНИЕ

В версиях Kafka, предшествующих 0.11.0, для пунктуации использовался метод `ProcessorContext.schedule(long time)`, вызывавший, в свою очередь, метод `Processor.punctuate` через заданные промежутки времени. Такой подход работал только при семантике времени потока, и оба метода в настоящий момент считаются устаревшими. Я иногда упоминаю в этой книге устаревшие методы, но в примерах используются только новейшие методы пунктуации.

Теперь, рассмотрев вопросы планирования, выполнения и пунктуации, перейдем к обработке входных записей.

### 10.2.3. Метод `process()`

Именно в методе `process()` производятся все расчеты для оценки показателей акций. При получении записи необходимо выполнить несколько шагов:

- проверить хранилище состояния на наличие объекта `StockPerformance`, соответствующего биржевому символу акции из этой записи;
- создать соответствующий объект `StockPerformance`, если в хранилище состояния его нет. Далее экземпляр `StockPerformance` суммирует значения для текущей цены акции и количества проданных акций и обновляет результаты;
- после достижения количества 20 транзакций для какого-либо вида акций начать расчеты.

Хотя вопрос финансового анализа выходит за рамки данной книги, я предлагаю вкратце рассмотреть эти расчеты. Для получения цены за акцию и количества проданных акций мы будем вычислять простое скользящее среднее (simple moving average, SMA). В сфере биржевой торговли SMA используются для вычисления среднего значения по наборам данных размером  $N$ .

В этом примере  $N = 20$ . Это ограничение означает, что при поступлении новой информации о сделках будут собираться данные о цене за акцию и количестве проданных акций для первых 20 транзакций. По достижении этого порога самое старое значение будет удаляться и добавляться самое новое. С помощью SMA мы получаем скользящее среднее цены акции и числа проданных акций за последние 20 сделок. Важно отметить, что пересчитывать все данные при поступлении новых значений не требуется.

На рис. 10.7 перечислены шаги, которые выполняет метод `process()`. Именно в методе `process()` проводятся все вычисления.

1) цена: \$10,79, количество акций: 5000;  
2) цена: \$11,79, количество акций: 7000



20) цена: \$12,05, количество акций: 8000

По мере поступления данных об акциях вычисляется скользящее среднее цены акции и количества проданных акций за последние 20 сделок. Кроме того, фиксируется отметка времени последнего обновления

Пока не наберется 20 сделок, среднее вычисляется по имеющимся на данный момент сделкам

1) цена: \$10,79, количество акций: 5000;  
2) цена: \$11,79, количество акций: 7000



20) цена: \$12,05, количество акций: 8000;

21) цена: \$11,75, количество акций: 6500;

22) цена: \$11,95, количество акций: 7300

По достижении 20 сделок информация о самой старой сделке удаляется и добавляется информация о самой новой.

Кроме того, за счет удаления самого старого значения обновляется среднее

**Рис. 10.7.** Пошаговое выполнение метода process() для анализа динамики акций

Взглянем теперь на код метода process() (его можно найти в файле `src/main/java/bbejeck/chapter_10/processor/StockPerformanceProcessor.java`) (листинг 10.8).

#### Листинг 10.8. Реализация метода process()

```
@Override
public void process(String symbol, StockTransaction transaction) {
    StockPerformance stockPerformance = keyValueStore.get(symbol); ← Извлекает предыдущую статистику по динамике акций, возможно пустую

    if (stockPerformance == null) { ← Создает новый объект StockPerformance, если его нет в хранилище состояния
        stockPerformance = new StockPerformance();
    }

    stockPerformance.updatePriceStats(transaction.getSharePrice()); ← Обновляет статистику цены для этой акции
    stockPerformance.updateVolumeStats(transaction.getShares()); ← Обновляет статистику по количеству проданных акций этого вида
    stockPerformance.setLastUpdateSent(Instant.now()); ← Фиксирует отметку времени последнего обновления

    keyValueStore.put(symbol, stockPerformance); ← Помещает обновленный объект StockPerformance в хранилище состояния
}
```

В методе process() мы прибавляем цену и количество акций в последней сделке к значениям в объекте StockPerformance.

Этот код вычисляет две основные статистики: скользящее среднее и разность текущих и новых значений цены/количество акций. Мы не будем вычислять среднее, пока не соберем данные о 20 транзакциях, поэтому откладываем выполнение каких-либо действий до момента получения узлом-обработчиком данных о 20 сделках. Накопив информацию о 20 сделках для конкретного вида акций, можно вычислить первое среднее значение, а затем разделить текущее значение цены акции или числа проданных акций на скользящее среднее, преобразуя результат в процентное соотношение.

## ПРИМЕЧАНИЕ

Если вы захотите взглянуть на вычисления, то найдете код `StockPerformance` в файле `src/main/java/bejeck/chapter_10/StockPerformance.java`.

В примере с интерфейсом `Processor` в листинге 10.3 мы по завершении выполнения метода `process()` отправляли записи далее по конвейеру. В данном же случае мы сохраняем итоговые результаты в хранилище состояния и оставляем отправку записей методу `Punctuator.punctuate`.

### 10.2.4. Выполнение пунктуатора

Мы уже обсудили семантику пунктуации и выполнение по расписанию, так что перейдем прямо к изучению метода `Punctuator.punctuate` (который можно найти в файле `src/main/java/bejeck/chapter_10/processor/punctuator/StockPerformancePunctuator.java`) (листинг 10.9).

#### Листинг 10.9. Код пунктуации

```
@Override
public void punctuate(long timestamp) {
    try (KeyValueIterator<String, StockPerformance> performanceIterator
        = keyValueStore.all()) { ←
        ← Извлекает итератор для обхода
        ← всех пар «ключ — значение»
        ← в хранилище состояния

        while (performanceIterator.hasNext()) {
            KeyValue<String, StockPerformance> keyValue =
                performanceIterator.next();
            String key = keyValue.key;
            StockPerformance stockPerformance = keyValue.value;

            if (stockPerformance != null) {
                if (stockPerformance.getPriceDifferential() >=
                    differentialThreshold ||

                    stockPerformance.getShareDifferential() >=
                    differentialThreshold) { ←
                    ← Проверяет
                    ← достижение порога
                    ← для текущего вида
                    ← акций
                    ←
                    ← По достижении
                    ← или превышении
                    ← порога запись
                    ← отправляется
                    context.forward(new Record<>(key,
                        stockPerformance,
                        timestamp));
                }
            }
        }
    }
}
```

Последовательность действий в методе `Punctuator.punctuate` проста. Он перебирает в цикле пары «ключ — значение» в хранилище состояния и, если вычисленное значение превысило заданный порог, отправляет запись далее по конвейеру.

Обратите особое внимание, что в отличие от предыдущих примеров, где записи отправлялись дальше только после фиксации или сброса кэша на диск, теперь мы сами определяем, когда это будет происходить. Кроме того, предполагаемое выполнение этого кода каждые 10 секунд не гарантирует фактическую отправку записей — результат вычислений должен достичь установленного порога. Отмечу также, что методы `Processor.process` и `Punctuator.punctuate` не выполняются параллельно.

## ПРИМЕЧАНИЕ

Хотя мы рассматриваем возможности доступа к хранилищу состояния, не помешает вспомнить архитектуру Kafka Streams и обсудить несколько важных нюансов. У каждой задачи `StreamTask` есть своя копия локального хранилища состояния, и объекты `StreamThread` не имеют совместно используемых задач или данных. По мере перемещения по топологии записи последовательно посещают каждый узел, двигаясь от родителя к потомку, а это значит, ни один узел-обработчик не использует хранилище состояния одновременно с другими узлами.

Этот пример — превосходное введение в написание пользовательских узлов-обработчиков, но можно пойти дальше и добавить новые структуры данных, а также совершенно новые способы агрегирования данных, отсутствующие в API. С учетом этого перейдем к добавлению агрегирования, управляемого данными.

## 10.3. АГРЕГИРОВАНИЕ, УПРАВЛЯЕМОЕ ДАННЫМИ

В главе 7, рассматривая операции с состоянием в Kafka Streams, вы познакомились с агрегированием. Но представьте, что у нас появились уникальные требования к агрегированию. В частности, вместо окон на основе времени нам понадобилось окно, основанное на определенных аспектах входных событий. Мы должны включать в агрегат только события, соответствующие заданным критериям, и пересыпать результаты, как только поступит запись, не соответствующая этим критериям.

Например, представьте, что мы отвечаем за производственную линию на заводе Waldo Widgets, изготавливающем популярные устройства. Для контроля за расходами и повышения эффективности мы установили несколько датчиков, которые постоянно отправляют информацию о важных показателях в процессе производства. Все датчики передают данные в Kafka. Со временем мы выяснили, что датчики температуры являются одними из лучших прогностических индикаторов неполадок в производственном процессе. Длительные скачки температуры почти всегда сопровождаются долгостоящей остановкой производственной линии, иногда на несколько часов, пока инженер не сможет обслужить машину или в худшем случае заменить ее.

Итак, мы определили, что нам нужно создать приложение Kafka Streams, отслеживающее показания датчиков температуры. Опираясь на свой опыт, мы разработали некоторые требования к точности информации. За эти годы работы мы подметили, что непосредственно перед возникновением проблемы начинают наблюдаться небольшие скачки температуры и постепенно периоды повышения температуры становятся все длиннее.

Нам нужно, чтобы агрегирование выполнялось только для замеров с повышенной температурой. Получение постоянного потока всех показаний температуры малопродуктивно, потому что в большом потоке информации можно не заметить

признаков приближения проблем. Мы потратили некоторое время на обдумывание точных требований и пришли к следующим критериям:

- если температура ниже заданного порога, то игнорировать ее;
- как только появляется значение, превышающее порог, открывается окно агрегирования;
- агрегирование продолжается, пока показания превышают пороговое значение;
- когда количество замеров достигнет заданного числа, агрегат должен быть отправлен дальше по конвейеру;
- когда показания упадут ниже порога, агрегат должен быть отправлен дальше по конвейеру;
- поскольку сетевое подключение к датчикам отличается известной нестабильностью, то, если в течение 10 секунд при открытом окне не поступает ни одного замера, следует отправить результат агрегирования дальше по конвейеру.

Такой подход позволит приложению визуализации отображать информацию, необходимую команде инженеров для принятия мер. Мы начинаем изучать Kafka Streams DSL API и сосредотачиваемся на оконном агрегировании. Прочитав документацию, мы решаем использовать агрегирование с применением `SessionWindow`, так как в этом случае агрегирование управляется поведением, а не временем. Однако хотелось бы иметь возможность выбирать информацию для включения в агрегат и определять момент, когда выдавать результаты. Поэтому мы решили взять за основу API узлов-обработчиков (Processor API) и реализовать свое оконное агрегирование, управляемое данными.

У нас уже есть опыт создания приложений Kafka Streams на основе API узлов-обработчиков, поэтому мы сразу же погружаемся в процесс и приступаем к созданию реализации, которая решает стоящую перед нами задачу. В листинге 10.10 приводится начало определения `ProcessorSupplier`, придуманного нами.

#### Листинг 10.10. ProcessorSupplier

```
public class DataDrivenAggregate implements
    ProcessorSupplier<String, Sensor, String, SensorAggregation> {
    private final StoreBuilder<?> storeBuilder; ←
    private final Predicate<Sensor> shouldAggregate; ←
    private final Predicate<Sensor> stopAggregation; ←
    public DataDrivenAggregate(final StoreBuilder<?> storeBuilder,
        final Predicate<Sensor> shouldAggregate,
        final Predicate<Sensor> stopAggregation) {
        this.storeBuilder = storeBuilder;
        this.shouldAggregate = shouldAggregate;
        this.stopAggregation = stopAggregation;
    }
```

Predicate, определяющий, когда должно начаться агрегирование

StoreBuilder, необходимый для создания хранилища состояний

Predicate, определяющий, когда должно закончиться агрегирование

Итак, при создании `ProcessorSupplier` мы должны передать конструктору два экземпляра `Predicate`. Первый определяет, когда начать агрегирование или включить текущую запись в имеющийся агрегат. Второй решает, когда отклонить запись и завершить начатое агрегирование. Рассмотрим реализацию `Processor`, которую возвращает `ProcessorSupplier` (листинг 10.11).

#### Листинг 10.11. Реализация интерфейса Processor

```
private class DataDrivenAggregateProcessor extends
    ContextualProcessor<String, Sensor, String, SensorAggregation> {
    KeyValueStore<String, SensorAggregation> store;
    long lastObservedStreamTime = Long.MIN_VALUE;

    @Override
    public void init(ProcessorContext<String, SensorAggregation> context) {
        super.init(context);
        store = context().getStateStore(storeBuilder.name()); ← Извлекает ссылку
        context().schedule(Duration.ofSeconds(10), ← на хранилище состояния
            PunctuationType.WALL_CLOCK_TIME,
            this::cleanOutDanglingAggregations); ← Планирует выполнение
    } ← пунктуации, чтобы
        // гарантировать
        // отправку результатов
        // остановленного
        // агрегирования
}
```

Инициализация экземпляра `Processor` проста и очень похожа на инициализацию других реализаций, которые мы видели выше в этой главе. Но самое интересное находится в методе `process` (листинг 10.12).

#### Листинг 10.12. Реализация метода DataDrivenAggregateProcessor.process

```
@Override
public void process(Record<String, Sensor> sensorRecord) {
    lastObservedStreamTime =
        Math.max(lastObservedStreamTime, ← Задает семантику времени
            sensorRecord.timestamp()); ← потока для обработчика
    SensorAggregation sensorAgg = store.get(sensorRecord.key());
    SensorAggregation.Builder builder;
    boolean shouldForward = false;
    if (shouldAggregate.test(sensorRecord.value())) { ← Первый предикат, определяющий
        if (sensorAgg == null) {
            builder = SensorAggregation.newBuilder();
            builder.setStartTime(sensorRecord.timestamp());
            builder.setSensorId(sensorRecord.value().getId());
        } else {
            builder = sensorAgg.toBuilder();
        }
        builder.setEndTime(sensorRecord.timestamp());
        builder.addReadings(sensorRecord.value().getReading());
        builder.setAverageTemp(builder.getReadingsList()
            .stream()
            .mapToDouble(num -> num)
            .average()
            .getAsDouble());
    }
```

```

sensorAgg = builder.build();
shouldForward =
    sensorAgg.getReadingsList().size() % emitCounter == 0;
store.put(sensorRecord.key(), sensorAgg);

} else if (stopAggregation.test(sensorRecord.value()) && sensorAgg != null) {
    store.delete(sensorRecord.key());
    shouldForward = true;
}
if (shouldForward) { ←
    context().forward(new Record<>(sensorRecord.key(),
        sensorAgg,
        lastObservedStreamTime));
}
}

```

Агрегирование имеет довольно простую логику, поэтому я коснусь только самых основных моментов этого метода. Первым делом он обновляет переменную `lastObservedStreamTime`. Время потока движется только вперед, поэтому мы не просто принимаем отметку времени из входной записи, а проверяем, поступила ли эта запись по порядку, и если это не так, то повторно используем существующее значение времени потока. Насколько важна переменная `lastObservedStreamTime`, вы увидите в следующем разделе, где мы рассмотрим метод, выполняющий пунктуацию.

Второй комментарий в листинге 10.12 касается сути этого узла-обработчика: определения необходимости включения записи в агрегат. Если запись нужно включить в агрегат, то создается новый объект агрегата (если он еще не существует) и запись добавляется в него. Кроме того, несколькими строками ниже переменной `shouldForward` присваивается значение `true` или `false` после проверки, достаточно ли замеров получено, чтобы потребовать выполнить отправку агрегата до закрытия окна. Количество замеров, соответствующих критерию, может быть велико, поэтому мы посчитали нужным периодически отправлять промежуточный результат агрегирования. Последняя строка в этом разделе сохраняет агрегат в хранилище состояния.

Если первый оператор `if` оценивает свое выражение как `false`, то управление будет передано второму предикату. Если запись соответствует условию `stopAggregation`, то это означает, что температура упала ниже порогового значения, и тогда, если имеется существующий агрегат, он удаляется из хранилища состояния, а переменной `shouldForward` присваивается значение `true`. Если температура ниже порога и нет существующего объекта агрегата, то узел-обработчик просто игнорирует запись. Наконец, если переменная `shouldForward` имеет значение `true`, то обработчик отправляет результат агрегирования всем нижестоящим узлам-обработчикам.

Итак, мы реализовали все основные требования к этому узлу-обработчику, создав оконный (установливающий начальную и конечную отметку времени) агрегат, основываясь на некоторых особенностях данных в записях — в данном случае на показаниях датчика температуры. Нам осталось позаботиться о последнем требовании. Мы должны прервать агрегирование и отправить результат, если в течение определенного периода не было получено никаких обновлений. Это последнее

требование реализует запланированную пунктуацию. Давайте посмотрим на код пунктуации (листинг 10.13).

### Листинг 10.13. Код пунктуации

```

void cleanOutDanglingAggregations(final long timestamp) {
    List<KeyValue<String, SensorAggregation>> toRemove = new ArrayList<>();
    try (KeyValueIterator<String, SensorAggregation> storeIterator
        = store.all()) { ←
        while (storeIterator.hasNext()) {
            KeyValue<String, SensorAggregation> entry = storeIterator.next();
            if (entry.value.getEndTime() < ←
                (lastObservedStreamTime - 10_000)) { ←
                toRemove.add(entry);
            }
        }
        toRemove.forEach(entry -> { ←
            store.delete(entry.key);
            context().forward(new Record<>(entry.key,
                entry.value,
                lastObservedStreamTime));
        });
    }
}

```

Получает итератор для обхода содержимого хранилища состояния

Каждая запись, не получившая обновления вовремя, удаляется из хранилища и отправляется дальше по конвейеру

Проверяет, находится ли отметка времени последнего замера в агрегате в пределах 10 секунд от текущего времени потока

Запланированная пунктуация будет выполняться каждые 10 секунд (по системному времени) и проверять все записи в хранилище состояния. Если отметка времени в какой-то записи отстает от текущего времени потока более чем на 10 секунд (то есть за это время она не обновлялась), то эта запись добавляется в список для удаления. После обхода всех записей, имеющихся в хранилище, выполняется обход списка, и каждая запись, присутствующая в нем, удаляется из хранилища и отправляется дальше по конвейеру.

На этом мы завершаем пример, демонстрирующий использование API узлов-обработчиков для реализации функциональности, недоступной в DSL. Это во многом надуманный пример, но его главной целью было показать, что, когда нужен более полный контроль над отправкой результатов агрегирования и пользовательских вычислений в конвейер, API узлов-обработчиков (Processor API) поможет в достижении этой цели.

## 10.4. ИНТЕГРАЦИЯ API УЗЛОВ-ОБРАБОТЧИКОВ И KAFKA STREAMS API

До сих пор мы изучали Kafka Streams и API узлов-обработчиков по отдельности, но кто сказал, что эти подходы нельзя сочетать? Давайте разберемся, зачем может понадобиться подобное сочетание.

Допустим, вы уже давно используете как Kafka Streams, так и API узлов-обработчиков. Вам больше нравится подход Kafka Streams, но вам хотелось бы включить

некоторые свои уже готовые узлы-обработчики в приложение Kafka Streams, поскольку они дают нужные вам возможности управления на более низком уровне. Или вам необходимо реализовать определенное поведение, не поддерживаемое DSL, но не во всей топологии, а только в ее части.

Kafka Streams API предоставляет метод `KStream.process`, с помощью которого можно интегрировать созданную с помощью API узлов-обработчиков функциональность. Добавленный в версии 3.0.0, этот метод предлагает новый подход к объединению API узлов-обработчиков и Kstream DSL из предыдущих версий. Новый метод `process` является значительным улучшением, обеспечивая столь желанную функциональность прямой пересылки записей нижестоящим узлам конвейера, которая ранее была доступна только путем использования метода `transformValues`.

#### ПРИМЕЧАНИЕ

Существует устаревший метод `KStream.process`, возвращающий тип `void` и принимающий параметр типа `org.apache.kafka.streams.processor.ProcessorSupplier`. Также устарели все методы `transformXXX`.

Рассмотрим на примере приложения биржевой аналитики (вы найдете его в файле `src/main/java/bbejeck/chapter_10/StockPerformanceDslAndProcessorApplication.java`), как можно объединить DSL и API узлов-обработчиков (листинг 10.14). Поскольку это приложение уже было описано выше в этой главе, я не буду повторять его описание, а покажу только как объединить Kafka Streams DSL и API узлов-обработчиков (некоторые детали для простоты опущены).

#### Листинг 10.14. Пример объединения DSL и API узлов-обработчиков

```
StreamsBuilder builder = new StreamsBuilder();
StoreBuilder<KeyValueStore<String, StockPerformance>> storeBuilder =
    Stores.keyValueStoreBuilder(storeSupplier,
        Serdes.String(),
        stockPerformanceSerde);

builder.stream(INPUT_TOPIC, ← Создает поток с помощью DSL
    Consumed.with(stringSerde, stockTransactionSerde))
    .process(new StockPerformanceProcessorSupplier(storeBuilder)) ←
    .peek(peekKV("StockPerformance"))
    .to(OUTPUT_TOPIC,
        Produced.with(stringSerde, stockPerformanceSerde));
    
```

Добавляет  
узел-обработчик  
для вычисления  
биржевой аналитики

Здесь создается экземпляр `KStream` с помощью DSL. Но затем в середину топологии внедряется пользовательский `StockPerformanceProcessor` (который создается с помощью `StockPerformanceProcessorSupplier`). По сути, этот прием помогает упростить топологию и использовать DSL для всего остального, кроме нестандартного узла-обработчика. Во многих случаях использование нового метода `KStream.process` оказывается наилучшим подходом для добавления пользовательской логики, ограниченной рамками одного узла-обработчика.

## ИТОГИ ГЛАВЫ

- API узлов-обработчиков обеспечивает большую гибкость за счет большего объема кода.
- Хотя код при использовании API узлов-обработчиков менее лаконичен, чем в случае Kafka Streams API, применять его достаточно просто, и «под капотом» Kafka Streams API задействует именно API узлов-обработчиков.
- При выборе используемого API учтите и возможность применения Kafka Streams API с интеграцией низкоуровневого метода `process()` по мере необходимости. Если у вас только один нестандартный узел-обработчик, то смешанный подход — это то, что вам нужно. Если таких узлов несколько или имеются особые требования к маршрутизации, то лучшим подходом может быть полный переход на API узлов-обработчиков.

# 11

## *ksqlDB*

### **В этой главе**

- ✓ Знакомство с ksqlDB.
- ✓ Подробности о потоковых запросах.
- ✓ Создание потоковых приложений с использованием инструкций SQL.
- ✓ Создание материализованных представлений потоков.
- ✓ Использование расширенных возможностей ksqlDB.

К настоящему моменту вы познакомились с несколькими компонентами платформы потоковой обработки событий Kafka: Kafka Connect — для интеграции с внешними системами и Kafka Streams — для создания приложений потоковой обработки событий. Эти два компонента вместе образуют фундамент для приложений потоковой обработки событий. В этой главе вы познакомитесь с ksqlDB — еще одним компонентом, который позволяет использовать Kafka Connect и Kafka Streams из кода на языке SQL. ksqlDB — это «потоковая база данных, специально созданная для потоковых приложений» (<https://ksqldb.io/>). Она позволит вам создавать мощные потоковые приложения, написав всего несколько операторов SQL.

Какие преимущества дает использование ksqlDB? Самое главное, что значительно упрощается процесс разработки приложений. С ksqlDB вам не придется писать программный код или файлы конфигурации — достаточно будет написать и выполнить SQL-запросы, а они, в свою очередь, запустят постоянно работающее приложение, от которого вы сможете получать мгновенные уведомления о событиях.

Представим, что мы работаем с бизнес-аналитиками в финтех-компании Big Short Equity. Они видели приложения, которые мы создали с помощью Kafka Streams, и хотели бы иметь возможность создавать приложения для финансового анализа в режиме, близком к реальному времени, но они не владеют программированием на Java. Мы могли бы удовлетворить потребности аналитиков и создать необходимые приложения Kafka Streams, но было бы гораздо эффективнее, если бы аналитики могли создавать их самостоятельно. Аналитики имеют богатый опыт в SQL, потому что большая часть их работы заключается в написании запросов к реляционным базам данных. В результате у нас возникла идея познакомить их с компонентом ksqlDB, поддерживающим масштабируемую распределенную потоковую обработку, в том числе агрегирование, соединения и оконные операции. Но в отличие от SQL-запросов к типичной реляционной базе данных, которые возвращают результаты и на этом останавливаются, запросы ksqlDB выполняются и возвращают данные непрерывно.

Вам понравится ksqlDB, потому что с его помощью вы сможете быстро создавать мощные приложения потоковой обработки событий за время, необходимое для написания SQL-запроса! Итак, в этой главе вы узнаете, как применить все, что вы узнали к данному моменту о создании приложений потоковой обработки событий, используя знакомый синтаксис SQL. Внутренне ksqlDB использует Kafka Streams, поэтому все идеи, представленные в предыдущих главах, применимы и здесь. Кроме того, сервер ksqlDB обеспечивает прямую интеграцию с Kafka Connect, так что вы сможете создать полноценное сквозное решение, не написав ни строчки программного кода. Сначала мы рассмотрим базовые понятия потока и таблицы, затем исследуем, как ksqlDB обрабатывает различные форматы данных, включая JSON, Avro и Protobuf, и, наконец, разберем продвинутые варианты агрегирования, соединения и оконных операций.

## 11.1. ЗНАКОМСТВО С KSQLDB

Выше в книге мы обсуждали понятия потока событий и потока обновлений. Напомню, что поток событий — это бесконечная последовательность независимых событий. Записи в потоке событий с одинаковым ключом не связаны друг с другом — каждая представляет отдельное событие. Поток обновлений немного отличается. Он тоже бесконечен, но события, имеющие тот же ключ, что и предшествовавшие им события, считаются обновлениями этих событий. ksqlDB поддерживает обе идеи, позволяя выполнять запросы к потоку или таблице. Кроме того, запросы могут выполняться к потоку или материализованному представлению на определенный момент времени.

Итак, напишем наш первый запрос ksqlDB. А чтобы сравнение с приложением Kafka Streams получилось более наглядным, мы повторим приложение Yelling, которое написали в главе 6.

### ПРИМЕЧАНИЕ

Развернуть ksqlDB можно одним из трех способов: в автономном режиме, как кластер на месте или в облаке Confluent Cloud. В этой книге мы будем использовать ksqlDB, развернутый в автономном режиме с помощью Docker. Далее в этой главе архитектура ksqlDB будет представлена более подробно.

Сначала на основе топика Kafka нужно создать поток STREAM (листинг 11.1).

#### **Листинг 11.1.** Создание потока STREAM в ksqlDB

```
ksql> CREATE STREAM input_stream (phrase VARCHAR) WITH
  (kafka_topic='src-topic', partitions=1, value_format='KAFKA');
```

Итак, первый шаг — создание потока STREAM. В этом примере мы создали STREAM с именем `input_stream` на основе топика `src-topic`.

#### **ПРИМЕЧАНИЕ**

Далее я буду предполагать, что для всех примеров ksqlDB вы используете один из файлов `docker compose (arm64_ksqldb-docker-compose.yml или x86_ksqldb-docker-compose.yml,` в зависимости от аппаратной архитектуры вашего ноутбука), включенных в примеры исходного кода для книги. После запуска команды `docker compose -f <имя_файла> up` вам нужно будет открыть новое окно терминала и выполнить в нем команду `docker exec -it ksqldb-cli ksql http://ksqldb-server:8088`. Она запустит сеанс интерфейса командной строки (CLI) ksqlDB. Прочтите файл `README` в примерах исходного кода для главы 11, где приводится исчерпывающий набор инструкций. Кроме того, эти команды должны также работать, если вы решите использовать ksqlDB в Confluent Cloud.

После выполнения команды `CREATE STREAM...` в окне терминала, где запущен клиент командной строки ksqlDB, вы должны увидеть сообщение, показанное в листинге 11.2.

#### **Листинг 11.2.** Результат выполнения инструкции CREATE STREAM...

```
Message
-----
Stream created
-----
```

#### **СОВЕТ**

Чтобы убедиться, что объект STREAM действительно был создан, выполните команду `show streams;` в окне, где запущен клиент командной строки ksqlDB.

Теперь, когда у нас есть объект STREAM, заполним входной топик, чтобы можно было начать кричать! Для создания записей в топике можно использовать KafkaProducer, но пока ограничимся командами SQL. Выполните несколько инструкций `INSERT`, как показано в листинге 11.3.

#### **Листинг 11.3.** Инструкции для заполнения потока данными

```
INSERT INTO input_stream (phrase) VALUES (
  'Chuck Norris finished World of Warcraft');
INSERT INTO input_stream (phrase) VALUES (
  'Chuck Norris first program was kill -9');
INSERT INTO input_stream (phrase) VALUES (
  'generate bricks-and-clicks content');
INSERT INTO input_stream (phrase) VALUES (
  'brand best-of-breed intermediaries');
....
```

Выполнение операторов `INSERT INTO...` — отличный способ быстро начать работу с ksqlDB. Но на практике, завершив быстрое прототипирование, вы захотите иметь

какое-то более эффективное средство ввода данных в топик, например KafkaProducer или другой запрос ksqlDB.

Следующий шаг — создание непрерывного запроса, или push-запроса. Прежде чем написать его, познакомимся с некоторой справочной информацией. В ksqlDB запросы основаны на данных из топика Kafka, а так как поток событий никогда не останавливается, запросы выполняются непрерывно. То есть после запуска запрос продолжит извлекать входные данные, пока вы явно не остановите его. Это и есть push-запросы, потому что они «проталкивают»<sup>1</sup> результаты клиенту, запустившему запрос. Клиенты, которым мы должны передать запрос ksqlDB, рассматриваются далее в этой главе. Помимо push-запросов, есть еще pull-запросы<sup>2</sup>, которые выдают результаты из материализованного хранилища, соответствующие определенному моменту времени. Если эти термины кажутся вам непонятными, не волнуйтесь, они будут объясняться далее в этой главе.

Завершив начальное знакомство, вернемся к написанию запроса, который станет нашим потоковым приложением (листинг 11.4).

#### **Листинг 11.4.** Непрерывно выполняющийся запрос, который станет нашим потоковым приложением

```
CREATE STREAM yelling AS      ← Создает новый поток STREAM
    SELECT UCASE(phrase) AS SHOUT   ← Выбирает столбцы и применяет функцию UCASE
    FROM input_stream
    EMIT CHANGES;   ← Инструкция EMIT CHANGES отправляет изменения
```

Поздравляю! Написав простой запрос, вы получили потоковое приложение! Обратите внимание, что для преобразования букв в верхний регистр мы использовали встроенную функцию ksqlDB. Доступно несколько таких встроенных функций. Мы будем рассматривать их по мере продвижения в этой главе.

Прежде чем продолжить, давайте сравним это приложение с нашим первым приложением Kafka Streams — приложением Yelling. Это поможет вам понять, что такое ksqlDB и на что он способен при разработке потокового приложения. Для начала взглянем на версию Kafka Streams (листинг 11.5).

#### **Листинг 11.5.** Версия приложения Yelling для Kafka Streams

```
Serde<String> stringSerde = Serdes.String();
StreamsBuilder builder = new StreamsBuilder();

builder.stream("src-topic",
    Consumed.with(stringSerde, stringSerde))
    .peek(sysout)
    .mapValues(value -> value.toUpperCase())
    .peek(sysout)
    .to("out-topic",
        Produced.with(stringSerde, stringSerde));
```

Приложения Kafka Streams реализуются довольно просто. Одно из преимуществ Kafka Streams DSL — это в первую очередь декларативный, а не императивный

<sup>1</sup> Push — «вталкивать, проталкивать». — Примеч. пер.

<sup>2</sup> Pull — «вытягивать, захватывать, получать». — Примеч. пер.

язык. То есть в коде на этом языке вы описываете, что хотите сделать, а не как это сделать. И в чем тут разница? А разница в том, что декларативный код выглядит проще и понятнее.

Например, представьте, что вы пригласили друга на ужин. Он спросил, нужно ли что-нибудь принести с собой, и вы попросили его захватить бутылку красного вина. Это было бы декларативное заявление. Вы оставили решение, где, когда и как ваш друг возьмет бутылку вина. А теперь представьте, что вы описываете другу, как добраться до винного магазина, где найти красное вино в самом магазине, что делать, если ваше любимое вино отсутствует в продаже, и т. д. — это набор императивных инструкций. Я мог бы продолжать и дальше, но вы наверняка уже поняли, что я хотел сказать.

Использование Kafka Streams вместо простых экземпляров `KafkaConsumer` и `KafkaProducer` значительно упрощает разработку приложений, но ksqlDB выводит разработку на еще более высокий уровень. В листинге 11.6 показано то же потоковое приложение, реализованное в виде запроса ksqlDB.

#### **Листинг 11.6.** Версия приложения Yelling для ksqlDB

```
CREATE STREAM yelling AS SELECT UCASE(phrase) AS SHOUT FROM input_stream
    EMIT CHANGES;
```

Для создания приложения Yelling в ksqlDB потребовался всего лишь односторочный оператор SQL.

Версия для Kafka Streams все еще выглядит довольно просто, но не забывайте, что ей нужно передать экземпляры Serde, создать экземпляр `StreamBuilder`, скомпилировать и запустить код и т. д. Все эти сложности начинают становиться более существенными при создании все более сложных приложений. Еще одно преимущество ksqlDB, и, пожалуй, самое важное, — использование языка запросов SQL открывает дверь перед теми, кто не владеет навыками программирования приложений.

Обратите внимание, что SQL-оператор ksqlDB на самом деле компилируется в приложение Kafka Streams. Поэтому даже при использовании SQL полезно понимать, как работает Kafka Streams.

Означает ли это, что ksqlDB может решить все наши проблемы и нам больше не понадобится Kafka Streams? Конечно нет. Никакой инструмент не сможет удовлетворить все потребности, и Kafka Streams всегда найдет себе применение в создании мощных приложений потоковой обработки событий. ksqlDB — это лишь еще один мощный инструмент в ваших руках.

Далее рассмотрим более сложный и реалистичный пример, и одновременно вы познакомитесь с абстракцией `TABLE`, которая представляет последнюю запись для заданного ключа.

## **11.2. ПОДРОБНОСТИ О ПОТОКОВЫХ ЗАПРОСАХ**

Ранее в книге вы познакомились с концепциями `KStream` и `KTable` в Kafka Streams (см. главы 6 и 7 соответственно). `KStream` — это поток событий, в котором записи (пары «ключ — значение») с одинаковым ключом представляют независимые события, а `KTable` — это поток обновлений, в котором записи с одинаковым ключом являются обновлениями предыдущих событий с тем же ключом. В ksqlDB имеются

аналогичные концепции **STREAM** и **TABLE**. Рассмотрим более реалистичный пример, чем приложение Yelling, и заодно более детально рассмотрим работу с ksqlDB.

Представьте, что мы запустили сайт, пропагандирующий занятия физкультурой, который призывает участников делать как можно больше шагов в день. При этом участники могут заниматься разными видами деятельности, которые засчитываются в шаги, а не только бегом или ходьбой. После того как некоторые влиятельные лица одобрили приложение, наш трафик значительно вырос, и мы хотели бы еще больше стимулировать рост, предложив возможность отображать лидеров, набравших больше всего шагов, практически в реальном времени.

У нас есть мобильное приложение, которое обновляет результаты пользователей на внутреннем сервере, работающем в облаке. Сервер получает информацию от мобильных приложений и передает обновления об активности пользователей в топик Kafka с именем `user_activity`. Этот топик является источником информации для публикации обновлений на веб-сайте, и наш первый шаг — создать **STREAM** на ее основе (листинг 11.7).

#### Листинг 11.7. Создание потока STREAM

```
CREATE STREAM user_activity (
    first_name VARCHAR,           ← Столбцы в потоке STREAM
    last_name VARCHAR,
    activity VARCHAR,
    event_time VARCHAR,
    steps INT

) WITH (kafka_topic='user_activity',
       partitions=4,
       value_format='JSON',
       timestamp = 'event_time',      ← Поле с отметкой
                                         времени деятельности
       timestamp_format = 'yyyy-MM-dd HH:mm:ss' ← Формат поля
                                                 с отметкой времени
);
```

У нас есть SQL-запрос, создающий поток `user_activity`. Событие в топике хранится в формате JSON, и мы указали структуру JSON, перечислив имена столбцов. Инструкция `WITH` содержит свойства, которые ksqlDB использует для обработки потока, и на этот раз мы добавили несколько новых элементов: свойства с именами `timestamp` и `timestamp_format`.

С помощью этих свойств можно указать, какое поле в записи следует использовать в качестве отметки времени. Как рассказывалось в главе 4, `KafkaProducer` встраивает отметку времени в запись перед тем, как отправить ее в Kafka. Производитель добавляет отметку времени, которая интерпретируется как время события. Обычно отметка времени, созданная производителем, достаточно близка к фактическому времени события. Но иногда бывает желательно использовать отметку времени, не ту, что добавляется производителем, а ту, что находится в значении записи.

Причин, почему предпочтительнее использовать отметку времени из значения записи, может быть много, но главная заключается в том, что отметка времени в записи точнее отражает момент, когда произошло событие. В нашем случае отметка времени в записи — это время, когда мобильное приложение на устройстве пользователя отправило запись на наш сервер. Отметка времени, установленная производителем,

отражает время получения записи от нашего сервера. Хотя на практике эти два времени должны быть очень близки, при присуждении очков или призов правильное отслеживание времени, когда запись была создана на пользовательском устройстве, играет особенно важную роль.

Итак, чтобы использовать встроенную отметку времени, мы определили свойство `timestamp` в инструкции `WITH` и тем самым указали, что время события должно извлекаться из заданного столбца, а не из метаданных, которые предоставляет Kafka. Поскольку время в поле `event_time` хранится в виде строки, мы должны сообщить ksqlDB его формат, что мы и сделали с помощью свойства `timestamp_format`.

На практике отметки времени в объектах событий обычно представлены значениями типа `long`, определяющими количество миллисекунд, прошедших от начала эпохи Unix (1 января 1970 года). В таких случаях столбец с отметкой времени объявляется с типом `BIGINT`, а в инструкции `WITH` указывается только имя столбца, потому что с таким полем ksqlDB может работать непосредственно. Учитывая последнее замечание, мы можем обновить запрос создания потока, как показано в листинге 11.8.

#### Листинг 11.8. Создание потока с временем события типа long

```
CREATE STREAM user_activity (first_name VARCHAR,
                            last_name VARCHAR,
                            activity VARCHAR,
                            event_time BIGINT, ← Поля с отметкой времени
                            steps INT                                объявлено с типом BIGINT

) WITH (kafka_topic='user_activity',
       partitions=4,
       value_format='JSON', ← Сообщает ksqlDB, что отметка времени
       timestamp = 'event_time'                      события находится в столбце event_time
);

```

Тип `BIGINT` используется потому, что это числовой 8-байтный тип в ksqlDB. В Java ему соответствует тип `long`. Теперь посмотрим, как использовать отметку времени, созданную производителем. В ksqlDB каждая запись получает системный столбец `ROWTIME`, содержащий отметку времени, полученную из базовой записи Kafka. Под словами «*системный столбец*» я подразумеваю, что он добавляется ksqlDB автоматически. Нам не нужно выполнять какие-либо действия, чтобы добавить его. Пример использования `ROWTIME` мы увидим в разделе 11.3. Прежде чем перейти к примеру приложения учета шагов, мы должны рассмотреть некоторую дополнительную информацию об инструкции `WITH`. Для начала обсуждения в листинге 11.9 представлена инструкция `WITH` из потока `user_activity`.

#### Листинг 11.9. Инструкция WITH из потока user\_activity

```
WITH (kafka_topic = 'user_activity', ← Задает топик Kafka, который послужит
      partitions = 4, ← источником данных для потока
      value_format = 'JSON', ← Количество разделов во входном топике
      timestamp = 'event_time', ← Формат значения в паре «ключ — значение»
      timestamp_format = 'yyyy-MM-dd HH:mm:ss' ← Столбец с отметкой времени
);

```

Сообщает ksqlDB формат представления отметки времени

## ПРИМЕЧАНИЕ

Выше в этом разделе мы уже обсудили свойства, связанные с отметками времени, поэтому я не буду рассматривать их снова.

Обратите внимание, что мы указали имя топика. Этот топик будет служить источником данных для потока записей, описывающих активность пользователя. Напомню, что ksqlDB применяет запрос к входным записям в топике, а не к таблице в реляционной базе данных. Элемент `partitions` указывает количество разделов топика. Обратите также внимание, что свойство `kafka_topic` всегда является обязательным.

Мы уже знаем, что внутренне ksqlDB использует Kafka Streams, тогда почему мы должны сообщать количество разделов входного топика? В конце концов, при работе с Kafka Streams мы указываем только имя топика, а все остальные хлопоты по организации правильного потребления записей из топика берет на себя Kafka Streams. Дело в том, что если при создании потока указанного топика не существует, то ksqlDB попытается его создать.

Но если топик существует, то ksqlDB использует его. Важно отметить, если вы указываете свойство `partitions` для `STREAM` и топик существует, то указанное вами количество разделов должно соответствовать фактическому. Иначе будет генерирована ошибка. Однако если топик существует, то можно спокойно опустить свойство `partitions`.

При разработке или прототипировании в локальной среде создание топиков с помощью ksqlDB может сэкономить время. Но в промышленном окружении лучше всегда создавать нужные топики заранее, чтобы обеспечить ясность относительно структуры приложений.

Последний параметр настройки потока, который мы рассмотрим здесь, — это `value_format`. Как нетрудно догадаться, он сообщает ksqlDB формат значения в паре «ключ — значение». Есть также настройка для ключа — `key_format`. В параметрах `key_format` и `value_format` допускается указывать следующие значения:

- `JSON`;
- `JSON_SR`;
- `AVRO`;
- `PROTOBUF`;
- `PROTOBUF_NOSR`;
- `NONE`;
- `KAFKA`;
- `DELIMITED`.

Я не буду здесь вдаваться в подробное обсуждение различных типов (их описание вы найдете в документации ksqlDB: <http://mng.bz/ZEZP>), однако считаю нужным отметить несколько важных моментов. Форматы `JSON_SR`, `AVRO` и `PROTOBUF` используют Schema Registry, а для того, чтобы использовать Schema Registry с ksqlDB, нужно задать конечную точку HTTP (как мы делали это с клиентами Kafka или Kafka Streams) через свойство `ksql.schema.registry.url`.

Подробнее параметры конфигурации ksqlDB мы рассмотрим в следующем разделе. Если вы используете Schema Registry, то ksqlDB автоматически зарегистрирует

схему, если потребуется. Однако если для заданного значения или ключа уже есть схема, то ее идентификатор можно указать в инструкции `WITH` при настройке свойств потока. Например, предположим, что в нашем приложении мы использовали Авто для значений. В этом случае мы бы описали поток, как показано в листинге 11.10.

#### **Листинг 11.10.** Определение идентификатора схемы при создании потока

```
CREATE STREAM user_activity
WITH ( kafka_topic = 'user_activity',
      value_format = AVRO,           ← Задает формат значения
      value_schema_id = 1           ← Задает идентификатор
                                существующей схемы
)
```

Здесь мы указали, что для сериализации значений используется формат Avro. Однако, чтобы дать возможность получить точную схему из Schema Registry, мы указали также идентификатор схемы. Эта возможность использовать точную версию схемы имеет большое значение, когда ksqlDB потребляет записи из топика или отправляет их в топик, с которой работают другие приложения, потому что все клиенты должны использовать одну и ту же схему, иначе приложение, скорее всего, будет сталкиваться с проблемами из-за ошибок форматирования данных. Если не указать идентификатор, то ksqlDB извлечет последнюю версию схемы, предположив, что имя субъекта задано в соответствии с шаблоном `<имя_топика>-key` или `<имя_топика>-value`. Мы подробно рассмотрели Schema Registry в главе 3, поэтому вы можете вернуться туда, чтобы освежить понимание концепций Schema Registry.

Отметьте также, что мы не определили столбцы для потока. Причина в том, что при использовании формата сериализации, поддерживаемого Schema Registry, ksqlDB автоматически определит имена столбцов и типы полей из схемы. В текущем примере мы указали имена и типы столбцов, потому что используем простой формат данных JSON и автоматически определить эти характеристики не удастся. Более подробно об использовании ksqlDB вместе с Schema Registry мы поговорим в разделе 11.5, а пока вернемся к созданию физкультурного приложения. Для общего знакомства в листинге 11.11 показан начальный запрос.

#### **Листинг 11.11.** Начальный запрос для создания физкультурного приложения

```
CREATE STREAM user_activity (first_name VARCHAR,
                            last_name VARCHAR,
                            activity VARCHAR,
                            event_time VARCHAR,
                            steps INT

) WITH (kafka_topic='user_activity',
       partitions=4,
       value_format='JSON',
       timestamp = 'event_time',
       timestamp_format = 'yyyy-MM-dd HH:mm:ss'
);
```

Создав этот поток, можно переходить к реализации часто запрашиваемой новой функции — таблицы лидеров. Мы суммируем количество шагов для заданного вида деятельности и сообщаем результаты в порядке от наибольшего к наименьшему. Чтобы

создать статистику для таблицы лидеров, нужно выполнить агрегирование, которое в данном случае представляет собой простую сумму столбца `steps` (листинг 11.12).

#### Листинг 11.12. Добавление суммирования для расчета лидеров в категории

```
CREATE TABLE activity_leaders AS ← Создаст таблицу
  SELECT
    last_name,           ← Для агрегирования шагов
    SUM(steps)          ← вызывается функция SUM
  FROM user_activity
  GROUP BY last_name   ← Группировка по столбцу last_name
  EMIT CHANGES;
```

При использовании агрегирования в ksqlDB результатом запроса является таблица TABLE, поэтому ее нужно явно создать с помощью запроса. Обратите также внимание, что, как и при запросе к реляционной базе данных, мы должны включить в инструкцию GROUP BY столбцы, присутствующие в операторе SELECT. В результате мы создали таблицу, или материализованное представление агрегата SUM. Запрос продолжит возвращать результаты, выбирая записи из базового потока `user_activity`, поэтому по мере добавления новых результатов в поток наша таблица будет обновляться. Кроме того, ksqlDB создает топик журнализации изменений, чтобы гарантировать резервное копирование агрегированных записей на случай, если сервер ksqlDB, выполняющий табличный запрос, столкнется с какими-либо проблемами.

Следует отметить, что за кулисами создания этой таблицы приводит к созданию в Kafka нового топика с именем по умолчанию `activity_leaders`, которое является именем таблицы. Если потребуется использовать другое имя для топика, то его можно задать в инструкции WITH. Как это сделать, я покажу ниже.

Изначально этот запрос работает автоматически и наши клиенты в демонстрационном приложении моментально реагируют на появление изменений в таблице лидеров. Но в своих отзывах пользователи указали, чтобы мы могли бы конкретизировать выводимую информацию. Во-первых, сейчас количество шагов суммируется по всем видам деятельности и отображается только с фамилией пользователя, что затрудняет различение пользователей приложения с одинаковой фамилией.

Ознакомившись с пожеланиями пользователей, мы решили приостановить выпуск окончательной версии и внести необходимые изменения. Поскольку мы находимся в среде разработки, то хотели бы очистить результаты выполнения табличного запроса. Поскольку мы планируем полностью переделать таблицу `activity_leaders`, также желательно удалить выходной топик. Поэтому следующим шагом мы выполним очистку, запустив команды, показанные в листинге 11.13.

#### Листинг 11.13. Команда для удаления таблицы и выходного топика

```
DROP TABLE activity_leaders DELETE TOPIC;
```

Эта команда удалит таблицу из ksqlDB и выходной топик. Обратите внимание, что она только помечает топик как удаленный, фактическое удаление топика выполняется брокером асинхронно в период очистки ресурсов.

## ПРИМЕЧАНИЕ

Инструкцию `DELETE TOPIC` в конце можно опустить. Если нужно сохранить записи в топике, то исключите инструкцию удаления топика из команды. Можно также добавить в команду проверку `IF EXISTS`, чтобы она не генерировала ошибку, если удаляемая таблица не существует. С учетом всех этих замечаний команда будет выглядеть как `DROP TABLE IF EXISTS activity_leaders DELETE TOPIC.`

Очистив существующую таблицу, переопределим табличный запрос и добавим в результаты имя пользователя и тип активности. Теперь, глядя на результаты, пользователи смогут отличать друг друга и видеть лидеров для каждого вида активности. Поскольку мы хорошо разбираемся в SQL, нам не потребуется много времени, чтобы придумать запрос (листинг 11.14).

### Листинг 11.14. Обновленный запрос с агрегированием лидеров по видам деятельности и полным именам

```
CREATE TABLE activity_leaders AS
SELECT
    first_name,
    last_name,
    activity,
    SUM(steps)
FROM user_activity
GROUP BY first_name, last_name, activity
EMIT CHANGES;
```

Обновленный запрос мало изменился: в него добавлена лишь выборка дополнительных столбцов `first_name` и `activity`. Поскольку мы добавили два столбца в часть `SELECT`, их также нужно добавить в `GROUP BY`, потому что при агрегировании нам нужно сгруппировать записи по выбранным полям для формирования уникальных результатов. Группировка действует как ключ для операции агрегирования. Теперь запускаем новый табличный запрос в ksqlDB CLI и... видим неожиданную ошибку (листинг 11.15).

### Листинг 11.15. Ошибка создания GROUP BY с несколькими столбцами

```
Key format does not support schema.
format: KAFKA
schema: Persistence{columns=[`FIRST_NAME` STRING KEY, `LAST_NAME`  
    STRING KEY, `ACTIVITY` STRING KEY], features=[]}
reason: The 'KAFKA' format only supports a single field. Got:  
[`FIRST_NAME` STRING KEY, `LAST_NAME` STRING KEY, `ACTIVITY` STRING KEY]
```

Это подробное сообщение хорошо объясняет ситуацию, но давайте разберемся с ней подробнее. Когда ksqlDB пытается создать выходной топик для таблицы, то по умолчанию выбирает тип первичного ключа KAFKA, то есть это должен быть скалярный тип, поддерживаемый Kafka, например строка или целое число. Но здесь мы используем составной ключ, полученный из трех столбцов. Это необходимо, потому что группировка выполняется по этим трем столбцам с целью гарантировать

уникальность каждой выходной записи. Однако в отсутствие схемы три поля не могут служить ключом.

К счастью, эта проблема имеет простое решение, нужно лишь указать формат ключа для новой таблицы, который поддерживает схему. В данном случае мы будем использовать JSON. Для этого немного изменим запрос и добавим инструкцию `WITH`, как показано в листинге 11.16.

**Листинг 11.16.** Обновленный запрос `CREATE TABLE`, определяющий формат ключа, который поддерживает схему

```
CREATE TABLE activity_leaders WITH (KEY_FORMAT = 'JSON') AS
SELECT
    first_name,
    last_name,
    activity,
    SUM(steps)
FROM user_activity
GROUP BY first_name, last_name, activity
EMIT CHANGES;
```

После добавления инструкции `WITH (KEY_FORMAT = 'JSON')` наша новая таблица будет использовать JSON и создавать составной ключ, содержащий столбцы `first_name`, `last_name` и `activity`. Но мы еще не закончили, так как это изменение хотя и обеспечит успешное выполнение запроса, но столбцы, перечисленные в `GROUP BY`, находятся в ключе и не будут отображаться в конечных результатах.

Поэтому нам нужно сообщить ksqlDB, что эти столбцы должны быть включены в значение результирующих записей (напомню, что Kafka работает с парами «ключ — значение»). И снова мы можем быстро добиться желаемого, использовав функцию `AS_VALUE`, которая также требует от ksqlDB скопировать ключ записи в ее значение. Нам нужно сохранить исходные столбцы в инструкции `SELECT` и добавить функцию `AS_VALUE` для каждого ключа, который должен быть скопирован в значение (листинг 11.17).

**Листинг 11.17.** Определение формата ключа и копирование ключей в значение

```
CREATE TABLE activity_leaders WITH (KEY_FORMAT = 'JSON') AS
SELECT
    first_name as first_name_key,          | Определяет псевдонимы
    last_name as last_name_key,           | ключам
    activity as activity_key,            |
    AS_VALUE(first_name) as first_name,   | Копирует часть ключа в значение и задает
    AS_VALUE(last_name) as last_name,     | псевдонимы для имен столбцов
    AS_VALUE(activity) as activity,
    SUM(steps) as total_steps
FROM user_activity
GROUP BY first_name, last_name, activity
EMIT CHANGES;
```

Теперь у нас есть действующий агрегирующий запрос с несколькими элементами в `GROUP BY`, по сути с составным ключом для каждой записи. Обратите внимание, что

такой подход требуется не всегда. Но необходимость группировки по нескольким столбцам возникает довольно часто, поэтому имело смысл уделить внимание этой ситуации.

Все, что мы видели до сих пор, — это лишь верхушка айсберга возможностей, имеющихся в ksqlDB. Мы можем создавать более сложные запросы и использовать в них встроенные функции, которые стоят того, чтобы с ними познакомиться. Но прежде, чем продолжить изучать возможности ksqlDB, сделаем небольшую паузу и обсудим некоторые из концептуальных типов запросов.

## 11.3. ПОСТОЯННЫЕ ЗАПРОСЫ, А ТАКЖЕ ЗАПРОСЫ PUSH И PULL

К настоящему моменту мы создали потоковое приложение, которое отображает обновления по мере появления в потоке `user_activity` новых пользовательских данных. После того как клиент запустит запрос, результаты будут непрерывно «проталкиваться» ему, если только он не завершит запрос явно. Но иногда бывает нужно получить только один результат вместо постоянного потока обновлений. Кроме того, может понадобиться запустить непрерывный запрос, который обслуживает не конкретный клиент, а может использоваться любым клиентом, отправляющим запрос, то есть что-то более постоянное. Далее мы обсудим, как реализовать оба этих подхода.

ksqlDB поддерживает три вида запросов. Один из них — постоянные (непрерывные) запросы, которые непрерывно обновляют поток или таблицу по мере появления новых входных записей. Запрос этого вида возвращает результаты клиенту, запустившему запрос. Примером постоянного запроса может служить таблица `activity_leaders`, созданная в предыдущем разделе.

Push-запросы являются отличным выбором для организации асинхронной обработки; вы отправляете команду или запрос, но не ждете немедленного ответа — запрос выполняется в фоновом режиме и вернет ответ позже. Конкретным примером асинхронной обработки является отправка электронного письма: вы пишете текст, а затем отправляете его, зная, что в какой-то момент получите ответ, но вы не ждете немедленного ответа.

Для случаев, когда требуется синхронная обработка, ksqlDB предлагает *pull-запросы*. Примером синхронной обработки может служить ситуация, когда вы наняли для ремонта дома бригаду строителей и им нужно ваше разрешение, чтобы удалить часть стены. Все работы останавливаются, пока они не получат ответ. Pull-запросы могут выполняться к потоку или таблице, и они имеют ограничения, о которых мы поговорим чуть позже.

Итак, возникает вопрос: какой тип запросов и когда следует использовать? Чтобы ответить на него, сравним все три типа запросов (постоянные, push и pull) друг с другом (рис. 11.1).

**Исходный поток**

```
CREATE STREAM MY_DATA_STREAM
    WITH(kafka_topic='data_topic',
        value_format='PROTOBUF'
    );
```

Постоянные запросы создают новый поток или таблицу на основе существующего и выполняют некоторый анализ

**Производный поток,  
выполняющий сложную  
обработку**

```
CREATE STREAM DATA_CRUNCHING AS
    SELECT field_1, field_2, ...
    FROM
        MY_DATA_STREAM
```



Результаты запроса постоянно  
сохраняются в топике

**Рис. 11.1.** Постоянные запросы выполняются на сервере и сохраняют результаты в топике

Постоянный запрос выполняется на сервере. Он сохраняет результаты в топике Kafka, поэтому они остаются доступными в течение всего времени хранения топика. Кроме того, запустив постоянный запрос, вы сможете поделиться его результатами с другими, поскольку любой клиент-потребитель Kafka сможет прочитать записи из топика. Непрерывно выполняющийся запрос можно рассматривать как рабочую лошадку, которая тянет основной груз по выполнению анализа в вашем потоковом приложении. При создании постоянного запроса можно использовать весь спектр SQL-команд, поддерживаемых в ksqlDB. Постоянные запросы имеют форму CREATE TABLE|STREAM AS SELECT....

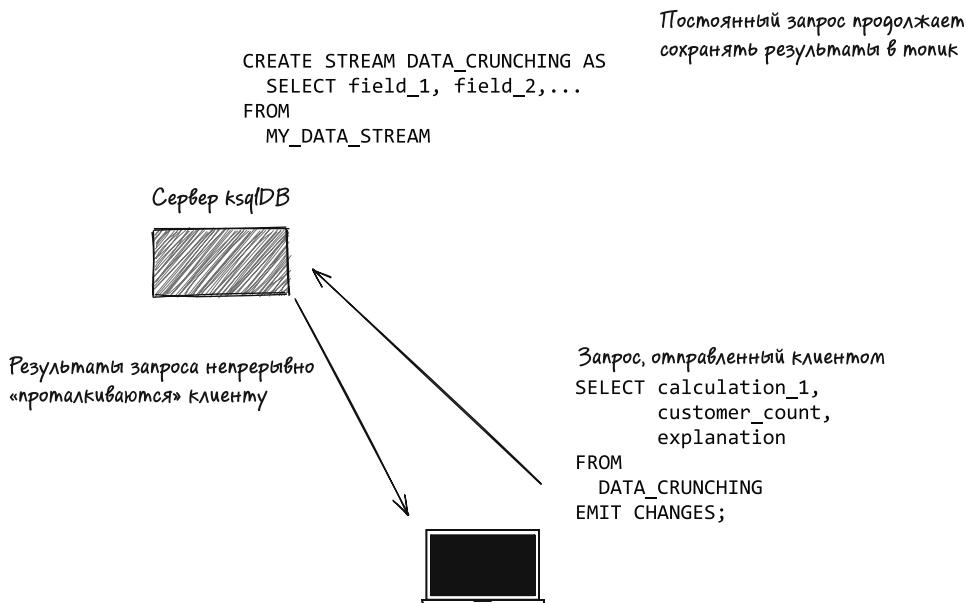
Push-запрос не сохраняет результаты в топике. Push-запрос возвращает свои результаты клиенту, запустившему запрос, но при этом результаты отправляются непрерывным потоком. Push-запрос можно представить как подписку на изменения в постоянном запросе (рис. 11.2).

В листинге 11.18 показано, как мог бы выглядеть push-запрос к `activity_leaders`.

#### Листинг 11.18. Пример push-запроса к таблице лидеров

```
SELECT
    last_name, activity, total_steps
FROM activity_leaders
EMIT CHANGES;
```

Запустив этот запрос, вы будете получать результаты по мере обновления информации об активности каждого пользователя. Но эти изменения нигде не сохраняются — результаты просто возвращаются клиенту, запустившему запрос из ksqlDB CLI, REST API или клиента ksqlDB для Java. Какие клиенты ksqlDB доступны, мы узнаем далее в этой главе. Этот запрос возвращает все обновления в таблицу, но вы можете уточнить результаты с помощью инструкции WHERE, как показано в листинге 11.19.



**Рис. 11.2.** Push-запросы не сохраняют результаты, а возвращают их клиенту, отправившему запрос

#### Листинг 11.19. Push-запрос с инструкцией WHERE

```

SELECT,
    last_name, activity, total_steps
FROM activity_leaders
WHERE total_steps > 1000
EMIT CHANGES;
    
```

Теперь вы будете получать только те обновления, в которых общее количество шагов превышает 1000. Условия в инструкции `WHERE` могут ссылаться на любой столбец, определяемый потоком или таблицей, включая псевдостолбцы `ROWTIME`, `ROWPARTITION` и `ROWOFFSET`, создаваемые ksqlDB автоматически. Эти столбцы добавляются для каждой записи, получаемой из входного топика Kafka, поддерживающей поток или таблицу. Давайте уделим немного времени каждому из них:

- `ROWTIME` — это отметка времени, связанная с записью Kafka, которую устанавливает производитель или брокер, в зависимости от конфигурации;
- `ROWPARTITION` — раздел топика, откуда была прочитана запись;
- `ROWOFFSET` — смещение записи в топике Kafka; каждая запись имеет смещение, представляющее ее логическую позицию.

В каких случаях могут пригодиться эти псевдостолбцы в запросе? Иногда бывает полезно отфильтровать результаты по внешним факторам. Например, вы можете решить просмотреть только события в заданном временном интервале. Даже если запись не определяет отметку времени как один из своих столбцов, вы все равно сможете отфильтровать результаты по значениям в `ROWTIME`.

Для уточнения результатов можно указать несколько условий. Я не буду перечислять их все здесь, но уверен, что вы знаете, как это делается. На данный момент вы должны заметить связь между постоянными и push-запросами. Постоянный запрос несет всю нагрузку и позволяет запускать push-запросы для получения подмножества интересующей вас информации. Поскольку push-запрос не сохраняет результаты, их можно использовать в качестве источника оповещения достижения заданного состояния.

Дождавшись требуемого состояния — скажем, когда определенный пользователь достигнет 10 000 шагов, — вы можете завершить запрос. С помощью инструкции `LIMIT` можно ограничить количество результатов. Например, представьте, что вы тестируете запрос в ksqlDB CLI и хотите просто убедиться, что он работает, и затем завершить его. В таком случае push-запрос в листинге 11.19 можно изменить, как показано в листинге 11.20.

#### **Листинг 11.20.** Push-запрос с инструкцией `LIMIT`

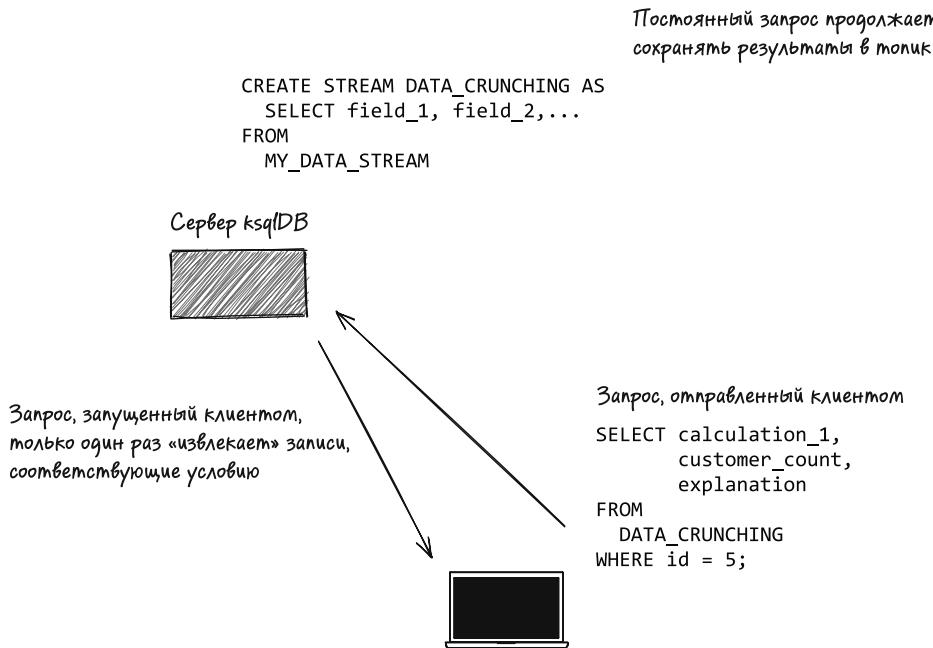
```
SELECT
    last_name, activity, total_steps
FROM activity_leaders
WHERE total_steps > 1000
EMIT CHANGES
LIMIT 10;
```

Теперь запрос будет автоматически завершаться после выдачи десяти результатов. Должен отметить, что в push-запросах можно использовать весь набор SQL-команд, поддерживаемых в ksqlDB. Между постоянными и push-запросами есть два основных различия.

1. Push-запросы не сохраняют результаты — они выводятся в консоль или возвращаются клиенту, запустившему запрос.
2. Push-запросы не являются общедоступными. Постоянный запрос обрабатывает входные записи *один раз* и сохраняет результаты в топике Kafka. С push-запросами ситуация иная: если отдельные клиенты запускают одни и те же запросы, то ksqlDB будет обрабатывать каждый из них независимо, даже если они возвращают одинаковый вывод (см. рис. 11.2).

Теперь перейдем к последнему типу запросов — pull-запросам (рис. 11.3). В отличие от постоянных и push-запросов, которые непрерывно обрабатывают входные записи, pull-запрос выполняется только один раз и затем завершается. При этом pull-запрос не сохраняет свои результаты в топике как постоянный запрос и возвращает их клиенту. Pull-запрос можно представить как однократное обращение с извлечением результата. Pull-запрос лучше всего подходит, когда нужен немедленный ответ, например, в сценарии «запрос — ответ».

Pull-запросы поддерживают только часть операторов ksqlDB SQL. Их можно использовать с любыми потоками или таблицами, созданными с помощью `CREATE TABLE as SELECT`, но нельзя с таблицами, созданными непосредственно на основе базового топика. Кроме того, pull-запросы не поддерживают инструкции `JOIN`, `GROUP BY`, `PARTITION BY` и `WINDOW`.



**Рис. 11.3.** Pull-запросы возвращают результаты на определенный момент времени только один раз; чтобы получить обновления, нужно повторно отправить запрос

Имеются также ограничения на содержимое инструкции WHERE: в ней должен использоваться столбец ключа, а сравнение возможно только с литеральным значением. Например, в листинге 11.21 показан pull-запрос к таблице activity\_leaders, созданной нами ранее.

**Листинг 11.21.** Пример pull-запроса с инструкцией WHERE, осуществляющей фильтрацию «поиск по ключу»

```

SELECT last_name, activity, total_steps
FROM activity_leaders
WHERE key_1 = 'Smith'
    
```

Этот запрос выберет из таблицы activity\_leaders, вернет записи с фамилией Smith и завершится. Чтобы получить какие-либо дополнительные обновления для этого запроса, его придется выполнить снова.

#### ПРИМЕЧАНИЕ

Существует настройка, которую можно задать, чтобы обеспечить более свободное использование инструкции WHERE. Если в сеансе CLI или на сервере ksqlDB присвоить параметру `ksql.query.pull.table.scan.enabled` значение `true`, то после этого станут доступны чуть более широкие возможности, например, вы сможете использовать неключевые столбцы или выполнять сравнение столбцов со столбцами. Дополнительную информацию о сканировании таблиц см. в документации ksqlDB: <http://mng.bz/lVld>.

Мы рассмотрели ряд вопросов о доступных типах запросов, а теперь подведем итоги и составим таблицу, которая поможет сравнить ситуации, в которых постоянные, push- и pull-запросы особенно эффективны (табл. 11.1).

**Таблица 11.1.** Когда использовать постоянные, push- и pull-запросы

Тип	Синтаксис	Лучше использовать	Режим выполнения	Сохраняет результат в топике
Постоянный	CREATE [STREAM TABLE] AS SELECT.. EMIT CHANGES	Когда допускаются асинхронное выполнение и тяжелая нагрузка на сервер	Непрерывный поток обновлений	Да
Push	SELECT [items] FROM [STREAM TABLE]... EMIT CHANGES	Когда допускается асинхронное выполнение, требуется уточненные запросы	Непрерывный поток обновлений	Нет
Pull	SELECT [items] FROM [STREAM Materialized TABLE]... WHERE	Когда запрос должен выполниться один раз и завершиться	Однократный, для получения обновлений необходимо запустить запрос повторно	Нет

Итак, согласно нашей таблице постоянные запросы имеют форму `CREATE STREAM|TABLE AS SELECT.. EMIT CHANGES` и сохраняют результаты в топике Kafka. Поскольку топики позволяют читать данные из них разным клиентам, постоянные запросы лучше всего подходят для выполнения тяжелых или сложных запросов. Изменения непрерывно выводятся в топик по мере поступления новых записей в поток или таблицу.

Push-запросы имеют форму `SELECT [items] FROM.. EMIT CHANGES`, они не сохраняют результаты в топике и постоянно отправляют их клиенту. В push-запросах можно использовать весь спектр SQL-инструкций, доступных в ksqlDB. Поскольку результаты не сохраняются, ksqlDB всегда обрабатывает одни и те же запросы от разных клиентов по отдельности. Push-запрос хорошо подходит для подписки на изменения в потоке или таблице, как это делается в событийно-ориентированных архитектурах, но запрос обычно намного проще.

Наконец, pull-запрос извлекает единственный результат и завершается; никаких обновлений по мере поступления новых записей не происходит. Pull-запрос имеет форму `SELECT [items] FROM...`. Результаты не сохраняются и возвращаются клиенту, отправившему запрос. Лучше всего pull-запросы подходят для сценариев «запрос — ответ», когда требуется получить единственный результат. Pull-запрос имеет ограничения на операторы SQL, которые он может использовать, особенно это касается инструкции `WHERE`.

Обычно принято запускать постоянные запросы на сервере ksqlDB и использовать в приложениях комбинации push- и pull-запросов для извлечения подмножества информации из результатов выполнения постоянных запросов. Мы завершили наше знакомство с типами запросов, но прежде, чем продолжить, давайте формализуем способы создания потоков или таблиц с помощью запросов различных типов.

## 11.4. СОЗДАНИЕ ПОТОКОВ И ТАБЛИЦ

К настоящему моменту вы узнали, что ksqlDB позволяет создавать потоки и таблицы разными способами, но пока не классифицировали их. Устранением этого пробела мы и займемся в данном разделе. Попытка классификации также будет прекрасным поводом для обсуждения интеграции с Schema Registry, потому что если поток или таблица определяется на основе топика, имеющего схему, то объявление столбцов в инструкции `CREATE` можно опустить. ksqlDB автоматически определит имена и типы на основе схемы. Сначала мы обсудим создание потоков и таблиц без схем, а затем перейдем к интеграции с Schema Registry.

Первая категория потока или таблицы, которую мы рассмотрим, может считаться базовым потоком или таблицей. Эти базовые потоки или таблицы создаются непосредственно на основе топика Kafka. Вы уже видели создание базового потока выше на примере потока `user_activity` — его определение повторно приводится в листинге 11.22.

### Листинг 11.22. Создание потока user\_activity

```
CREATE STREAM user_activity (first_name VARCHAR,
                            last_name VARCHAR,
                            activity VARCHAR,
                            event_time VARCHAR,
                            steps INT

) WITH (kafka_topic='user_activity',
       partitions=4,
       value_format='JSON',
       timestamp = 'event_time',
       timestamp_format = 'yyyy-MM-dd HH:mm:ss'
);
```

Этот запрос создает поток на основе топика `user_activity`. Как вы наверняка помните, топик Kafka хранит записи в виде пар «ключ — значение», но, как определено здесь, этот поток содержит только значения. Ключи в каждой записи имеют значение `null`. В ksqlDB ключи в потоке можно не определять, и в этом примере топик `user_activity` не имеет ключей. А если у топика есть ключи? Как бы тогда выглядела инструкция `CREATE STREAM`? Допустим, что топик `user_activity` имеет непустые ключи — целое число, представляющее идентификатор пользователя. Тогда мы могли бы изменить инструкцию `CREATE`, как показано в листинге 11.23.

**Листинг 11.23.** Создание потока user\_activity

```
CREATE STREAM user_activity ( user_id INT KEY,           ← Объявление ключа для потока
                            first_name VARCHAR,
                            last_name VARCHAR,
                            activity VARCHAR,
                            event_time VARCHAR,
                            steps INT

) WITH (kafka_topic='user_activity',
       partitions=4,
       key_format='KAFKA'   ← Определение формата ключа
       value_format='JSON',
       timestamp='event_time',
       timestamp_format='yyyy-MM-dd HH:mm:ss'
);
```

Итак, чтобы добавить ключ, единственное, что нужно сделать, — объявить столбец соответствующего типа с зарезервированным словом `KEY`, которое сообщает ksqlDB, что это ключ пары «ключ — значение». Нужно также указать формат представления ключа, что мы и сделали, указав формат `KAFKA` — один из основных форматов, поддерживаемых Kafka, наряду с `String`, `Long` и `Integer`. Как мы видели, когда обсуждали клиенты Kafka и Kafka Streams, ключ управляет секционированием входных записей — в отсутствие ключа записи равномерно распределяются по разделам.

Аналогично на основе имеющегося топика можно создать таблицу. Возьмем наш поток `user_activity` и создадим на его основе таблицу с именем `user_activity_table`, как показано в листинге 11.24.

**Листинг 11.24.** Создание таблицы user\_activity\_table

```
CREATE TABLE user_activity_table (user_id INT PRIMARY KEY,      ← Объявление первичного
                                  first_name VARCHAR,
                                  last_name VARCHAR,
                                  activity VARCHAR,
                                  event_time VARCHAR,
                                  steps INT

) WITH (kafka_topic='user_activity',
       partitions=4,
       key_format='KAFKA'   ← Определение
       value_format='JSON',  | формата ключа
       timestamp='event_time',
       timestamp_format='yyyy-MM-dd HH:mm:ss'
);
```

Создание таблицы похоже на создание потока, но с одним существенным отличием. В потоке столбец со спецификатором `KEY` можно опустить, а в таблице он является обязательным. Мы указали формат ключа, следуя тем же правилам, что и в предыдущем примере определения потока. Кроме `KAFKA`, в качестве формата ключа можно также использовать `AVRO`, `PROTOBUF` и `JSON_SR` (для схемы `JSON`), но мы на протяжении всей книги использовали и будем использовать в своих примерах только ключи типа `KAFKA`.

Между потоком и таблицей также есть некоторые различия в семантике ключей и значений. В потоке, как мы выяснили, запись может иметь пустой (`null`) ключ, но если попытаться внести такую запись в таблицу, то она будет отброшена. Еще одно отличие, которое мы рассмотрели выше: в потоке записи с одинаковым ключом не влияют друг на друга, они остаются независимыми.

Однако в таблице, как и в таблице реляционной базы данных, может быть только одна запись с конкретным значением первичного ключа, поэтому входная запись с тем же ключом, что и у имеющейся в таблице, будет интерпретироваться как обновление и заменит существующую. Есть разница и в семантике пустого (`null`) значения в потоке и в таблице. Пустое значение в потоке не имеет особого значения, но пустое значение в таблице интерпретируется как «надгробие». Пустое значение помечает запись для удаления из таблицы и лежащего в ее основе топика. Давайте обобщим эти различия в табл. 11.2.

**Таблица 11.2.** Потоки и таблицы

Поток	Таблица
Ключ необязателен	Ключ обязателен; записи без ключа будут отбрасываться
Ключи не обязательно должны быть уникальными; записи с одинаковым ключом не связаны друг с другом	Ключи должны быть уникальными; записи с одинаковым ключом считаются обновлениями существующей записи
Пустые ( <code>null</code> ) значения не имеют особой интерпретации	Пустые ( <code>null</code> ) значения интерпретируются как «надгробия», они помечают запись для удаления из таблицы
Тип KEY	Тип PRIMARY KEY

В качестве дополнительного примечания отмечу, что запись с пустым (`null`) значением не удаляется немедленно. Она лишь помечается для удаления. За кулисами в основе таблицы лежит топик, который является *сжатым*, и запись, помеченная для удаления, будет удалена, только когда запустится процесс чистки журнала. Чистка журнала производится через регулярные интервалы в соответствии с настройками.

Создав потоки или таблицы с помощью инструкции `CREATE` на основе топиков Kafka, вы не будете запрашивать их напрямую. Эти потоки и таблицы просто переносят топик Kafka в ksqlDB. А на их основе вы будете создавать производные потоки и таблицы с результатами, возвращаемыми клиенту в случае push- и pull-запросов (напомню, что push-запросы выполняются бесконечно, пока не будут завершены явно, а pull-запросы выполняются один раз и завершаются) или сохраняемыми в топике.

Итак, у нас есть три основных типа потоков и таблиц.

1. Базовые потоки или таблицы, предоставляющие тему ksqlDB для дальнейших запросов.
2. Постоянные запросы, которые публикуют результаты в топике Kafka.
3. Запросы push и pull, которые можно применять к базовому потоку или таблице, а также постоянные запросы.

Теперь, завершив знакомство с типами запросов, перейдем к форматам ключей и значений, а также к интеграции с Schema Registry.

## 11.5. ИНТЕГРАЦИЯ С SCHEMA REGISTRY

Schema Registry относительно легко интегрируется с ksqlDB и дает значительное преимущество при определении потока или таблицы, где есть схема. Схема содержит имена полей и их типы, что позволяет опустить определения столбцов при создании потока или таблицы. Например, давайте еще раз взглянем на определение таблицы `user_activity_table` и предположим, что значение имеет формат Protobuf (листинг 11.25).

**Листинг 11.25.** Определение `user_activity_table` при наличии схемы Protobuf в Schema Registry

```
CREATE TABLE user_activity_table (user_id INT PRIMARY KEY ← Определение
) WITH (kafka_topic='user_activity_proto',  

       partitions=4,  

       key_format='KAFKA' ← Определение формата ключа  

       value_format='PROTOBUF', ← Определение формата значения  

       timestamp='event_time',  

       timestamp_format='yyyy-MM-dd HH:mm:ss'  

);
```

Как и прежде, мы должны определить первичный ключ, поскольку это базовый тип KAFKA, но мы опускаем описание столбцов, и ksqlDB определяет их имена и типы из схемы. Такая ситуация, когда используется тип KAFKA и тип, поддерживаемый Schema Registry, известна как частичная ссылка на схему. Если бы ключ тоже имел тип Protobuf, то мы могли бы еще больше сократить определение таблицы, как показано в листинге 11.26.

**Листинг 11.26.** Определение `user_activity_table` при наличии схемы Protobuf для ключа и значения

```
CREATE TABLE user_activity_table  

    WITH (kafka_topic='user_activity_proto',  

          partitions=4,  

          key_format='PROTOBUF' ← Определение формата ключа  

          value_format='PROTOBUF', ← Определение формата значения  

          timestamp='event_time',  

          timestamp_format='yyyy-MM-dd HH:mm:ss'  

);
```

Я покажу обновленное определение только для таблицы `user_activity_table`, потому что изменения в определении потока `user_activity` при использовании схемы будут выглядеть совершенно идентично. Из этого правила, позволяющего опускать определения столбцов при наличии схем, есть исключение: когда используется только подмножество столбцов. Например, вернемся к таблице `user_activity_table`, но теперь предположим, что мы собираемся использовать только три столбца: `last_name`, `activity` и `steps`. В таком случае определение таблицы будет выглядеть так, как показано в листинге 11.27.

**Листинг 11.27.** Определение user\_activity\_table с подмножеством столбцов

```
CREATE TABLE user_activity_table (user_id INT PRIMARY KEY, ← Определение столбца-ключа
                                last_name VARCHAR, ← Определение столбцов
                                activity VARCHAR, из значения
                                steps INT

) WITH (kafka_topic='user_activity_proto',
       partitions=4,
       key_format='KAFKA',
       value_format='PROTOBUF'
);
```

В этом примере, даже притом что значение находится в формате Protobuf, мы должны объявить имена и типы столбцов, потому что используем только подмножество из них. Автоматическое определение имен и типов столбцов в ksqlDB сокращает объем определения потока или таблицы, но снижает ясность, потому что теперь вам придется либо выполнить инструкцию `DESCRIBE`, либо получить описание схемы, чтобы узнать имена и типы столбцов.

Прежде чем закончить обсуждение интеграции ksqlDB и Schema Registry, рассмотрим еще два важных момента, связанных с определением типов данных и регистрацией схемы в ksqlDB, а также с преобразованием типов данных.

Когда создается постоянный запрос — с использованием синтаксиса `CREATE STREAM AS SELECT...` — он наследует формат ключа и значения из базового потока или таблицы (рис. 11.4). Напомню, что постоянные запросы сохраняют свои результаты в топике с именем, совпадающим с именем потока или таблицы.

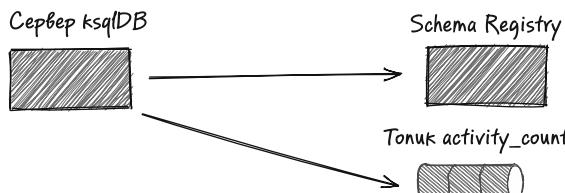
**Исходный запрос**

```
CREATE STREAM user_activity (
    first_name VARCHAR,
    ...
) WITH (kafka_topic='user_activity',
       value_format='PROTOBUF'
);
```

CREATE TABLE activity\_count AS  
SELECT  
 last\_name,  
 COUNT(activity) AS ACTIVITY\_COUNT  
FROM user\_activity  
GROUP BY last\_name

Таблица наследует тип значений Protobuf,  
потому что именно он используется  
в базовом запросе

*Зарегистрирует схему значения activity\_count-value в Schema Registry*



**Рис. 11.4.** При создании нового постоянного запроса на основе существующего он по умолчанию наследует формат данных

Это означает, что если значение находится в формате Protobuf, то ksqlDB зарегистрирует новую схему в Schema Registry, используя субъект потока или таблицы, за которым следует слово `value`. Например, вернемся к потоку `user_activity`, но на этот раз используем значение типа Protobuf (листинг 11.28).

#### **Листинг 11.28.** Поток `user_activity` в формате Protobuf

```
CREATE STREAM user_activity (first_name VARCHAR,
                            last_name VARCHAR,
                            activity VARCHAR,
                            event_time VARCHAR,
                            steps INT

) WITH (kafka_topic='user_activity',
       partitions=4,
       value_format='PROTOBUF',
       timestamp = 'event_time',
       timestamp_format = 'yyyy-MM-dd HH:mm:ss'
);
```

Здесь определения столбцов можно было бы опустить, но мы все же добавили их, чтобы прояснить пример. Допустим, что нам нужен постоянный запрос для подсчета количества действий по фамилии пользователя. В итоге у нас получится постоянный запрос, показанный в листинге 11.29.

#### **Листинг 11.29.** Постоянный запрос, наследующий формат значения

```
CREATE TABLE activity_count AS
SELECT
    last_name,
    COUNT(activity) AS ACTIVITY_COUNT
FROM user_activity
GROUP BY last_name
EMIT CHANGES;
```

Создав эту таблицу, ksqlDB зарегистрирует схему с именем `activity_count-value` с форматом Protobuf, потому что исходный поток имеет этот формат. А теперь предположим, что материализованный топик, полученный из запроса, должен нести данные в формате JSON, потому что некоторые клиенты-потребители не поддерживают другие форматы (рис. 11.5).

Для ksqlDB это не проблема, и при необходимости вы сможете с легкостью изменить формат данных из базового постоянного запроса, переопределив формат значений, как показано в листинге 11.30.

#### **Листинг 11.30.** Переопределение формата значений, получаемых из исходного потока

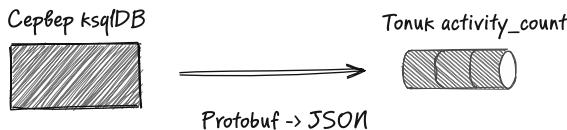
```
CREATE TABLE activity_count WITH (value_format = 'JSON') AS
SELECT
    last_name,
    COUNT(activity) AS ACTIVITY_COUNT
FROM user_activity
GROUP BY last_name
EMIT CHANGES;
```

*Исходный запрос*

```
CREATE STREAM user_activity (
    f_irst_name VARCHAR,
    .
    .
) WITH (kafka_topic='user_activity',
       value_format='PROTOBUF'
);
```

*Новый запрос*

```
CREATE TABLE activity_count WITH (value_format = 'JSON') AS
SELECT
    last_name,
    COUNT(activity) AS ACTIVITY_COUNT
FROM user_activity
GROUP BY last_name
```



**Рис. 11.5.** ksqlDB позволяет изменять тип данных на лету при создании нового постоянного запроса

Получившийся топик `activity_count` будет содержать записи со значениями в формате JSON. Эту особенность постоянных потоков можно использовать для преобразования формата записей из исходных потоков/таблиц. Поскольку push- и pull-запросы передают свои результаты клиенту в десериализованной форме, в них нет ни возможности, ни необходимости преобразовывать формат.

В заключение главы я хотел бы привести последний пример преобразования формата сериализации. Допустим, у нас есть поток данных IoT в формате Avro и вам нужно преобразовать все записи в формат Protobuf для поддержки большего количества клиентов. В листинге 11.31 показан имеющийся у нас поток со значениями в формате Avro.

**Листинг 11.31.** Поток со значениями в формате AVRO

```
CREATE STREAM IoT_TEMP_AVRO (device_id INT KEY, temp DOUBLE)
WITH (kafka_topic = "iot_temp", value_format 'AVRO');
```

Наша цель — создать идентичный поток, но со значениями в формате Protobuf. Для этого мы можем создать новый поток, выбрав все записи из потока `IoT_TEMP_AVRO`, как показано в листинге 11.32.

**Листинг 11.32.** Создание нового потока со значениями в другом формате сериализации

```
CREATE STREAM IoT_TEMP_PROTOBUF WITH (value_format 'PROTOBUF') AS
SELECT * FROM IoT_TEMP_AVRO;
```

Итак, мы создали идентичный поток со значениями в другом формате сериализации, написав всего одну строку SQL! Теперь, когда вы познакомились с основами ksqlDB, обсудим некоторые расширенные возможности, включая соединение и агрегирование.

## 11.6. РАСШИРЕННЫЕ ВОЗМОЖНОСТИ KSQLDB

К настоящему моменту вы узнали, как с помощью ksqlDB создавать потоки и таблицы, но нередко для решения сложных задач нужны дополнительные возможности. Рассмотрим сценарий из предыдущей главы, в котором у нас были два разных потока покупок, сделанных в кофейнях, открытых внутри магазинов, и в самих магазинах. Напомню, что там мы соединяли покупки, сделанные в течение 30 минут, чтобы начислить клиентам дополнительные баллы. Начнем с создания потока для каждой категории (листинг 11.33).

**Листинг 11.33.** Создание потоков для разных категорий покупок в ksqlDB

```
CREATE STREAM coffee_purchase_stream (custId VARCHAR KEY,
                                      drink VARCHAR,
                                      drinkSize VARCHAR,
                                      price DOUBLE,
                                      purchaseDate BIGINT)
    WITH (kafka_topic = 'coffee-purchase',
          partitions = 1,
          value_format = 'PROTOBUF',
          timestamp = 'purchaseDate'
    );
CREATE STREAM store_purchase_stream(custId VARCHAR KEY,
                                     credit_card VARCHAR,
                                     purchaseDate BIGINT,
                                     storeId VARCHAR,
                                     total DOUBLE)
    WITH (kafka_topic = 'store-purchase',
          partitions = 1,
          value_format = 'PROTOBUF',
          timestamp = 'purchaseDate'
    );
```

Следующим шагом после создания потоков реализуем само соединение. Но прежде уделим немного времени обсуждению требований. Как и в Kafka Streams, для выполнения соединения оба потока должны иметь *совместимое секционирование*, то есть базовые топики должны иметь одинаковое количество разделов и одинаковые ключи (в роли ключей должны выступать поля одного и того же типа). В нашем случае оба топика имеют четыре раздела, а в роли ключа в обоих потоках используется идентификатор клиента, поэтому они могут участвовать в соединении. Теперь рассмотрим инструкцию SQL, выполняющую соединение (листинг 11.34).

Итак, мы выбрали из каждого потока поля с идентификатором клиента и общей суммой покупки в магазине и с помощью оператора `CASE` по общей сумме, потраченной клиентом, определяем количество бонусных баллов. Оператор `CASE` предлагает элегантный способ оценки других условий на основе значения поля, являющегося частью запроса.

**Листинг 11.34.** Создание соединения «поток — поток» для определения вознаграждения клиентов

```
CREATE STREAM customer-rewards-stream AS
    SELECT c.custId AS customerId, ← Выбирает идентификатор клиента
        s.total AS amount, ← Выбирает общую сумму покупок
        CASE ← Инструкция CASE определяет количество бонусных баллов
            WHEN s.total < 25.00 THEN 15
            WHEN s.total < 50.00 THEN 50
            ELSE 75
        END AS reward_points
    FROM coffee-purchase-stream c
        INNER JOIN store-purchase-stream s
        WITHIN 30 MINUTES GRACE PERIOD 2 MINUTES ← Определяет окно соединения
        ON c.custId = s.custId ← Протяженностью 30 минут
                                и с 2-минутным периодом отсрочки
                                Условие, требующее совпадения
                                идентификаторов клиентов
```

### ПРИМЕЧАНИЕ

Для соединений «поток — поток» в ksqlDB поддерживаются также внешние левые соединения (`LEFT OUTER`), внешние правые (`RIGHT OUTER`) и внешние полные (`FULL OUTER`), которые имеют ту же семантику, что и в `KStream`.

Но в ksqlDB участвовать в соединениях могут не только потоки, но также таблицы. При выполнении соединения потока с таблицей результат будет отправляться только при появлении новых записей на стороне потока (как и в Kafka Streams). Поэтому соединение потоков и таблиц, как правило, применяется для обогащения данных на стороне потока за счет данных из таблицы.

Например, рассмотрим результаты только что реализованного соединения «поток — поток». Один из столбцов, спроектированных в соединение, — идентификатор клиента, но было бы хорошо добавить более полную информацию о клиенте. Для этого соединим поток `customer-rewards-stream` с таблицей клиентов, которая содержит полную информацию обо всех покупателях, участвующих в программе лояльности.

Предположим, что у нас есть коннектор-приемник, который экспортирует все записи из таблицы `members` в топик Kafka с именем `rewards-members`, поэтому первым делом создадим таблицу в ksqlDB (листинг 11.35).

**Листинг 11.35.** Создание таблицы поиска в ksqlDB на основе существующего топика

```
CREATE TABLE rewards_members (
    member_id VARCHAR PRIMARY KEY,
    first_name VARCHAR,
    last_name VARCHAR,
    address VARCHAR,
    year_joined INT)
    WITH (kafka_topic = 'rewards-members',
        partitions = 1,
        value_format = 'PROTOBUF')
);
```

Теперь, создав таблицу, настроим соединение с `customer-rewards-stream`. В этом случае не все покупатели являются участниками программы лояльности, поэтому

используем внешнее левое соединение LEFT OUTER, чтобы позже отфильтровать записи, не содержащие информации о клиентах (листинг 11.36).

### Листинг 11.36. Добавление информации о клиенте с помощью внешнего левого соединения

```
CREATE STREAM enriched-rewards-stream ← Инструкция создания потока
  WITH (kafka_topic='customer-rewards-stream',
        value_format='PROTOBUF') AS
    SELECT crs.custID as customer_id,
           rm.first_name + ' ' + rm.last_name as name,
           rm.year_joined as member_since
           crs.amount as total_purchase,
           crs.reward_points as points
      FROM customer-rewards-stream crs
      LEFT OUTER JOIN rewards-members rm
        on crs.customerId = rm.member_id ← Оператор LEFT OUTER JOIN соединяет
                                         записи из потока и таблицы
                                         с совпадающими идентификаторами
```

Задает имя выходного топика, поскольку нам хотелось бы, чтобы имя топика отличалось от имени потока

Выбирает поля из потока и таблицы

Мы создали обогащенный поток, соединив поток вознаграждений с таблицей, содержащей информацию о клиентах. Мы выбрали соединение LEFT OUTER JOIN, поэтому будем получать информацию о покупках, даже если для клиента в потоке `customer-rewards-stream` не найдется соответствующей записи в таблице `rewards-members`, правда, при этом все поля, представляющие данные из таблицы, будут иметь значение `null`.

Это был пример объединения потока и таблицы, но ksqlDB поддерживает также соединения между таблицами. Уникальность соединений двух таблиц в ksqlDB заключается в том, что они поддерживают соединение по внешним и первичным ключам. Соединение по внешнему ключу может пригодиться, когда соединение производится по первичному ключу в одной таблице и столбцу, не являющемуся первичным ключом, в другой таблице. Например, возьмем таблицу `activity-count`, созданную выше в этой главе, и соединим ее с таблицей `rewards-members` из примера соединения потока и таблицы. Для простоты я повторю определение обеих таблиц (листинг 11.37).

### Листинг 11.37. Две таблицы, участвующие в соединении по внешнему ключу

```
CREATE TABLE activity_count WITH (value_format = 'JSON') AS
  SELECT
    last_name,
    COUNT(activity) AS ACTIVITY_COUNT
   FROM user_activity
  GROUP BY last_name
  EMIT CHANGES;

CREATE TABLE rewards_members (member_id VARCHAR PRIMARY KEY,
                             first_name VARCHAR,
                             last_name VARCHAR,
                             address VARCHAR,
                             year_joined INT)
  WITH (kafka_topic = 'rewards_members',
        partitions = 3,
        value_format = 'PROTOBUF'
);
```

Допустим, та же компания розничной торговли купила наше физкультурное приложение, для которого мы создали соединения «поток — поток» и «поток — таблица», и хотела бы награждать участников программы лояльности баллами за использование этого приложения в их магазинах. У нас уже есть таблица `rewards-members`, поэтому мы можем соединить ее с таблицей `activity-counts`.

В настоящий момент роль первичного ключа в таблице `activity-counts` играет поле `last_name`, а в таблице `rewards-members` — поле с идентификатором участника программы лояльности, но в ней тоже есть столбец `last_name`, а значит, мы сможем выполнить соединение с `rewards-members.last_name` по внешнему ключу. Поскольку соединение производится по столбцу, который является частью значения, на нас не распространяется ограничение, требующее совместимого секционирования. А объясняется это просто: так как соединение происходит по полю в значении, у нас нет никакой возможности узнать, к какому разделу принадлежит запись с искомым полем в значении, поскольку записи распределяются между разделами по ключу.

### СОВЕТ

Если понадобится изменить ключ потока или таблицы в ksqlDB, то используйте инструкцию `SELECT FROM` и `partitionBy=<столбец>` в инструкции `WITH`, чтобы получить корректный ключ.

Итак, создадим соединение этих двух таблиц (листинг 11.38).

#### **Листинг 11.38.** Соединение по внешнему ключу таблиц `activity_count` и `rewards-members`

```
CREATE TABLE rewards-members-fitness-count AS
SELECT * FROM
activity_count ac JOIN rewards-members rm
    ON ac.last_name = rewards-members.last_name
EMIT CHANGES;
```

Почти без усилий мы соединили таблицу `activity_count` с информацией об участниках программы лояльности, используя внешний ключ в другой таблице.

Прежде чем завершить эту главу, обсудим одну из самых мощных особенностей ksqlDB. Мы узнали, что ksqlDB поддерживает различные форматы данных: Avro, Protobuf и JSON. Но до сих пор все наши объекты были плоскими, то есть существовал один объект верхнего уровня и все нужные нам поля были атрибутами этого объекта. А как быть, если данные имеют многоуровневую организацию? Рассмотрим схему JSON в листинге 11.39.

#### **Листинг 11.39.** Схема JSON с вложенными структурами

```
"event_id": 1234,
"school_event": {
    "type": "registration",
    "date": "2023-02-18",
    "student": {
        "first_name": "Rocky",
        "last_name": "Squirrel",
        "id": 1234567,
        "email": "rsquirl@gmail.com"
    }
}
```

```

},
"class": {
    "name": "Geology-100",
    "room": "23RF",
    "professor": {
        "first_name" : "Bullwinkle",
        "last_name" : "Moose"
        "other_classes" : ["Geology-200", "Rocks-400",
                           "Earth Minerals-304"]
    }
}
}

```

Это многоуровневая структура JSON. Как можете видеть из схемы, она содержит массу полезной информации, но как ее смоделировать и получить к ней доступ? К счастью, ksqlDB позволяет легко и просто обращаться к вложенным данным.

Для этого в ksqlDB используется тип данных **STRUCT**, который отображает строковые (VARCHAR) ключи в произвольные значения. При определении потока можно с помощью **STRUCT** описать схему вложенных данных. Например, для данных с только что представленной схемой JSON можно определить поток, как показано в листинге 11.40.

#### **Листинг 11.40.** Использование типа данных **STRUCT** для представления данных с многоуровневой организацией

```

CREATE STREAM school_event_stream (
    event_id INT,
    event STRUCT<type VARCHAR,           ← Внешняя
          date VARCHAR,                  | структура
          student STRUCT<first_name VARCHAR,   ← Первая вложенная
                                         | структура
                                         last_name VARCHAR,
                                         id BIGINT,
                                         email VARCHAR
          >,                                |
    class STRUCT<name VARCHAR,
                 room VARCHAR,
                 professor STRUCT<first_name VARCHAR,
                                   last_name VARCHAR,
                                   other_classes ARRAY<VARCHAR>
                 >
          >
)
WITH (kafka_topic='school_events',
      partitions=1,
      value_format='JSON'
)

```

Как видите, вложенные объекты определяются точно так же, как столбцы любого типа. Сначала указывается имя, за которым следует тип, то есть **STRUCT<**, и далее

перечисляются имена и типы полей объекта. По достижении последнего поля объекта его определение закрывается символом >. Этот процесс повторяется каждый раз, когда в схеме встречается объект.

Для доступа к вложенным полям нужно указать имя самого внешнего ключа и стрелку -> для разыменования объекта, повторяя этот шаблон по мере необходимости. Чтобы увидеть, как это выглядит на практике, предположим, что мы решили написать запрос, возвращающий курсы для каждого данного студента, рекомендуемые на основе других курсов, преподаваемых профессором класса, который студент в настоящее время посещает. Запрос показан в листинге 11.41.

#### **Листинг 11.41.** Извлечение вложенных данных для выработки рекомендаций

```
SELECT
    event->student->id as student_id,
    event->student->email as student_email,
    event->class->professor->other_classes as suggested
FROM
    school_event_stream
EMIT CHANGES
```

Для доступа к вложенным данным используется шаблон name->, пока не будет достигнуто требуемое поле. Этому же шаблону можно следовать для доступа к отдельным элементам массива или словаря. Например, предположим, что мы решили вернуть только одно предложение — первый элемент из other\_classes, для чего обновили запрос, как показано в листинге 11.42.

#### **Листинг 11.42.** Запрос вложенных данных и извлечение конкретного элемента массива

```
SELECT
    event->student->id as student_id,
    event->student->email as student_email,
    event->class->professor->other_classes[1] as suggested
FROM
    school_event_stream
EMIT CHANGES
```

Теперь наш запрос вернет только одно предложение.

#### **СОВЕТ**

Чтобы получить доступ к отдельным элементам из вложенного словаря, можно использовать выражение name->map\_name[ 'key' ], а если элемент словаря, в свою очередь, является структурой, то к его элементам можно было бы обратиться, используя тот же синтаксис разыменования.

Файлы SQL, файлы Docker Compose и инструкции по запуску примеров, показанных в этой главе, вы найдете в примерах исходного кода для книги в папке `streams/src/main/java/bbejeck/chapter_11`.

## ИТОГИ ГЛАВЫ

- ksqlDB – это база данных для потоковой обработки событий, в которой можно создавать потоковые приложения, используя знакомый синтаксис SQL. Написанные вами запросы будут постоянно обрабатывать события, поступающие в топик Kafka, и могут сохранять результаты в топике или возвращать их напрямую клиентскому приложению.
- В ksqlDB можно создать поток **STREAM** или таблицу **TABLE**, имеющие ту же семантику, что и потоки с таблицами в Kafka Streams. **STREAM** – это неограниченный поток независимых событий, а **TABLE** – поток обновлений, где пары «ключ – значение» являются обновлениями предыдущих событий с тем же ключом.
- В ksqlDB поддерживаются разные типы запросов – исходные запросы, создающие поток **STREAM** или таблицу **TABLE** на основе топика Kafka, и постоянные запросы, которые выбирают все или некоторые столбцы из исходного запроса и сохраняют их результаты в топике Kafka. Результаты постоянных запросов могут потребляться несколькими клиентами. Push-запрос выбирает подмножество столбцов из постоянного запроса и отправляет результаты напрямую клиенту. Push-запросы выполняются бесконечно, пока клиент не закроет соединение. Pull-запрос используется клиентами единолично – ksqlDB будет выполнять идентичные запросы от разных клиентов, даже если они возвращают идентичные результаты. Pull-запрос выполняется один раз и завершается, а также имеет некоторые ограничения на поддерживаемые им инструкции SQL.
- ksqlDB поддерживает все форматы сериализации Schema Registry. Потоки и таблицы наследуют формат ключей и значений из базовых топиков или исходных потоков и таблиц. Однако формат потока или таблицы легко изменить с помощью инструкции **WITH**, указав другой формат ключа и/или значения. ksqlDB автоматически зарегистрирует схему при создании потока или таблицы на основе постоянного запроса. Вам не придется объявлять имена столбцов и их типы в определении потока или таблицы при использовании формата Avro, Protobuf или JSON для представления ваших событий.
- ksqlDB предлагает богатый выбор библиотечных функций агрегирования, таких как **COUNT**, **SUM** и **AVE**.
- С помощью соединений «поток – поток» можно создавать новые потоки или обогащать данные в потоке с помощью соединения «поток – таблица». Соединения «поток – поток» и «поток – таблица» требуют, чтобы обе стороны имели совместимое секционирование (один и тот же тип ключа и одинаковое количество разделов). Поддерживаются также соединения «таблица – таблица» по первичному и даже по внешнему ключу, если вдруг понадобится выполнить соединение по полю в значении одной из таблиц.
- При необходимости можно запрашивать данные с многоуровневой организацией, моделируя схему потока или таблицы с помощью типа данных **STRUCT**, а затем используя оператор **->** для разыменования объектов и перехода к нужному полю.

# 19

## *Spring Kafka*

### **В этой главе**

- ✓ Что такое Spring и когда его использовать с Kafka.
- ✓ Знакомство с механизмом внедрения зависимостей.
- ✓ Создание приложений Kafka с помощью Spring Kafka.
- ✓ Создание приложений Kafka Streams с помощью Spring.

В этой главе вы узнаете, как использовать еще одну библиотеку с открытым исходным кодом — Spring — для создания приложений Kafka и Kafka Streams. Но прежде, чем углубиться в детали, давайте коротко познакомимся с фреймворком Spring и выясним, чем он так привлекателен. Первоначально фреймворк Spring был контейнером IoC (inversion of control — «инверсия управления»), разработанным Родом Джонсоном (Rod Johnson).

Принцип инверсии управления гласит, что основная программа или приложение не управляет настройкой зависимостей. Если вы уже знакомы со Spring, то можете пропустить это введение и перейти к следующему разделу, демонстрирующему использование Spring Kafka для создания приложений производителя и потребителя Kafka.

### **12.1. ВВЕДЕНИЕ В SPRING**

Обычное приложение без поддержки IoC, например система обработки платежей, напрямую создает все компоненты приложения, взаимодействующие между собой. По сути, приложение управляет всеми своими частями. Это управление подразумевает необходимость знания конкретных типов компонентов. В отличие от него

приложение с поддержкой IoC получает только ссылки на типы интерфейсов компонентов и не создает их экземпляры — за создание и внедрение экземпляров в приложение отвечает контейнер. Такой подход делает приложения более гибкими и легко тестируемыми, поскольку позволяет изменять реализацию по мере необходимости. Тестирование становится более простым, потому что при внедрении интерфейсов можно подставлять в приложение фиктивные экземпляры.

Внедрение зависимостей (dependency injection) — один из способов реализации IoC. Внешний механизм внедряет зависимости через конструкторы, методы-сеттеры или даже через поля. Именно такой механизм и реализует фреймворк Spring: механизм для связывания приложений из компонентов, каждый из которых использует только интерфейсы. Со временем Spring превратился в большой успешный проект, переросший рамки контейнера IoC.

А теперь, чтобы понять, почему вдруг может возникнуть желание использовать Spring, рассмотрим конкретный пример платежной системы, упомянутой выше. Вне всяких сомнений, платежному приложению необходимо сетевое подключение для получения и отправки платежной информации. Кроме того, вам, как разработчику такого приложения, почти наверняка захочется использовать что-то иное, чем низкоуровневые сетевые сокеты, например создать отдельный программный компонент, чтобы платежному процессору не нужно было знать о тонкостях подключения к сети и взаимодействия с ней. Таким образом, базовый скелет класса платежного процессора может выглядеть примерно так, как показано в листинге 12.1.

### Листинг 12.1. Класс PaymentProcessor

```
public class PaymentProcessor implements Processor { ← Класс PaymentProcessor

    private NetworkClient networkClient; ← Класс PaymentProcessor имеет
                                            зависимость — NetworkClient

    public PaymentProcessor(NetworkClient networkClient) { ← Удовлетворяет зависимость, передавая экземпляр
        this.networkClient = networkClient; ← NetworkClient через параметр конструктора
    }

    public void handlePayment() {
        payment = networkClient.receive(); ← Использует экземпляр NetworkClient
        // здесь выполняется некоторая работа для выполнения работы
        networkClient.send(processedPayment);
    }
}
```

Рассматривая класс `PaymentProcessor`, можно заметить, что в своей работе он зависит от `NetworkClient`. Но обратите внимание, что код ничего не знает о `NetworkClient`, кроме имен общедоступных методов в интерфейсе. Это очень удобно, так как в `PaymentProcessor` нет необходимости знать что-то еще, что выходит за рамки контракта, определяемого интерфейсом. Почему это важно? Со временем вы будете вносить изменения в проект и в том числе, возможно, в реализацию `NetworkClient`, но для данного кода это не имеет никакого значения. `PaymentProcessor`

просто ожидает получить что-то, что реализует требуемый интерфейс. В противном случае вам пришлось бы отыскать весь код, использующий конкретную реализацию, и обновить его.

Следуя такому подходу, вы получаете дополнительную выгоду — простоту тестирования. В идеале было бы желательно иметь возможность написать тест, проверяющий только логику `PaymentProcessor`, без использования настоящего сетевого соединения. Поэтому с этой целью можно внедрить в `PaymentProcessor` «фиктивную» реализацию интерфейса `NetworkClient`, которая не подключается к Интернету, а просто передает заранее определенную вами информацию. Фиктивные модели и различные подходы к тестированию мы более подробно рассмотрим в главе 14.

Итак, теперь мы видим преимущество использования внедрения зависимостей в приложениях. Однако пока остается неясным, как внедрить требуемые классы? Эту задачу решает контейнер Spring. Spring предоставляет различные аннотации, которые можно использовать для «аннотирования» разнообразных отношений между типами. Рассмотрим пример (листинг 12.2).

#### Листинг 12.2. Аннотации к классам, определяющие отношения между ними

```
@Component ← Помечает PaymentProcessor
public class PaymentProcessor implements Processor { ← как компонент в контейнере Spring

    private NetworkClient networkClient;

    @Autowired ← Указывает на необходимость
    public PaymentProcessor(NetworkClient networkClient) { ← внедрения NetworkClient
        this.networkClient = networkClient; ← в конструктор PaymentProcessor
    }
    ...
}

...
← Помечает SecureNetworkClient
@Component ← как компонент в контейнере
public class SecureNetworkClient implements NetworkClient { ←

    public SecureNetworkClient(...) {
        ...
    }
    ...
}
```

На этапе запуска приложения Spring просканирует все классы и подключит зависимости на основе их аннотаций. Проще говоря, Spring позаботится о том, чтобы связать все части вместе.

На этом я завершаю краткое введение в фреймворк Spring. В следующем разделе вы увидите, как можно использовать Spring для создания приложений с поддержкой Kafka.

### ДОПОЛНИТЕЛЬНЫЕ РЕСУРСЫ

Spring Kafka — весьма обширная тема, поэтому я не буду описывать все детали, а затрону только то, что вам понадобится, чтобы начать использовать Spring в комбинации с Kafka и Kafka Streams. Чтобы получить более полную информацию о Spring, я бы посоветовал начать со следующих книг от Manning Publications:

- Craig Walls, *Spring in Action*<sup>1</sup>.
- Somnath Musib, *Spring Boot in Practice*.
- Laurențiu Spilcă, *Spring Security in Action*<sup>2</sup>.
- Cătălin Tudose, *Java Persistence with Spring Data and Hibernate*.

Можно также обратиться к превосходной документации Spring, доступной по адресу <https://docs.spring.io/spring-kafka/reference/index.html>.

## 12.2. ИСПОЛЬЗОВАНИЕ SPRING ДЛЯ СОЗДАНИЯ ПРИЛОЖЕНИЙ С ПОДДЕРЖКОЙ KAFKA

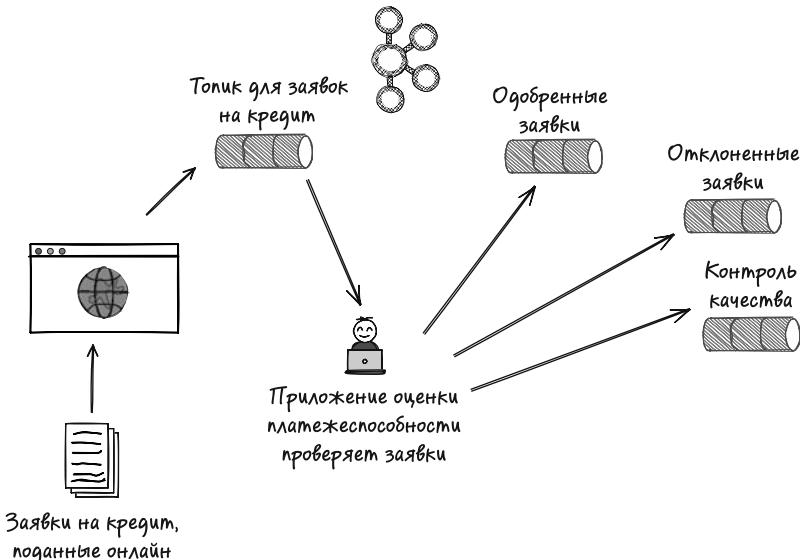
Есть два варианта создания приложений Kafka с помощью Spring: использовать стандартный фреймворк Spring, требующий больше настроек, или Spring Boot — расширение фреймворка Spring. Spring Boot поддерживает более разумный подход, следуя принципу преимущества соглашений перед настройками. Spring Boot берет на себя управление многими мелкими деталями, необходимыми для использования Kafka вместе с Spring. В принципе, можно использовать любой из подходов, но я расскажу только о подходе с использованием Spring Boot, потому что он проще и по умолчанию предоставляет такие замечательные возможности, как встроенный веб-сервер.

Представим, что мы работаем в стартапе, специализирующемся на обработке онлайн-заявок на ипотеку, авто- и бизнес-кредиты. Наша компания Dime предлагает существенно более низкие процентные ставки и планирует стать прибыльной за счет увеличения количества выдаваемых кредитов, что по задумке должно компенсировать снижение дохода от процентов. Наша цель — обеспечить быстрое рассмотрение кредитных заявок, максимально автоматизировав процесс их подачи (рис. 12.1).

Наше приложение пересыпает заявки на кредит, заполненные на сайте компании, в топик Kafka, откуда их извлекает и обрабатывает сложное приложение оценки платежеспособности клиента. Результаты оценки платежеспособности отправляются в один из трех топиков: в первый посыпаются принятые заявки, во второй — отклоненные и в третий — выборочные заявки для проверки в отделе контроля качества,

<sup>1</sup> Уоллс К. Spring в действии. — М., 2022.

<sup>2</sup> Спилкэ Л. Spring Security в действии. — М., 2024.



**Рис. 12.1.** Процесс подачи онлайн-заявки на кредит

который контролирует строгость процесса кредитования. Давайте посмотрим, как настроить приложение Spring Boot с помощью Kafka (листинг 12.3).

### Листинг 12.3. Объявление класса, играющего роль класса конфигурации в Spring Boot

```
@Configuration
public class LoanApplicationProcessingApplication {
    ...
}
```

Аннотация @Configuration отмечает класс как конфигурационный класс для контейнера Spring

Аннотация уровня класса указывает, что этот класс будет играть роль класса конфигурации для нашего приложения. По умолчанию Spring Boot отыщет в каталоге `src/main/resources` файл с именем `application.properties` и подставит все значения, что определены в этом файле, в поля с аннотацией `@Value` в этом классе. Обратите внимание, что если дать файлу другое имя или поместить его в другой каталог, то об этом нужно будет сообщить фреймворку Spring, чтобы тот смог найти файл (листинг 12.4).

За исключением некоторых дополнительных полей для внедрения значений свойств и компонентов `NewTopic` для создания требуемых топиков, в классе конфигурации больше ничего нет! Как видите, Spring Boot избавляет от необходимости определять значительный объем настроек инфраструктуры в приложении Spring Kafka. Но такое упрощение возможно только иногда, как будет показано далее в этой главе, когда изменим требования к приложению. Однако для задач, где нужны классы инфраструктуры Kafka, Spring Boot предоставляет их автоматически и тем самым открывает более быстрый путь в разработке.

**Листинг 12.4.** Базовая конфигурация, объем которой значительно меньше, если используется Spring Boot

```
@Configuration
public class LoanApplicationProcessingApplication {
    @Value("${application.group}")
    private String groupId;

    @Value("${loan.app.input.topic}")
    private String loanAppInputTopic;

    // Другие настройки опущены для простоты

    @Bean
    public NewTopic loanAppInputTopic() {
        return new NewTopic(loanAppInputTopic,
            partitions,
            replicationFactor);
    }

    // Другие bean-компоненты NewTopic для простоты опущены
}
```

Внедренная  
конфигурация

Компонент NewTopic,  
создающий входной топик

Прежде чем рассматривать конкретные компоненты Spring Kafka, посмотрим, как запустить приложение Spring Boot (листинг 12.5).

**Листинг 12.5.** Главный класс для запуска приложения Spring Boot

```
@SpringBootApplication(scanBasePackages =
    "bbejeck.spring.application")      ← Определяет приложение Spring Boot
public class LoanApplicationProcessingApplication {

    public static void main(String[] args) {

        SpringApplicationBuilder applicationBuilder =
            new SpringApplicationBuilder(
                LoanApplicationProcessingApplication.class) ← Создает экземпляр SpringApplicationBuilder
                    .web(WebApplicationType.NONE); ← Задает тип приложения без веб-сервера
            applicationBuilder.run(args); ← Запускает приложение
    }
}
```

Чтобы создать приложение Spring Boot, нужно добавить аннотацию `@SpringBootApplication` перед классом с методом `main`, запускающим приложение. Необходимо также предоставить пакеты с компонентами, которые контекст Spring должен выбирать и включать.

Как показано в листинге 12.5, мы используем класс `SpringApplicationBuilder` для сборки приложения, он также создает `ApplicationContext` — основной интерфейс для настройки различных компонентов, выбранных для помещения в контейнер Spring. Вызов `.web(WebApplicationType.NONE)`; сообщает контейнеру, что наше приложение не должно содержать встроенный веб-сервер.

По умолчанию Spring Boot запускает веб-сервер Apache Tomcat, но в этом приложении он нам не понадобится, поэтому мы устанавливаем тип `NONE`. Ниже в этой главе мы создадим еще одно приложение Spring Boot, в котором будем использовать

встроенный веб-сервер. Приложение в листинге 12.5 запускается вызовом метода `SpringApplicationBuilder.run`.

## СОВЕТ

Сгенерировать скелет приложения Spring Boot можно, перейдя по ссылке <https://start.spring.io/>.

К данному моменту вы узнали, как настроить приложение Spring Kafka и как его запустить. Далее мы рассмотрим различные компоненты приложения и то, как они связаны друг с другом. Мы не будем уделять много внимания бизнес-логике приложения, потому что здесь это неважно, вы будете добавлять ее сами в свои приложения.

### 12.2.1. Компоненты приложения Spring Kafka

В обработку заявок на кредиты вовлечены два основных компонента: один получает информацию о заявке и применяет алгоритм одобрения к каждой входной записи. После этого результаты обработки выдаются в топики Kafka: одобренные — в один топик, отклоненные — в другой, а некоторые еще и в третий топик для контроля качества. Определенное количество обработанных заявок на кредит выбирается случайным образом для проверки (человеком!), чтобы убедиться, что алгоритм работает в точном соответствии с ожиданиями.

Итак, рассмотрим `NewLoanApplicationProcessor`, начав с объявления класса и конструктора (листинг 12.6).

**Листинг 12.6.** Объявление класса `NewLoanApplicationProcessor` и его конструктор

```
@Component
public class NewLoanApplicationProcessor {
    // Аннотация @Component помечает этот
    // класс как компонент контейнера Spring

    @Value("${accepted.loans.topic}")
    private String acceptedLoansTopic; // Внедряет имена различных выходных
                                        // топиков через свойства

    @Value("${rejected.loans.topic}")
    private String rejectedLoansTopic;

    @Value("${qa.application.topic}")
    private String qaLoansTopic; // Переменная
                                // экземпляра
                                // для ссылки на объект
                                // KafkaTemplate

    private final KafkaTemplate<String, LoanApplication> kafkaTemplate; // Аннотация @Autowired осуществляет внедрение любых
                                                                        // зависимостей через параметры конструктора

    @Autowired
    public NewLoanApplicationProcessor(
        KafkaTemplate<String, LoanApplication> kafkaTemplate) {
        this.kafkaTemplate = kafkaTemplate;
    }
}
```

Перед объявлением имени класса находится аннотация `@Component`. В момент запуска приложения контейнер Spring отыскивает все классы с этой аннотацией и включает их в контекст приложения. Наличие аннотации `@Component` перед объявлением класса позволяет внедрять в него зависимости или использовать сам класс как зависимость, если обнаружатся ссылки в других классах.

Далее можно видеть внедрение в переменные свойств, содержащих имена различных выходных топиков, и этот код должен показаться вам знакомым, потому что вы уже видели нечто подобное в классе конфигурации. Наконец, взгляните на последние строки в листинге 12.6. `private final KafkaTemplate<String, LoanApplication> kafkaTemplate;` — это переменная, в которую будет помещена ссылка на экземпляр `KafkaTemplate`. Обратите также внимание на аннотацию `@Autowired` перед конструктором, она сообщает контейнеру Spring, что тот должен внедрить любые найденные параметры в вызов конструктора. В нашем случае конструктор получит экземпляр `KafkaTemplate`, созданный при запуске контейнера.

### ПРИМЕЧАНИЕ

Аннотацию `@Autowired` также можно использовать на уровне поля.

Теперь перейдем к главной части приложения — к методу обработки заявки на кредит (некоторые детали для простоты опущены) (листинг 12.7).

#### Листинг 12.7. Метод обработки заявки на кредит

```
@KafkaListener(topics = "${loan.app.input.topic}", groupId = "${application.group}")
public void doProcessLoanApplication(LoanApplication loanApplication) {
    boolean loanApproved = debtRatio <= 0.33 &&
        loanApplication.getCreditRating() > 650;
    String topicToSend = loanApproved ? acceptedLoansTopic :
        rejectedLoansTopic; ← Определяет выходной топик
    ←                               по результату проверки заявки
    LoanApplication processedLoan = LoanApplication.Builder.newBuilder(
        loanApplication).withApproved(loanApproved).build();
    kafkaTemplate.send(topicToSend, processedLoan.getCustomerId(),
        processedLoan); ← Использует KafkaTemplate для отправки
    if (random.nextInt(100) > 75) { ← результатов обработки заявок
        kafkaTemplate.send(qaLoansTopic, processedLoan.getCustomerId(),
            processedLoan); ← Случайно выбирает некоторые
    }                                ← заявки для контроля качества
}
```

Встретив аннотацию `@KafkaListener` перед объявлением этого метода, контейнер Spring создаст экземпляр `KafkaListener` с помощью `ConcurrentKafkaListenerContainerFactory`, который обертывает `KafkaConsumer`, потребляющий записи из топиков, указанных в объявлении `@KafkaListener`. Очень скоро мы подробнее исследуем связь между слушателями, потребителями и топиками, а пока продолжим рассмотрение аннотации `@KafkaListener`.

`@KafkaListener(topics = "${loan.app.input.topic}")` указывает имя топика для прослушивания, на который фактически будет подписан `KafkaConsumer`. Далее с помощью `groupId = "${application.group}"`) задается идентификатор группы потребителей. Как видите, для обоих атрибутов мы не задаем жестко запрограммированных значений, а используем операции подстановки свойств, благодаря чему получаем более гибкий код. Если в будущем понадобится изменить имена топиков

или идентификатор группы, то нам достаточно будет лишь обновить файл свойств, не изменения и не перекомпилируя код.

Здесь также можно заметить воплощение идеи управляемого сообщениями POJO (Plain Old Java Object – простой объект Java), потому что в этом классе нет ничего характерного для Kafka. Но, применив одну строку кода, мы добавили в простой класс Java возможность получать сообщения от брокера Kafka.

После обработки заявки на кредит (я намеренно опустил этот код, так как он неважен для изучения приемов использования Spring) мы выбираем топик для отправки заявки кредита, исходя из результатов ее проверки. Затем мы используем `KafkaTemplate` для возврата записи с обработанной заявкой на кредит в топик Kafka.

Здесь можно заметить дополнительные удобства, которые предоставляет `KafkaTemplate`. Мы указываем только имя топика, ключ и значение. Как мы видели в главе 4, посвященной клиентам Kafka, перед отправкой записи с помощью `KafkaProducer` нам сначала нужно было создать экземпляр `ProducerRecord`. Метод `KafkaTemplate#send` возвращает объект `ListenableFuture<SendResult<K, V>>`, и, чтобы получить результат отправки, нужно дождаться завершения асинхронной операции, вызвав метод `ListenableFuture#get`, но он заблокирует приложение, пока не будет получен результат. Но есть другое решение: мы можем передать в `ListenableFuture` объект обратного вызова, который обработает результат отправки асинхронно.

В нашем примере мы не захватываем возвращаемый `ListenableFuture`, но мы еще вернемся к использованию `KafkaTemplate` и изменим приложение обработки заявок на кредит. Но прежде рассмотрим еще один процесс, осуществляющий постобработку заявок после их оценки (листинг 12.8).

#### Листинг 12.8. Класс, осуществляющий постобработку заявок после их оценки

```
@Component ← Применение аннотации @Component
public class CompletedLoanApplicationProcessor {

    @KafkaListener(topics = "${accepted.loans.topic}",
                  groupId = "${accepted.group}")
    public void handleAcceptedLoans(LoanApplication acceptedLoan) {
        ...
    }

    @KafkaListener(topics = "${rejected.loans.topic}",
                  groupId = "${rejected.group}")
    public void handleRejectedLoans(LoanApplication rejectedLoan) {
        ...
    }

    @KafkaListener(topics = "${qa.application.topic}",
                  groupId = "${qa.group}")
    public void handleQALoans(LoanApplication qaLoan) {
        ...
    }
}
```

← Внедряет контейнеры прослушивателей для каждого топика постобработки

Мы не будем тратить много времени на подробное обсуждение, потому что вы уже знакомы с аннотациями `@Component` и `@KafkaListener`, но есть еще кое-что, что мы должны знать о `@KafkaListener`. Я уже говорил, что контейнер Spring создаст экземпляр `KafkaListenerContainer` для каждой аннотации `@KafkaListener`, встреченной при запуске приложения. Если вы определите еще один метод с аннотацией `@KafkaListener` (с другим идентификатором группы), то Spring создаст отдельный контейнер слушателя.

У нас есть четыре метода с аннотацией `@KafkaListener`, поэтому для приложения будет запущено четыре экземпляра `KafkaListenerContainer`. Каждый контейнер слушателя будет включать один экземпляр `KafkaConsumer`, подписанный на топики, указанные в аннотации. Потребитель, созданный фабрикой контейнеров, будет потреблять записи из всех разделов топика. Прежде чем перейти к следующему разделу, рассмотрим еще один вопрос, касающийся аннотации `@KafkaListener`.

Во всех примерах, представленных к данному моменту, мы размещали аннотацию на уровне метода. Однако ее также можно использовать на уровне класса, но это потребует некоторой дополнительной работы. Рассмотрим простой пример (листинг 12.9).

#### Листинг 12.9. Применение `@KafkaListener` на уровне класса требует аннотирования методов

```
@Component
@KafkaListener(topics = "${loan.app.input.topic}", groupId = " ${application.group}")
public class NewLoanApplicationProcessorListenerClassLevel {
    @KafkaHandler      ← Применение @KafkaHandler к методу,
    public void doProcessLoanApplication(LoanApplication loanApplication) { ← обрабатывающему заявки на кредит
        // Обработка заявки на кредит
    }
    @KafkaHandler(isDefault = true) ← Обработчик по умолчанию для обработки
    public void handleUnknownObject(Object unknown) { ← объектов неизвестных типов или типов,
        // Обработка неизвестного объекта
    }
}
```

Как видите, чтобы объявить класс прослушивателем Kafka, достаточно разместить аннотацию `@KafkaListener` перед его объявлением вместе с аннотацией `@Component`. При этом мы должны хотя бы один метод снабдить аннотацией `@KafkaHandler`. Мы добавили `@KafkaHandler` перед двумя методами: `doProcessLoanApplication` и `handleUnknownObject`, последний из которых мы дополнительно объявили прослушивателем по умолчанию, добавив атрибут `isDefault`. При использовании аннотации `@KafkaListener` на уровне класса Spring будет выбирать метод для вызова, основываясь на типе его параметра.

Это означает, что все методы с аннотацией `@Handler` должны принимать один параметр, и разные методы не должны иметь параметр одного типа. В нашем примере с аннотацией `@KafkaListener` на уровне класса мы добавили обработчик для записей Kafka, содержащих значение с типом, отличным от `LoanApplication`. Однако этот способ нельзя назвать лучшим вариантом использования `@KafkaListener` на

уровне класса, я демонстрирую этот подход только для полноты картины. По моему глубокому убеждению, аннотацию `@KafkaListener` следует использовать на уровне класса только при потреблении нескольких разных типов из одного топика. Рассмотрим следующий пример (листинг 12.10).

**Листинг 12.10.** Применение `@KafkaListener` на уровне класса при потреблении нескольких разных типов

```

@Component
@KafkaListener(topics = "${multi.topic.input}", groupId =
    "${multi.input.group}")
public class NewLoanApplicationProcessorListenerClassLevel {

    @KafkaHandler   ←———— Обработчик для объектов типа String
    public void doSomethingWithLong(String someString) {
        // Обработка объекта String
    }

    @KafkaHandler   ←———— Обработчик для объектов типа Long
    public void doSomething(Long longNumber) {
        // Обработка объекта Long
    }

    @KafkaHandler   ←———— Обработчик для объектов типа Double
    public void doSomething(Double doubleNumber) {
        // Обработка объекта Double
    }

    @KafkaHandler(isDefault = true)   ←———— Обработчик по умолчанию
    public void handleUnknownObject(Object unknown) {
        // Обработка объекта неизвестного типа
    }
}

```

Каждый метод принимает параметр своего типа, благодаря чему Spring может выбрать нужный для обработки той или иной записи. Теперь рассмотрим вопрос: когда следует помещать `@KafkaListener` на уровне класса, а когда на уровне метода? На этот вопрос нет четкого и быстрого ответа, но я лично придерживаюсь устоявшегося подхода: всегда стараться использовать `@KafkaListener` на уровне метода. Каждый топик должен нести события одного типа и иметь осмысленное имя, описывающее назначение топика. Конечно, всегда есть исключения из правил, которые вам придется учитывать, и если один из топиков несет события нескольких типов, то для его обслуживания можно использовать аннотацию `@KafkaListener` на уровне класса.

На этом мы завершаем создание нашего первого приложения Spring Kafka. Однако нам есть еще о чём поговорить, включая более продвинутую функциональность, которая имеет широкое применение при создании приложений потоковой обработки событий. Единственного потребителя может быть достаточно для обработки тем с одним разделом или с низким уровнем трафика. Но как быть, если понадобится обеспечить более быстрое потребление (то есть не в одном потоке выполнения)? А также как получить ключ записи Kafka и другие метаданные (отметку времени, номер раздела) или организовать отправку записей разных типов? Все эти вопросы мы рассмотрим далее.

## 12.2.2. Расширенные требования к приложению

Наше приложение онлайн-кредитования работает прекрасно, и дела идут в гору. Но теперь мы начали задумываться об изменениях, которые могут улучшить приложение.

1. Добавить больше разделов во входной топик.
2. После увеличения количества разделов хотелось бы распараллелить приложение и создать по отдельному потребителю для каждого раздела.
3. Извлекать ключ и отметку времени из входной записи.
4. Отслеживать смещения и отметки времени созданных записей.

Этот список пожеланий выглядит довольно объемным, но, к счастью, изменения, необходимые для их воплощения, легко реализуются. Мы сосредоточимся только на изменениях, характерных для приложения Spring Kafka. Полагаю, что вы уже знаете, как реализовать такие изменения, как получение номера раздела и предметного объекта (соответствующие изменения вы найдете в примерах исходного кода).

Посмотрим, как увеличить количество разделов и как запустить несколько потребителей в приложении, чтобы увеличить его пропускную способность. В главе 4 мы рассмотрели клиенты Kafka и узнали, что единицей распараллеливания топика Kafka является раздел. Таким образом, чтобы увеличить пропускную способность приложения Kafka, нужно добавить больше разделов (или заранее создать избыточные разделы с прицелом на будущее), чтобы назначить по одному потребителю для каждого раздела. Благодаря наличию выделенного потребителя для каждого раздела можно максимизировать пропускную способность приложения (конечно же, я обобщаю, потому что всегда есть исключения и всегда могут возникнуть различные ограничения).

Мы провели анализ и определили, что для оптимальной пропускной способности входной топик для заявок на кредиты должен иметь три раздела. Это число учитывает текущее количество подаваемых заявок и ожидаемое увеличение в ближайшее время. Мы также решили увеличить количество разделов в топиках постобработки, но в меньшей степени — до двух разделов в каждой из них. Мы рассуждали примерно так: каждая заявка на кредит будет существовать только в одном из двух состояний — «одобрена» или «отклонена», — поэтому при рейтинге одобрения, колеблющемся в районе 50 %, в каждый топик постобработки будет передаваться только половина ожидаемого максимального трафика заявок.

Выше в этой главе вы видели, что при использовании Spring Boot и аннотации `@EnableKafka` контейнер Spring автоматически создает `ConcurrentKafkaListenerContainerFactory`. Освежим нашу память: фабрика контейнеров слушателей создает `KafkaListenerContainer` для каждого метода, отмеченного аннотацией `@KafkaListener`. Поэтому если увеличить количество разделов во входном топике, то в текущей реализации мы будем иметь один потребитель, опрашивающий все разделы, но нам нужно иметь отдельный потребитель для каждого раздела. К счастью, несмотря на то, что Spring Boot предоставляет множество функциональных возможностей «из коробки», всегда есть возможность изменить конфигурацию, что мы сейчас и сделаем.

Первым шагом создадим Spring Bean для `ConcurrentKafkaListenerContainerFactory`, чтобы установить конкретное свойство `concurrencyLevel` (листинг 12.11).

#### Листинг 12.11. Увеличение уровня параллелизма для фабрики контейнеров слушателей

```
Создаст новый компонент
@Bean
public ConcurrentKafkaListenerContainerFactory<String, LoanApplication>
    kafkaListenerContainerFactory() {
    ConcurrentKafkaListenerContainerFactory<String, LoanApplication>
        kafkaListenerContainerFactory =
            new ConcurrentKafkaListenerContainerFactory<>();
    kafkaListenerContainerFactory.setConsumerFactory(consumerFactory());
    kafkaListenerContainerFactory.setConcurrency(partitions);
    return kafkaListenerContainerFactory;
}
```

Задает уровень параллелизма  
по количеству разделов

Эта конфигурация похожа на ту, которую мы создали в приложении без Spring Boot ранее, с одним существенным отличием: здесь мы используем метод `setConcurrency` и передаем ему число настроенных разделов. В результате такой установки уровня параллелизма мы получаем по одному `KafkaConsumer` на раздел, что нам и нужно для максимальной пропускной способности.

Другой момент, который следует учесть: мы использовали то же имя для метода, что и имя по умолчанию, ожидаемое контейнером. Почему? Используя имя `kafkaListenerContainerFactory` в качестве имени метода (напомню, что в отсутствие атрибута `name` имя метода становится именем bean-компонента), контейнер Spring выберет вашу пользовательскую фабрику контейнеров вместо создания фабрики по умолчанию с тем же именем. Благодаря этому «теневому» процессу именования bean-компонентов наш класс-обработчик автоматически будет использовать обновленную фабрику контейнеров без каких-либо изменений в коде, что сохранит гибкость кода при выборе и использовании любого контейнера.

Но это не значит, что вы всегда должны использовать одно и то же имя при переопределении значения, используемого Spring Boot по умолчанию. Вы можете дать ему любое имя, когда используете пользовательскую фабрику. В таком случае вам нужно будет лишь явно сообщить аннотации `@KafkaListener`, какую фабрику контейнеров использовать, добавив атрибут `containerFactory` (листинг 12.12).

#### Листинг 12.12. Добавление пользовательской фабрики контейнеров прослушивателей

```
// В конфигурационном классе
@Bean("custom-container")
public ConcurrentKafkaListenerContainerFactory<K, V>
    customContainerFactory()
```

Сообщает KafkaListener, какую фабрику  
контейнеров использовать

```
// В классе с аннотацией @Component
@KafkaListener(groupId = "${application.group}",
    containerFactory = "custom-container",
```

Явно задает имя  
пользовательского контейнера

Этот код представляет альтернативное решение. Здесь мы создаем настроенный контейнер в классе конфигурации и указываем имя в аннотации `@Bean`, а в классе `@Component` используем то же имя в атрибуте `containerFactory`.

## ПРИМЕЧАНИЕ

Чтобы увеличить уровень параллелизма в `ConcurrentKafkaListenerContainerFactory`, также можно определить конфигурационный параметр `spring.kafka.listener.concurrency` с желаемым числом. Это можно сделать, например, так: `spring.kafka.listener.concurrency=${num.partitions}`, если предположить, что `num.partitions` определяется выше в файле `application.properties`. Я решил создать пользовательскую фабрику контейнеров, чтобы показать пример переопределения значений по умолчанию, предоставляемых Spring Boot. Другой подход заключается в прямом определении атрибута `concurrency` с помощью аннотации `@KafkaListener`.

Теперь, достигнув желаемого соотношения по одному потребителю на раздел, мы должны учесть еще один фактор: теперь наше приложение стало многопоточным.

Контейнер Spring создает несколько потоков в соответствии с настроенным уровнем параллелизма. Создание новых потоков выполнения необходимо для увеличения пропускной способности. Но это означает, что они будут вызывать метод слушателя *параллельно*. Параллельные вызовы допустимы, если метод потокобезопасен (то есть не создает и не использует общее изменяемое состояние).

## СОВЕТ

Потоки выполнения, связанные с определенным слушателем Kafka, легко найти, если добавить `id` в аннотацию `@KafkaListener`. Spring будет использовать предоставленный вами идентификатор как часть имени потока, что позволит определить, какой поток осуществляет потребление записей из того или иного топика.

В нашем примере приложение использует один общий экземпляр `KafkaTemplate` во всех потоках, но, поскольку базовый `KafkaProducer` не представляет опасности в многопоточной среде, шаблон также потокобезопасен. Всегда старайтесь писать методы прослушивателя Kafka так, чтобы они не имели хранимого состояния, в противном случае вам придется задействовать механизмы синхронизации, чтобы гарантировать получение детерминированных результатов.

## ПРИМЕЧАНИЕ

Я не собираюсь рассматривать потоки Java или методы синхронизации. Желающие углубиться в эту тему без труда найдут множество полезных ссылок, выполнив простой поиск в Google.

Прежде чем двигаться дальше, реализуем еще два улучшения приложения из нашего списка.

1. Извлечь и записать в журнал ключ и отметку времени из выходной записи.
2. Получив результат обработки заявки, записать в журнал смещение и отметку времени произведенной записи. Кроме того, если при создании записи произошла ошибка, то у нас не будет ни смещения, ни отметки времени, поэтому также следует вывести в журнал описание ошибки.

Чтобы получить ключ, отметку времени и другие метаданные из входной записи Kafka, можно воспользоваться аннотацией `@Header`, применив ее к дополнительному параметру метода слушателя. Изменения, которые нужно внести в класс обработки заявок, будут выглядеть так, как показано в листинге 12.13.

**Листинг 12.13.** Извлечение ключа и отметки времени из входной записи Kafka

```
public void doProcessLoanApplication(LoanApplication loanApplication,
    @Header(Kafka Headers.RECEIVED_TIMESTAMP) long timestamp, ← Извлечь отметку
    @Header(Kafka Headers.RECEIVED_MESSAGE_KEY) String key) ← времени
                                                                из записи Kafka
                                                                из записи Kafka
```

Добавив аннотацию `@Header` с нужным заголовком, можно получить отметку времени и ключ сообщения. Доступны и другие заголовки, несущие информацию о топике, разделе и смещении.

**СОВЕТ**

Информация в заголовках потребляемых записей доступна в свойствах `KafkaHeaders.RECEIVED_X`. Класс `KafkaHeaders` предоставляет свойства для записей производителей и потребителей, причем имена свойств потребляемых записей начинаются с `RECEIVED`.

Теперь, получив ключ сообщения и отметку времени входной записи, перейдем к следующей задаче — зарегистрировать в журнале смещение и отметку времени созданной записи после обработки заявки. Выше в этой главе мы увидели, что класс `NewLoanApplicationProcessor` после обработки заявки возвращает запись в Kafka со статусом одобрения, определяющим топик. В листинге 12.14 для напоминания показана строка кода, отвечающая за создание записи.

**Листинг 12.14.** Отправка записи с обработанной заявкой на кредит

```
kafkaTemplate.send(topicToSend, processedLoan.getCustomerId(),
    processedLoan);
```

Мы также видели, что метод `KafkaTemplate.send` возвращает `ListenableFuture<SendResult<K, V>>`, и как раз в этой точке можно было бы извлечь отметку времени и смещение отправленной записи. Но проблема в том, что метод `ListenableFuture.get` блокирует основной поток выполнения приложения, пока не получит ответ, из-за чего приложение будет вынуждено ждать завершения запроса на отправку записи. Чтобы решить эту проблему, влияющую на производительность, мы используем другой подход. В данном случае мы можем передать возвращаемому объекту `ListenableFuture` ссылку на объект обратного вызова, который выполнится асинхронно после завершения запроса, независимо от того, был ли он успешным или нет (листинг 12.15).

**Листинг 12.15.** Объект обратного вызова для получения информации после завершения запроса

```
private final ListenableFutureCallback<SendResult<String, LoanApplication>>
    produceCallback = new ListenableFutureCallback<>() { ← Создает экземпляр ListenableFutureCallback
    @Override
    public void onFailure(Throwable ex) { ← Метод onFailure для регистрации ошибок
        LOGGER.error("Problem producing a record", ex);
    }
    @Override
    public void onSuccess(SendResult<String, LoanApplication> result) { ← Регистрация метаданных в случае успешного выполнения запроса на отправку записи
        RecordMetadata metadata = result.getRecordMetadata();
        LOGGER.info("Produced a record to topic {} at offset{} at time {}", metadata.topic(), metadata.offset(), metadata.timestamp());
    }
};
```

После создания объекта обратного вызова осталось только передать его в объект `CompletableFuture`, полученный вызовом метода `KafkaTemplate.send` (листинг 12.16).

**Листинг 12.16.** Передача объекта обратного вызова для обработки уведомлений о выполненных запросах

```
ListenableFuture<SendResult<String, LoanApplication>> ← Сохраняет ListenableFuture в переменной после вызова send
➥ produceResult =
➥ kafkaTemplate.send(topicToSend,
    processedLoan.getCustomerId(),
    processedLoan); ← Передает объект обратного вызова для обработки результатов выполнения запроса
produceResult.addCallback(produceCallback);
```

Теперь мы будем получать уведомления с результатами обработки запросов на отправку записей даже в случае сбоя. Этот процесс похож на тот, что мы видели в главе о клиентах Kafka, с той лишь разницей, что при работе напрямую с `KafkaProducer` мы передавали обратный вызов в виде параметра методу `KafkaProducer.send`.

На этом мы завершаем создание приложения Kafka с использованием Spring Kafka и далее посмотрим, как использовать Spring Kafka с приложениями Kafka Streams.

## 12.3. SPRING KAFKA STREAMS

Как мы видели, Spring Kafka в комбинации с Spring Boot значительно упрощает создание приложений на основе Kafka, однако Spring Boot также предлагает поддержку приложений Kafka Streams. Эта поддержка отличается тем, что Kafka Streams уже абстрагирует многие детали работы с Kafka. Как и при использовании любого инструмента или фреймворка, этот подход имеет свои компромиссы, которые приходится учитывать. В данном случае я имею в виду упрощение разработки приложений за счет утраты детального контроля над процессом их создания.

Использование Spring Boot с Kafka Streams дает некоторые преимущества в виде уменьшения объема инфраструктурного кода, но за счет некоторой потери контроля

над процессом построения приложения по сравнению с использованием Spring только для внедрения зависимостей. Мы рассмотрим оба подхода, чтобы вы могли решить, какой вариант лучше подходит в вашем случае. Мы также обсудим одну из замечательных особенностей использования Spring Boot с Kafka Streams; приложения Spring Boot по умолчанию запускают встроенный веб-сервер.

Наличие веб-сервера в приложении Kafka Streams является реальным преимуществом, когда требуется организовать обработку интерактивных запросов. Интерактивные запросы позволяют напрямую запрашивать результаты операций с состоянием, выполняемых в Kafka Streams. То есть вы можете включить в свое приложение Spring Kafka Streams веб-классы для обработки входящих запросов и с их помощью обслуживать интерактивные запросы. Это может упростить архитектуру вашего приложения, но об этом мы поговорим чуть позже.

Рассмотрим пример создания простого приложения Kafka Streams, сначала с использованием утилит Spring Boot, а затем с применением Spring для внедрения зависимостей. Итак, мы решили переделать свое кредитное приложение Kafka, задействовав возможности Kafka Streams. Оно будет использовать ту же логику одобрения кредита, но на этот раз записи будут обрабатываться не с помощью слушателей и `KafkaTemplate`, а с помощью Kafka Streams. Это позволит выполнять агрегирование по результатам проверки заявок в одном месте. Наш первый шаг к использованию Spring Kafka с Kafka Streams — добавить аннотации в определение класса конфигурации (листинг 12.17).

#### Листинг 12.17. Добавление аннотаций в определение класса конфигурации

```
@SpringBootApplication(scanBasePackages = {"bbejeck.spring.datagen",
                                             "bbejeck.spring.streams.boot"})
@EnableKafka ← Стандартная аннотация, добавляющая поддержку Kafka
@EnableKafkaStreams ← Аннотация, добавляющая поддержку Kafka Streams
@Configuration ← Отмечает класс как класс конфигурации приложения
public class KafkaStreamsLoanApplicationApplication {

    // Элементы конфигурации опущены для простоты
}
```

Кое-что из этого кода мы уже видели, а вот с аннотацией `@EnableKafkaStreams` встретились впервые. Она необходима для активации поддержки приложений Kafka Streams в Spring Boot.

#### СОВЕТ

Обратите внимание, что в листинге 12.17 указаны пакеты, где следует искать компоненты для включения в контейнер. Этого не требуется, если все необходимое для приложения находится в том же пакете, что и аннотированный класс конфигурации.

Благодаря применению аннотации `@EnableKafkaStreams` Spring создаст обертку вокруг экземпляра `KafkaStreams` и будет управлять жизненным циклом (запуском и остановкой) потоков данных в приложении. Прежде чем мы приступим к созданию самого приложения Kafka Streams, добавим еще кое-что из настроек (листинг 12.18).

**Листинг 12.18.** Настройки Kafka Streams, необходимые при использовании `@EnableKafkaStreams`

```

@Bean(name =
    KafkaStreamsDefaultConfiguration.
    &gt; DEFAULT_STREAMS_CONFIG_BEAN_NAME) ← Аннотация @Bean
    |                               | задает имя компонента
KafkaStreamsConfiguration kafkaStreamsConfiguration() {
    Map<String, Object> streamsConfigMap = new HashMap<>(); ← Создает HashMap
    streamsConfigMap.put(StreamsConfig.APPLICATION_ID_CONFIG,
        "loan-processing-app");
    streamsConfigMap.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
        bootstrapServers);
    return new KafkaStreamsConfiguration(streamsConfigMap); ← Возвращает экземпляр
}                                     | KafkaStreamsConfiguration

```

При запуске приложения с аннотацией `@EnableKafkaStreams` Spring Boot будет искать Spring Bean с именем `defaultKafkaStreamsConfig`, содержащий настройки для создания `StreamBuilderFactoryBean`, который, в свою очередь, создает экземпляр `StreamsBuilder` и управляет запуском и остановкой приложения Kafka Streams. Если вам не требуется ничего менять в самом экземпляре `KafkaStreams`, то на этом подготовку к запуску приложения Kafka Streams можно считать завершенной. Осталось только создать само приложение.

Но прежде, чем приступить к созданию приложения, давайте кратко рассмотрим шаги, которые можно предпринять, чтобы получить доступ к базовому экземпляру `KafkaStreams`, если понадобится. Для таких случаев Spring предоставляет `StreamsBuilderFactoryBeanCustomizer` — функциональный интерфейс или интерфейс с одним абстрактным методом. Функциональные интерфейсы идеально подходят для работы, поскольку позволяют использовать лямбда-выражение Java вместо конкретного экземпляра объекта. А когда может понадобиться доступ к объекту `KafkaStreams`? Представьте ситуацию, когда возникла необходимость, чтобы `StateListener` уведомлял нас о переходе `KafkaStreams` в состояние выполнения и мы могли получить информацию о назначении разделов топика активным задачам.

Итак, чтобы настроить `StateListener`, нужно определить два новых bean-компоненты в классе конфигурации. Первый — `KafkaStreamsCustomizer`, который дает доступ к `KafkaStreams` до его запуска. Второй — `StreamsBuilderFactoryBeanCustomizer`, который принимает `KafkaStreamsCustomizer` для применения изменений (листинг 12.19).

**Листинг 12.19.** Компонент `KafkaStreamsCustomizer` для настройки `StateListener`

```

@Bean
KafkaStreamsCustomizer getKafkaStreamsCustomizer() {
    return kafkaStreams ->
        kafkaStreams.setStateListener((newState, oldState) -> { ← Устанавливает
            if (newState == KafkaStreams.State.RUNNING) { | прослушиватель
                LOG.info("Streams now in running state"); | состояний
                kafkaStreams.metadataForLocalThreads() | в Kafka Streams
                    .forEach(tm -> LOG.info("{} active task info: {}", |
                        tm.threadName(), tm.activeTasks())); ← Выводит в журнал
            } | текущие активные
        }); | задачи для всех
}

```

Итак, с помощью `KafkaStreamsCustomizer` можно получить доступ к экземпляру `KafkaStreams` — в данном случае это делается для настройки `StateListener`. Чтобы поместить этот bean-компонент в `StreamsBuilderFactoryBean`, создадим второе определение bean-компонента, как показано в листинге 12.20.

#### Листинг 12.20. Конфигуратор для настройки объекта KafkaStreams

```
@Bean
StreamsBuilderFactoryBeanCustomizer kafkaStreamsCustomizer() { ← Создает конфигуратор
    return streamsFactoryBean ->
    streamsFactoryBean
        .setKafkaStreamsCustomizer(getKafkaStreamsCustomizer()); ← Устанавливает
}                                            KafkaStreamsCustomizer
```

Добавив определения этих двух bean-компонентов в класс конфигурации, мы открываем возможность обратиться к объекту `KafkaStreams`.

Теперь, узнав, как получить доступ к объекту `KafkaStreams` в приложении Spring Boot, создадим само приложение. Создание приложения Kafka Streams с аннотацией `@EnableKafkaStreams` отличается от того, что мы видели раньше. Чтобы увидеть различия, перейдем сразу к примеру. Допустим, мы решили преобразовать имеющееся кредитное приложение, использующее производители и потребители Kafka, в приложение Kafka Streams (листинг 12.21).

#### Листинг 12.21. Преобразование приложения на основе производителя и потребителя Kafka в приложение Kafka Streams

```
@Component ← Объявляет класс компонентом
public class LoanApplicationProcessor {

    @Value("${loan.app.input.topic}")
    private String loanAppInputTopic;

    @Autowired
    public void loanProcessingTopology(StreamsBuilder builder) { ← Метод, конструирующий
        KStream<String, LoanApplication> processedLoanStream = ← Создает KStream
        builder.stream(loanAppInputTopic,
            Consumed.with(stringSerde,
                loanApplicationSerde))
            .mapValues(loanApp -> {
        double monthlyIncome = loanApp.getReportedIncome() / 12;
        double monthlyDebt = loanApp.getReportedDebt() / 12;
        double monthlyLoanPayment =
            loanApp.getAmountRequested() / (loanApp.getTerm() * 12);
        double debtRatio =
            (monthlyDebt + monthlyLoanPayment) / monthlyIncome;

        boolean loanApproved =
            debtRatio <= 0.33 && loanApp.getCreditRating() > 650;

        return LoanApplication.Builder.newBuilder(loanApp)
            .withApproved(loanApproved)
            .build();
    });
}
```

Код в листинге 12.21 — типичный пример приложения Spring. Он объявляет класс как `@Component` и внедряет необходимые объекты с помощью аннотации `@Autowired`, но на этот раз в параметры метода, а не конструктора. Я уже упоминал некоторые различия ранее, и если приглядеться внимательнее, то можно заметить, что метод `loanProcessingTopology` возвращает тип `void`. Метод не возвращает экземпляр `StreamsBuilder`, поскольку им управляет контейнер Spring.

При запуске контейнера для сборки потокового приложения Spring передаст синглтон `StreamsBuilder` всем методам, которые ссылаются на него и отмечены аннотацией `@Autowired`. Затем, запуская экземпляр `KafkaStreams`, `StreamsBuilderFactoryBean` вызовет метод `StreamsBuilder.build`.

Благодаря такому подходу к управлению экземпляром `StreamsBuilder` со стороны Spring можно распределить конструирование топологии по разным классам. Однако я все же рекомендую помещать эту логику в один класс, ведь, когда придет время отлаживать какие-либо проблемы, будет намного проще отследить, что не так, проанализировав всю топологию в одном месте.

Мы только что увидели, как создать приложение Kafka Streams с помощью Spring Boot и аннотации `@EnableKafkaStreams`. Такой подход действительно упрощает задачу, потому что Spring берет на себя большую часть хлопот по настройке. Но ничего не дается бесплатно. Здесь мы получаем простоту запуска и управления Kafka Streams за счет потери некоторой видимости того, что происходит с приложением.

Но есть и другой компромисс, на который можно пойти: отказаться от части удобств в обмен на больший контроль и видимость топологии Kafka Streams. Он рассматривается далее. Мы по-прежнему будем использовать Spring для связывания приложения. Но на этот раз всю ответственность за сборку топологии и управление жизненным циклом объекта `KafkaStreams` мы возложим на себя. Мы возьмем предыдущее приложение Kafka Streams и внесем небольшие изменения, начав с класса, который создает топологию (некоторые детали опущены для простоты) (листинг 12.22).

#### Листинг 12.22. Переименование класса для обозначения его роли

```
@Component  
public class LoanApplicationTopology { ← Новое имя класса  
}  
}
```

Первое изменение — переименование `LoanApplicationProcessor` в `LoanApplicationTopology`. Изменяя имя, мы сообщаем, что класс будет содержать всю топологию для приложения Kafka Streams, а не только один узел-обработчик или его часть. Далее обновим сигнатуру метода `loanProcessingTopology`, что повлечет за собой еще одно изменение (листинг 12.23).

Прежде всего для поддержки изменения метода нужно обновить конструктор, чтобы обеспечить автоматическое внедрение `KafkaStreamsConfiguration`. Далее вы увидите, как это связано с изменением метода. Второе, что мы сделали, — изменили имя метода на `topology`, чтобы отразить его роль, и третье — удалили параметр `StreamsBuilder`, создав его напрямую. Мы также заменили возвращаемый тип `void` на `Topology`, отразив изменение в последней строке метода, где вызываем `StreamsBuilder.build` и возвращаем экземпляр `Topology`.

**Листинг 12.23.** Обновление метода и явное создание StreamsBuilder

```

private final KafkaStreamsConfiguration streamsConfigs;

@Autowired
public LoanApplicationTopology( ← Внедряет конфигурацию
    KafkaStreamsConfiguration streamsConfigs) { ← через параметры конструктора
    this.streamsConfigs = streamsConfigs;
}

public Topology topology() { ← Имя метода изменено и был
    StreamsBuilder builder = new StreamsBuilder(); ← удален параметр StreamsBuilder
    // Конструирование топологии
    return builder.build(streamsConfigs.asProperties()); ← Создает StreamsBuilder
    // Конструирование топологии
    // Возвращает топологию
}

```

Следующее изменение – объявление класса для поддержки создания экземпляра `KafkaStreams` и управления жизненным циклом потокового приложения (некоторые детали для простоты опущены) (листинг 12.24).

**Листинг 12.24.** Класс поддержки Kafka Streams

```

@Component
public class KafkaStreamsContainer {

    @Autowired
    public KafkaStreamsContainer(
        final LoanApplicationTopology loanApplicationTopology, ← Внедрение объекта
        final KafkaStreamsConfiguration appConfiguration) { ← LoanApplicationTopology
        this.loanApplicationStream = loanApplicationTopology; ← Внедрение настроек
        this.appConfiguration = appConfiguration; ← приложения
    }
}

```

Мы объявили класс `KafkaStreamsContainer` для сборки и запуска приложения Kafka Streams. Обратите внимание, что конструктор имеет аннотацию `@Autowired` и два параметра: `LoanApplicationTopology` и `KafkaStreamsConfiguration`, которые автоматически будут внедрены фреймворком Spring. Далее посмотрим, как использовать эти два объекта для запуска приложения Kafka Streams (листинг 12.25).

**Листинг 12.25.** Запуск приложения Kafka Streams

```

@PostConstruct ← Аннотация PostConstruct
public void init() {
    Properties properties = appConfiguration.asProperties();
    Topology topology = loanApplicationStream.topology(); ← Создание топологии
    kafkaStreams = new KafkaStreams(topology, properties); ← Сборка экземпляра
    kafkaStreams.setStateListener((newState, oldState) -> {
        if (newState == KafkaStreams.State.RUNNING) {
            LOG.info("Streams now in running state");
            kafkaStreams.metadataForLocalThreads().forEach(tm ->
                LOG.info("{} assignments {}", tm.threadName(), tm.activeTasks()));
        }
    });
    kafkaStreams.start(); ← Запуск приложения KafkaStreams
}

```

Затем мы добавили метод `init`, который создает экземпляр `KafkaStreams` и запускает его. А когда мы должны вызвать этот метод? Эту задачу Spring решает автоматически благодаря добавленной нами аннотации `PostConstruct`. Когда аннотация `PostConstruct` добавляется в класс компонента, контейнер Spring вызывает отмеченный метод, после того как объект будет полностью сконструирован, то есть в нашем случае, внедрив требуемые зависимости, Spring вызовет метод `init` после создания экземпляра `KafkaStreams`. Но выше мы заявили, что полностью будем обрабатывать жизненный цикл, поэтому возникает следующий вопрос: как мы будем останавливать приложение `Kafka Streams`? Для этого добавим еще один метод (листинг 12.26).

#### Листинг 12.26. Остановка приложения Kafka Streams

```
@PreDestroy ← Аннотация PreDestroy
public void tearDown(){
    kafkaStreams.close(Duration.ofSeconds(10)); ← Остановит KafkaStreams
}
```

Для остановки приложения мы используем аналогичный подход, добавив аннотацию `PreDestroy` перед методом, который контейнер Spring должен вызвать перед тем, как удалит управляемый компонент в процессе завершения работы.

Теперь все изменения внесены, и мы видим все детали, касающиеся конструирования приложения Kafka Streams, его запуска и остановки. Здесь нет правильного или неправильного решения, и все сводится к личным предпочтениям и имеющимся требованиям.

## ИТОГИ ГЛАВЫ

- Spring Kafka предоставляет множество абстракций, упрощающих работу с Kafka. Вы можете использовать стандартный подход Spring, но тогда придется добавить конфигурацию для классов поддержки `KafkaTemplate` и `KafkaListener`. При использовании Spring Boot, напротив, приходится писать меньше кода, определяющего настройки, благодаря добавлению аннотаций `@SpringBoot` и `@EnableKafka` на уровне класса. Часто настройки также помещаются в класс конфигурации, отмеченный аннотацией `@Configuration`. Как известно, Spring Boot следует принципу преимущества соглашений перед настройками, поэтому если вас устраивают значения по умолчанию, то вам не придется уделять много внимания конфигурации.
- При настройке уровня параллелизма для `KafkaListener` контейнер Spring создаст соответствующее количество потоков, запускающих `KafkaConsumer` для слушателя, а вы должны гарантировать безопасность метода с аннотацией `@KafkaListener` в многопоточном окружении, если количество потоков больше одного.
- Spring Boot также предоставляет аннотацию `@EnableKafkaStreams`, которая реализует управление жизненным циклом приложения Kafka Streams. Но использовать ее не обязательно, и если вы предпочитаете иметь более полный контроль над своим приложением, то можете использовать Spring только для внедрения зависимостей.

# 13

## Интерактивные запросы *Kafka Streams*

### В этой главе

- ✓ Как запросить состояние приложения Kafka Streams.
- ✓ Что требуется для включения поддержки интерактивных запросов.
- ✓ Создание приложения, поддерживающего интерактивные запросы, с помощью Spring Boot.

Ранее в книге вы узнали, как создавать приложения Kafka Streams с состоянием. Когда в приложение добавляется операция с состоянием, например агрегирование, Kafka Streams создает хранилище состояния для хранения результатов вычислений.

Операции агрегирования и подобные им нужны для получения информации об объединенных значениях. Например, представим, что приложение отслеживает неудачные попытки входа в систему. Слишком большое число попыток указывает на то, что кто-то пытается получить незаконный доступ к учетной записи или машины. Другой пример, не связанный с криминалом, — подсчет количества посещений страниц веб-сайта с целью определить наиболее выгодное время показа рекламы. Я мог бы продолжить перечислять примеры, но полагаю, что суть вы поняли. Однако, чтобы такая информация была полезной, у нас должна иметься возможность быстро получить ее. В противном случае она будет бесполезна.

В этой главе вы узнаете, как включить поддержку интерактивных запросов (interactive queries, IQ) в приложениях Kafka Streams, а также посмотрите, как с помощью Spring Boot создать веб-приложение мониторинга, отображающее результаты операций с состоянием. Для этого мы возьмем за основу пример приложения Kafka Streams, созданного нами ранее в этой книге.

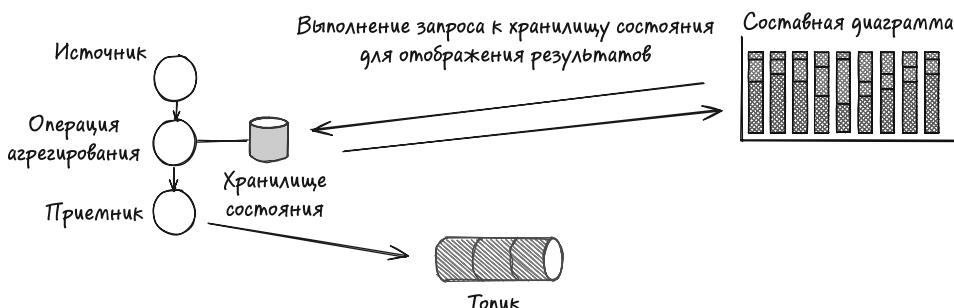
## 13.1. KAFKA STREAMS И ОБМЕН ИНФОРМАЦИЕЙ

Обычно, чтобы просмотреть результаты вычислений, выполненных приложением, необходимо сначала экспорттировать результаты в реляционную базу данных. Затем подключить приложение мониторинга к базе данных (БД), чтобы извлечь и отобразить информацию. На рис. 13.1 показано, как это может выглядеть.



**Рис. 13.1.** Экспортирование результатов операции с состоянием в базу данных для ее просмотра из приложения мониторинга

На этой схеме легко увидеть, как течет поток информации. Приложение выполняет вычисления с состоянием и экспортит результаты в базу данных, а приложение мониторинга запрашивает базу данных, чтобы получить информацию для отображения. Такой способ доступа к информации, сгенерированной приложением, вполне приемлем. Но Kafka Streams позволяет упростить этот процесс обмена информацией, воспользовавшись поддержкой IQ. IQ предоставляет возможность напрямую запрашивать информацию, содержащуюся в хранилище состояния. Такой прямой доступ поможет упростить общую архитектуру приложения, устранив ненужные движущиеся части, и получать данные по мере их вычисления. На рис. 13.2 показана схема, иллюстрирующая мои слова.



**Рис. 13.2.** Получение результатов операции с состоянием в приложении мониторинга непосредственно из источника

Как видите, прямой доступ к информации в хранилище состояния позволяет анализировать события по мере их появления, а также упростить архитектуру, убрав базу данных и шаги, связанные с передачей информации.

#### ПРИМЕЧАНИЕ

Я не утверждаю, что IQ избавляет от необходимости в реляционной базе данных. Базы данных наверняка всегда будут нужны в ваших приложениях. Но благодаря IQ у нас теперь есть еще один вариант, который в определенных ситуациях поможет получать необходимую информацию более эффективно.

## 13.2. ИНТЕРАКТИВНЫЕ ЗАПРОСЫ

IQ предоставляют возможность получать результаты операции с состоянием непосредственно из хранилища состояния. То есть мы можем просматривать состояние операции в режиме реального времени. Чтобы включить поддержку IQ, нужно выполнить два небольших шага.

1. Определить конфигурационный параметр `application.server`, содержащий имя хоста и номер порта, чтобы клиенты могли подключиться к экземпляру Kafka Streams. Если имеется несколько экземпляров приложения Kafka Streams (все с одинаковым идентификатором приложения), то для каждого экземпляра нужно задать уникальную комбинацию «хост:порт».
  2. Дать каждому хранилищу состояния уникальное имя с помощью конфигурационного объекта `Materialized`. Конечно, Kafka Streams даст свое имя каждому хранилищу состояния, если его не существует, но такие хранилища недоступны для запросов, если явно не указать их имена, а с этим могут возникнуть проблемы.
- Листинг 13.1 демонстрирует, что я имею в виду.

**Листинг 13.1.** Именование хранилищ состояния, чтобы получить возможность опрашивать их

```
Materialized.<String, LoanAppRollup>as(
    Stores.inMemoryKeyValueStore(loanAppStoreName))
    .withKeySerde(stringSerde)
    .withValueSerde(loanAppRollupSerde))
```

Задает тип хранилища с помощью StoreSupplier, которому нужно передать имя

```
Materialized.<String, LoanAppRollup>as(
    loanAppStoreName)
    .withKeySerde(stringSerde)
    .withValueSerde(loanAppRollupSerde))
```

Указывается только имя хранилища, а в качестве типа выбирается тип по умолчанию

```
Materialized.with(stringSerde, loanAppRollupSerde)
```

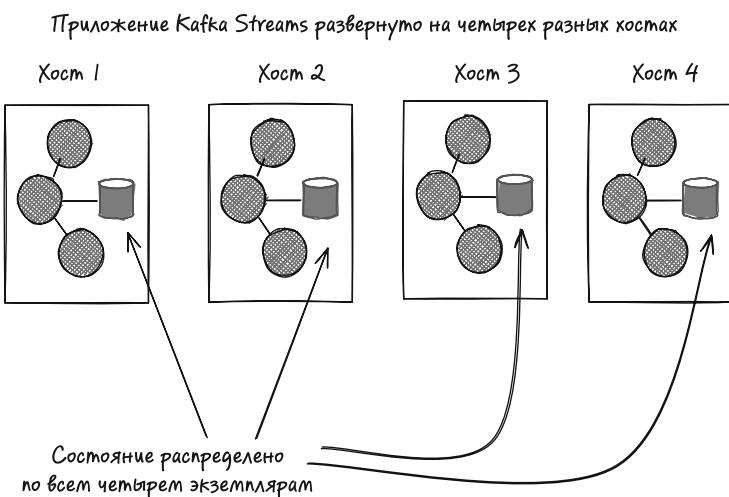
Передаются только объекты Serdes для (де)сериализации, а имя не указывается

В первом случае с помощью `StoreSupplier` мы указали, что будем использовать хранилище в памяти с заданным именем. Во втором случае мы указали только имя, положившись на тип хранилища по умолчанию, который не изменяется. В любом случае оба подхода включают поддержку запросов к хранилищу. Выбор `StoreSupplier` не имеет значения. При использовании любого из них мы должны указать имя

хранилища. В третьем случае мы не указываем ни имени, ни поставщика, поэтому такое хранилище будет недоступно для запросов.

Напомню, что Kafka Streams создает хранилище состояния для каждой задачи, и поскольку каждая задача представляет один раздел, мы получаем отдельное хранилище для каждого раздела. Это важно для нашего обсуждения, потому что при развертывании нескольких экземпляров приложения Kafka Streams каждый экземпляр отвечает только за подмножество общего числа задач (разделов), так как Kafka Streams распределяет нагрузку между приложениями. Например, если исходный топик имеет шесть разделов, а мы запустили три экземпляра приложения, то каждый из них будет отвечать за два раздела.

Поскольку хранилища состояний являются локальными для каждого экземпляра, эффект распределения нагрузки заключается в том, что состояние распределяется по нескольким машинам. Рисунок 13.3 поможет вам понять эту идею.

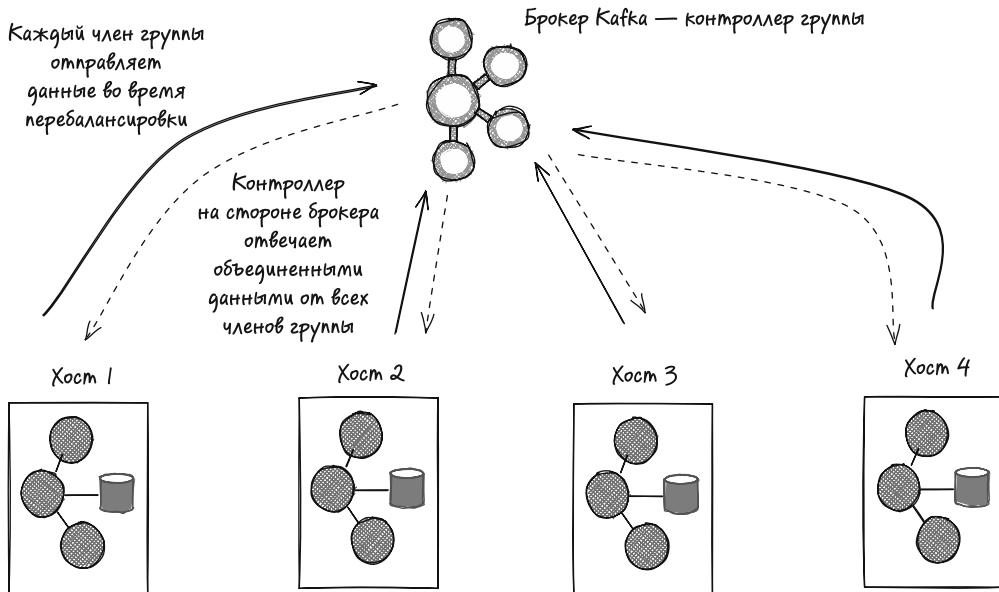


**Рис. 13.3.** Состояние в Kafka Streams распределено по нескольким машинам

Учитывая, что состояние распределено, а хранилища состояний в Kafka Streams являются хранилищами пар «ключ — значение», как узнать, какому экземпляру посыпать запрос? Оказывается, Kafka Streams предоставляет всю необходимую инфраструктуру, поэтому нам не нужно знать, какой экземпляр отвечает за раздел, где находится искомый ключ. Мы просто выбираем произвольный экземпляр, и если в его хранилище состояния нет требуемого ключа, то он перешлет запрос нужному экземпляру и вернет результат.

Такая маршрутизация запросов возможна благодаря способности протокола балансировки кодировать произвольные данные в полезной нагрузке. Поэтому, когда экземпляры приложений Kafka Streams балансируются, в дополнение

к предоставлению всех разделов, за которые каждый из них отвечает, они также получают индивидуальные конфигурации `application.server`. Взгляните на рис. 13.4, иллюстрирующий этот процесс.



**Рис. 13.4.** Протокол балансировки распределяет метаданные между экземплярами с одинаковым идентификатором приложения

Итак, каждое приложение получает метаданные с информацией обо всех разделах, за которые отвечают другие экземпляры (с тем же идентификатором приложения). Кроме того, каждому известна информация о сервере приложений, поэтому, когда приложению Kafka Streams посыпается запрос с заданным ключом, оно определяет раздел, в который попадет ключ, вычисляя хеш ключа по модулю количества разделов (предполагая, что секционирование производится по хешу). Если текущий экземпляр приложения Kafka Streams не владеет этим разделом, то оно может узнать, какому из экземпляров он принадлежит и пересыпает запрос, поскольку также знает хост и порт для запросов. Этот обмен метаданными и пересылка запросов — довольно сложный процесс, но иллюстрация на рис. 13.5 должна помочь понять, как он действует.

Kafka Streams обеспечивает внутреннюю инфраструктуру для обмена метаданными между экземплярами приложений с одинаковым идентификатором, но уровень связи между ними этого не делает. Поэтому нам необходимо реализовать обслуживание запросов и внутреннюю связь, что мы и сделаем в следующем разделе.

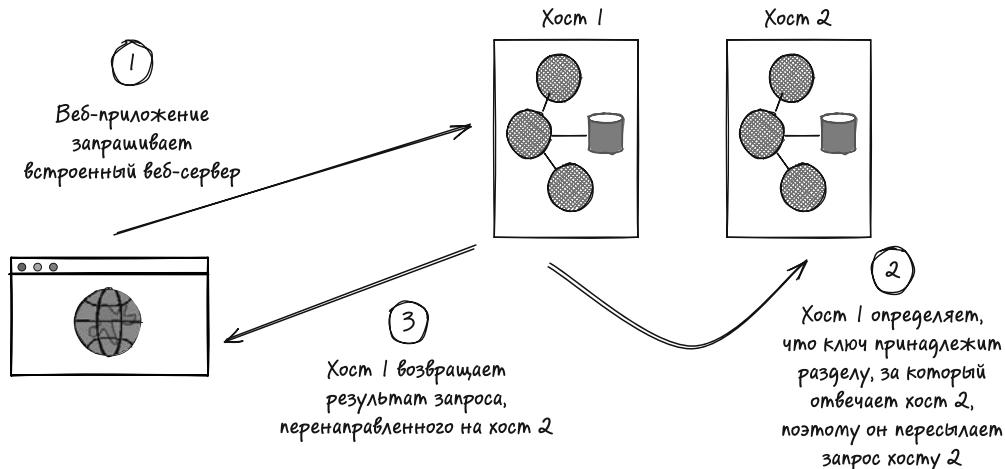


Рис. 13.5. Определение хоста для обработки запроса

### 13.2.1. Создание приложения интерактивных запросов с помощью Spring Boot

Далее мы шаг за шагом рассмотрим процесс создания слоя IQ в приложении Kafka Streams. Самое интересное, что мы уже заложили основу приложения в примере `bbejeck.spring.streams.container.KafkaStreamsLoanApplicationApplication`. Теперь нам предстоит выполнить следующие шаги.

1. Включить встроенный веб-сервер в приложении Spring Boot при запуске.
2. Добавить `RestController` для обработки входящих запросов, отправленных пользователями или родственными приложениями.
3. Внутри контроллера определим класс, реализующий логику обработки запросов.
4. Создадим HTML-страницу для выполнения вызовов REST API и постоянного отображения результатов рассмотрения заявок на кредит на одной странице.

В процессе разработки мы также познакомимся с последней версией IQ (`IQv2`), которая значительно лучше первой версии.

Итак, рассмотрим каждый пункт нашего списка по порядку, начав с включения веб-сервера при запуске. В нашем предыдущем приложении Spring Boot в главном методе мы явно отключили веб-сервер, как показано в листинге 13.2.

#### Листинг 13.2. Отключение веб-сервера

```
SpringApplicationBuilder applicationBuilder =
new SpringApplicationBuilder(LoanApplicationProcessingApplication.class)
    .web(WebApplicationType.NONE); ← Явное отключение
                                            веб-сервера
```

Мы отключили веб-сервер, так как он нам был не нужен. Мы не предусматривали прием и обработку внешних запросов в нашем приложении. Но теперь нам нужно, чтобы приложение принимало HTTP-запросы и позволяло узнать состояние заявки на кредит. Для этого следует изменить параметр-перечисление `WebApplicationType` в вызове конструктора `SpringApplicationBuilder` (листинг 13.3).

### Листинг 13.3. Включение веб-сервера в приложении Spring Boot

```
SpringApplicationBuilder applicationBuilder =
new SpringApplicationBuilder(KafkaStreamsLoanApplicationApplication.class)
    .web(WebApplicationType.SERVLET); ← Включение веб-сервера
```

То есть мы должны лишь изменить тип приложения с `NONE` на `SERVLET`. После чего веб-сервер будет автоматически запускаться вместе с приложением.

#### ПРИМЕЧАНИЕ

Сервлеты – это технология Java, которую можно развернуть на веб-серверах для создания веб-приложений. Мы не будем вдаваться в подробности, но при желании дополнительную информацию вы найдете по адресу <http://mng.bz/0laJ>.

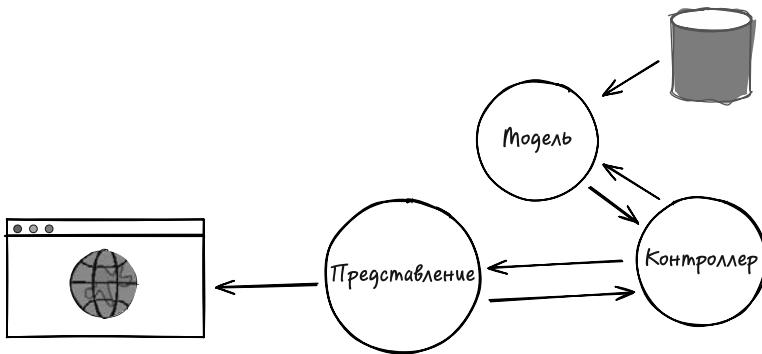
Следующий шаг – определение класса для обработки входящих запросов, как показано в листинге 13.4 (некоторые детали опущены для простоты).

### Листинг 13.4. Определение контроллера для обработки HTTP-запросов

```
@RestController ← Аннотация RestController
@RequestMapping("/loan-app-iq") ← Определяет базовый URL, который
public class LoanApplicationController {                                обслуживается этим контроллером
}
```

Первый шаг – объявить класс и добавить две новые аннотации на уровне класса. Аннотация `@RestController` выполняет два основных действия: отмечает класс как веб-контроллер и включает автоматическое преобразование объекта, который возвращается в ответ на запрос, в формат JSON. Аннотация `@RequestMapping` выбирает классы и методы для обработки входящих HTTP-запросов. Аннотация на уровне класса указывает, что этот контроллер обрабатывает все запросы, поступающие на заданный базовый URL.

Прежде чем двигаться дальше, уделим минуту определению термина «контроллер». Веб-контроллер является частью шаблона проектирования MVC (Model-View-Controller – «Модель – представление – контроллер») веб-приложений. В этом разделе мы создадим приложение Spring MVC. Модель представляет данные, представление отвечает за визуальное представление компонента, а контроллер – за прием и обработку входящих запросов и возврат ответа, отображаемого через компонент представления. На рис. 13.6 показано, как компоненты MVC в веб-приложении сочетаются друг с другом.



**Рис. 13.6.** Компоненты MVC в веб-приложении

Теперь, познакомившись с шаблоном проектирования MVC, вернемся к созданию контроллера. К настоящему моменту мы создали объявление класса и добавили аннотации, необходимые, чтобы получить контроллер, обрабатывающий HTTP-запросы. Теперь рассмотрим поближе, что нужно контроллеру для работы (см. листинг 13.5).

#### Листинг 13.5. Зависимости, необходимые контроллеру

```

@RestController
@RequestMapping("/loan-app-iq")
public class LoanApplicationController {

    @Value("${store.name}")      ← Поля для имени хранилища состояния
    private String storeName;

    @Value("${application.server}") ← Внедрение конфигурационного
    private String applicationServer;   параметра application.server

    private final KafkaStreams kafkaStreams; ← Поле для ссылки на экземпляр KafkaStreams
    private final RestTemplate restTemplate; ← RestTemplate используется
                                              для обработки HTTP REST-вызовов

    @Autowired
    public LoanApplicationController(KafkaStreams kafkaStreams,
                                      RestTemplate restTemplate) {
        this.kafkaStreams = kafkaStreams;
        this.restTemplate = restTemplate;
    }
}
  
```

Здесь мы видим поля для внедрения настроек, которые уже видели ранее, и аннотацию `@Autowired`, которая внедряет в параметр конструктора экземпляр `KafkaStreams`, встроенный в `KafkaStreamsContainer`, и экземпляр `RestTemplate` из Spring. Мы будем использовать их для связи с другими приложениями Kafka Streams при обработке запросов с ключами, за которые текущий экземпляр приложения не отвечает. Чтобы внедрить экземпляр `KafkaStreams`, нужно внести небольшое изменение в класс `KafkaStreamsContainer` (листинг 13.6).

**Листинг 13.6.** Экспортирование KafkaStreams как bean-компоненты Spring

```
@Bean
public KafkaStreams kafkaStreams() {
    return kafkaStreams;
}
```

Аннотация @Bean наделяет метод ролью провайдера bean-компонента Spring

Если добавить этот метод с аннотацией @Bean в класс KafkaStreamsContainer и при этом в контейнере есть другой класс, содержащий ссылку на объект KafkaStreams как зависимость, то контейнер Spring выполнит этот метод kafkaStreams для внедрения объекта в класс. Вызывая конструктор LoanApplicationController, контейнер Spring в точности следует этому процессу. Вы увидите, как контроллер использует объект KafkaStreams далее, когда мы перейдем к методу обработки запросов (листинг 13.7).

**Листинг 13.7.** Метод, аннотированный как обработчик входящих запросов для заданной категории кредитов

```
@GetMapping(value = "/loantype/{category}")
public QueryResponse<LoanAppRollup> getCategoryRollup(@PathVariable
    String category) {
    KeyQueryMetadata keyMetadata = getKeyValueMetadata(symbol,
        Serdes.String().serializer());
    if (keyMetadata == null) {
        return QueryResponse.withError(String.format(
            "ERROR: Unable to get key metadata after %d retries", MAX_RETRIES));
    }
}
```

Объявляет метод обработчиком запросов на получение информации об отдельных заявках на кредиты

Переменная пути в URL представляет категорию кредита

Аннотация @GetMapping на уровне метода LoanApplicationController.getCategoryRollup означает, что ему будут передаваться веб-запросы с URL `http://loan-app-iq/loantype/<category>`. Объект KeyQueryMetadata играет важную роль, потому что содержит информацию, помогающую определить, отвечает ли текущий запрашиваемый экземпляр за раздел, которому принадлежит ключ, указанный в запросе. getKeyMetadata — это просто внутренний метод и стоит того, чтобы кратко рассмотреть происходящее в нем (некоторые детали для простоты опущены) (листиング 13.8).

**Листинг 13.8.** Реализация метода getKeyMetadata, извлекающего KeyQueryMetadata

```
private <K> KeyQueryMetadata getKeyMetadata(K key,
    Serializer<K> keySerializer) {
    int currentRetries = 0;
    KeyQueryMetadata keyMetadata =
        kafkaStreams.queryMetadataForKey(storeName,
            key,
            keySerializer);
```

Выполняет метод KafkaStreams для извлечения информации о ключе

```
return keyMetadata;
```

В этом блоке кода можно увидеть одну из причин, по которой нам нужна ссылка на объект `KafkaStreams`: чтобы извлекать необходимую информацию о ключе, указанном в запросе. Одним из критических полей `KeyQueryMetadata` является поле `HostInfo`, которое содержит имя хоста и номер порта сервера, где находится экземпляр Kafka Streams, содержащий искомые данные. Чтобы получить возможность сравнивать имя хоста, мы используем конфигурационный параметр `application.server` для создания объекта `HostInfo` при конструировании контроллера (листинг 13.9).

#### Листинг 13.9. Создание объекта HostInfo на основе конфигурационного параметра `application.server`

```
@PostConstruct
public void init() {
    String[] parts = applicationServer.split(":");
    thisHostInfo = new HostInfo(parts[0],
        Integer.parseInt(parts[1])); ← Создает объект HostInfo на основе конфигурационного параметра
}
```

В методе `init` (выполняется контейнером Spring после вызова конструктора контроллера) мы создаем объект `HostInfo`. Он будет использоваться для сравнения с аналогичным объектом, который возвращается в метаданных, как показано в листинге 13.10 (некоторые детали опущены для простоты).

#### Листинг 13.10. Сравнение HostInfo из метаданных ключа с HostInfo текущего хоста

```
if (targetHostInfo.equals(thisHostInfo)) { ← Сравнивает целевой хост с текущим

    Set<Integer> partitionSet =
        Collections.singleton(keyMetadata.partition());

    StateQueryResult<LoanAppRollup> keyQueryResult =
        kafkaStreams.query(StateQueryRequest.inStore(storeName) ← Создает объект StateQueryRequest для выполнения запроса
            .withQuery(query)
            .withPartitions(partitionSet));

    QueryResult<LoanAppRollup> queryResult =
        keyQueryResult.getOnlyPartitionResult(); ← Извлекает результат в объект QueryResult

} else { ← Блок else обрабатывает ситуацию, когда текущий хост не является целевым
    String path = "/loantype/" + type;
    String host = targetHostInfo.host();
    int port = targetHostInfo.port();
    queryResponse = doRemoteRequest(host, port, path); ← Посыпает запросциальному узлу
}
```

Чтобы определить, правильному ли хосту был отправлен запрос, мы сравниваем объект `HostInfo`, полученный из метаданных ключа, с объектом, созданным контроллером. Если они одинаковы, то с помощью метода `KafkaStreams.query` выполняется запрос, а полученные результаты извлекаются и возвращаются. В противном случае мы извлекаем информацию о хосте и порте и отправляем запрос удаленному хосту с помощью вызова REST API. Я опустил некоторые детали в этом листинге, поэтому уделим минуту и рассмотрим их сейчас.

## ПРИМЕЧАНИЕ

Эти примеры кода создают несколько внутренних объектов, которые предназначены только для поддержки выполнения примера и не являются частью Spring или Kafka Streams. Я не буду описывать их, поскольку они никак не связаны с тем, что мы изучаем, но вы можете найти их реализацию в примерах исходного кода для книги.

Критически важным компонентом IQ является интерфейс `Query<R>`, который должны реализовать все запросы. В настоящее время существует четыре реализации:

- `KeyQuery`;
- `RangeQuery`;
- `WindowKeyQuery`;
- `WindowRangeQuery`.

`KeyQuery` используется для поиска результатов по отдельному ключу. `RangeQuery` — по диапазону ключей. `WindowKeyQuery` применяется при работе с оконными хранилищами состояний и позволяет указать время начала и конца просматриваемого интервала. `WindowRangeQuery` тоже предназначен для поиска в оконных хранилищах и, помимо временного диапазона, позволяет также задать диапазон ключей. За исключением конкретных параметров, предоставляемых отдельному объекту `Query`, все эти реализации используются похожим образом. Объект `KeyQuery` создается вызовом статического фабричного метода, предоставляемого классом (листинг 13.11).

### Листинг 13.11. Создание экземпляра KeyQuery

```
KeyQuery<String, LoanAppRollup>
    keyQuery = KeyQuery.withKey(loanType);
```

При вызове методу `KeyQuery.withKey` передается искомый ключ, а в ответ он возвращает объект `KeyQuery`. Тип `KeyQuery` — это ожидаемый результат запроса, ключ и значение.

Следующий шаг — создать объект `StateQueryRequest`, который затем передается методу `KafkaStreams.query`. Он использует шаблон построителя, как показано в листинге 13.12.

### Листинг 13.12. Создание экземпляра StateQueryRequest

```
StateQueryRequest.inStore(storeName)      ← Имя хранилища
    .withQuery(query)          ← Объект Query
    .withPartitions(partitionSet)) ← Соответствующие разделы
```

Объект `StateQueryRequest` содержит важную информацию, в том числе: имя хранилища состояния, объект запроса и раздел, где находится искомый ключ (мы получили его ранее из объекта `KeyMetadata`). Помимо обязательных параметров — имени хранилища и самого запроса, — другие параметры `StateQueryRequest` являются необязательными. К их числу относятся список разделов, требующих активных задач, и величина приемлемой задержки перед отправкой запроса резервным задачам. Запросы с привлечением резервных задач мы рассмотрим ниже в этой главе.

После выполнения метода `KafkaStreams.query` можно сохранить полученный `StateQueryResult` в переменной (листинг 13.13).

#### Листинг 13.13. Сохранение результатов запроса в переменной

```
StateQueryResult<LoanAppRollup> stateQueryResult = kafkaStreams.query(StateQueryRequest.inStore(storeName)
    .withQuery(query)
    .withPartitions(partitionSet));
```

Сохранит результаты запроса в переменной

Получив объект `StateQueryResult`, из него можно извлечь базовый результат, полученный из хранилища состояния. Общий результат запроса сохраняется в `Map`, роль ключа в котором играет раздел, а роль значения — объект `QueryResult`. В этом случае мы знаем, что результат относится только к одному разделу, поскольку это запрос ключа, а значит, мы можем использовать удобный метод `StateQueryResult.getOnlyPartitionResult` (листинг 13.14).

#### Листинг 13.14. Извлечение результата запроса

```
QueryResult<LoanAppRollup> queryResult = stateQueryResult.getOnlyPartitionResult();
```

Получает результат из единственного раздела

```
LoanAppRollup loanAppRollup = queryResult.getResult();
```

Извлекает результат `LoanAppRollup`

Теперь, получив результат, мы можем вернуть его веб-приложению, как показано в листинге 13.15 (некоторые детали опущены для простоты).

#### Листинг 13.15. Возврат результата компоненту представления веб-приложения

```
queryResponse = QueryResponse.withResult(queryResult.getResult().value());
// можно также добавить некоторые метаданные из запроса
return queryResponse;
```

Получает необработанный результат запроса

Возвращает результат представлению

После получения необработанного результата мы сохраняем его в пользовательском объекте, способном хранить метаданные для отображения в представлении. Веб-контроллер автоматически преобразует все данные в формат JSON, подготовливая их к отображению. В случае получения результатов из нескольких разделов мы можем получить `Map` и извлечь из него данные в цикле, как показано в листинге 13.16 (некоторые детали опущены для простоты).

#### Листинг 13.16. Извлечение результатов, полученных из нескольких разделов

```
Map<Integer, QueryResult<KeyValueIterator<String,
    LoanAppRollup>>> allPartitionsResult =
    result.getPartitionResults();
```

Извлекает `Map` с результатами

```
allPartitionsResult.forEach((key, queryResult) -> {
    // Выполнить некоторые операции с результатами
});
```

Выполняет цикл по содержимому `Map`

Отметим пару важных моментов, которые следует иметь в виду. Во-первых, при попытке извлечь результаты, полученные из нескольких разделов, с помощью метода `getOnlyPartition`, объект `QueryResult` выдаст исключение. Правило в этом случае очевидно. При запросе единственного ключа (то есть `KeyQuery`) можно смело

использовать прием извлечения из одного раздела, а во всех остальных случаях необходимо выполнить итерацию по полученному ассоциативному массиву Map.

К настоящему моменту мы видели, как обрабатывается запрос, когда искомый ключ находится на текущем хосте, но как быть, если для получения ключа нужно выполнить запрос к удаленному хосту? В этом случае контроллер использует внутренний метод `doRemoteQuery`, который мы видели выше в этом разделе. Реализация этого метода показана в листинге 13.17 (некоторые детали опущены для простоты).

#### Листинг 13.17. Выполнение запроса к удаленному хосту

```
private <V> QueryResponse<V> doRemoteRequest(String host,
                                              int port,
                                              String path) {
    QueryResponse<V> remoteResponse;
    try {
        remoteResponse = restTemplate.getForObject(BASE_IQ_URL + path,
                                                    QueryResponse.class,
                                                    host,           | Выполняет запрос к удаленному
                                                    port);         | хосту вызовом RestTemplate
    } catch (RestClientException exception) { ← Перехватывает возможные исключения
        remoteResponse = QueryResponse.withError(exception.getMessage());
    }
    return remoteResponse;
}
```

В случае когда ключ хранится в другом экземпляре приложения Kafka Streams, необходимо использовать `RestTemplate` для отправки вызова REST API удаленному хосту, который выполнит те же действия, что и текущий хост, и вернет результат, который затем текущий хост отправит представлению для отображения.

Прежде чем закончить эту главу, обсудим несколько дополнительных вопросов. Во-первых, подход с запросом диапазона ключей. В отличие от `KeyQuery`, с помощью которого можно запросить только один хост, запрос диапазона должен обрабатываться всеми экземплярами Kafka Streams из-за распределенного размещения разделов. Чтобы запустить запрос ко всем экземплярам приложения, нужно узнать, какие из них содержат искомое хранилище состояния и какие разделы им назначены (листинг 13.18).

#### Листинг 13.18. Подготовка запроса диапазона ключей ко всем экземплярам

```
Collection<StreamsMetadata> streamsMetadata =
    kafkaStreams.streamsMetadataForStore(storeName); ← Получит
List<LoanAppRollup> aggregations = new ArrayList<>();          | метаданные
streamsMetadata.forEach(streamsClient -> { ← Выполнит обход всех
    Set<Integer> partitions =                                | объектов StreamsMetadata
        getPartitions(streamsClient.topicPartitions());          | экземплярах
    QueryResponse<List<LoanAppRollup>> queryResponse =
        doRangeQuery(streamsClient.hostInfo(), ← Kafka Streams
                      Optional.of(partitions),
                      Optional.empty(),
                      lower,
                      upper);                                         | Выполнит запрос диапазона
                                                               | к удаленным хостам
```

Для выполнения запроса диапазона ключей по всем хостам нужно использовать `KafkaStreams.streamsMetadataForStore`, возвращающий метаданные о каждом экземпляре приложения Kafka Streams, имеющего хранилище в своей топологии. Затем можно выполнить обход метаданных в цикле и выполнить локальный запрос к текущему экземпляру и удаленные к остальным.

Второй вопрос, который мы должны обсудить, — возможность запросить резервную задачу. Когда экземпляр Kafka Streams переходит в автономный режим, выполняется перебалансировка для перераспределения разделов между оставшимися активными экземплярами. Это может занять некоторое время, поэтому Kafka Streams предлагает компромисс между доступностью и согласованностью, позволяя запрашивать резервные задачи. Компромисс заключается в том, что резервные задачи могут отставать от основных и хранить не все данные, которые имеются у активных экземпляров, но они все равно могут обслуживать запросы и возвращать результаты.

Чтобы запросить резервный сервер, нужно предусмотреть обработку любых ошибок, которые могут возникнуть при запросе активного хоста, перехватывая `RestClientException`. А столкнувшись с ошибкой, переслать запрос резервному серверу, как показано в листинге 13.19.

**Листинг 13.19.** Отправка запроса резервному серверу в случае ошибки обработки запроса активным хостом

```
if (queryResponse.hasError() && !standbyHosts.isEmpty()) {
    Optional<QueryResponse<LoanAppRollup>> standbyResponse =
        standbyHosts.stream()
            .map(standbyHost -> doKeyQuery(standbyHost,
                keyQuery,
                keyMetadata,
                symbol,
                HostStatus.STANDBY))
            .filter(resp -> resp != null && !resp.hasError())
            .findFirst(); ← Прерывает цикл при получении результата
    if (standbyResponse.isPresent()) {
        queryResponse = standbyResponse.get();
    }
}

return queryResponse; ← Возвращает результат
```

При обнаружении ошибки и включении режима опроса резервных задач, получив ошибку обработки запроса активной задачей, можно переслать запрос резервной задаче. Для использования этой возможности следует включить резервные задачи через конфигурацию.

## ИТОГИ ГЛАВЫ

- Поддержка интерактивных запросов (IQ) дает возможность напрямую запрашивать информацию, содержащуюся в хранилище состояния.
- Прямой доступ к результатам операций с состоянием упрощает архитектуру и в некоторых случаях может избавить от необходимости использовать реляционную БД для отображения результатов.

- Передача имени материализованному объекту напрямую или через `StoreSupplier` позволяет хранилищу обрабатывать запросы.
- Spring Boot по умолчанию автоматически запускает веб-сервер вместе с приложением. Этот встроенный веб-сервер отлично подходит для приложений Kafka Streams, поскольку он автоматически доступен слою `IQ`.
- При использовании аннотации `@RestController` в классе контроллера методы, обрабатывающие веб-запросы, будут автоматически преобразовывать возвращаемые объекты с ответом в формат JSON.

# 11

## Тестирование

### В этой главе

- ✓ Разница между модульным и интеграционным тестированием.
- ✓ Тестирование производителей и потребителей Kafka.
- ✓ Создание тестов для операторов Kafka Streams.
- ✓ Тестирование топологии Kafka Streams.
- ✓ Разработка эффективных интеграционных тестов.

К настоящему моменту мы рассмотрели основные стандартные блоки, используемые при создании приложений Kafka Streams: производители и потребители Kafka, Kafka Connect и Kafka Streams. Но я не упомянул еще один важнейший элемент разработки приложений: тестирование. Одна из главных концепций, на которой мы сосредоточим наше внимание, — размещение бизнес-логики в автономных классах, полностью независимых от приложения Kafka Streams, с целью упрощения тестирования. Я полагаю, что вы знаете, насколько важно тестирование, но все же упомяну две основные, с моей точки зрения, причины, почему тестирование не менее важно, чем сам процесс разработки.

Во-первых, разрабатывая свой код, вы подписываете негласное соглашение относительно того, что можно ожидать от кода. Единственный способ доказать, что код работает, — тщательно протестировать его. При тестировании вы должны учесть широкий спектр возможных входных данных и сценариев, чтобы убедиться в должной работе кода в допустимых условиях. Во-вторых, всестороннее тестирование необходимо, так как неизбежны изменения программного обеспечения. Хороший

набор тестов обеспечивает немедленную обратную связь в случае, когда код начинает вести себя не так, как ожидалось.

Кроме того, при масштабном рефакторинге или добавлении новой функциональности благополучное выполнение тестов даст вам уверенность, что вы выпускаете надежное и работоспособное приложение. Тестируя приложения Kafka Streams не всегда просто, даже если вы понимаете, насколько это важно. Можно запустить на выполнение простую топологию и посмотреть на результаты, но у этого подхода есть один недостаток. Нам нужен набор воспроизводимых (repeatable) тестов, которые можно запустить в любой момент в качестве составной части сборки. Еще одним важным аспектом тестирования является необходимость запускать тесты как можно быстрее. Поэтому вы, вероятно, предпочтете запускать большую часть своих тестов без брокера Kafka. Тестируя без брокера Kafka — один из самых важных моментов в этой главе. Но будут моменты, когда для эффективного тестирования вам понадобится действующий брокер. Это разница между использованием и не использованием брокера при тестировании определяет границу между модульным и интеграционным тестированием.

## 14.1. РАЗНИЦА МЕЖДУ МОДУЛЬНЫМ И ИНТЕГРАЦИОННЫМ ТЕСТИРОВАНИЕМ

В этом разделе дается субъективное определение модульного и интеграционного тестирования. Модульное тестирование я определяю как тестирование, проверяющее определенный подкомпонент приложения, определенную точку в логике. Например, предположим, что у нас есть приложение для международной торговли, которое, получая заказ, должно немедленно конвертировать сумму сделки в доллары США. Модульное тестирование сможет проверить только часть конвертации валюты с использованием отдельных тестов для разных типов ожидаемых валют или одного параметризованного теста.

Такие тесты обычно проверяют методы классов и выполняются очень быстро. Проблема модульного тестирования заключается в том, что часто существуют внешние зависимости — например, для запуска приложения может потребоваться брокер Kafka. Поэтому, если следовать нашему непосредственному примеру, возникает вопрос: должны ли мы запустить брокер Kafka, чтобы проверить конвертацию различных типов валют? Ответ на этот вопрос: нет, и я объясню его с помощью аналогии.

Представьте, что вы играете в спектакле в местном театре и должны выучить свою роль. Вам не нужны другие актеры на сцене, чтобы сделать это, только суплер, который подскажет вам необходимые реплики. Суплер также может подсказать вам, все ли вы сделали правильно.

То же относится и к модульному тестированию. Нам не нужна вся система для тестирования определенной части, нам нужен лишь механизм для предоставления входных данных и возможность проверить результат. Этот механизм называется фиктивным объектом (имитацией). Фиктивный объект имеет тот же интерфейс, что и удаленный подключаемый компонент, но не имеет фактического поведения. Мы явно указываем фиктивному объекту, что он должен делать, например представлять определенные значения, а затем проверяем результаты.

Вернемся к нашей аналогии со спектаклем. Иногда будет возникать необходимость собрать всех вместе на репетицию, чтобы убедиться, что все актеры знают свои реплики и взаимодействуют друг с другом так, как ожидается. Этот тип тестирования в программировании называется интеграционным тестированием. Это лишь пробный прогон программного обеспечения, но в этом прогоне не участвуют имитации. Все внешние компоненты — настоящие.

Интеграционные тесты тоже важны, но их количество будет отличаться от количества модульных тестов. Одна из главных причин, почему модульных тестов должно быть больше, чем интеграционных, — скорость выполнения. Типичный модульный тест выполняется за доли секунды, а интеграционный может выполняться до нескольких секунд и даже минут. Такая продолжительность не обременительна, когда выполняется один тест, но, когда тестов сотни или тысячи, вы поймете, почему необходимо, чтобы набор тестов выполнялся как можно быстрее.

Итак, какой признак отличает модульные тесты от интеграционных? Как я уже говорил выше, модульные тесты применяются для проверки поведения отдельных методов. Признаком, говорящим о необходимости интеграционного теста, является потребность проверить реальное поведение удаленного компонента. Например, может понадобиться посмотреть, как действует клиентское приложение Kafka при выполнении перебалансировки. В этом случае нам потребуется интеграционный тест, запускающий реальную перебалансировку и проверяющий поведение приложения. Продолжая наш пример с приложением Kafka, мы можем запустить интеграционный тест с брокером Kafka в образе Docker и проверить поведение в комплексе с действующим брокером. Однако такой тест будет целиком выполнятьсь в локальной среде разработки (то есть на нашем ноутбуке). Давайте рассмотрим табл. 14.1, в которой обобщается разница между модульным и интеграционным тестированием.

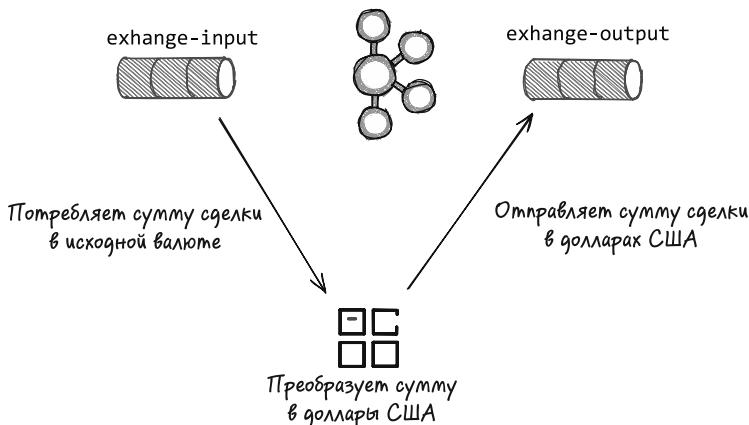
**Таблица 14.1.** Сравнение модульного и интеграционного тестирования

Тип	Цель	Скорость тестирования	Объемы использования
Модульные	Уровень метода, мелкая логика, изолированные объекты	Доли секунды	Большинство
Интеграционные	Сквозное тестирование, интеграция компонентов, крупные блоки логики	От секунд до минут	Меньшинство

### 14.1.1. Тестирование производителей и потребителей Kafka

Предположим, у нас есть приложение, выполняющее простую операцию конвертации валюты, как показано на рис. 14.1.

Результаты сделок в валюте, отличной от валюты США, выводятся в топик Kafka с именем `exchange-input`. Наше приложение потребляет данные из этого топика, конвертирует сумму в валюту США, а затем выводит преобразованную сумму в другой топик Kafka `exchange-output`. Сокращенный код показан в листинге 14.1 (некоторые детали опущены для простоты).



**Рис. 14.1.** Приложение конвертации валют в доллары США

#### Листинг 14.1. Класс конвертации валют CurrencyExchangeClient

```
public void runExchange() {
    while (keepRunningExchange) {
        ConsumerRecords<String, CurrencyExchangeTransaction> consumerRecords =
            exchangeConsumer.poll(Duration.ofSeconds(5));
        consumerRecords.forEach(exchangeTxn -> {
            CurrencyExchangeTransaction tx = exchangeTxn.value();
            double convertedAmount =
                tx.currency().exchangeToDollars(tx.amount());
            CurrencyExchangeTransaction converted =
                new CurrencyExchangeTransaction(convertedAmount,
                    CurrencyExchangeTransaction.Currency.USD);
            ProducerRecord<String, CurrencyExchangeTransaction> producerRecord =
                new ProducerRecord<>(outputTopic, converted);
            exchangeProducer.send(producerRecord...)
        });
    }
}
```

Это простое приложение, и нам нужно проверить процесс конвертации валют. Напишем для этого тест и запустим его как модульный тест, что означает, что ему не потребуется действующий брокер Kafka. Мы разработали класс так, что только интерфейсы `Consumer` и `Producer` являются ожидаемыми типами (листинг 14.2).

#### Листинг 14.2. Объявление интерфейсов в конструкторе

```
public CurrencyExchangeClient(
    final Consumer<String, CurrencyExchangeTransaction> exchangeConsumer,
    final Producer<String, CurrencyExchangeTransaction> exchangeProducer,
    final String inputTopic,
    final String outputTopic) {
    this.exchangeConsumer = exchangeConsumer;
    this.exchangeProducer = exchangeProducer;
    this.inputTopic = inputTopic;
    this.outputTopic = outputTopic;
}
```

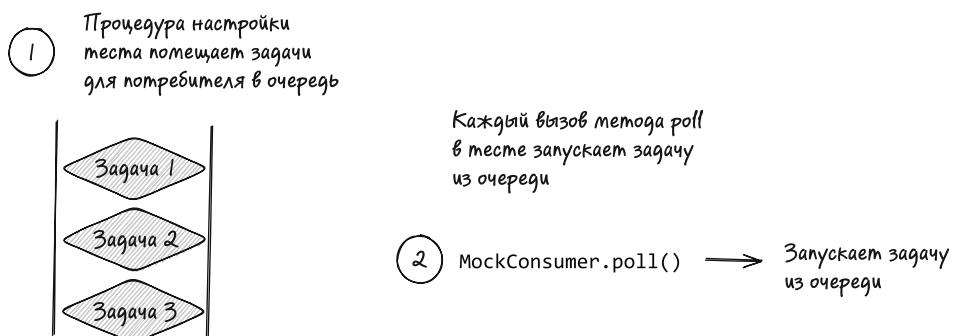
Параметр типа `Consumer`

Параметр типа `Producer`

Указав, что наше приложение использует только интерфейсы (верное решение во всех отношениях), мы подготовились к быстрому тестированию класса `CurrencyExchangeClient` с помощью классов `MockConsumer` и `MockProducer`. `MockConsumer` реализует интерфейс `Consumer`, а `MockProducer` — интерфейс `Producer`, поэтому мы можем использовать их в teste и выполнить без привлечения действующего брокера. Мы также можем проверить взаимодействие потребителя и производителя и конечный результат.

Главный недостаток использования фиктивных объектов — необходимость описывать все взаимодействия и шаги, которые они должны выполнить. Начнем с потребителя. Есть несколько моментов, которые следует учитывать при создании теста. Клиент конвертации валют работает в цикле бесконечно, пока не выполнится метод `CurrencyExchangeClient.close`. А так как он работает в цикле, то, как только мы вызовем `CurrencyExchangeClient.runExchange` в teste, управление уже не вернется тесту, пока цикл не завершится, что порождает ситуацию «курицы и яйца».

Итак, нам нужен простой способ остановить цикл без запуска дополнительного потока в teste. Циклы — обычное решение при работе с `KafkaConsumer`. В идеале мы хотели бы, чтобы потребитель работал бесконечно, так как потоки событий никогда не останавливаются. К счастью, `MockConsumer` предоставляет эту возможность в виде метода `schedulePollTask`, который позволяет поставить задачу в очередь, которую потребитель будет выполнять для каждого вызова `poll(Duration)`. Схематически этот процесс показан на рис. 14.2.



**Рис. 14.2.** Описание поведения потребителя с помощью очереди задач

Итак, вызовы `MockConsumer.schedulePollTask` будут добавлять задачи для потребителя (в виде экземпляров `Runnable`), которые тот будет выполнять по порядку при каждом вызове `poll()`. Посмотрим, как это реализуется (листинг 14.3).

### Листинг 14.3. Использование `schedulePollTask`

```
@Test
void runExchangeApplicationTest()
CurrencyExchangeClient exchangeClient = new CurrencyExchangeClient(
    mockConsumer,
    mockProducer,
    "input",
    "output");
    
```

Создает экземпляр клиента для тестирования

```

mockConsumer.schedulePollTask(() -> {
    final Map<TopicPartition, Long> beginningOffsets = new HashMap<>();
    TopicPartition topicPartition = new TopicPartition("input", 0);
    beginningOffsets.put(topicPartition, 0L);
    mockConsumer.rebalance(Collections.singletonList(topicPartition));
    mockConsumer.updateBeginningOffsets(beginningOffsets);
});

```

Добавляет задачу в очередь

Здесь мы создали экземпляр класса для тестирования, затем добавили в очередь первую задачу для потребителя. В данном случае все эти настройки необходимы для приведения `MockConsumer` в исходное состояние. Обратите внимание, что задачи, которые мы предоставляем, не должны возвращать записи, которые потребитель вернет из вызова `poll`. Это произвольный код, который нам нужно запустить. Однако в какой-то момент мы должны предоставить записи, чтобы потребитель мог вернуть их, выполняя код в цикле, что мы и сделаем дальше (листинг 14.4).

#### **Листинг 14.4.** Добавление задач, возвращающих записи

```

mockConsumer.schedulePollTask(() -> {
    mockConsumer.addRecord(new ConsumerRecord<>("input", 0, 0, null,
        euroTransaction));
    mockConsumer.addRecord(new ConsumerRecord<>("input", 0, 1, null,
        gbpTransaction));
    mockConsumer.addRecord(new ConsumerRecord<>("input", 0, 2, null,
        jpyTransaction));
});

```

С помощью этой задачи мы создаем три записи, которые потребитель вернет из следующего вызова `poll`, и все эти записи должны быть обработаны именно в этом порядке и переданы производителю. Наконец, мы добавим еще одну задачу, которая закроет клиентское приложение (листинг 14.5).

#### **Листинг 14.5.** Добавление конечной задачи, которая завершит цикл конвертации валют

```

mockConsumer.schedulePollTask(exchangeClient::close); ←
exchangeClient.runExchange(); ← Запускает приложение внутри теста

```

Добавляет вызов метода, который завершит приложение

Итак, последняя задача, добавляемая в очередь, вызывает метод `CurrencyExchangeClient.close`, который закроет приложение. Следующая инструкция запустит `CurrencyExchangeClient` внутри теста. Цикл выполнит три итерации, потому что мы предоставили три задачи, после чего приложение будет корректно завершено. Но нам также нужно проверить, получил ли производитель ожидаемые записи с суммами сделок, конвертированными в доллары США (листинг 14.6).

#### **Листинг 14.6.** Проверка производителя в teste

```

List<CurrencyExchangeTransaction> actualTransactionList = mockProducer
    .history() ← Использует метод history
    .stream() | объекта MockProducer
    .map((ProducerRecord::value))
    .toList();

assertThat(actualTransactionList.get(0), equalTo(expectedEUROToUS));
assertThat(actualTransactionList.get(1), equalTo(expectedGBDToUS));
assertThat(actualTransactionList.get(2), equalTo(expectedJPYToUS));

```

Чтобы убедиться, что производитель получил записи с правильно конвертированными суммами сделок и в правильном порядке, мы вызываем метод `MockProducer.history`, затем отображаем полученный список записей `ProducerRecord` в список значений и сравниваем их по одному с ожидаемыми значениями, которые были определены ранее в тесте. Весь этот тест можно найти в файле `streams/src/test/java/bbejeck/chapter_14/CurrencyExchangeClientTest.java` в примерах исходного кода для книги. В `custom-connector/src/test/java/bbejeck/chapter_5` вы также найдете тесты для коннектора Kafka, разработанного в главе 5. Я не включил этот код в данную главу, так как он во многом повторяется и тоже использует фиктивные объекты.

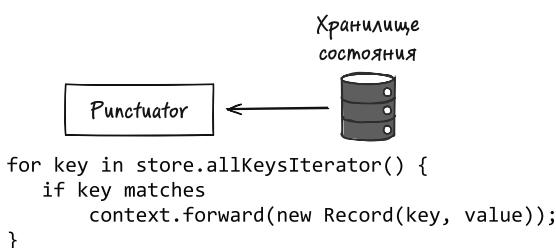
Мы протестируем приложение `CurrencyExchangeClient` в модульном teste, который выполняется очень быстро. Мы можем и должны добавить также дополнительные тесты для проверки различных условий ошибок, но я не буду их здесь рассматривать. Теперь давайте продолжим обсуждение тестирования с помощью фиктивных объектов и рассмотрим приемы тестирования операторов Kafka Streams.

### 14.1.2. Тестирование операторов Kafka Streams

Топологии Kafka Streams практически всегда содержат одну или несколько операций, принимающих интерфейс с единственным абстрактным методом (*single abstract method, SAM*). Их применение позволяет быстро создавать приложения без определения конкретных классов, подставляя лямбда-выражения, удовлетворяющие требования к поведению. Но такой подход делает практически невозможным создание теста, проверяющего только одну конкретную операцию, потому что ее реализация определяется прямо в коде. Другой вариант — создать конкретный класс, реализующий ожидаемый интерфейс SAM. Однако некоторые из этих интерфейсов могут работать с другими объектами Kafka Streams внутри приложения, что усложняет тестирование.

Например, рассмотрим интерфейс `Punctuator`. Поскольку у него есть только один метод `punctuate`, он квалифицируется как интерфейс SAM. Обычно `Punctuator` не работает сам по себе и требует взаимодействия с другими объектами.

В этом разделе я покажу, как сымитировать произвольные интерфейсы и объекты с помощью фреймворка Mockito (<https://site.mockito.org/>), упрощающего тестирование взаимодействий с внешними объектами. Для этого примера мы создадим тест для класса `bbejeck.chapter_9.punctuator.StockPerformancePunctuator` (рис. 14.3).



**Рис. 14.3.** Punctuator проверяет все записи в хранилище состояний и пересыпает те из них, которые соответствуют условию

При выполнении этот экземпляр пунктуатора проверяет содержимое хранилища состояния и пересыпает все записи, соответствующие определенным критериям. Я не буду здесь обсуждать использование пунктуаторов в Kafka Streams, а сосредоточусь исключительно на тестировании. Чтобы получить представление об объектах, с которыми взаимодействует пунктуатор, взглянем на конструктор (листинг 14.7).

**Листинг 14.7.** Конструктор класса Punctuator определяет типы объектов, с которыми осуществляются взаимодействия

```
public StockPerformancePunctuator(double differentialThreshold,
    ProcessorContext<String, StockPerformance> context,
    KeyValueStore<String, StockPerformance> keyValueStore) {
    this.differentialThreshold = differentialThreshold;
    this.context = context;
    this.keyValueStore = keyValueStore;
}
```

Как видите, первый параметр конструктора — это примитивный тип Java и не вызывает проблем при тестировании, а два других являются интерфейсами Kafka Streams и имеют ожидаемое поведение во время вызова пунктуации. В частности, код будет перебирать все записи из хранилища состояния и пересыпать те, что соответствуют критериям оценки. Однако благодаря использованию фиктивных объектов написать этот тест будет просто. Сначала рассмотрим создание фиктивных объектов для теста (листинг 14.8).

**Листинг 14.8.** Создание необходимых фиктивных объектов

```
@BeforeEach
public void setUp() {
    context = mock(ProcessorContext.class); ← Создаст фиктивный объект
    keyValueStore = mock(KeyValueStore.class); ← для использования в роли ProcessorContext
    stockPerformancePunctuator =
        new StockPerformancePunctuator(
            differentialThreshold, ← Создаст фиктивный объект
            context, ← для использования в роли KeyValueStore
            keyValueStore); ← Передача всех параметров конструктору
}
```

Чтобы создать фиктивный объект, достаточно вызвать `Mockito.mock` (здесь получен в результате статического импорта) и передать класс или интерфейс объекта в фиктивный объект. В нашем случае Mockito возвращает объект, реализующий ожидаемый интерфейс, поэтому, чтобы удовлетворить требования к параметрам, мы передаем возвращаемые объекты конструктору `StockPerformancePunctuator`.

Фиктивные объекты удовлетворяют требованиям интерфейса, но у них отсутствует какое-либо поведение, поэтому следующим шагом нужно сообщить фиктивным объектам, как они должны себя вести, что мы и сделаем в тестовом методе (некоторые детали опущены для простоты) (листинг 14.9).

**Листинг 14.9.** Тест для StockPerformancePunctuator

```

@Test
void shouldPunctuateRecordsTest() {
    StockPerformance stockPerformance =
        getStockPerformance(); ← Создает необходимый объект Protobuf
    Iterator<KeyValue<String, StockPerformance>>
    storeKeyValues =
        List.of(KeyValue.pair("CLFT", stockPerformance)).iterator(); ← Создает экземпляр Iterator для передачи фиктивному объекту KeyValueStore
    long timestamp = Instant.now().toEpochMilli();

    Record<String, StockPerformance> record =
        new Record<>("CFLT", stockPerformance, timestamp); ← Возвращаемая запись
    when(keyValueStore.all()) ← Сообщает фиктивному хранилищу пар «ключ — значение», что нужно сделать
        .thenReturn(           при вызове метода all
        TestUtils.kvIterator(storeKeyValues.iterator()));
    context.forward(record); ← Устанавливает ожидаемый вызов для имитации ProcessorContext
    stockPerformancePunctuator.punctuate(timestamp); ← Вызывает тестируемый метод
    verify(context, times(1)).forward(record); ← Проверяет работу имитации ProcessorContext

```

Некоторые из этих шагов создают объекты, необходимые для работы теста. Мы создаем объект `StockPerformance` с нужными свойствами, удовлетворяющими критериям оценки. Затем создаем `Iterator`, предварительно создав `ArrayList`, содержащий объект `KeyValue`, который должно вернуть хранилище. Далее создаем экземпляр `Record`, чтобы передать его имитации `ProcessorContext` для пересылки.

Далее мы сообщаем фиктивному объекту `KeyValueStore`, что при вызове метода `all` следует вернуть этот фиктивный экземпляр `KeyValueIterator` (созданный вспомогательным методом путем обертывания экземпляра `Iterator` в `KeyValueIterator`), который будет использовать фактический итератор, созданный ранее. Затем мы определяем поведение имитации `ProcessorContext`. В частности, она должна ждать вызова метода `forward` с объектом `Record`, созданным ранее.

Наконец мы вызываем метод `Punctuator.punctuate`, который выполнит код внутри метода, включая фиктивные объекты, и проверяем, сделал ли фиктивный объект `ProcessorContext` то, что от него ожидалось. Класс `TestUtils` вы найдете в файле `streams/src/main/java/bbejeck/utils/TestUtils.java`.

В этом разделе мы узнали, как проводить модульное тестирование операторов Kafka Streams с использованием фиктивных объектов взамен настоящих. Далее мы протестируем приложение Kafka Streams без фиктивных объектов и брокера.

### 14.1.3. Тестирование топологии

Тестирование приложений Kafka Streams должно осуществляться на двух уровнях. Обычно желательно писать реализацию различных операций, фильтров, отображений, агрегаторов и так далее в виде конкретных классов, чтобы потом иметь возможность тестировать их по отдельности. Но Kafka Streams DSL принимает их как лямбда-выражения, что дает возможность написать полноценное приложение без особых усилий. Но даже при тестировании этих различных функций по отдельности

с помощью модульных тестов важно иметь тесты, которые проверяют топологию целиком, чтобы убедиться, что все узлы работают вместе, как ожидалось.

Но есть ли возможность разрабатывать быстрые тесты, если приложение Kafka Streams предполагает работу с брокером Kafka? Да, есть, и создать такой быстрый модульный тест для топологии нам поможет класс `TopologyTestDriver`, разработанный специально для тестирования топологий Kafka Streams (вместе с хранилищами состояний) без привлечения брокера. Лучший способ познакомиться с `TopologyTestDriver` — опробовать его на практике. Начнем с нашего первого примера приложения Kafka Streams — приложения Yelling.

Поскольку это наш первый опыт использования `TopologyTestDriver`, мы рассмотрим каждую часть настройки теста, но в последующих примерах я буду показывать только основную логику тестирования (листинг 14.10).

#### Листинг 14.10. Настройка теста для приложения Yelling

```
@Test
@DisplayName("Should Yell At Everyone")
void yellingTopologyTest() {
    KafkaStreamsYellingApp yellingApp = new KafkaStreamsYellingApp(); ← Создает экземпляр KafkaStreamsYellingApp
    Topology yellingTopology = yellingApp.topology(new Properties()); ← Извлекает объект Topology из приложения
    Serializer<String> stringSerializer = Serdes.String().serializer(); ← Создает сериализатор для объектов String
    Deserializer<String> stringDeserializer =
        Serdes.String().deserializer(); ← Создает десериализатор для объектов String
```

Настройка начинается как обычно: метод теста декорируется аннотациями `@Test` и `@DisplayName`. Внутри метода мы сначала создаем экземпляр `KafkaStreamsYellingApp`, который понадобится в следующей строке для извлечения топологии с помощью метода `KafkaStreamsYellingApp.topology`. Это та же самая топология, которая создается при запуске приложения. Мы также создаем сериализатор и десериализатор — они понадобятся нам на следующем шаге, когда мы перейдем к созданию более важных частей теста (листинг 14.11).

#### Листинг 14.11. Создание экземпляра `TopologyTestDriver`, а также входного и выходного топиков

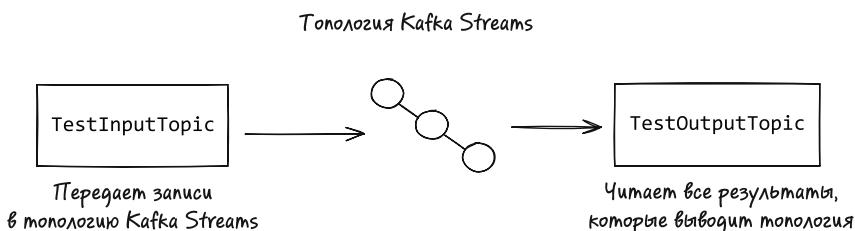
```
try (TopologyTestDriver driver =
        new TopologyTestDriver(yellingTopology)) { ← Создает экземпляр TopologyTestDriver
    TestInputTopic<String, String> inputTopic = ← Создает топик TestInputTopic
        driver.createInputTopic("src-topic",
            stringSerializer,
            stringSerializer);
    TestOutputTopic<String, String> outputTopic = ← Создает топик TestOutputTopic
        driver.createOutputTopic("out-topic",
            stringDeserializer,
            stringDeserializer);
```

На следующем шаге мы создаем экземпляр `TopologyTestDriver`, который будет служить платформой для запуска тестируемой топологии, переданной конструктору в параметре. Класс `TopologyTestDriver` имеет несколько перегруженных конструкторов, мы рассмотрим их позже с другими примерами.

Далее нужно создать `TestInputTopic` для передачи записей в тестируемую топологию. При создании объекта `TestInputTopic` в первом параметре конструктору нужно передать имя топика, и это имя должно совпадать с именем топика, использованным при построении топологии. В других параметрах передаются сериализаторы для ключей и значений, созданные при запуске теста. `TestInputTopic` будет сериализовать все переданные ключи и значения, чтобы приложение Kafka Streams получало ожидаемые массивы байтов, как при реальном запуске.

Нам также понадобится выходной топик для топологии, куда будут записываться результаты. Его мы получаем, когда создаем `TestOutputTopic`. В первом параметре конструктору точно так же передается имя топика, в частности совпадающее с именем выходного топика, используемым топологией в реальном приложении. В других параметрах передаются десериализаторы ключей и значений: топология сериализует выводимые данные, поэтому нам понадобится возможность преобразовать их обратно в конкретные объекты для проверки результатов теста.

Следующий шаг — отправка входных записей в топологию и последующая проверка выходных данных, что выглядит примерно так, как показано на рис. 14.4.



**Рис. 14.4.** Передача записей в топологию и последующее чтение выходных данных

То есть процесс вызывает соответствующий метод `TestInputTopic` для отправки значений в топологию, а затем использует `TestOutputTopic` для получения результатов (листинг 14.12).

#### Листинг 14.12. Отправка записей в топологию и проверка результатов

```

List<String> inputValues = List.of("if you don't eat your meat", ←
    "you can't have any pudding!", ←
    "How can you have any pudding", ←
    "if you don't eat your meat!" ); ←
    | Составляет список
    | входных значений

inputTopic.pipeValueList(inputValues); ←
    | Псыает все входные
    | данные в топологию

List<String> expectedOutput = inputValues.stream() ←
    .map(String::toUpperCase) ←
    .toList(); ←
    | Создает список
    | ожидаемых значений

List<String> actualOutput = outputTopic.readValuesToList(); ←
    | Читает результаты
    | из топологии в список

assertThat(actualOutput, equalTo(expectedOutput)); ←
    | Сравнивает полученные
    | результаты с ожидаемыми
  
```

В этом последнем фрагменте примера мы сначала создаем образец входных данных (при тому, кто сможет назвать исполнителя и песню, которую мы использовали). Затем мы отправляем образец в топологию с помощью метода `TestInputTopic.pipeValueList`. В этом случае, поскольку наше приложение Kafka Streams работает только со значениями из пар «ключ — значение», мы можем передать все значения в списке. Но послать можно не только список значений — `TestInputTopic` предлагает ряд дополнительных методов для передачи входных тестовых данных. Я перечислю некоторые из них в листинге 14.13.

#### Листинг 14.13. Другие методы для отправки входных данных в `TestInputTopic`

<code>pipeInput(K key, V value)</code>	← Помыает ключ и значение	Помыает ключ и значение вместе с отметкой времени события
<code>pipeInput(K key, V value, Instant)</code>	←	
<code>pipeInput(TestRecord&lt;K, V&gt; testRecord)</code>	←	Помыает объект <code>TestRecord</code>

Хочу отметить, что существуют также перегруженные версии методов, перечисленных в листинге 14.13, которые принимают параметры типа `List`. Варианты, посылающие пары «ключ — значение», принимают параметр типа `List<KeyValue>`. В каких случаях следует использовать те или иные версии методов? Строгих правил на этот счет нет, но некоторые общие рекомендации вы найдете в табл. 14.2.

**Таблица 14.2.** Общие рекомендации по выбору метода в зависимости от параметров

Параметры метода	Причина выбора
Одиночное значение или список значений	Простая топология, входной топик не имеет ключей, отсутствуют операции с состоянием, или операции с состоянием извлекают ключ из значения
Одиночная пара «ключ — значение» или список таких пар	Операции с состоянием, входной топик имеет ключи
Одиночный объект <code>TestRecord</code> или список таких объектов	Топология использует отметки времени и заголовки, тест будет продвигать время потока, исходя из отметок времени в записях
<code>Instant</code> и <code>Duration</code>	<code>Instant</code> предоставляет отметку времени для данной записи. Перегруженные версии с параметрами <code>Instant</code> и <code>Duration</code> используют <code>Instant</code> в качестве начальной отметки времени, а <code>Duration</code> — для продвижения времени потока с каждой следующей записью

Как показано в табл. 14.2, выбор метода для передачи входных данных в топологию производится не случайно и во многом зависит от действий топологии. Не менее разнообразен и набор методов для чтения результатов из приложения Kafka Streams с помощью `TestOutputTopic` (листинг 14.14).

#### Листинг 14.14. Методы для чтения результатов из теста

```
readKeyValue()
readKeyValuesToList()
readRecord()
readRecordsToList()
```

Выбор метода для проверки результатов часто зависит от того, как входные записи передаются в тест. Строгих правил, определяющих выбор, нет, но лично я предпредлагаю отправлять входные данные списком и затем списком же читать результаты и сравнивать их с ожидаемыми значениями. Можно также передать одну запись, проверить результат, передать другую запись, опять проверить результат и т. д.

В этом подразделе вы познакомились с настройкой `TopologyTestDriver` для тестирования простого приложения Kafka Streams, но этот класс может тестировать не только простые топологии, но и очень сложные, и я покажу несколько примеров в следующем подразделе.

Но прежде я хотел бы отметить еще одно применение `TopologyTestDriver`, выходящее за рамки тестирования правильности приложений. `TopologyTestDriver` можно также использовать для быстрого создания прототипов приложений Kafka Streams. `TopologyTestDriver` предлагает значительный объем функциональных возможностей, которые могут пригодиться для наблюдения за поведением приложения Kafka Streams. Среди возможностей, недоступных в `TopologyTestDriver`, — назначение задач, перебалансировка, повторное секционирование и т. д. Тем не менее с его помощью можно ускорить разработку, создав прототип приложения Kafka Streams, не нуждающееся в брокере Kafka. Конечно, нам неизбежно понадобятся тесты с действующим брокером, и я расскажу об этом немного позже в этой главе.

#### 14.1.4. Тестирование сложных приложений Kafka Streams

В этом подразделе мы рассмотрим применение `TopologyTestDriver` для тестирования более сложных приложений Kafka Streams. Поскольку выше вы уже видели полный пример создания теста от начала до конца, я покажу только отдельные разделы теста, напрямую касающиеся тестирования сложных приложений Kafka Streams. В качестве первого примера мы рассмотрим приложение с состоянием, которое выполняет операцию `reduce`. Должен упомянуть, что при тестировании топологий с состоянием `TopologyTestDriver` не буферизует никакие записи — для каждой входной записи генерируется выходная запись.

В главе 7 мы обсуждали создание приложений Kafka Streams, включающие операции с состоянием, и `reduce` является одной из таких операций, поэтому рассмотрим тест для одного из прежних примеров (`bbejeck.chapter_7.StreamsPokerGameInMemoryStoreReducer`).

##### Листинг 14.15. Настройка теста операции `reduce`

```
try (TopologyTestDriver driver = new TopologyTestDriver(topology)) {
    TestInputTopic<String, Double> inputTopic = driver.createInputTopic...
    TestOutputTopic<String, Double> outputTopic = driver.createOutputTopic...

    inputTopic.pipeInput("Anna", 65.75); ← Трижды вызывает
    inputTopic.pipeInput("Matthias", 55.8);   метод pipeInput
    inputTopic.pipeInput("Neil", 47.43);
```

Приложение Kafka Streams онлайн-игры в покер получает входные данные от игроков, роль ключа в которых играет имя пользователя, а роль значения — текущий счет игрока. Итак, в начале теста введем три счета и убедимся, что операция `reduce`

выполняется по порядку, для чего прочитаем результаты обработки трех записей и убедимся, что они следуют в порядке ввода (листинг 14.16).

#### Листинг 14.16. Проверка порядка обработки записей в операции reduce

```
Прочитает первую запись  
из выходного топика  
KeyValue<String, Double> actualKeyValue = outputTopic.readKeyValue(); ←  
assertThat(actualKeyValue, equalTo(KeyValue.pair("Anna", 65.75))); ←  
  
actualKeyValue = outputTopic.readKeyValue();  
assertThat(actualKeyValue, equalTo(KeyValue.pair("Matthias", 55.8)));  
  
actualKeyValue = outputTopic.readKeyValue();  
assertThat(actualKeyValue, equalTo(KeyValue.pair("Neil", 47.43)));  
  
Проверка  
порядка  
следования  
пар «ключ —  
значение»
```

Этот блок кода читает три записи и проверяет, соответствует ли порядок результатов порядку, в каком были переданы входные данные. Пока все хорошо, но мы можем протестировать еще один уровень. Текущее приложение является приложением с состоянием, а значит, результат последней операции `reduce` сохраняется в хранилище состояния. Пока мы проверили лишь вывод, но мы также можем проверить содержимое хранилища состояния и убедиться, что оно соответствует последнему полученному результату (листинг 14.17).

#### Листинг 14.17. Проверка соответствия содержимого хранилища состояния последнему результату

```
KeyValueStore<String, Double> kvStore =  
    driver.getKeyValueStore("memory-poker-score-store"); ←  
  
assertThat(kvStore.get("Anna"), is(65.75)); ←  
assertThat(kvStore.get("Matthias"), is(55.8)); ←  
assertThat(kvStore.get("Neil"), is(47.43));  
  
Извлекает хранилище  
состояния из топологии  
  
Проверяет, соответствует ли содержимое  
последнему полученному результату
```

Для проверки содержимого хранилища состояния `TopologyTestDriver` предоставляет метод `getKeyValueStore`, позволяющий извлечь `KeyValueStore` по имени, что мы и делаем здесь, после чего проверяем, соответствует ли содержимое последнему полученному результату.

#### СОВЕТ

`TopologyTestDriver` предлагает несколько методов для получения хранилищ разных типов: сеансовых, оконных и с отметками времени. В случаях, когда имя хранилища не указано, Kafka Streams генерирует его автоматически. Метод `getAllStateStores` возвращает ассоциативный массив `Map` со всеми хранилищами. Получив такой массив, можно в цикле проверить все хранилища и выбрать нужные.

В завершение теста мы передаем множество записей в случайном порядке. Мы уже убедились, что обработка производится по порядку, поэтому теперь мы проверяем соответствие конечного полученного результата ожидаемому. А так как мы просто вводим массу случайных записей, нам нужен способ получить конечный результат для каждого ключа (листинг 14.18).

**Листинг 14.18.** Получение конечного результата для каждого ключа

```
Map<String, Double> allOutput = outputTopic.readKeyValuesToMap(); ←
    assertThat(allOutput.get("Neil"), is(252.43));
    assertThat(allOutput.get("Anna"), is(185.75));
    assertThat(allOutput.get("Matthias"), is(180.8));
```

Получает последнюю запись для каждого ключа

```
assertThat(kvStore.get("Anna"), is(185.75)); ← Проверяет конечное
assertThat(kvStore.get("Matthias"), is(180.8)); состояния хранилища
assertThat(kvStore.get("Neil"), is(252.43));
```

Здесь для получения последнего результата по ключу используется метод `TestOutputTopic.readKeyValuesToMap`, который возвращает окончательное табличное представление результатов, где более поздние записи обновляют и заменяют предыдущие. Для проверки каждого результата в отдельности можно использовать один из методов `TestOutputTopic.readXXXTolist`.

**ПРИМЕЧАНИЕ**

Встроенные в Kafka Streams операции агрегирования избавляют от необходимости тестиировать содержимое хранилища. Однако я рекомендую все же проверять содержимое хранилищ в приложениях Kafka Streams с помощью API узлов-обработчиков (Processor API), и в этом примере я показал, как это сделать.

Вы только что узнали, как тестировать приложения с состоянием. Теперь перейдем к тестированию приложения, поведение которого зависит от отметок времени в записях, и в качестве примера используем приложение `StockPerformanceApplication` из главы 10. Кратко напомню, что `StockPerformanceApplication` выдает записи только после операции пунктуации, которая выполняется через каждые 10 секунд по времени потока. Это означает, что пунктуация выполняется только тогда, когда время потока продвигается отметками времени в записях. Для управления поведением пунктуации нам нужно организовать передачу соответствующих отметок времени, как показано в листинге 14.19, где приводится код теста из `bbejeck.chapter_10.StockPerformanceApplicationTest` (некоторые детали опущены для простоты).

**Листинг 14.19.** Передача отметок времени для управления временем потока

```
try (TopologyTestDriver driver = new TopologyTestDriver(topology)) {
    inputTopic.pipeInput("ABC", transactionOne, instant); ← Передаст первую
    inputTopic.pipeInput("ABC", transactionTwo,           запись с текущим
                           instant.plus(15, ChronoUnit.SECONDS)); ← временем
    inputTopic.pipeInput("ABC", transactionThree,
                           instant.plus(25, ChronoUnit.SECONDS)); ← Добавит 15 секунд
    // пунктуация должна выполниться три раза
    assertThat(outputTopic.getQueueSize(), is(3L));      к отметке времени
                                                       во второй записи
                                                       Передаст последнюю запись,
                                                       увеличив время на 25 секунд
```

Итак, для управления поведением отметки времени мы используем метод `TestInputTopic.pipeInput`, который принимает следующие параметры: ключ, значение и отметку времени в форме `java.time.Instant` для записи. Передача в отметках времени последовательно увеличивающихся значений обеспечит соответствующий ход времени потока и выполнения требуемого количества пунктуаций. Здесь мы

проверяем, сколько раз Kafka Streams вызовет `punctuate`, подсчитывая записи во внутренней очереди `TestOutputTopic`.

### СОВЕТ

Для приложений Kafka Streams с пунктуацией, основанной на системном времени, придется явно перемещать внутреннее системное время в `TopologyTestDriver` с помощью метода `advanceWallClockTime(Duration advanceAmount)`. Мы уже обсуждали системное время в предыдущей главе.

Следует отметить, что метод `pipeInput`, принимающий параметр с отметкой времени, продвигает время потока, отслеживаемое `TopologyTestDriver`, и не меняет внутреннее время события `TestInputTopic`. Это означает следующее: если передать следующую запись `TestInputTopic` без явной передачи отметки времени, то время события в этой записи станет начальной отметкой времени `TestInputTopic`, в данном случае время ее создания, потому что, когда отметка времени не передается явно, используется текущее время события. Чтобы продвинуть время события, мы должны использовать подход, показанный в листинге 14.20.

**Листинг 14.20.** Передача отметок времени путем продвижения времени события входного топика

```
inputTopic.pipeInput("ABC", transactionOne);           Продвинет время события  
inputTopic.advanceTime(Duration.ofSeconds(15));      топика на 15 секунд  
inputTopic.pipeInput("ABC", transactionTwo);          ←  
inputTopic.advanceTime(Duration.ofSeconds(25));      Продвинет время события  
inputTopic.pipeInput("ABC", transactionThree);        топика еще на 25 секунд
```

Этот пример реализует аналогичное поведение и проверяет Kafka Streams, выполняя три пунктуации. Но какой подход лучше? Ответ на этот вопрос зависит от того, как структурирован тест. Если отправляется небольшое количество записей, то установка каждой отметки времени вручную подходит как нельзя лучше, так мы получаем возможность видеть отметку времени для каждой записи. Однако если тест генерирует большое количество входных данных, то установка отметки времени для каждой записи может оказаться обременительной задачей, и в этом случае эффективнее продвигать время события `TestInputTopic` с применением различных интервалов.

Прежде чем закончить наше знакомство с `TopologyTestDriver`, рассмотрим еще один пример, затрагивающий отметки времени в записях, — пример с оконным агрегированием в Kafka Streams. В качестве примера мы протестируем приложение `bbejeck.chapter_9.window.StreamsCountTumblingWindowSuppressedEager`, которое подсчитывает количество входных записей в окне размером 1 минута без периода отсрочки.

Если в топологии присутствует оконная операция с подавлением, то Kafka Streams не будет выдавать результат, пока окно не закроется. Для проверки результата мы должны отправить записи с отметками времени, продвигающими время потока вперед. А чтобы продвинуть время потока в окне, применим подход, аналогичный тому, который мы использовали для проверки пунктуации в `bbejeck.chapter_9.window.StreamsCountTumblingWindowSuppressedEagerTest` (листинг 14.21).

### **Листинг 14.21.** Установка отметки времени для вывода результатов оконной операции с подавлением

В этом тесте мы сначала сгенерировали десять записей, отправили их в топологию и убедились, что библиотека Kafka Streams ничего выдала, проверив очередь выходного топика. Затем мы добавили еще одну запись и явно продвинули ее отметку времени на 1 минуту 15 секунд вперед. Поскольку мы установили размер окна равным 1 минуте, то Kafka Streams должна отправить счетчик событий из предыдущего окна, равный 10.

## **ПРИМЕЧАНИЕ**

Этот прием с установкой отметки времени для продвижения времени потока можно применять для тестирования любых оконных операций в приложениях Kafka Streams.

К настоящему моменту вы узнали о модульном тестировании клиентов производителей и потребителей Kafka, а также приложений Kafka Streams без привлечения действующего брокера. Но модульное тестирование без брокера — это только часть картины. Часто бывает нужно протестировать взаимодействие приложения с брокером Kafka, о чём мы и поговорим в нашем последнем разделе.

## **14.1.5. Эффективное интеграционное тестирование**

Основное отличие интеграционных тестов для приложений Kafka Streams, которое сразу бросается в глаза, — это дополнительный код, необходимый для взаимодействия с живым брокером. Кроме того, нам придется учитывать, что при работе с реальными приложениями Kafka Streams использует кэширование, что особенно важно, когда в топологии имеется операция с состоянием. Мы все так же будем писать тесты с использованием JUnit 5, но они будут работать немного иначе, в основном из-за того, что им требуется время для запуска брокера, с чего мы и начнем.

Для представления брокера Kafka в интеграционных тестах мы будем использовать **TestContainers** (<https://www.testcontainers.org/>) – библиотеку для Java, которая позволяет получать доступ к внешним компонентам, работающим в контейнерах Docker, непосредственно из тестов JUnit. Я предполагаю, что вы знакомы с Docker, но если вам нужна дополнительная информация, то перейдите на сайт Docker: <https://www.docker.com/>.

## **ПРИМЕЧАНИЕ**

Для использования библиотеки Testcontainers в тестах JUnit 5 требуются следующие зависимости: `org.testcontainers:junit-jupiter:1.17.1` и `org.testcontainers:kafka:1.17.1`. Они уже включены в примеры исходного кода для книги, тем не менее считаю нужным упомянуть о них для вашего сведения.

Начнем знакомство с интеграционным тестированием с создания теста для приложения `bbejeck.chapter_9.window.StreamsCountHoppingWindow`. Библиотека `Testcontainers` предоставляет аннотации, которые мы будем использовать в нашем коде для управления жизненным циклом контейнера Kafka Docker. Давайте начнем писать тест прямо сейчас (листинг 14.22).

#### Листинг 14.22. Интеграционный тест с аннотациями `Testcontainers`

```
@Testcontainers ← Добавляет аннотацию @Testcontainers к объявлению класса
class StreamsCountHoppingWindowIntegrationTest {

    @Container ← Задает контейнер Kafka для теста
    private static final KafkaContainer kafka =
        new KafkaContainer(
            DockerImageName.parse("confluentinc/cp-kafka:7.5.1")); ← Создает KafkaContainer
                                                               и сохраняет его
                                                               в статическом поле теста
```

Сначала мы создаем тестовый класс `StreamsCountHoppingWindowIntegrationTest` и добавляем на уровне класса аннотацию `@Testcontainers`, которая является расширением JUnit Jupiter и автоматически управляет жизненным циклом любых контейнеров в teste, отыскивая любые поля с аннотацией `@Container`. Здесь также можно видеть аннотацию `@Container`, отмечающую поле `KafkaContainer`. Когда поле определяется как статическое, то все тестовые методы будут использовать один и тот же контейнер, то есть контейнер будет запускаться до выполнения первого теста и останавливаться после завершения последнего.

Если определить поле контейнера как нестатическое, то контейнер будет запускаться для каждого теста и останавливаться по его завершении. Если у вас нет особой причины запускать отдельный контейнер для каждого теста, то я бы рекомендовал использовать подход со статическим полем, так как это уменьшит затраты процессорного времени, потому что контейнер будет запускаться и останавливаться только один раз.

Экземпляр `KafkaContainer` создается путем передачи объекта `DockerImageName`, которому, в свою очередь, передается строка в стандартном формате Docker (реестр/имя:тег).

#### ПРИМЕЧАНИЕ

Я не буду объяснять, как работает Docker, но могу порекомендовать книги издательства Manning, посвященные этой теме: *Docker in Practice*<sup>1</sup> Иана Милла (Ian Miell) и Эйдана Хобсона Сейерса (Aidan Hobson Sayers) (2019) и *Docker in Action* Джекфа Николоффа (Jeff Nickoloff) и Стивена Кюнцли (Stephen Kuenzli) (2019).

Теперь мы настроили контейнер Kafka Docker для теста! Осталось обсудить элементы, которые нам понадобятся для запуска приложения Kafka Streams под управлением теста. Поскольку Kafka Streams получает входные данные из топиков и выдает результаты в топики Kafka, нам понадобится `KafkaProducer` для отправки данных в топик и `KafkaConsumer` для проверки результатов. Посмотрим, что конкретно нам понадобится (некоторые детали опущены для простоты) (листинг 14.23).

<sup>1</sup> Миллан И., Сейерс Э. Х. Docker на практике. — М., 2020.

**Листинг 14.23.** Добавление конфигураций производителя и потребителя

```

@BeforeEach
public void setUp() { ← Метод setup вызывается
    перед каждым тестом

    streamsCountHoppingWindow = new StreamsCountHoppingWindow();
    kafkaStreamsProps.put("bootstrap.servers",
        kafka.getBootstrapServers()); ←

    kafkaStreamsProps.put(StreamsConfig.APPLICATION_ID_CONFIG,
        "hopping-windows-integration-test");

    // Конфигурация производителя
    producerProps.put("bootstrap.servers",
        kafka.getBootstrapServers()); ← Настройка
    producerProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
        StringSerializer.class); конфигурационного
    producerProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
        StringSerializer.class); параметра
                                                                bootstrap.servers
                                                                для клиентов

    // Конфигурация потребителя
    consumerProps.put("bootstrap.servers",
        kafka.getBootstrapServers()); ←

    consumerProps.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
        StringDeserializer.class);
    consumerProps.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
        LongDeserializer.class);
    consumerProps.put(ConsumerConfig.GROUP_ID_CONFIG,
        "integration-consumer");
    consumerProps.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,
        "earliest"); ← Создает необходимые
                                                                топики

    Topics.create(kafkaStreamsProps,
        streamsCountHoppingWindow.inputTopic,
        streamsCountHoppingWindow.outputTopic); ←
}

```

В методе `setUp` мы определяем все необходимые конфигурационные параметры для Kafka Streams и клиентов-производителей и клиентов-потребителей. Мы не будем вручную создавать `KafkaProducer` и `KafkaConsumer`, а доверим производство и потребление записей по мере необходимости некоторым статическим вспомогательным методам, о которых я вскоре расскажу. Последний фрагмент кода в методе `setUp` создает требуемые топики. Есть также парный метод `tearDown`, который выполняется после завершения каждого теста (листинг 14.24).

**Листинг 14.24.** Метод `tearDown`, который выполняется после каждого теста

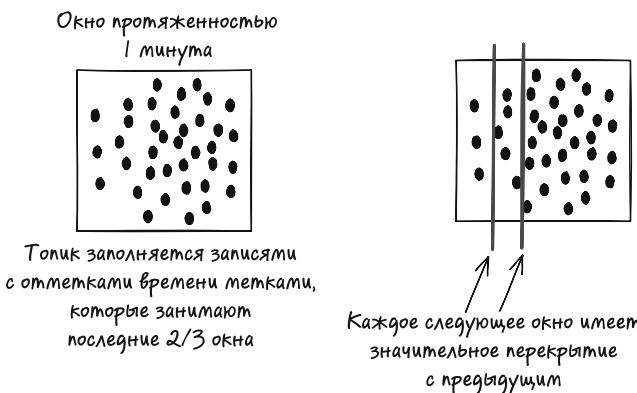
```

@AfterEach
public void tearDown() {
    Topics.delete(kafkaStreamsProps,
        streamsCountHoppingWindow.inputTopic,
        streamsCountHoppingWindow.outputTopic);
}

```

Метод `tearDown` удаляет все топики, созданные до запуска теста. Важно следовать этой практике создания и удаления топиков для каждого теста, чтобы гарантировать начало тестирования с чистой отправной точки и отсутствие искажений в результатах, обусловленных предыдущим тестом. Для создания и удаления топиков мы используем служебный класс `bvejeck.utils.Topics`, имеющийся в примерах исходного кода для книги.

Теперь перейдем к сути интеграционного теста. Поскольку тестируемое приложение Kafka Streams, `StreamsCountHoppingWindow`, использует прыгающие окна (в которых шаг меньше размера окна) протяженностью 1 минута и с шагом 10 секунд, мы хотели бы передавать значения так, чтобы каждый следующий шаг перекрывался с предыдущим. Это означает, что каждый раз, когда окно продвигается вперед на 10 секунд, счетчик включает предыдущие результаты, пока не будет достигнут размер окна. После этого счетчик начнет уменьшаться. На рис. 14.5 показано, что должен проверить наш тест.



**Рис. 14.5.** В ходе интеграционного тестирования прыгающего окна счетчик должен увеличиваться, пока окно не продвинется на размер окна, а затем уменьшаться

## ПРИМЕЧАНИЕ

В предыдущей главе я уже рассматривал работу с окнами Kafka Streams, поэтому здесь сосредоточусь только на деталях, имеющих отношение к тестированию.

Теперь, когда все настройки завершены, приступим к написанию самого теста (листинг 14.25).

В тестовом методе мы создаем экземпляр `KafkaStreams` и перед запуском настраиваем `StateListener` для перехода Kafka Streams в состояние `RUNNING`. После этого нужно немного подождать, пока приложение Kafka Streams не будет готово к работе. Этот шаг не является обязательным, но я предпочитаю начинать выполнять тест только после того, как приложение перейдет в известное состояние.

**Листинг 14.25.** Тест для приложения Kafka Stream с прыгающим окном

```

@Test
@DisplayName("Integration test for hopping window")
void shouldHaveHoppingWindowsTest() {
    final Topology topology =
        streamsCountHoppingWindow.topology(kafkaStreamsProps); ← Создание топологии
    AtomicBoolean streamsStarted = new AtomicBoolean(false);
    try (KafkaStreams kafkaStreams =
        new KafkaStreams(topology, kafkaStreamsProps)) { ← Устанавливает StateListener в состояние RUNNING
        kafkaStreams.cleanUp(); ← Запускает экземпляра KafkaStreams
        kafkaStreams.setStateListener((newState, oldState) -> { ← Ожидает состояния RUNNING
            if (newState == KafkaStreams.State.RUNNING) {
                streamsStarted.set(true);
            }
        });
        kafkaStreams.start(); ←
        while (!streamsStarted.get()) { ←
            time.sleep(250); ←
        }
    }
}

```

Теперь перейдем к генерированию входных данных (листинг 14.26).

**Листинг 14.26.** Генерирование записей для приложения Kafka Streams

```

long startTimestamp = Instant.now().toEpochMilli(); ← Первой отметке времени присваивается текущее время
for (int i = 1; i <= 6; i++) {
    List<KeyValue<String, String>> list =
        Stream.generate(() ->
            KeyValue.pair("Foo", "Bar"))).limit(i).toList(); ← Генерирует список пар KeyValue
    TestUtils.produceKeyValuesWithTimestamp( ←
        streamsCountHoppingWindow.inputTopic,
        list, ← Вызывает вспомогательный
        producerProps, ← метод для отправки записей
        startTimestamp,
        Duration.ofMillis(100L));
    startTimestamp += 10_000; ← Увеличивает отметки времени на величину шага
}

```

Здесь мы используем вспомогательный метод `TestUtils.produceKeyValuesWithTimestamp`, который можно найти в примерах исходного кода для книги. Этот метод выполняет все необходимые действия, чтобы создать сообщения для обработки в `StreamsCountHoppingWindow`. В каждой итерации мы отправляем записи, количество которых соответствует индексу цикла, и увеличиваем отметку времени на 10 секунд, чтобы гарантировать, что следующая партия созданных записей окажется в следующем окне. Мы ожидаем, что каждое продвижение будет включать сумму предыдущих записей до границы размера окна, а затем количество начнет уменьшаться. Чтобы проверить ожидания, добавим в тест следующий код (листинг 14.27).

**Листинг 14.27.** Подготовка списка ожидаемых записей, которые, как предполагается, будут произведены в Kafka Streams

```
List<KeyValue<String, Long>> expectedKeyValues =
    List.of(KeyValue.pair("Foo", 1L),
    KeyValue.pair("Foo", 3L),
    KeyValue.pair("Foo", 6L),
    KeyValue.pair("Foo", 10L),
    KeyValue.pair("Foo", 15L),
    KeyValue.pair("Foo", 21L), ← Достигнут максимальный
    KeyValue.pair("Foo", 20L), размер окна 1 минута
    KeyValue.pair("Foo", 18L),
    KeyValue.pair("Foo", 15L),
    KeyValue.pair("Foo", 11L),
    KeyValue.pair("Foo", 6L)); ← Вспомогательный метод
                                                                для потребления записей

List<KeyValue<String, Long>> actualConsumed =
    TestUtils.readKeyValues(streamsCountHoppingWindow.outputTopic, ←
        consumerProps,
        45_000, ← Максимальное время, в течение которого тест будет ждать результаты
        12); ←

assertThat(actualConsumed, equalTo(expectedKeyValues)); ←
                                                                Проверит соответствие фактических
                                                                результатов ожидаемым
                                                                Максимальное количество
                                                                извлекаемых записей. Получив
                                                                это количество результатов,
                                                                метод вернет управление
```

Итак, мы создаем список ожидаемых записей, в которых счетчик сначала увеличивается на количество отправленных записей плюс предыдущий счетчик ( $1 + 2 + 3 + 4\dots$ ), а затем уменьшается. После этого вызываем вспомогательный метод, чтобы потребить записи из топика, вернуть результаты в тест и проверить соответствие результатов нашим ожиданиям. При каждом запуске этот тест должен выполняться успешно. Код в этом разделе сначала создает ожидаемый результат для сравнения, а потом использует вспомогательные методы, которые можно повторно использовать в разных интеграционных тестах. Я не включил примеры интеграционного тестирования для классов производителя и потребителя. Однако они следуют той же схеме: объявляется тестовый класс с аннотацией `@Testcontainer` и с полем `@Container`, а в методах класса используются вспомогательные методы для производства и потребления записей. Эти интеграционные тесты вы найдете в примерах исходного кода для книги.

## ИТОГИ ГЛАВЫ

- Тестирование — важнейшая часть разработки, и каждый компонент приложения должен быть протестирован в достаточной мере.
- Модульные тесты должны составлять большую часть набора тестов, но вам также нужно написать интеграционные тесты, чтобы получить уверенность, что все работает так, как и ожидалось.

- Фиктивные объекты `MockProducer` и `MockConsumer` удобны тем, что позволяют тестиировать клиентские приложения без привлечения действующего брокера Kafka.
- Операторы Kafka Streams (отображения, фильтрации, пунктуации и т. д.) лучше оформлять как конкретные классы. Это позволит вам писать изолированные тесты для каждого оператора.
- Тестируя ожидаемое поведение операторов Kafka Streams (отображений, агрегаторов и пунктуаторов) следует с помощью модульных тестов. В модульных тестах можно также использовать `TopologyTestDriver` для тестирования всей топологии без привлечения действующего брокера Kafka.
- `TopologyTestDriver` позволяет создавать экземпляры `TestInputTopic` и `TestOutputTopic` и использовать их для отправки записей в топологию и последующего извлечения результатов.
- При тестировании приложения Kafka Streams с помощью `TopologyTestDriver` может потребоваться задать отметки времени, чтобы обеспечить продвижение окна.
- Для интеграционных тестов следует использовать библиотеку `Testcontainers`, которая предоставляет брокер Kafka для нужд тестирования и заботится о запуске контейнера и управлении его жизненным циклом.
- По умолчанию расширение JUnit 5 `Testcontainers` будет запускать новый контейнер для каждого теста, что может привести к значительным затратам времени. Чтобы уменьшить эти затраты, можно сделать поле `Testcontainers` статическим, и в таком случае контейнер будет запускаться только один раз для всех тестов в тестовом классе.

# *Приложения*

<https://t.me/javalib>

# *Практикум по совместимости схем*

В этом приложении вы познакомитесь с пошаговым руководством по обновлению схем в различных режимах совместимости. Мы изменим режим совместимости схем, внесем изменения, протестируем эти изменения и, наконец, запустим обновленные производители и потребители, чтобы увидеть, как действуют различные режимы совместимости. Я уже внес все изменения. Вам остается только прочитать и выполнить указанные команды. Здесь мы рассмотрим три подпроекта: `sr-backward`, `sr-forward` и `sr-full`. Каждый содержит производители, потребители и схемы, обновленные и настроенные для репрезентативного режима совместимости.

## **ПРИМЕЧАНИЕ**

Программный код и файлы `build.gradle` подпроектов имеют много совпадений. Это было сделано намеренно, так как я хотел сделать каждый модуль независимым от других. Цель этих модулей — помочь изучить развитие схемы в реестре схем и понять, какие изменения в связи с этим необходимо внести в реализацию производителей и потребителей Kafka, а не как настроить идеальный проект Gradle!

В этом разделе я расскажу, как Schema Registry обеспечивает совместимость с клиентами. Чтобы усвоить правила совместимости схем в фреймворках сериализации, каждый из них нужно рассматривать отдельно. Правила разрешения схем в Avro доступны по адресу <http://mng.bz/ngE2>. Правила обратной совместимости в Protobuf описаны в спецификации языка по адресу <http://mng.bz/v8r4>.

А теперь рассмотрим различные режимы совместимости. Для каждого режима мы рассмотрим изменения, внесенные в схему, и выполним несколько шагов, необходимых для успешной миграции схемы.

Для наглядности каждая миграция схемы в разных режимах совместимости имеет свой подмодуль Gradle в исходном коде книги. Я поступил так потому, что каждое

изменение файла схемы Avro приводит к созданию разных структур классов Java при сборке кода. Поэтому, чтобы не заставлять вас переименовывать файлы, я выбрал структуру, в которой каждый тип миграции может действовать независимо. В типичной среде разработки вам не понадобится такая практика. Вы просто измените файл схемы, сгенерируете новый код Java и обновите производители и потребители в одном и том же проекте.

Все примеры миграции схемы изменяют исходный файл схемы `avenger.avsc`. Для справки в листинге A.1 приводится исходный код схемы, чтобы было легче увидеть изменения, внесенные для каждой миграции схемы.

#### **Листинг A.1.** Исходный код схемы Avenger Avro

```
{
  "namespace": "bbejeck.chapter_3.avro",
  "type": "record",
  "name": "AvengerAvro",
  "fields": [
    {"name": "name", "type": "string"},
    {"name": "real_name", "type": "string"},
    {"name": "movies", "type": [
      {"type": "array", "items": "string"},
      "default": []
    ]}
  ]
}
```

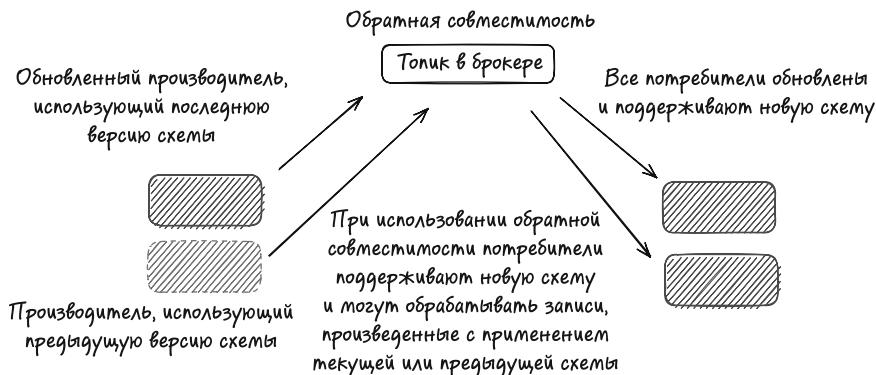
#### **ПРИМЕЧАНИЕ**

Для развития схемы при использовании разных типов совместимости я создал три подмодуля в исходном коде: `sr-backward`, `sr-forward` и `sr-full`. Эти подмодули полностью независимы и намеренно содержат повторяющийся код и настройки. Модули обновляют схемы, производители и потребители для каждого режима совместимости. Я сделал это, чтобы упростить процесс изучения и позволить вам без помех просматривать изменения и запускать новые примеры.

## **A.1. ОБРАТНАЯ СОВМЕСТИМОСТЬ**

Обратная совместимость — это тип совместимости по умолчанию для миграций. При обратной совместимости поддержка новой схемы добавляется сначала в код потребителя. После этого обновленные потребители оказываются способными читать записи, сериализованные с применением новой или предыдущей схемы (рис. А.1).

Как показано на рис. А.1, потребитель может работать и с предыдущей, и с новой схемой. В режиме обратной совместимости допускаются такие изменения, как удаление полей или добавление необязательных полей (необязательное поле — это поле, для которого схема определяет значение по умолчанию). Если сериализованные байты не содержат необязательного поля, то десериализатор использует указанное значение по умолчанию при десериализации байтов обратно в объект.



**Рис. А.1.** При обратной совместимости сначала в код потребителя добавляется поддержка новой схемы. После этого обновленные потребители могут читать записи, сериализованные с применением новой или предыдущей схемы

Прежде чем начать, запустим производитель с первоначальной схемой. Благодаря этому после следующего шага у нас будут иметься записи, сериализованные с применением как старой, так и новой схемы, и мы сможем увидеть обратную совместимость в действии. Запустите docker командой `docker-compose up -d`, а затем выполните команды, показанные в листинге А.2.

#### Листинг А.2. Создание записей с использованием первоначальной схемы

```
./gradlew streams:registerSchemasTask           Регистрация первоначальной
./gradlew streams:runAvroProducer               схемы avengers.avsc
                                                Запуск производителя, использующего
                                                первоначальную схему
```

Теперь у нас в топике будут иметься записи, сериализованные с использованием первоначальной схемы. Когда мы выполним следующий шаг, наличие этих записей поможет нам прояснить, как работает обратная совместимость, так как потребитель сможет принимать записи, используя старую и обновленную схему. Итак, обновим первоначальную схему, удалив поле `real_name` и добавив поле `powers` со значением по умолчанию (листинг А.3).

#### ПРИМЕЧАНИЕ

Файл схемы и код вы найдете в подмодуле `sr-backward` в каталоге `src/main/avro` в примерах исходного кода.

#### Листинг А.3. Обновление схемы с сохранением обратной совместимости

```
{
  "namespace": "bbejeck.chapter_3.avro",
  "type": "record",
  "name": "AvengerAvro",
  "fields": [
    {"name": "name", "type": "string"}, Поле powers заменило
    {"name": "powers", "type": "array", "items": "string"}, поле real_name
  ]
}
```

```

    "default": []
},
{"name": "movies", "type":
    {"type": "array", "items": "string"},
    "default": []
}
]
}

```

← Как того требует обратная совместимость, для поля powers  
определенено значение по умолчанию: пустой список

Теперь, обновив схему, мы должны проверить ее совместимость перед выгрузкой в Schema Registry. К счастью, протестировать новую схему совсем не сложно. Для тестирования совместимости мы используем `testSchemasTask` в модуле `sr-backward` из плагина Gradle. Итак, сначала проверим совместимость, запустив следующую команду в корне проекта (листинг А.4).

#### **Листинг А.4.** Проверка обратной совместимости новой схемы

```
./gradlew :sr-backward:testSchemasTask
```

#### **ВНИМАНИЕ**

Для успешного выполнения примера необходимо набрать команду в точности так, как она показана в листинге А.4, включая начальный символ `:`.

После запуска команда `testSchemasTask` должна вывести `BUILD SUCCESSFUL`, сообщая тем самым, что новая схема обратно совместима с существующей. `testSchemasTask` вызывает Schema Registry для сравнения новой схемы с текущей, чтобы убедиться в ее совместимости. Теперь, когда мы знаем, что новая схема обратно совместима, продолжим и зарегистрируем ее с помощью следующей команды (листинг А.5).

#### **Листинг А.5.** Регистрация новой схемы

```
./gradlew :sr-backward:registerSchemasTask
```

Команда регистрации выведет в консоль сообщение `BUILD SUCCESSFUL`. Прежде чем перейти к следующему шагу, выполним команду REST API и посмотрим последнюю версию схемы `avro-avengers-value`:

```
curl -s "http://localhost:8081/subjects/avro-avengers-value/
versions/latest" | jq .'
```

Эта команда должна вернуть примерно следующий результат:

```
{
  "name" : "avro-avengers-value",
  "version" : 2,
  "schema" : "{\"type\": \"record\", \"namespace\": \"bbejeck.chapter_3\",
  \"name\": \"AvengerAvro\"..."}
```

Как показывают результаты, номер версии схемы увеличился с 1 до 2. После внесения этих изменений нам нужно обновить клиенты, начав с потребителя. Режим обратной совместимости требует сначала обновить потребителей, чтобы они могли обрабатывать записи, созданные с использованием новой схемы.

Например, изначально потребители обрабатывали поле `real_name`, но мы удалили его из схемы, поэтому мы должны также удалить ссылки на него в потребителе.

Мы также добавили поле `powers`, а значит, должны добавить возможность работы с этим полем. Это также подразумевает, что мы должны генерировать новые объекты модели.

Раньше, когда мы запускали команду `clean`, `build`, она генерировала правильные объекты для всех модулей. Поэтому сейчас делать этого не нужно.

Обратите внимание, что, поскольку мы действуем в режиме обратной совместимости, ничего страшного не произойдет, если обновленный потребитель получит записи в предыдущем формате. Обновление игнорирует поле `real_name`, а поле `powers` имеет значение по умолчанию.

После обновления потребителя нужно обновить производители. `AvroProducer` в подмодуле `sr-backward` уже включает все необходимые обновления. Теперь выполните следующую команду, чтобы производитель отправил записи с использованием новой схемы (листинг А.6).

#### Листинг А.6. Создание записей с применением новой схемы

```
./gradlew :sr-backward:runAvroProducer
```

Вы увидите длинный поток текста, за которым следует знакомая строка `BUILD SUCCESSFUL`. Как вы наверняка помните, мы запускали команду создания новых записей в первоначальной версии подмодуля всего несколько минут назад. Теперь, запустив производитель, поддерживающий новую схему, мы получили в топике смесь записей со старой и новой схемой. Но наш потребитель должен уметь обрабатывать оба типа, поскольку мы действуем в режиме обратной совместимости.

Теперь, запустив потребитель, вы должны увидеть записи, созданные с предыдущей схемой, и записи, созданные с новой схемой. Выполните следующую команду, чтобы запустить обновленный потребитель (листинг А.7).

#### Листинг А.7. Потребление записей после обновления схемы

```
./gradlew :sr-backward:runAvroConsumer
```

В консоли вы должны увидеть первые записи с полем `powers`, имеющим значение `[]` в консоли. Пустое значение отмечает старые записи, для которых использовано значение по умолчанию, поскольку в них нет поля `powers`.

#### ПРИМЕЧАНИЕ

В этом примере наш потребитель прочитал все записи, имеющиеся в топике, потому что мы использовали новый идентификатор `group.id` для потребителя в модуле `sr-backward` и настроили его на чтение с самого раннего доступного смещения, если прежде приложение с этим `group.id` не извлекало записи из топика. Тот же `group.id` мы используем в остальных примерах и модулях совместимости, и потребитель будет читать только вновь созданные записи. Подробности, касающиеся конфигурации `group.id`, вы найдете в главе 4.

## A.2. ПРЯМАЯ СОВМЕСТИМОСТЬ

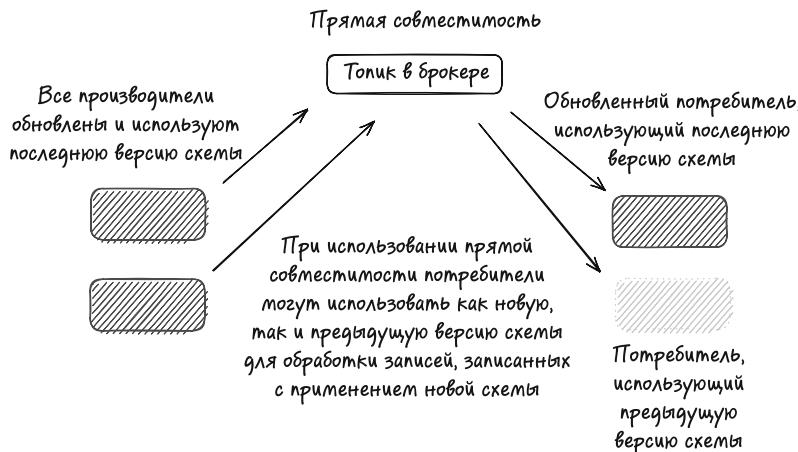
Прямая совместимость — это зеркальное отражение обратной совместимости относительно изменений полей. При использовании прямой совместимости можно добавлять поля и удалять необязательные. Давайте продолжим и снова обновим схему, создав файл `avenger_v3.avsc`, который можно найти в каталоге `sr-forward/src/main/avro` (листинг А.8).

**Листинг A.8.** Обновление схемы Avenger с сохранением прямой совместимости

```
{
  "namespace": "bbejeck.chapter_3.avro",
  "type": "record",
  "name": "AvengerAvro",
  "fields": [
    { "name": "name", "type": "string" },
    { "name": "powers", "type": {
        "type": "array", "items": "string" },
      "default": []
    },
    { "name": "nemeses", "type": {
        "type": "array", "items": "string"
      }
    }
  ]
}
```

Добавлено новое поле nemeses

В этой новой версии схемы мы удалили поле `movies`, которое по умолчанию содержит пустой список, и добавили новое поле `nemeses`. При соблюдении режима прямой совместимости сначала обновляется код клиента-производителя (рис. А.2).



**Рис. А.2.** При прямой совместимости поддержка новой схемы сначала добавляется в код производителя, а потребители сохраняют способность обрабатывать записи, сериализованные с применением как новой, так и предыдущей схемы

Первоочередное обновление производителей гарантирует правильное заполнение новых полей и их доступность только в записях в новом формате. Обновление потребителей можно отложить — они смогут работать с новой схемой без обновления кода, так как новые поля будут просто игнорировать, а удаленными полям будут присваивать значения по умолчанию.

Теперь нужно изменить режим совместимости с `BACKWARD` на `FORWARD`. В подмодуле `sr-forward` конфигурация плагина Schema Registry имеет следующий раздел кода, устанавливающий совместимость (листинг А.9).

**Листинг А.9.** Объявление совместимости в build.gradle для подмодуля sr-forward

```
config {
    subject('avro-avengers-value', 'FORWARD')
}
```

Теперь, изменив режим совместимости в конфигурации, выполним следующую команду (листинг А.10).

**Листинг А.10.** Изменение режима совместимости на FORWARD

```
./gradlew :sr-forward:configSubjectsTask
```

Как мы уже видели раньше, эта команда выводит BUILD SUCCESSFUL в консоль. Чтобы подтвердить режим совместимости для нашего субъекта, можете использовать команду REST API из листинга А.11.

**Листинг А.11.** Команда REST API, проверяющего настроенный режим совместимости для субъекта

```
curl -s "http://localhost:8081/config/avro-avengers-value" | jq .'
```

Команда jq в конце этой командной строки форматирует возвращаемый код JSON, и в консоли вы должны увидеть примерно такой вывод, как в листинге А.12.

**Листинг А.12.** Форматированный ответ с конфигурацией

```
{
    compatibility: FORWARD
}
```

Теперь, настроив субъект avro-avengers-value с прямой совместимостью, протестируем новую схему, выполнив команду в листинге А.13.

**Листинг А.13.** Проверка прямой совместимости новой схемы

```
./gradlew :sr-forward:testSchemasTask
```

Эта команда должна вывести в консоль сообщение BUILD SUCCESSFUL, и после этого можно зарегистрировать новую схему (листинг А.14).

**Листинг А.14.** Регистрация новой схемы

```
./gradlew :sr-forward:registerSchemasTask
```

Теперь запустим обновленный производитель, как показано в листинге А.15, для отправки записей в новом формате.

**Листинг А.15.** Запуск обновленного производителя для создания записей с применением новой схемы

```
./gradlew :sr-forward:runAvroProducer
```

Далее запустим необновленный потребитель, то есть не поддерживающий новую схему, как показано в листинге А.16.

**Листинг А.16.** Запуск необновленного потребителя

```
./gradlew :sr-backward:runAvroConsumer
```

Как показывают результаты, в режиме прямой совместимости даже необновленный потребитель может обрабатывать записи, созданные с использованием новой схемы. Теперь снова создадим несколько записей, чтобы проверить *обновленный* потребитель (листинг A.17).

#### **Листинг A.17.** Повторный запуск производителя

```
./gradlew :sr-forward:runAvroProducer
```

Запустим обновленный потребитель, поддерживающий новую схему (листинг A.18).

#### **Листинг A.18.** Запуск обновленного потребителя

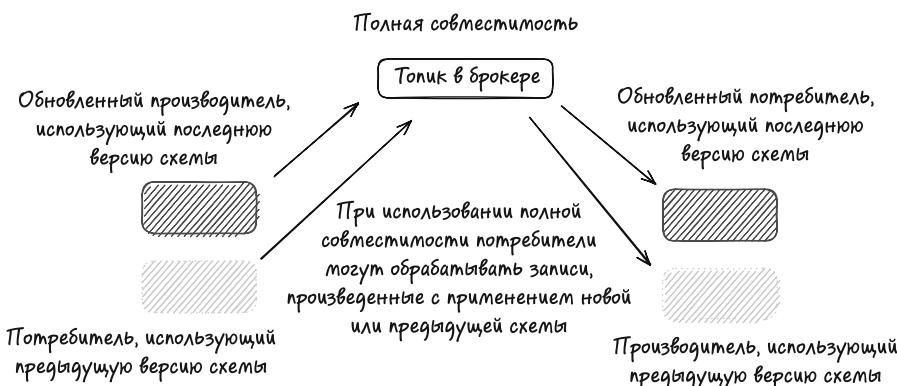
```
./gradlew :sr-forward:runAvroConsumer
```

В обоих случаях потребитель благополучно прочитает записи, но вывод в консоли будет отличаться из-за добавления в потребитель поддержки новой схемы.

Итак, вы увидели совместимость двух типов: обратную и прямую. Как следует из названия совместимости, она учитывает возможность изменения записей в одном направлении. При обратной совместимости сначала обновляются потребители, поскольку записи могут поступать как в новом, так и в старом формате. При прямой совместимости сначала обновляются производители, чтобы гарантировать, что с момента перехода записи будут иметь только новый формат. Последняя стратегия совместимости, которую мы рассмотрим, — полная совместимость.

## A.3. ПОЛНАЯ СОВМЕСТИМОСТЬ

В режиме полной совместимости можно добавлять или удалять поля, но есть одна загвоздка. Любые изменения должны касаться только необязательных полей (рис. A.3).



**Рис. A.3.** При полной совместимости производители могут посыпать записи с использованием предыдущей или новой схемы, а потребители могут обрабатывать записи, сериализованные с применением новой или предыдущей схемы

Поскольку поля в обновленной схеме являются необязательными, эти изменения совместимы с существующими производителями и потребителями, и порядок обновления в таком случае зависит только от вашего решения. Потребители смогут читать записи, созданные с помощью новой или старой схемы. Рассмотрим порядок обновления схемы в режиме полной совместимости (листинг А.19).

**Листинг А.19.** Обновление схемы с сохранением полной совместимости (avengers\_v4.avsc)

```
{  
    "namespace": "bbejeck.chapter_3.avro",  
    "type": "record",  
    "name": "AvengerAvro",  
    "fields": [  
        { "name": "name", "type": "string" },  
        { "name": "yearPublished", "type": "int", "default": 1960 }, ← Новое необязательное  
        { "name": "realName", "type": "string", "default": "unknown" }, ← поле yearPublished  
        { "name": "partners", "type": {  
            "type": "array", "items": "string" }, ← Новое поле  
            "default": [] } partners  
        },  
        { "name": "nemeses", "type": {  
            "type": "array", "items": "string" }, ← Вновь  
            "default": [] } возвращено  
    ]  
}
```

Прежде чем обновить схему, снова запустим производитель, чтобы получить несколько записей в прежнем формате (листинг А.20).

**Листинг А.20.** Создание нескольких записей перед изменением схемы

```
./gradlew :sr-forward:runAvroProducer
```

В результате мы получим пакет записей, которые попробуем прочитать обновленным потребителем. Но прежде изменим режим совместимости, на этот раз на FULL (листинг А.21).

**Листинг А.21.** Изменение режима совместимости на FULL

```
./gradlew :sr-full:configSubjectsTask
```

Следя нормальному течению процесса, проверим совместимость схемы перед ее изменением (листинг А.22).

**Листинг А.22.** Проверка полной совместимости новой схемы

```
./gradlew :sr-full:testSchemasTask
```

После проверки совместимости можно произвести регистрацию новой схемы (листинг А.23).

**Листинг А.23.** Регистрация новой схемы

```
./gradlew :sr-full:registerSchemasTask
```

Зарегистрировав новую версию схемы, немного поиграем с порядком производимых и потребляемых записей. Поскольку все обновления схемы касаются только необязательных полей, порядок обновления производителей и потребителей не имеет значения.

Недавно мы создали пакет записей с применением предыдущей версии схемы. Это было сделано, чтобы продемонстрировать возможность использовать обновленный потребитель в режиме полной совместимости для чтения старых записей. Напомню, что раньше, в режиме прямой совместимости, было важно гарантировать, что обновленные потребители увидят записи только в новом формате.

Запустите обновленный потребитель для чтения записей, созданных с использованием предыдущей схемы, и посмотрите, что произойдет после запуска команды в листинге A.24.

**Листинг A.24.** Чтение записей обновленным потребителем

```
./gradlew :sr-full:runAvroConsumer
```

Он выполнится благополучно! Теперь изменим порядок операций и сначала запустим обновленный производитель (листинг A.25).

**Листинг A.25.** Создание записей с использованием новой схемы

```
./gradlew :sr-full:runAvroProducer
```

А затем запустим необновленный потребитель и попробуем прочитать записи в новом формате (листинг A.26).

**Листинг A.26.** Чтение записей в новом формате необновленным потребителем

```
./gradlew :sr-forward:runAvroConsumer
```

Поэкспериментировав с различными версиями производителей и потребителей, мы убедились, что в режиме полной совместимости порядок обновления производителей и потребителей не имеет значения и зависит только от наших предпочтений.

# *Ресурсы Confluent*

## **Б.1. CONFLUENT CLOUD**

Confluent Cloud — это отказоустойчивый масштабируемый сервис потоковой обработки данных на основе Apache Kafka®, предоставляемый как полностью управляемый сервис. Confluent Cloud позволяет быстро масштабировать кластер Kafka без затрат ресурсов на вашем ноутбуке. Используя интерфейс командной строки Confluent (CLI), можно быстро и без особого труда развернуть кластер, создать и удалить топики и включить поддержку Schema Registry. Можно также попробовать недавний продукт от Confluent — Flink SQL.

## **Б.2. ИНТЕРФЕЙС КОМАНДНОЙ СТРОКИ CONFLUENT**

Интерфейс командной строки Confluent CLI — это мощный инструмент, который упрощает работу с Confluent Cloud. С его помощью можно создавать и удалять кластеры, а также читать и отправлять записи в топики из командной строки. Установить инструмент CLI в macOS можно командой `brew install confluentinc/tap/cli`, но вообще поддерживаются все основные операционные системы. Перейдите по адресу <http://mng.bz/KZMZ>, где приводятся исчерпывающие инструкции по различным вариантам установки CLI.

Возможности CLI можно расширять с помощью плагинов. Официальный репозиторий плагинов Confluent находится по адресу <https://github.com/confluentinc/cli-plugins>. Найти доступные для установки плагины можно командой `confluent plugin search`. Например, запустите плагин `kickstart` командой `confluent cloud kickstart` — и он автоматически подготовит кластер Kafka, включит реестр схем,

создаст ключи API и сгенерирует файл свойств для подключения клиентов. Чтобы увидеть полный список поддерживаемых параметров, запустите команду `confluent cloud kickstart -h`.

## **Б.3. ЗАПУСК CONFLUENT ЛОКАЛЬНО**

CLI также может запустить брокер Kafka в Docker локально (<http://mng.bz/9dMo>), для чего нужно выполнить команду `confluent local kafka start`. Эта команда извлечет образ Kafka Docker и запустит его. Но для этого у вас должны быть установлены Docker и Docker Desktop.

# *Работа с Avro, Protobuf и JSON Schema*

---

## **B.1. APACHE AVRO**

Схема Avro поддерживает как простые, так и составные типы. В числе простых типов поддерживаются `null`, `boolean`, `int`, `long`, `float`, `double`, `bytes` и `string`. В числе составных типов поддерживаются `record`, `enum`, `array`, `map`, `union` и `fixed`. Полное описание простых и составных типов вы найдете в документации Avro по адресу <http://mng.bz/Y7Oo> и <http://mng.bz/GZ2M> соответственно.

В книге в основном используется составной тип `record`, соответствующий объекту. Вы также можете столкнуться с несколькими примерами, использующими простые значения, но в основном мы будем придерживаться типа `record`. Также мы будем использовать объединения, тип `union`, особенно при работе со стратегиями `RecordNameStrategy` и `TopicRecordName`. Тип `union` очень полезен и определяется в виде массива. Объединения позволяют указать, что поле может иметь один или несколько типов. Я покажу применение типа `union` в следующих примерах.

Тип `record` содержит элементы `name`, `doc`, `aliases` и `fields`. Элемент `fields` может иметь свойства `name`, `type`, `doc`, `default`, `order` и `aliases`. На рис. B.1 показана схема Avro с краткими примечаниями, описывающими каждое из этих свойств.



**Рис. B.1.** Сравнение схем с вложенными записями и ссылками на схемы

### B.1.1. Значения по умолчанию и псевдонимы

Свойство `default` определяет значение поля при десериализации объекта в формате Avro, если оно отсутствует. Другими словами, определение значения по умолчанию в Avro равнозначно определению поля как необязательного. В отношении совместимости необязательные значения в значительной степени определяют, является ли схема обратно или прямо совместимой.

Псевдоним `alias` ссылается на имя предыдущей схемы, которую должна представлять текущая схема. Это означает, например, следующее: если вы развиваете схему с именем `event`, создали новую ее версию с именем `event_v2` и указали псевдоним `event` для новой схемы, то читателям новая схема будет также доступна под именем `event`. То же относится к полям с псевдонимами. Использование псевдонимов, совпадающих с именами полей в старой схеме, позволит читателю использовать псевдоним. Псевдонимы могут пригодиться, например, если вы обновили имя поля, но хотите, чтобы новая схема оставалась обратно совместимой для нижестоящих пользователей, использующих прежнее имя поля.

## B.1.2. Объединения

Я упомянул, что объединения (тип `union`) — это важная концепция схем Avro. Объединение позволяет указать, что поле может иметь один из нескольких типов. Рассмотрим для начала следующий простой пример:

```
{  
    "type": "record",  
    "namespace": "bbejeck.chapter_3",  
    "name": "transaction",  
    "fields" : [  
        {"name": "txn_type", "type": "string"},  
        {"name": "identifier", "type": ["string", "long"]}] }  
}  
} ]
```

Поле `identifier` имеет тип `union`; оно может содержать строку или длинное целое

Этот пример схемы демонстрирует простое объединение. Поле `identifier` может содержать значение типа `string` или `long`. Но тип `union` может быть очень сложным и широко используется при работе со стратегиями `RecordNameStrategy` и `TopicRecordName`.

Например, расширим пример схемы и используем тип `record` для представления транзакции. Каждая транзакция может быть записью одного из трех типов — `Purchase`, `Return` или `Exchange`. Соответствующая схема будет выглядеть примерно так:

```
{  
    "type": "record",  
    "namespace": "bbejeck.chapter_3.nested"  
    "name": "transaction",  
  
    "fields" : [  
        {"name": "txn_type", "type": [ ] } ] ← Тип union объявляется как массив  
        {  
            "type": "record",  
            "namespace": "bbejeck.chapter_3.nested",  
            "name": "Purchase", } ← Запись типа Purchase  
            "fields": [  
                {"name": "item", "type": "string"},  
                {"name": "amount", "type": "double"}  
            ],  
            {  
                "type": "record",  
                "namespace": "bbejeck.chapter_3.nested",  
                "name": "Return", } ← Запись типа Return  
                "fields": [  
                    {"name": "item", "type": "string"},  
                    {"name": "amount", "type": "double"}  
                ],  
                {  
                    "type": "record",  
                    "namespace": "bbejeck.chapter_3.nested",  
                    "name": "Exchange", } ← Запись типа Exchange  
                ]  
            },  
        ]  
    } ]
```

```

    "fields": [
        {"name": "item", "type": "string"},
        {"name": "amount", "type": "double"},
        {"name": "new_item", "type": "string"}
    ]
}

],
{
    "name": "identifier", "type": "long"
}
}

```

Поле с типом `union` может принимать значение одного из трех типов `record`, описанных в этой схеме. Если вложенные типы имеют несколько полей и особенно если они содержат другие типы-объединения, то определение схемы может быстро стать чрезвычайно громоздким. К счастью, Schema Registry поддерживает ссылки на схемы.

## B.2. PROTOCOL BUFFERS

Для организации взаимодействий между программами, написанными на разных языках, в Google разработали Protocol Buffers (<https://developers.google.com/protocol-buffers>). Что такое Protocol Buffers? Приведу определение с домашней страницы Protocol Buffers:

Protocol Buffers – это расширяемый механизм для сериализации структурированных данных, не зависящий ни от языка программирования, ни от платформы, разработанный в Google. Представьте себе XML, но более компактный и простой. Вы один раз определяете, как должны структурироваться ваши данные, и используете генерированный исходный код на разных языках для быстрой записи/чтения данных в/из различных потоков.

*Определение с домашней страницы Protocol Buffers*

Поскольку цель Protocol Buffers – обеспечить возможность сериализации данных в представление, не зависящее от языка программирования, этот формат идеально подходит для использования с Kafka и Schema Registry. Нетрудно понять, что Protocol Buffers – это обширная тема, поэтому я буду акцентировать внимание только на конкретных деталях. Но в этом разделе я расскажу, что вам понадобится, чтобы начать использовать Protocol Buffers вместе с Schema Registry.

### ПРИМЕЧАНИЕ

В настоящее время доступны версии Protocol Buffers 2 и 3. Последняя версия – 3, и именно на ней я сосредоточусь. Кроме того, несмотря на то что Protocol Buffers поддерживает разные языки программирования, я буду рассматривать только использование этого формата в Java.

Чтобы использовать Protocol Buffers, нужно установить компилятор Protobuf. Если вы используете macOS, как я, то установить его можно с помощью homebrew (<https://brew.sh/>), выполнив команду `brew install Protobuf`.

## ПРИМЕЧАНИЕ

Для работы с Protobuf доступны свои инструменты командной строки, но примеры исходного кода для книги используют gradle. Поэтому для их опробования вы будете использовать `Protobuf-gradle-plugin`, разработанный в Google (<https://github.com/google/Protobuf-gradle-plugin>).

Protocol Buffers поддерживает понятие сообщения — `message` — вершины иерархии записей. Сообщение `message` может иметь одно или несколько полей. Схемы сообщений в формате Protocol Buffers сохраняются в файлах с расширением `.proto`, в отличие от Avro, где используются файлы с расширением `.avsc`.

Поля, содержащиеся в сообщении `message`, могут иметь скалярный тип — `long`, `int`, `float`, `double`, `boolean`, — `String` для Java API и `ByteString`, аналогичный массиву байтов в Java. Компилятор Protocol Buffers поддерживает генерацию кода из файлов `.proto`, а также позволяет использовать сложные структуры, поэтому сообщения могут иметь вложенную организацию. Рассмотрим простой файл `.proto`, чтобы закрепить новые знания (листинг B.1).

### Листинг B.1. Простая схема Protobuf в файле avenger\_v1.proto

```
syntax = "proto3"; // Указывает, что используется версия Protobuf 3. Если атрибут syntax не указан, то предполагается версия proto2

package proto_files; // Объявление пакета Protobuf для создания пространств имен сообщений в различных файлах proto

option java_package = "bbejeck.chapter_3.proto"; // Параметр, специфический для Java. Если java_package опущен, то генерируемый код помещается в объявление пакета

option java_multiple_files = true; // Код на Java для каждого сообщения message, сгенерированный из объявлений Protobuf, будет помещен в отдельный файл .java

message Avenger { // Начало объявления сообщения message
    string name = 1; // Определение единичного поля
    string real_name = 2; // Определение единичного поля
    repeated string movies = 3; // Определение повторяющегося поля
}

}
```

Формат определения схемы Protobuf относительно прост. В этом примере я хочу отметить лишь одно: в Protobuf версии 3 все поля считаются необязательными.

Если при десериализации сообщения обнаружится, что в определении формата отсутствует какое-то поле, то Protobuf использует значение по умолчанию для данного типа. Например, для числовых типов по умолчанию используется значение 0, для логических типов — `false`, а для строк — пустая строка. Значение по умолчанию для отсутствующих полей типа `message` зависит от языка программирования, но в случае Java это будет `null`.

Обратите внимание, что имени поля мы присвоили число. Protobuf использует это число для идентификации поля в сообщении в двоичном формате сообщения. Это означает, что, начав использовать сообщение, уже нельзя будет повторно использовать числа, назначенные полям.

Числа от 1 до 15 занимают всего 1 байт в закодированном формате, поэтому ими обычно обозначаются часто встречающиеся поля сообщений. Для получения

дополнительной информации о нумерации полей обращайтесь к разделу *Assigning Field Numbers* в документации ProtoBuf (<http://mng.bz/RZGK>).

Поля в сообщении могут быть единичными (по умолчанию) или повторяющимися. Схема, которую мы только что рассмотрели, содержит примеры полей обоих видов. Первые три поля в нашей схеме являются единичными. Единичные поля создаются по умолчанию, и поэтому в объявление не нужно добавлять никакой дополнительной информации. Поле `repeated string movies = 3;` в предыдущей схеме — повторяющееся, то есть оно может содержать 0 или более значений. Повторяющиеся поля подобны массивам с изменяемым размером, и в Java API они представлены типом `List<T>`.

## B.2.1. Сложные сообщения

ProtoBuf позволяет внутри сообщения определять поле с типом сообщения. Кроме того, поле в сообщении может ссылаться на другое сообщение. Рассмотрим предыдущий пример сообщения `Avenger`. Допустим, мы решили добавить в сообщение дополнительную информацию о фильме, кроме названия, и определили новый тип сообщения `AvengerMovie` в файле `proto` из предыдущего примера (листинг B.2).

**Листинг B.2.** Добавление нового сообщения с информацией о фильме в `avenger_v1.proto`

```
syntax = "proto3";

package proto_files;

option java_package = "bbejeck.chapter_3.proto";
option java_multiple_files = true;

message Avenger { ← Содержит дополнительное
    message AvengerMovie { ← Определяет поле
        string title = 1;
        string year = 2;
        string producer = 3;
    }

    string name = 1;
    string real_name = 2;
    repeated AvengerMovie = 3;
}
```

В этом примере я определил вложенный тип и использовал его для определения поля, но сделано это было исключительно в демонстрационных целях, потому что в действительности совершенно не обязательно объявлять вложенные типы и использовать их для определения полей, если только в этом нет явной необходимости.

### ПРИМЕЧАНИЕ

Рискуя констатировать очевидное, хочу подчеркнуть, что нумерация полей во вложенных типах сообщений не зависит от нумерации во вмещающих типах. Другими словами, в определении вложенного типа нумерация полей начинается с 1.

Protobuf позволяет вкладывать определения типов сообщений на любом уровне. Но помните о сложности поддержки вложенных объявлений. В какой-то момент файлы с глубоко вложенными типами станут очень трудно поддерживать. Иногда лучше разбить сложные вложенные типы на proto-файлы. К счастью, Protobuf предоставляет простой способ работы с отдельными proto-файлами.

## В.2.2. Импорт

Protobuf позволяет определить несколько типов сообщений в одном файле, но это возможно не всегда. Например, у вас может появиться необходимость использовать файл proto из другого модуля. Какова бы ни была причина, возможность импорта играет важную роль, позволяя повторно использовать существующие определения, имеющиеся в базе кода. Чтобы импортировать файл proto, нужно добавить в начало своего файла инструкцию `import`. Вернемся к файлу примера `avenger_v2.proto` и переместим определение сообщения `AvengerMovie` в отдельный файл (листинг В.3).

### Листинг В.3. Отдельный файл avenger\_movie.proto

```
syntax = "proto3";

package proto_files;

option java_package = "bbejeck.chapter_3.proto";
option java_multiple_files = true;

message AvengerMovie {
    string title = 1;
    string year = 2;
    string producer = 3;
}
```

Чтобы теперь использовать сообщение `AvengerMovie`, скопируйте файл `avengerV1.proto` в файл `avengerV2.proto` и добавьте оператор `import` в начало файла, например:

```
syntax = "proto3";

package proto_files;

import "avenger_movie.proto";

option java_package = "bbejeck.chapter_3.proto";
option java_multiple_files = true;

message AvengerV2 {
    string name = 1;
    string real_name = 2;
    repeated AvengerMovie = 3;
}
```

Как поступить — объединить ли несколько определений типов сообщений в одном файле или сохранить их в отдельных файлах, зависит от функциональности, но чаще всего — от конкретного варианта использования.

Прежде чем завершить обсуждение Protobuf, я хочу отметить еще один аспект, который позволяет использовать несколько типов событий в одном топике.

### B.2.3. Тип oneof

В Protobuf имеется тип `oneof`, который позволяет объявить в сообщении поле, которое может иметь один из указанных типов. Тип `oneof` в Protobuf похож на объединение типов `union` в Avro. Важно отметить, что поле типа `oneof` нельзя определить как элемент верхнего уровня, его обязательно нужно заключить в сообщение, то есть его нельзя определить как отдельно стоящий элемент в файле proto.

Рассмотрим пример использования типа `oneof`. За основу возьмем все тот же пример покупки из раздела, где описывалось объединение типов в Avro, чтобы можно было воочию сравнить `union` из Avro и `oneof` из Protobuf (листинг B.4).

#### Листинг B.4. Объявление сообщения Protobuf с полем типа oneOf

```
syntax = "proto3";

package proto_files;

option java_package = "bbejcek.chapter_3.proto";
option java_multiple_files = true;

message TransactionType {

    message Purchase { ← Объявление сообщения Purchase
        string item = 1;
        double amount = 2;
    }

    message Return { ← Объявление сообщения Return
        string item = 1;
        double amount = 2;
    }

    message Exchange { ← Объявление сообщения Exchange
        string item = 1;
        double amount = 2;
        string new_item = 3;
    }

    string identifier = 1;

    oneof txn_type { ← Поле типа oneof
        Purchase purchase = 2;
        Return return = 3;
        Exchange exchange = 4;
    }
}
```

Как показано здесь, в файле proto нужно определить все поддерживаемые типы сообщений, а затем в объявлении поля (в этом примере `oneof txn_type`) перечислить типы значений, которые может содержать поле. Поддержка полей, которые могут принимать несколько типов, имеет большое значение для использования `RecordNameStrategy` или `TopicRecordNameStrategy`, поскольку эти стратегии позволяют передавать в топики записи нескольких типов.

В этом примере имеет смысл определить все типы сообщений в одном файле, так как они тесно связаны. Однако если понадобится передавать в топик записи нескольких типов, а файлы proto находятся в разных местах, то можно использовать ссылки на схему. Давайте посмотрим, как это сделать.

Я не буду показывать отдельные файлы в этом примере, но вы можете смело предположить, что каждое определение вложенного типа сообщения помещено в свой собственный файл. Таким образом, обновленный файл схемы теперь будет выглядеть, как показано в листинге B.5.

#### Листинг B.5. Схема Protobuf с полем типа oneof в файле transaction\_type.proto

```
syntax = "proto3";

package proto_files;
import "purchase.proto";           ← Добавлены
import "return.proto";
import "exchange.proto";           ← инструкции import

option java_package = "bbejeck.chapter_3.proto";
option java_multiple_files = true;

message TransactionType {

    string identifier = 1;
    oneof txn_type {             ← Определение поля типа oneof ссылается
        Purchase purchase = 2;
        Return return = 3;
        Exchange exchange = 4;
    }
}
```

Этот файл, который импортирует объявления типов из других файлов, выглядит намного проще. Теперь, создав файл со ссылками на схемы, посмотрим, как зарегистрировать новую схему, чтобы получить возможность использовать различные типы событий (листинг B.6).

#### ПРИМЕЧАНИЕ

При использовании ссылок на схемы необходимо задействовать стратегию `TopicNameStrategy`, которая определяет субъект для поиска схемы с учетом имени топика. Применение этой стратегии играет важную роль, потому что она применяет ограничения «субъект — топик» для всех типов в ссылках на схемы *только* к топику, использующему ссылки.

**Листинг B.6.** Ссылка на схему Protobuf

```
{
    "schema" : "syntax = \"proto3\";           ← Часть запроса со схемой,
                                                содержащей ссылки

    package proto_files;

    import \\"purchase.proto\\";
    import \\"return.proto\\";
    import \\"exchange.proto\\";

    option java_package = \\"bbejeck.chapter_3.proto\\";
    option java_multiple_files = \\"true\\";

    message TransactionType {
        string identifier = 1;                ← Ссылки на схемы по именам
                                                типов сообщений
        oneof txn_type {
            Purchase purchase = 2;
            Return return = 3;
            Exchange exchange = 4;
        }
    }

    ",                                     ← Задает тип схемы, если это объявление
                                                опустить, то по умолчанию будет
                                                использован формат AVRO
    "schemaType": "PROTOBUF",             ←

    "references": [                         ← Ссылки на уже зарегистрированные
                                                схемы для этих объектов событий
    {
        "name": "purchase.proto",
        "subject": "purchase",
        "version": 1
    },
    {
        "name": "return.proto",
        "subject": "return",
        "version": 1
    },
    {
        "name": "exchange.proto",
        "subject": "exchange",
        "version": 1
    }
]
}
```

**COBET**

При использовании ссылок на схемы в Protobuf не отключайте `auto_schema.registration`, потому что Protobuf рекурсивно загружает все указанные схемы.

Теперь мы знаем, как регистрировать схемы Protobuf и использовать несколько схем через ссылки. Далее перейдем к генерации кода с помощью компилятора Protobuf.

### **B.2.4. Генерация кода**

Protobuf, как и Avro, предоставляет механизм генерации кода на основе файла схемы. Я уже говорил о важности генерации кода в разделе об Avro, но повторю еще раз: вы должны сгенерировать исходный код с определениями типов объектов, которые собираетесь использовать в своих приложениях.

Самое замечательное в инструментах генерации кода — они снимают бремя создания объектов моделей, которые будут использоваться. Помимо избавления от утомительной рутины, генерация кода гарантирует соответствие объектов спецификациям схемы.

Protobuf поддерживает генерацию кода на нескольких языках, но в этой книге мы используем только генерацию кода на Java. Теперь уделим минуту рассмотрению некоторых особенностей генерации исходного кода на Java для схемы.

Protobuf предоставляет инструмент `protoc`, генерирующий исходный код на Java из файлов proto. Но вместо него мы будем использовать плагин `gradle`. Плагин сам вызывает `protoc`, но в этом случае генерация кода является частью процесса сборки проекта, выполняется автоматически и снимает с нас часть хлопот.

Чтобы опробовать пример, перейдите в корневой каталог, куда вы клонировали исходный код примеров для книги. Выполните команду `./gradlew clean build`, а затем перейдите в каталог `build/generated-main-proto-java/main/java/bbejeck.chapter_3/proto`. Там вы увидите файл `AvengerProto.java`.

Как и в случае с файлами, сгенерированными Avro, объекты Java, сгенерированные Protobuf, являются неизменяемыми. Учитывая эту особенность, Protobuf предоставляет объекты-построители `Builder`, конструирующие объекты сообщений. Каждый тип сообщения получает свой объект `Builder`, доступный как `MessageName.Builder`. Вложенные типы сообщений тоже получают свои объекты `Builder`, на которые можно ссылаться так: `OuterMessage.NestedMessage.Builder`.

#### **ПРИМЕЧАНИЕ**

Что такое паттерн построителя? Этот паттерн предлагает гибкий способ создания объектов. Требуемые поля добавляются не через параметры конструктора, а через методы объекта-построителя. Каждый такой метод представляет поле объекта. Это один из паттернов проектирования «Банды четырех» (<https://martinfowler.com/bliki/GangOfFour.html>).

Примеры файлов proto имеют два параметра, специфические для Protobuf Java API: `java_package` и `java_multiple_files`:

- `java_package` — если задано, определяет пакет для сгенерированного кода на Java, в противном случае Protobuf использует поле `package`. Для организации файлов proto в пространстве имен лучше использовать поле `package`, а `java_package` применять для задания правильного имени пакета Java;

- `java_multiple_files` – значение `true` в этом параметре сообщает Protobuf, что все сообщения верхнего уровня и перечисления, которые определены в файле proto, должны помещаться в отдельные файлы `.java` на уровне пакета. В противном случае сообщения и перечисления будут определены как члены объемлющего внешнего класса.

Итак, теперь вы знаете, как использовать встроенные инструменты Protobuf для генерации файлов исходного кода. Помните: Protobuf поддерживает много языков, но мы сосредоточились только на языке Java, который я использовал в книге. Возможно, вы заметили, что инструменты Avro и Protobuf, несмотря на различия, генерируют довольно похожий код. Некоторые незначительные различия можно обнаружить во вспомогательных методах, которые Protobuf генерирует для поля `oneof`.

## B.2.5. Конкретные и динамические типы

Когда десериализаторы Schema Registry Protobuf создают объект из сериализованного представления, этот объект может иметь конкретный или динамический тип. Этот подход к типизации напоминает подход, используемый в Avro. В зависимости от информации, доступной при десериализации, объект может получить тип конкретного класса или `GenericRecord`.

Обычно предпочтительнее использовать тип конкретного класса, потому что структура объекта известна заранее. Например десериализуя и работая с объектом `AvengerProto`, вы точно знаете, как обращаться с ним в вашем коде. Но, как и в случае с универсальным типом Avro, при работе с динамическим типом Protobuf вам придется исследовать объект, чтобы узнать, какие поля он содержит и какие данные в конечном счете представляет.

Теперь, познакомившись с основами использования Protobuf, перейдем к третьему формату схем, поддерживаемому Schema Registry: JSON.

## B.3. JSON SCHEMA

Перейдем к третьему типу схемы, поддерживаемому Schema Registry: JSON Schema. В последующих примерах с JSON Schema я буду использовать версию draft-07 (<http://json-schema.org/draft-07/schema#>).

### ПРИМЕЧАНИЕ

Как и в разделах, посвященных Avro и Protobuf, я не буду вдаваться в детали использования схемы JSON, но расскажу достаточно, чтобы вы смогли начать. За полной информацией обращайтесь по адресу <https://json-schema.org/>.

При работе со схемами JSON вы заметите некоторое сходство со схемами Avro, поскольку обе они определяются в формате JSON. Однако формат JSON Schema имеет некоторые отличия, из-за чего вы не можете повторно использовать схему Avro для определения схемы JSON Schema.

Посмотрим, как выглядит определение схемы JSON Schema. Чтобы упростить сравнение с другими форматами, снова воспользуемся объектом `Avenger` (листинг B.7).

**Листинг B.7.** Определение объекта Avenger в формате JSON Schema

```
{
  "$schema" : "http://json-schema.org/draft-07/schema#", ← Объявление версии схемы
  "title" : "Avenger",
  "description" : "A JSON schema of Avenger object", ← Определять заголовок не обязательно, но его наличие дает дополнительные удобства
  "javaType": "bbejeck.chapter_3.SimpleAvenger",
  "type" : "object",
  "properties" : { ← Объявление типа, в этом случае object. Это аналог типа record в Avro и message в Protobuf
    "name" : { ← В разделе properties перечисляются поля объекта
      "type": "string"
    },
    "movies" : {
      "type": "array",
      "items": {
        "type": "string"
      },
      "default": []
    },
    "realName": {
      "type": "string"
    }
  },
  "required" : ["name", "realName"] ← Это объявление обязательного поля
}
```

Как видите, схема JSON Schema довольно близка к Avro. Тип `object` — это один из простых типов, по сути, ассоциативный массив с атрибутом `properties`, представляющим ключи и значения, содержащиеся в объекте.

К простым типам также относятся `array`, `boolean`, `integer`, `null`, `number` и `string`. Свойства объекта по умолчанию считаются необязательными. Чтобы объявить поле обязательным, его имя нужно поместить в массив `required`.

**B.3.1. Вложенные объекты**

JSON Schema также поддерживает вложенные типы объектов. Расширим пример, добавив поле `movies`, как мы делали это раньше (листинг B.8).

**Листинг B.8.** JSON Schema с вложенным типом `object`

```
{
  "$schema" : "http://json-schema.org/draft-07/schema#",
  "title" : "Avenger",
  "description" : "A JSON Avenger object with a nested object",
  "javaType" : "bbejeck.chapter_3.SimpleAvenger"
  "type" : "object",
  "properties" : {

    "name" : {
      "type" : "string"
    },
    "realName": {
      "type" : "string"
    },
    "movies" : {
      "type": "array",
      "items": {
        "type": "string"
      }
    }
  }
}
```

```

"movies" : { ← Добавлено свойство movies
    "type" : "array",
    "items" : { Свойство movies — это массив, и в нем
        "type" : "object", ← нужно указать тип элементов
        "javaType": "bbejeck.chapter_3.AvengerMovie" ← Задает имя класса
        "properties" : AvengerMovie
            {
                "title" : {
                    "type" : "string"
                },
                "year" : {
                    "type" : "string"
                },
                "producer" : {
                    "type" : "string"
                }
            },
            "required" : ["title", "year", "producer"] ← Все три свойства
        }                                     вложенного типа,
    },
    "required" : ["name", "realName"]
}

```

Этот пример показывает, что все свойства вложенного типа `object` в массиве являются обязательными, но сам массив `movies` не является таковым. Однако вложение структур `object` может быстро усложнить управление ими. К счастью, JSON Schema тоже поддерживает ссылки, что может помочь сделать основную структуру `object` более управляемой.

## B.3.2. Ссылки JSON

Итак, изменим эту схему и используем ссылки на определения объектов (листинг B.9).

**Листинг B.9.** Схема JSON Schema со ссылками

```

{
    "$schema" : "http://json-schema.org/draft-07/schema#",
    "title" : "Avenger",
    "description" : "A JSON schema of Avenger object with an
        internal schema ref",
    "definitions" : {
        "avenger_movie" : { ← Ключевое слово definitions
            "type" : "object" ,
            "properties" : {
                "title" : { "type" : "string" },
                "year" : { "type" : "string" },
                "producer" : { "type" : "string" }
            },
            "required" : ["title", "year", "producer"]
        }
    }
}

```

```

},
"description" : "A JSON schema of Avenger object",
"type" : "object",
"javaType": "bbejeck.chapter_3.SimpleAvenger",
"properties" : {

    "name" : {
        "type" : "string"
    },
    "realName": {
        "type" : "string"
    },
    "movies" : {
        "type" : "array",
        "items": { "$ref" : "#definitions/avenger_movie" } ← Ссылка на avenger_movie
    }
}

}

```

В процессе интерпретации ключ `$ref` заменяется содержимым определения, на которое указывает ссылка. Обновленная схема получилась такой же длинной, как и предыдущая, но читается намного легче. Однако в атрибуте `$ref` можно также указывать ссылки на отдельные файлы со схемами JSON Schema. Вернемся к схеме, которую мы только что рассмотрели, и предположим, что извлекли определение `avenger_movie` в файл `avenger_movie.schema.json`. Теперь мы можем обновить схему, подставив ссылку на файл, — поведение `$ref` при этом останется прежним: он подставит содержимое файла на свое место. В листинге B.10 показана обновленная схема со ссылкой на внешний файл.

#### Листинг B.10. Схема JSON Schema со ссылкой на внешний файл

```

{
    "$schema": "http://json-schema.org/draft-07/schema#",
    "title": "Avenger with an external schema reference",
    "description": "A JSON schema with refs to external schemas",
    "javaType": "bbejeck.chapter_3.ComplexAvenger", ← Изменяет имя класса Java,
    "type": "object",                                         чтобы отразить изменения в схеме
    "properties": {
        "name": {
            "type": "string"
        },
        "realName": {
            "type": "string"
        },
        "movies": {
            "type": "array",
            "items": {
                "$ref": "avenger_movie.schema.json"
            }
        }
    }
}

```

Использовав внешние ссылки, мы еще больше упростили схему. Обратите внимание: в этом примере предполагается, что файл, указанный в ссылке, находится в том же каталоге, что и ссылающийся на него файл.

### ПРИМЕЧАНИЕ

При работе со схемами JSON полезно иметь под рукой валидатор, помогающий убедиться, что схема построена правильно. На момент написания этой книги существовал онлайн-валидатор по адресу <https://www.jsonschemavalidator.net/>, который я посчитал полезным.

JSON Schema предоставляет несколько ключевых слов, позволяющих объединять схемы: `allOf`, `anyOf`, `oneOf` и `not`. Далее мы рассмотрим ключевое слово `oneOf`, цель которого – дать возможность использовать несколько типов событий Schema Registry в одном топике. Ключевое слово `oneOf` действует подобно ключевому слову `oneof` в Protobuf.

Далее представлен пример импортирования схемы JSON Schema из Schema Registry.

### B.3.3. Ссылки на схему JSON Schema в Schema Registry

Вы уже познакомились с созданием сложных схем и ссылками в JSON Schema, поэтому сразу перейдем к регистрации схемы JSON Schema, импортирующей другую схему. И снова рассмотрим наиболее удобочитаемую схему, которую зарегистрируем в Schema Registry. Информация в разделе "schema" записывается в одну строку без переносов (листинг B.11).

**Листинг B.11.** Схема JSON Schema со ссылками на другие схемы

```
{
  "schema" : ← Схема JSON
    {
      "\"schema\" : \"http://json-schema.org/draft-07/schema#\",
      "\"title\" : \"JSON schema example\",
      "\"description\" : \"Schema imports with JSON\",
      "\"javaType\" = \"transactionInfo\" ← Определяет имя класса

      "\"oneOf\" : [
        { '\"$ref\" : \"purchase.schema.json\" },
        { '\"$ref\" : \"return.schema.json\" },
        { '\"$ref\" : \"exchange.schema.json\" }
      ]
    },
    "schemaType": "JSON", ← Определяет тип схемы как JSON

    "references" : [ ← Ссылки на уже зарегистрированные схемы JSON
      {
        "name": "purchase.schema.json",
        "subject": "purchase",
        "version": 1
      },
    ]
}
```

Поле типа `oneOf` со ссылками

```
{  
    "name": "return.schema.json",  
    "subject": "return",  
    "version": 1  
},  
{  
    "name": "exchange.schema.json",  
    "subject": "exchange",  
    "version": 1  
}  
]  
}
```

Кроме некоторых синтаксических отличий, формат определения ссылок на схемы JSON похож на формат, используемый в Avro и Protobuf. При использовании ссылок на схемы в схемах JSON Schema необходимо внести те же изменения в конфигурацию, что и в Avro. Как вы наверняка помните, мы должны установить свойства `auto.register.schema=false` и `use.latest.version=true`. Глава 4 рассказывает о настройке различных типов схем в клиентах (производителях и потребителях).

Есть еще один важный момент при работе со схемами JSON, особенно в языках со статической типизацией, таких как Java. Это определение имени класса. Если сравнить схемы JSON со схемами Avro или Protobuf, то можно заметить отсутствие информации, которую можно использовать в качестве имени пакета или класса. Но схемы JSON Schema позволяют добавлять поля, не перечисленные в спецификации схемы явно. Это называется моделью с открытым содержимым.

Модель с открытым содержимым дает возможность добавить поле верхнего уровня в файл схемы и задать имя класса для десериализации. Поддержку дополнительных полей можно отключить, установив `"additionalProperties": false` в схеме объекта; если этого не сделать, то по умолчанию будет предполагаться, что этот параметр имеет значение `true`. При использовании схем JSON добавление поля верхнего уровня, задающего имя класса, необходимо для десериализации, когда в топике могут иметься сообщения нескольких типов.

К данному моменту мы рассмотрели достаточно информации, чтобы начать понимать схемы JSON Shema. Но схема — это спецификация, а нам нужно работать с объектами, которые следуют спецификации схемы. В следующем разделе мы обсудим, как перейти от схемы к объектам.

### **B.3.4. Генерация кода на основе схемы JSON Schema**

Как я уже говорил, JSON Shema отличается от Avro и Protocol Buffers в плане генерации исходного кода. Разница в том, что схема JSON Shema — это спецификация, а Avro и Protocol Buffers — это фреймворки с полным набором инструментов. Поэтому генерация кода — встроенная возможность в Avro и Protobuf. Но для генерации кода на основе схем JSON Shema приходится пользоваться сторонними инструментами.

Есть несколько популярных инструментов для работы с объектами Java и JSON. И несмотря на наличие множества неплохих вариантов, в этой книге я буду

использовать утилиты FasterXML/Jackson (<https://github.com/FasterXML/jackson>). Jackson предоставляет поддержку сериализации и аннотации, которые можно использовать с объектами Java. Но я буду использовать поддержку аннотаций, реализованную в дополнительной библиотеке с открытым исходным кодом — Jackson jsonSchema Generator (<https://github.com/mbknor/mbknor-jackson-jsonSchema>). Jackson также поддерживает генерацию схем, но библиотека mbknor-jackson-jsonSchema имеет возможности, выходящие за рамки Jackson.

Прежде чем перейти к дополнительным аннотациям, рассмотрим класс Java, использующий базовые аннотации (некоторые детали опущены для простоты) (листинг B.12).

**Листинг B.12.** Код на Java с аннотациями Jackson

```
public class Customer {
    @JsonProperty
    private String name;

    @JsonProperty
    private int id;

    @JsonProperty
    private String email;

}
```

Аннотации дают нам несколько вариантов. Во-первых, не нужно генерировать схему физически. Мы можем положиться на сериализаторы JSON в Schema Registry для разбора и регистрации схемы в процессе сериализации. Но если в одном топике имеются события нескольких типов, то необходимо добавить поле верхнего уровня, определяющее имя класса, как обсуждалось в предыдущем разделе. И в таком случае нужно сгенерировать физическую схему.

Необходимость добавления поля верхнего уровня, определяющего имя класса, — это то, где в игру вступает mbknor-jackson-jsonSchema. Библиотека mbknor-jackson-jsonSchema предоставляет аннотацию `@JsonSchemaInject`, которая поддерживает внедрение фрагментов JSON Schema. Давайте обновим предыдущий пример и добавим аннотацию `@JsonSchemaInject` (некоторые детали опущены для простоты) (листинг B.13).

**Листинг B.13.** Код на Java с аннотациями для внедрения имени класса

```
@JsonSchemaInject(strings=
    {@JsonSchemaString(
        path="javaType",
        value="bbejeck.chapter_3.codegen.Customer")})
public class Customer {
    @JsonProperty
    @JsonProperty
    private String name;

    @JsonProperty
    private int id;
```

```
@JsonProperty  
private String email;  
  
}
```

Код остался прежним, мы лишь добавили аннотацию `@JsonSchemaInject`, которая создает поле верхнего уровня `javaClassName` и включает поддержку нескольких типов событий в одном топике. Теперь, чтобы сгенерировать физическую схему с помощью генератора `mbknor-jackson-jsonSchema`, можно выполнить код в листинге В.14.

#### Листинг В.14. Использование генератора схем

```
theClass = Class.forName(args[0]);  
ObjectMapper mapper = new ObjectMapper();  
JsonSchemaGenerator schemaGenerator = new JsonSchemaGenerator(mapper);  
JsonNode jsonNode = schemaGenerator.generateJsonSchema(theClass);  
String schema = mapper.writerWithDefaultPrettyPrinter()  
    .writeValueAsString(jsonNode);
```

Результирующая схема будет выглядеть так, как показано в листинге В.15.

#### Листинг В.15. Сгенерированная схема JSON Schema

```
{  
  "$schema" : "http://json-schema.org/draft-04/schema#",  
  "title" : "Customer",  
  "type" : "object",  
  "additionalProperties" : false,  
  "javaType" : "bbejeck.chapter_3.codegen.Customer",  
  "properties" : {  
    "name" : {  
      "type" : "string"  
    },  
    "id" : {  
      "type" : "integer"  
    },  
    "email" : {  
      "type" : "string"  
    }  
  },  
  "required" : [ "id" ]  
}
```

Пример генерации схемы имеется в исходном коде для книги в пакете `bbejeck.chapter_3.codegen`.

Подход, который мы рассмотрели в этом разделе, называется «сначала код, потом схема». Мы начинаем с существующих или пишем свои объекты Java, применяем аннотации, а затем генерируем схему. Однако кто-то может посчитать, что писать код вручную неудобно, и предпочел бы сначала написать схему, а затем сгенерировать код. Поэтому далее мы рассмотрим генерацию кода из схемы.

Другой инструмент, <https://github.com/joelittlejohn/jsonschema2pojo>, предлагает возможность генерации кода на Java из схемы JSON. Библиотека `jsonschema2pojo`

поддерживает также генерацию с использованием аннотаций. Поддерживаемые стили аннотаций предназначены для следующих инструментов Java-JSON: Jackson 1.x, Jackson 2.x, Gson и Moshi.

#### ПРИМЕЧАНИЕ

Я уже упоминал Jackson, но здесь я представляю два новых инструмента Java – JSON. Gson (<https://github.com/google/gson>) от Google и Moshi (<https://github.com/square/moshi>) от Square. Оба являются библиотеками для преобразования объектов Java в JSON-представление и обратно. Я не буду вдаваться в подробности ни об одном из них. Но если вы захотите узнать больше о любом из них, то переходите по предлагаемым ссылкам.

Уникальная особенность библиотеки jsonschema2pojo – она предлагает онлайн-решение, <http://www.jsonschema2pojo.org/>, способное генерировать исходный код на Java прямо в браузере. Имеется также версия в виде инструмента командной строки и плагинов для Maven и Gradle. Поскольку проект с примерами для этой книги основан на Gradle, я буду использовать плагин `gradle`.

#### ПРИМЕЧАНИЕ

В репозитории GitHub проекта jsonschema2pojo предлагается устаревший плагин `gradle`, работающий только с версиями Gradle < 7.0. Но на основе плагина `gradle` был создан плагин <https://github.com/eirnpyum/js2p-gradle>, который продолжает развиваться, поэтому я буду использовать его в этой книге.

Чтобы сгенерировать код на Java на основе схемы, выполните команду `./gradlew build`. Запуск сборки одновременно сгенерирует исходный код для Avro, Protobuf и JSON Schema. Я не буду вдаваться в подробности настройки разных параметров сборки, но вы можете заглянуть в файл `build.gradle`, чтобы получить дополнительную информацию.

Итак, на этом мы завершаем обсуждение генерации кода на основе схем JSON Schema. Далее мы поговорим о настройке различных сериализаторов, предоставляемых Schema Registry для работы с Avro, Protobuf и JSON Schema.

#### ПРИМЕЧАНИЕ

Читателям, использующим macOS (или Linux), библиотеку jsonschema2pojo можно установить с помощью Homebrew: `brew install jsonschema2pojo`. Для тех, кто не знаком с Homebrew, отмечу, что это менеджер пакетов для установки программного обеспечения. Узнать больше о Homebrew можно на его домашней странице: <https://brew.sh/>.

### B.3.5. Конкретные и обобщенные типы

Как в Avro и Protobuf, в JSON Schema доступны две категории объектов при десериализации. При использовании имени конкретного класса десериализатор Schema Registry возвращает объект этого типа. Если конкретный класс не настроен или десериализатор не сможет определить класс, то он вернет объект `JsonNode`. Работа с `JsonNode` аналогична работе с DOM (document object model – «объектная модель документа») XML.

Чтобы использовать ссылки на схемы, эти схемы сначала нужно зарегистрировать. Затем в ссылающейся схеме нужно настроить три параметра для каждой схемы, на которую она ссылается.

1. Имя ссылки.
2. Субъект, под которым зарегистрирована схема.
3. Точная версия схемы, указанной в субъекте (заданном в п. 2).

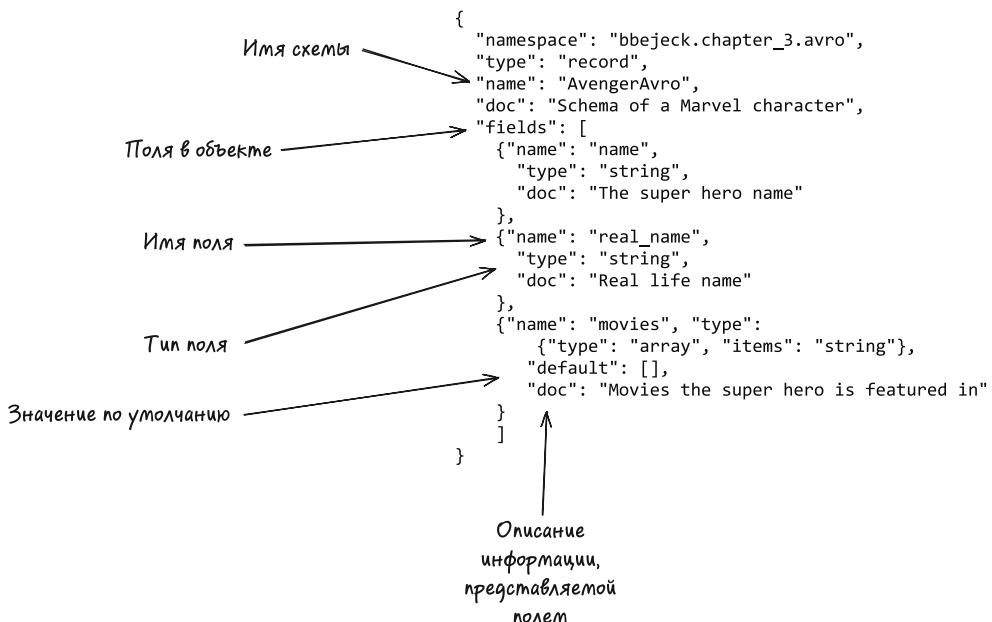
В Avro имя ссылки — это полное имя файла схемы. В Protobuf — имя файла proto, а в JSON — URL.

Ссылки на схемы удобны, когда есть топик, ожидающий объекты одного типа, и еще один топик, в котором объекты содержат поля того же типа, что и объекты в первом. Однако использование ссылок на схемы не является обязательным требованием, поскольку в таких случаях имеется более одного топика, требующего одну и ту же схему. Ссылки на схемы удобно использовать для упрощения объединений типов, потому что ссылка в объявлении поля выглядит менее громоздко, чем полное определение сложной схемы.

### ПРИМЕЧАНИЕ

Schema Registry также поддерживает ссылки на схемы Protobuf и JSON. Мы еще вернемся к ним в последующих разделах. Но при этом мы будем рассматривать только синтаксические особенности, характерные для Protobuf и JSON, поскольку в общем и целом мы уже рассмотрели их выше.

Взгляните на рис. В.2, поясняющий сказанное мною.



**Рис. В.2.** Ссылки на схемы удобны, когда уже есть схема для другого топика, который нужно использовать повторно

Теперь, когда вы узнали, как работают ссылки на схемы, вернемся к примеру из подраздела B.1.2, где рассказывалось об объединениях в Avro, и посмотрим, как выглядят ссылки.

```
{
  "type": "record",
  "namespace": "bbejeck.chapter_3"
  "name": "Transaction",
  "fields" : [
    {"name": "txn_type", "type": [
      {"name": "identifier", "type": "long"}  

    ]},
    {"name": "txns", "type": "array", "items": {
      "type": "record", "namespace": "bbejeck.chapter_3", "name": "Txn", "fields": [
        {"name": "id", "type": "long"},  

        {"name": "type", "type": "union", "variants": [
          {"type": "string", "enum": ["Purchase", "Return", "Exchange"]},  

          {"type": "null"}  

        ]},  

        {"name": "amount", "type": "long"},  

        {"name": "date", "type": "string", "format": "date-time"}  

      ]},  

      {"name": "meta", "type": "map", "valuesType": "string"}  

    ]}  

}
```

Использует объединение типов для представления поля, которое может иметь значение одного из трех типов

`bbejeck.chapter_3.Purchase`,  
`bbejeck.chapter_3.Return`,  
`bbejeck.chapter_3.Exchange`

Итак, в этой схеме мы используем объединение union для поля `txn_type`. По сравнению со схемой, что приводится выше, эта схема намного проще, поскольку в ней указан небольшой массив ссылок, а не полных определений схем.

Теперь рассмотрим еще один пример, и на этот раз посмотрим, как можно зарегистрировать схему со ссылками. Сначала приведу пример JSON, который можно использовать для регистрации схемы со ссылками:

```
{
  "schema" : {
    "type": "record",
    "namespace": "bbejeck.chapter_3",
    "name": "transaction",
    "fields": [
      {"name": "txn_type", "type": [
        "Purchase",  

        "Return",  

        "Exchange"
      ]},  

      {"name": "identifier", "type": "long"}
    ]
  },
  "schemaType": "AVRO",
  "references" : [
    {
      "name": "bbejeck.chapter_3.Purchase",
      "subject": "purchase",
      "version": 1
    }
  ]
}
```

Часть запроса со схемой, содержащей ссылки

Ссылка на схему по имени класса

Ссылки на уже зарегистрированные схемы для этих объектов

```
{  
    "name": "bbejeck.chapter_3.Return",  
    "subject": "return",  
    "version": 1  
,  
{  
    "name": "bbejeck.chapter_3.Exchange",  
    "subject": "exchange",  
    "version": 1  
}  
]  
}
```

Как видите, схема JSON публикуется вместе с необходимыми ссылками. Этот формат обязательен, в противном случае, если просто отправить список ссылок, Schema Registry не будет знать, с какой схемой связать ссылки.

#### ПРИМЕЧАНИЕ

Представленная здесь схема JSON отформатирована для удобства чтения. А вообще, схема должна быть оформлена как строковое значение без переносов строк.

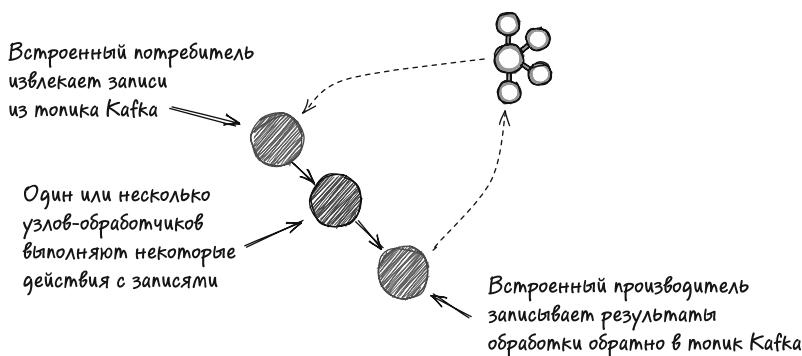


# Архитектура Kafka Streams

В книге вы узнали, что архитектура приложения Kafka Streams имеет вид ориентированного ациклического графа, состоящего из узлов обработки, называемого топологией. Вы увидели, как добавлять узлы-обрабочики в топологию для обработки событий из топика Kafka. Но нам еще нужно обсудить, как Kafka Streams помещает события в топологию, как происходит их обработка и как обработанные события возвращаются обратно в топик Kafka. В этом приложении мы подробно рассмотрим все эти вопросы.

## Г.1. ОБЩИЙ ВИД АРХИТЕКТУРЫ

На рис. Г.1 показано обобщенное представление того, что мы собираемся обсудить.



**Рис. Г.1.** Обобщенное представление приложения Kafka Streams. Здесь есть три секции: потребление, обработка и производство

Как показано на рис. Г.1, работу любого приложения Kafka Streams можно разделить на три этапа, такие как:

- 1) потребление событий из топика Kafka;
- 2) назначение, распределение и обработка событий;
- 3) запись результатов обработки событий обратно в топик Kafka.

Мы уже рассмотрели клиенты Kafka и знаем, что Kafka Streams является абстракцией над ними, поэтому не будем снова вдаваться в эти подробности. Вместо этого я решил объединить потребление и производство в более общее обсуждение клиентов, а затем углубиться в архитектуру Kafka Streams и рассмотреть назначение, распределение и обработку событий.

## Г.2. ПОТРЕБИТЕЛИ И ПРОИЗВОДИТЕЛИ В KAFKA STREAMS

Как показывают примеры в книге, приложение Kafka Streams создает экземпляр `KStream`, который начинается с узла-источника и заканчивается узлом-приемником. Для потребления событий из узла-источника и записи результатов обработки обратно в Kafka в приложениях Kafka Streams используются встроенные клиенты `KafkaConsumer` и `KafkaProducer` соответственно.

Но откуда берутся клиенты? Для этого в Kafka Streams используется внутренний класс `DefaultKafkaClientSupplier`, который реализует интерфейс `KafkaClientSupplier`, предоставляющий все клиенты, необходимые приложению Kafka Streams.

### ПРИМЕЧАНИЕ

`KafkaClientSupplier` имеет методы, которые предоставляют административный клиент, потребитель восстановления, глобальный потребитель и клиент-производитель. Мы уже рассмотрели административный клиент в главе 4. Потребитель восстановления и глобальный потребитель — это простые объекты `KafkaConsumer` с определенными ролями в Kafka Streams, и мы обсудим их позже.

Иногда пользователям требуются свои собственные клиенты. В большинстве случаев в этом нет необходимости, тем не менее это возможно. Например, вы можете написать свой код, расширяющий возможности наблюдаемости, и добавить его в свои классы, наследующие `KafkaConsumer` или `KafkaProducer`.

Объект `KafkaStreams` имеет несколько перегруженных конструкторов, и некоторые из них принимают интерфейс `KafkaClientSupplier` в качестве параметра. Таким образом, чтобы задействовать свои клиенты в приложении Kafka Streams, нужно реализовать экземпляр `KafkaClientSupplier` и передать его конструктору в одном из параметров, как в листинге Г.1 (см. `bbejeck.chapter_6.client_supplier.CustomKafkaStreamsClientSupplier.java`).

### Листинг Г.1. Передача в Kafka Streams своей версии клиента

```
KafkaStreams kafkaStreams =
new KafkaStreams(topology,
properties,
new CustomKafkaStreamsClientSupplier());
```

`CustomKafkaStreamsClientSupplier` в примерах исходного кода не делает ничего особенного. Он просто показывает, как можно внедрить свои реализации клиентских интерфейсов. Когда могут потребоваться свои реализации клиентов? В прошлом я видел, как в некоторых компаниях хотели использовать свою специфическую логику аудита. Но, повторю еще раз, в большинстве случаев возможностей клиентов, встроенных в Kafka Streams, вполне достаточно.

Естественным продолжением обсуждения является ситуация, когда не требуется использовать собственные клиенты, но желательно определить некоторые конфигурации для удовлетворения конкретных потребностей. Конфигурации по умолчанию, используемые в Kafka Streams, прекрасно подходят для большинства случаев, но иногда бывает нужно задать свою, нестандартную конфигурацию.

Определить свои конфигурации клиентов просто, достаточно задать нужные свойства при построении объекта `KafkaStreams`. Kafka Streams передаст заданные вами свойства каждому внутреннему объекту, требующему настройки. Любые конфигурации, не соответствующие ожидаемым, игнорируются.

Например, предположим, что у нас есть несколько узлов-обработчиков, которые требуют много времени для выполнения своей работы, и высока вероятность, что обработка не будет завершена до того, как встроенный потребитель прервет ожидание по тайм-ауту и выполнит еще один вызов `poll`, прежде чем он будет исключен из группы потребителей. Чтобы этого не произошло, можно увеличить максимальное время, которое `KafkaConsumer` проводит в ожидании между вызовами `poll`. Код в листинге Г.2 показывает, как можно выполнить необходимые настройки.

#### Листинг Г.2. Настройка конфигурации для встроенного KafkaConsumer

```
props.put(StreamsConfig.APPLICATION_ID_CONFIG, "yelling_app_id");
props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
props.put(ConsumerConfig.MAX_POLL_INTERVAL_MS_CONFIG, 45_000);
```

Увеличит максимальный  
интервал опроса

Итак, нам потребовалось лишь задать значение для параметра `max.poll.interval.ms` наряду со свойствами, специфичными для потока. Но обратите внимание, что при настройке параметров Kafka Streams применит их во всех случаях, где они востребованы. В этом примере увеличенное время, разрешенное между вызовами `poll`, будет применено к основному потребителю (потребителю, который получает записи для передачи в топологию), потребителю восстановления и глобальному потребителю.

Чтобы избежать конфликтов свойств, Kafka Streams предоставляет префиксы для выбора конкретного клиента, для которого предназначены настройки. Поэтому, чтобы настроить только основной потребитель, следует изменить конфигурацию, как показано в листинге Г.3.

#### Листинг Г.3. Увеличение максимального интервала опроса только для основного потребителя

```
props.put(
    StreamsConfig.mainConsumerPrefix(ConsumerConfig.MAX_POLL_INTERVAL_MS_CONFIG), 45_000);
```

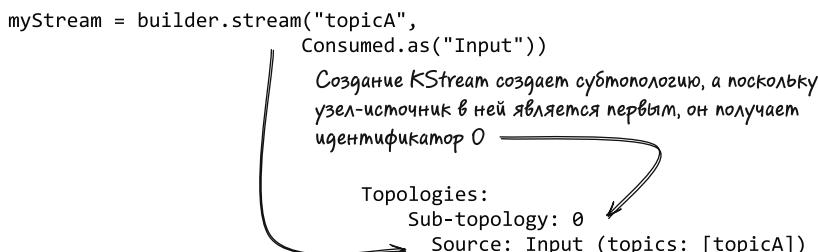
Настройки, выполненные вызовом метода `StreamsConfig.mainConsumerPrefix`, затронут только основной потребитель. Имеются и другие подобные методы: `consumerPrefix`, `producerPrefix`, `topicPrefix` (используется для внутренних топиков, созданных Kafka Streams), `restoreConsumerPrefix`, `globalConsumerPrefix` и `adminClientPrefix`. Я рекомендую использовать эти методы при настройке любых клиентов.

Последнее замечание, касающееся настройки клиентов в Kafka Streams: не все конфигурационные параметры клиентов можно изменить с помощью приложения Kafka Streams. Например, параметр `auto.commit` получает значение `false` и присвоить ему значение `true` невозможно. Это связано с тем, что Kafka Streams может обработать не все записи из предыдущего вызова `poll` к тому времени, когда будет сделан следующий. Поэтому значение `false` в `auto.commit` гарантирует повторную обработку всех частично обработанных записей в случае сбоя. Полное объяснение, когда не следует использовать автоматическое подтверждение транзакций, вы найдете в подразделе 4.3.5.

На этом мы завершаем обсуждение клиентов, встроенных в Kafka Streams, и теперь я предлагаю рассмотреть архитектуру Kafka Streams и ее влияние на обработку записей в топологии.

## Г.3. НАЗНАЧЕНИЕ, РАСПРЕДЕЛЕНИЕ И ОБРАБОТКА СОБЫТИЙ

В этом разделе мы обсудим структуру приложения Kafka Streams и то, как оно организует и распределяет работу. В подразделе 6.6.3 мы уже рассматривали структуру приложений Kafka Streams и то, как каждый узел-источник создает свою субтопологию (рис. Г.2).

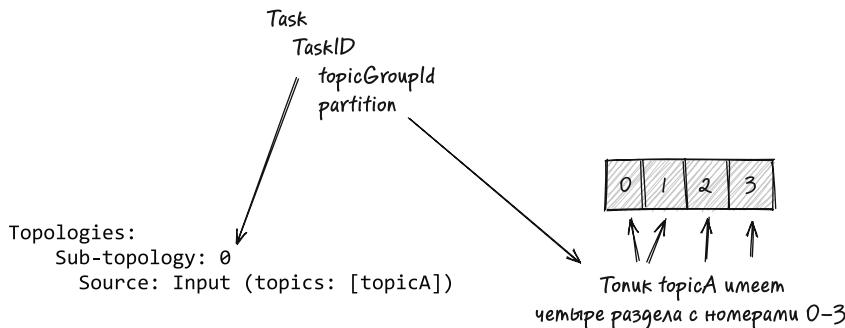


**Рис. Г.2.** Создание экземпляра KStream с узлом-источником создает субтопологию

Субтопология — это дискретная часть приложения со своим узлом-источником и узлами-обработчиками, в число которых может входить конечный узел (скорее всего, узел-приемник для записи результатов обратно в топик Kafka). В главе 6 приводилось несколько диаграмм, показывающих, что такое субтопология. Но прежде, чем продолжить, давайте обсудим, как Kafka Streams определяет, как будут обрабатываться записи.

Kafka Streams использует идею задачи (**Task**) как базовую единицу работы. Задача **Task** обрабатывает записи из определенной субтопологии и номера раздела. Это разделение иллюстрирует рис. Г.3.

Как показано на рис. Г.3, каждая задача **Task** привязана к определенной субтопологии и номеру раздела, из которого она будет извлекать записи для обработки. Обратите внимание, что я использую термин «номер раздела» вместо «раздел». Его значение мы обсудим позже в этом разделе.

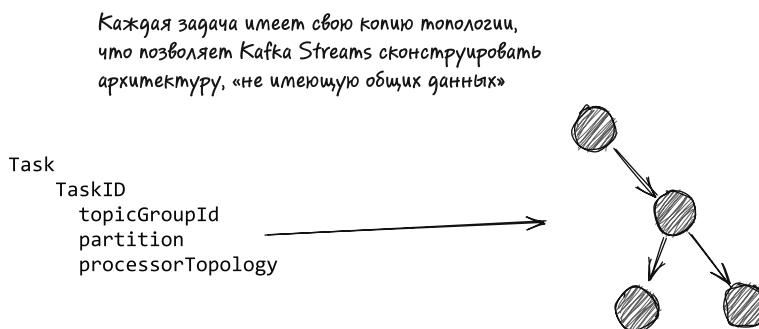


**Рис. Г.3.** Назначение задач в Kafka Streams

#### ПРИМЕЧАНИЕ

Я не буду вдаваться в подробности классов и имен полей, а больше сфокусируюсь на концептуальном представлении, поскольку детали реализации могут меняться со временем.

Каждая задача имеет полную копию конкретной субтопологии, за выполнение которой она отвечает. Взгляните на рис. Г.4, иллюстрирующий этот момент.

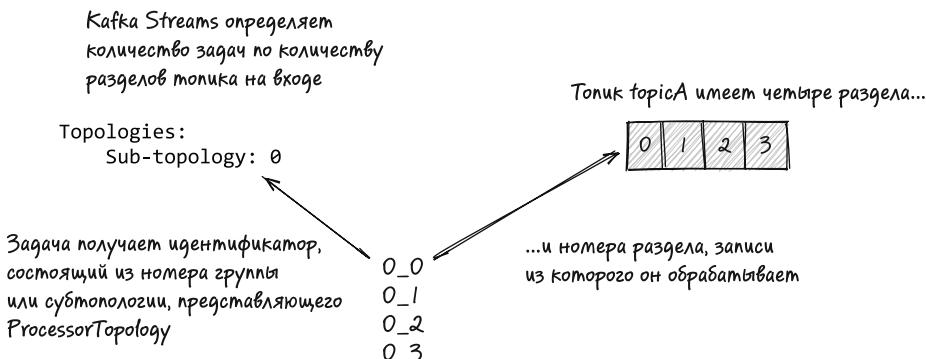


**Рис. Г.4.** Каждая задача имеет копию конкретной субтопологии, ответственной за обработку записей

Поскольку каждая задача имеет свою копию назначенной ей субтопологии, Kafka Streams получает возможность сконструировать архитектуру, не имеющую общих данных, то есть каждая задача обрабатывает записи независимо от других задач. Этот

принцип отсутствия общих данных и работы со своей копией топологии является основополагающим, как отмечалось в главе 7, когда мы говорили о состоянии.

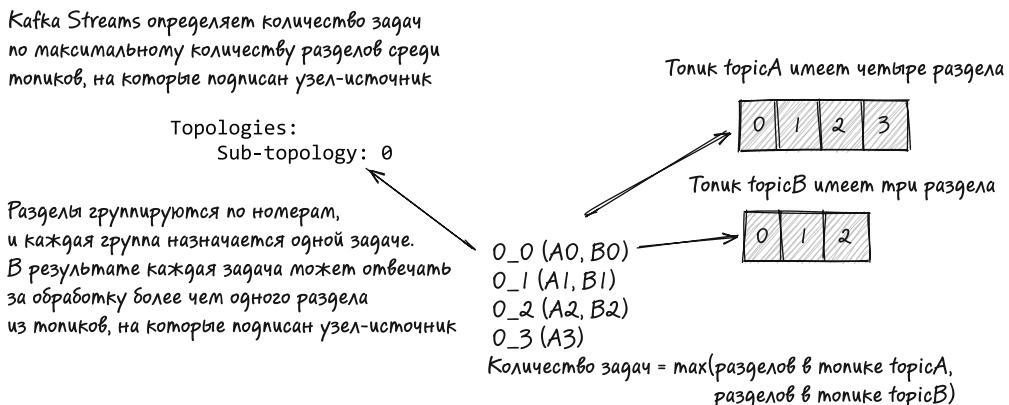
К настоящему моменту вы узнали, что Kafka Streams использует задачи как единицы работы. Но как определяется необходимое количество задач? Количество задач задается количеством разделов в базовом топике. Начнем с простейшего случая с одним узлом-источником и одним топиком. Эта архитектура изображена на рис. Г.5.



**Рис. Г.5.** В случае с одним топиком количество задач равно количеству разделов в нем

Как показывает иллюстрация, для топика с четырьмя разделами создаются четыре задачи. Каждая задача представлена номером раздела субтопологии, который также можно увидеть в идентификаторах задач, которые Kafka Streams выводит в журналы. Итак, первая задача  $0\_0$  отвечает за обработку записей в первой субтопологии 0 из раздела 0 топика **topicA**.

Но, как уже отмечалось ранее, узел-источник может быть подписан на несколько топиков, поэтому взгляните на рис. Г.6, показывающий, как наличие нескольких топиков может повлиять или не повлиять на общее количество задач.

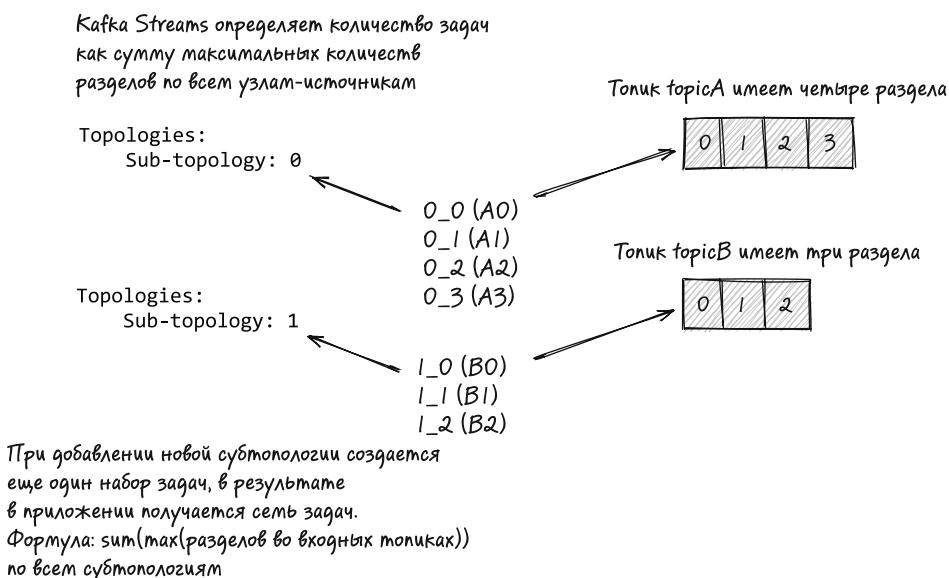


**Рис. Г.6.** При наличии нескольких топиков в одном узле-источнике общее количество задач равно максимальному количеству разделов среди топиков

Как видите, после добавления еще одного топика с тремя разделами общее количество задач не изменилось — их осталось четыре. Однако в трех задачах увеличилось количество разделов, которые они обрабатывают. Это связано с тем, что Kafka Streams группирует разделы по их номерам и каждую группу назначает одной задаче. Таким образом, задача `0_0`, соответствующая первой субтопологии, обрабатывает любые записи из разделов с номером 0 во всех топиках, на которые подписан этот конкретный узел-источник.

То же относится и к другим задачам. Четвертая задача обрабатывает только записи из топика `topicA`. Это означает, что при добавлении нескольких топиков в узел-источник общее количество задач равно максимальному количеству разделов среди всех входных топиков. Если записать это в виде уравнения, то оно будет выглядеть так: количество задач =  $\max(\text{разделов в топике } \text{topicA}, \text{ разделов в топике } \text{topicB})$ . Поскольку новый топик имеет три раздела, общее количество задач остается неизменным.

Наконец, рассмотрим случай, когда в приложение добавляется новый экземпляр `KStream` со своим узлом-источником (рис. Г.7).



**Рис. Г.7.** При наличии нескольких узлов-источников общее количество задач равно сумме максимальных количеств разделов во всех узлах-источниках

При добавлении еще одного узла-источника, который создает еще одну субтопологию, Kafka Streams добавляет еще три задачи, и теперь их в общей сложности получается семь. Соответственно, мы можем обновить нашу формулу определения количества задач, которая выражается как сумма максимальных количеств разделов во всех субтопологиях. Знание количества созданных задач необходимо для получения максимальной производительности от ваших приложений. В следующем разделе мы обсудим эту концепцию более подробно.

## ПРИМЕЧАНИЕ

Каждая задача выполняется независимо и не обменивается информацией или состоянием с другими задачами.

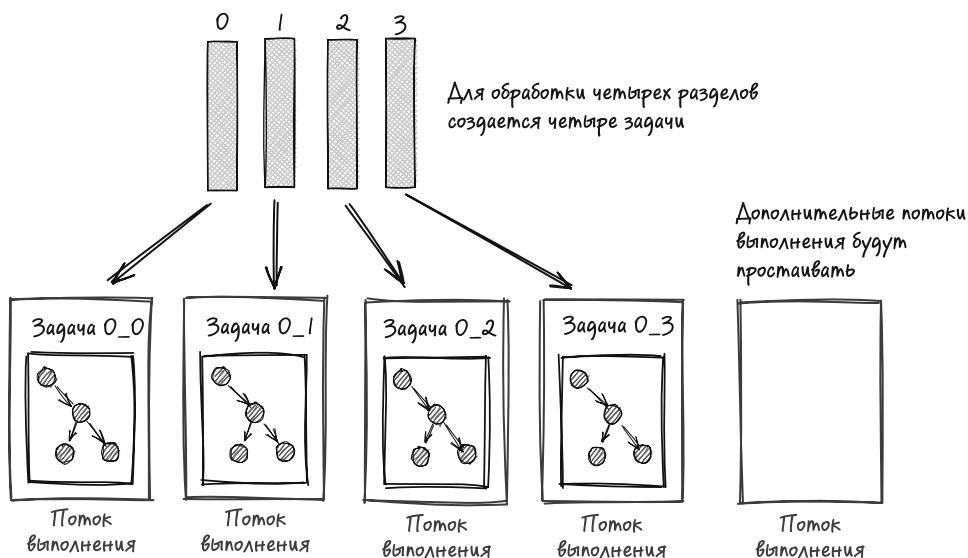
Теперь, рассмотрев единицу работы в приложении Kafka Streams, поговорим о ее выполнении.

## Г.4. ПОТОКИ ВЫПОЛНЕНИЯ В KAFKA STREAMS: STREAMTHREAD

Для запуска задач Kafka Streams использует StreamThread. Обратите внимание, что StreamThread расширяет класс `java.lang.Thread`.

По умолчанию в приложении Kafka Streams запускается один поток выполнения. Чтобы увеличить число потоков, нужно присвоить конфигурационному параметру `StreamsConfig.NUM_STREAM_THREADS_CONFIG` значение больше 1. Возникает вопрос: сколько потоков выполнения следует задать? Чтобы ответить на него, нужно учесть число ядер на компьютере, но пока давайте придерживаться соображений, касающихся только для Kafka Streams.

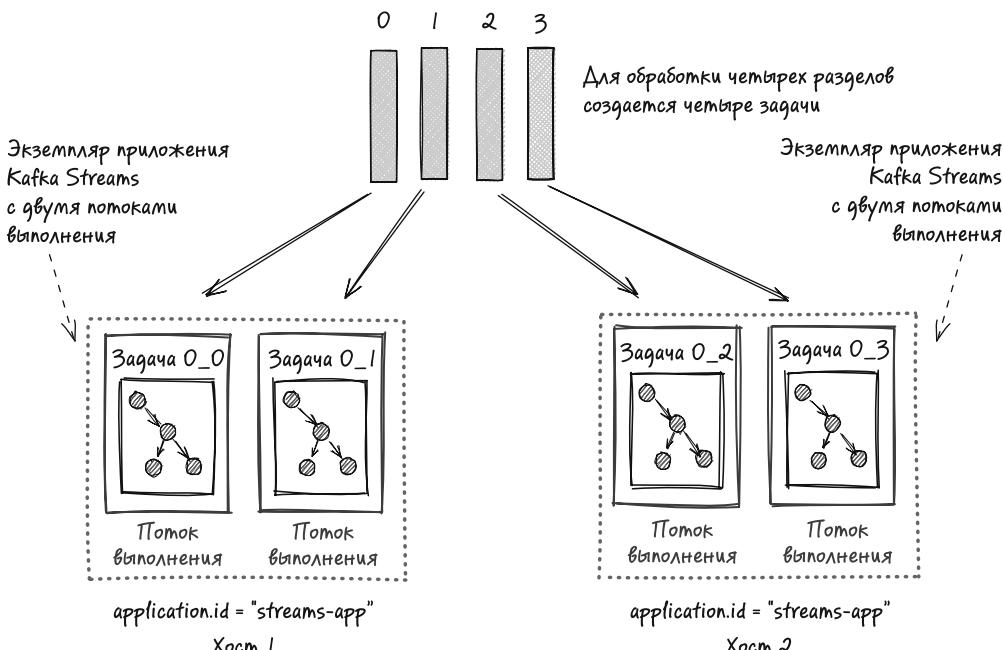
Из обсуждения поведения `KafkaConsumer` в главе 4 мы знаем, что максимальное количество активных клиентов-потребителей соответствует количеству разделов. Любые дополнительные клиенты будут простаивать. Таким образом, здесь применяются те же правила, но максимальное количество потоков выполнения зависит от количества задач (рис. Г.8).



**Рис. Г.8.** Вы можете запустить столько же потоков выполнения, сколько имеется задач, любые дополнительные потоки будут простаивать

В нашем первом примере можно запустить до четырех потоков выполнения. Kafka Streams равномерно распределит задачи по запущенным потокам, но если запустить больше потоков, то избыточные потоки будут простаивать, потому что для них не найдется задач.

Но при определении правильного количества потоков выполнения для использования и исходя из соображений пропускной способности мы должны учитывать не только потоки выполнения. Экземпляры Kafka Streams с одинаковым `application.id` считаются одним логическим приложением. То есть несколько приложений с одинаковым `application.id` проходят через схожий процесс назначения задач. Взгляните на рис. Г.9, иллюстрирующий сказанное.

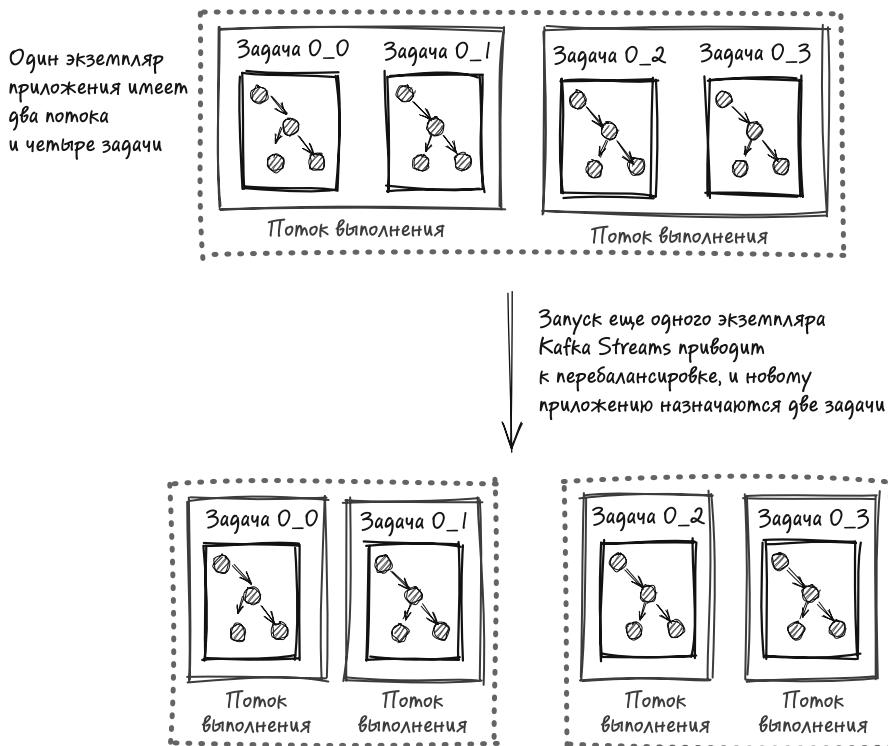


**Рис. Г.9.** Несколько экземпляров приложения Kafka Streams с одинаковым идентификатором представляют одно логическое приложение, поэтому задачи будут распределяться между экземплярами

Если запустить два экземпляра приложения с двумя потоками в каждом, то каждому экземпляру будет назначено две задачи, то есть на каждый поток выполнения придется по одной задаче. Таким образом, основное правило заключается в том, чтобы подобрать такую комбинацию задач и экземпляров приложений, чтобы она соответствовала максимальной пропускной способности.

Эта гибкость экземпляров приложений и потоков выполнения приводит к мощной концепции Kafka Streams: динамическому членству. В главе 4 о клиентах мы обсудили, как работает протокол перебалансировки потребителей. Когда меняется

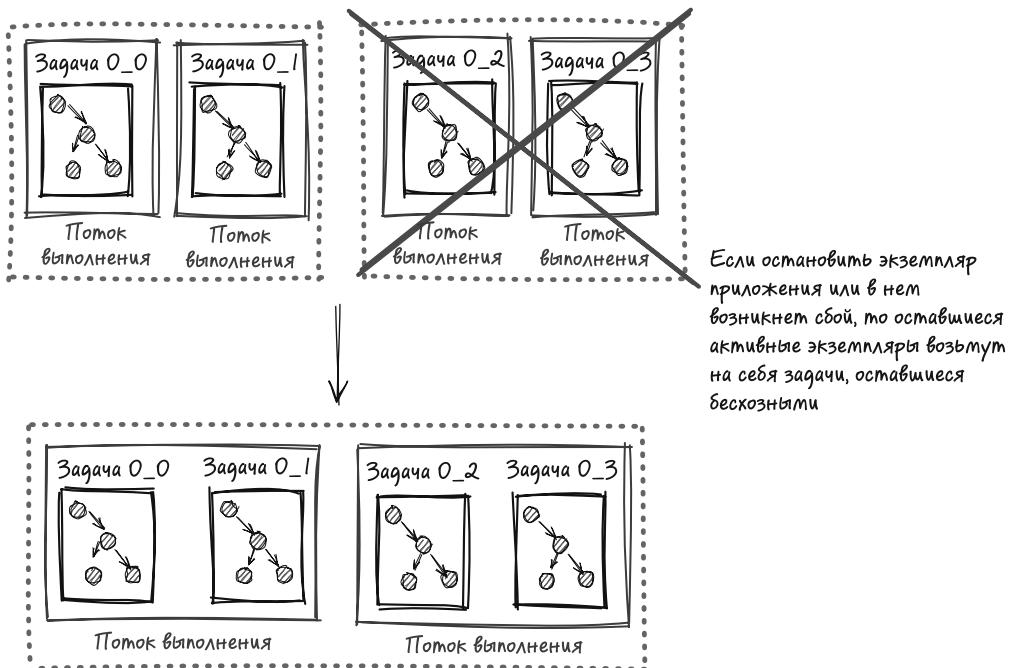
количество членов в группе, происходит перебалансировка, перераспределяющая ресурсы. Поскольку Kafka Streams использует встроенных потребителей Kafka, то для приложений применяется та же функциональность перебалансировки. Этот процесс иллюстрирует рис. Г.10.



**Рис. Г.10.** Запуск второго экземпляра приложения Kafka Streams приводит к перебалансировке и перераспределению задач для нового экземпляра

Когда экземпляр приложения Kafka Streams включается в работу, текущие участники отказываются от некоторых задач в пользу нового экземпляра приложения. Обратное тоже верно. Когда экземпляр прекращает работу, Kafka Streams перераспределяет его задачи между другими активными экземплярами (рис. Г.11). Такое динамическое назначение задач дает нам возможность реагировать на изменения спроса на обработку и запускать или завершать приложения «на лету», а Kafka Streams автоматически осуществит перераспределение.

Как показывает иллюстрация, когда спрос низкий, можно запустить один экземпляр приложения Kafka Streams с одним потоком выполнения, обрабатывающим все шесть задач. Но когда количество событий увеличивается и требуется больше вычислительной мощности, можно запустить еще два экземпляра приложения, и после перебалансировки первоначальное приложение отдаст четыре задачи, а вновь запущенные экземпляры получат по две задачи.



**Рис. Г.11.** Kafka Streams также динамически обрабатывает выбывающие приложения и перераспределяет их задачи между оставшимися активными экземплярами

Мы рассмотрели, как максимизировать пропускную способность, комбинируя количество потоков выполнения в экземпляре Kafka Streams и общее количество экземпляров. Но есть еще одна более тонкая ситуация, которую я хотел бы осветить: когда требуется максимизировать пропускную способность при наличии узла-источника с несколькими топиками. Как вы наверняка помните, количество задач для узла-источника с несколькими топиками определяется как максимальное количество разделов среди всех входных топиков.

Например, предположим, что вы создаете приложение Kafka Streams, которое запускается, как показано в листинге Г.4.

#### Листинг Г.4. Определение потока с одним узлом-источником и несколькими входными топиками

```
StreamsBuilder builder = new StreamsBuilder();
// Топик topicA имеет 4 раздела, а topicB - 3
KStream<String, String> myStream = builder.stream(List.of("topicA", "topicB"));
myStream.filter(..).mapValues(..).to(..);
```

Из предыдущего обсуждения мы знаем, что в итоге получим четыре задачи, хотя входных разделов семь. Запустить такое количество потоков выполнения, чтобы на каждый приходилось по одной задаче, не проблема, но у нас может появиться доступ к нескольким серверам, и мы сможем распределить приложение по нескольким серверам.

В идеале желательно, чтобы на каждый поток выполнения приходилось по одной задаче, так как это максимизирует пропускную способность. Но, поскольку мы объединили топики, у нас есть три задачи, обслуживающие два топика с тремя разделами в каждой. В этом случае мы можем внести некоторые незначительные изменения в приложение и увеличить количество задач до числа входных разделов, что обеспечит максимальную пропускную способность за счет назначения каждому потоку выполнения одной задачи.

Для этого, как показано в листинге Г.5, создадим отдельный `KStream` для каждого входного топика, а затем каждый объект `KStream` передадим методу `buildStream`, который добавляет операции в топологию.

**Листинг Г.5.** Создание отдельного объекта `KStream` для каждого топика, чтобы максимизировать количество задач

```
StreamsBuilder builder = new StreamsBuilder();
KStream<String, String> streamA = builder.stream("topicA");
KStream<String, String> streamB = builder.stream("topicB");
```

```
buildStream(KStream<String, String> sourceStream) {
    return sourceStream.filter(..).mapValues(..);
}

buildStream(streamA).to("output");
buildStream(streamB).to("output");
```

Внеся это небольшое изменение, мы перешли от одной субтопологии к двум, и в результате теперь у нас семь задач вместо четырех, что позволит запустить отдельный поток выполнения для каждой задачи и получить максимальную пропускную способность.

У нас появился некоторый повторяющийся код, но мы постарались свести его к минимуму. Это дублирование кода можно рассматривать как небольшую плату за максимальную пропускную способность. Я представил этот случай, чтобы продемонстрировать связь между задачами и субтопологиями. Это не лучшая практика, но она стоит того, чтобы учесть такую возможность при рассмотрении факторов производительности для приложений Kafka Streams.

## Г.5. ОБРАБОТКА ЗАПИСЕЙ

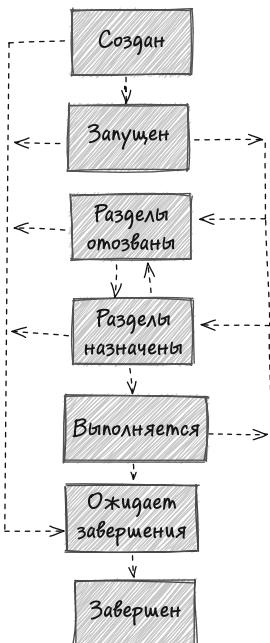
Вы узнали, что задача является единицей работы, и увидели, как запускать потоки выполнения в приложении Kafka Streams. А теперь в завершение обсуждения поговорим о некоторых деталях, касающихся работы задач Kafka Streams. Для этого рассмотрим жизненный цикл одного потока выполнения `StreamThread` в приложении Kafka Streams. Для целей обсуждения предположим, что мы уже создали и развернули свое приложение Kafka Streams и оно успешно работает.

Вызов метода `KafkaStreams.start()` в приложении запустит все настроенные экземпляры `StreamThread`.

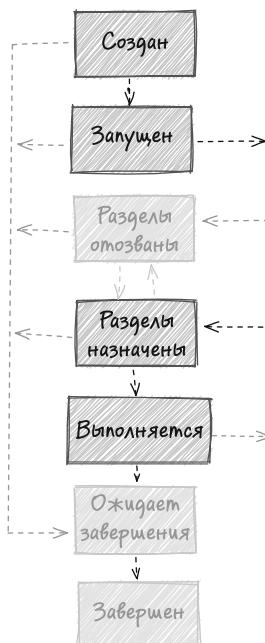
## ПРИМЕЧАНИЕ

Для ясности мы будем обсуждать только один `StreamThread` в одном приложении Kafka Streams, но на практике может быть несколько вариантов.

`StreamThread` имеет несколько состояний, в которые он может перейти, как показано на рис. Г.12. Здесь приведен полный график состояний, но в случае успешного начального запуска приложения поток выполнения `StreamThread` будет переходить из состояния в состояние, как показано на рис. Г.13.



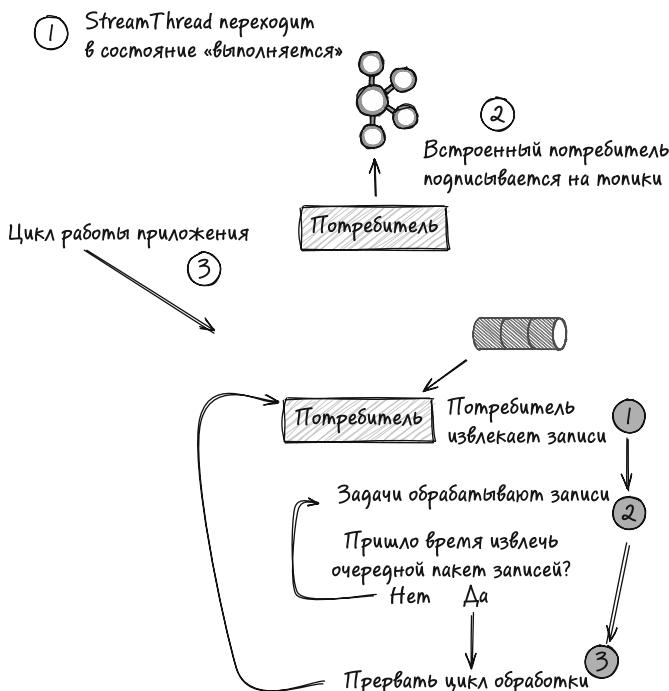
**Рис. Г.12.** StreamThread имеет несколько состояний, в которые он может переходить при обработке событий



**Рис. Г.13.** Как StreamThread переходит из состояния в состояние после успешного начального запуска

На этой иллюстрации видно, как поток выполнения запускается, а затем проходит через последовательность состояний «запущен», «разделы назначены» и, наконец, «выполняется». В последнем из этих состояний поток обрабатывает события. Переходим сразу к состоянию «выполняется», поскольку назначение разделов (задач) мы уже обсудили в предыдущем разделе. Я не собираюсь описывать каждую деталь жизненного цикла во время выполнения потока, а отмечу лишь основные моменты, которые следует знать при запуске приложения Kafka Streams.

Для справки мы будем следовать за диаграммой на рис. Г.14, иллюстрирующей происходящее с потоком во время выполнения им своей работы.



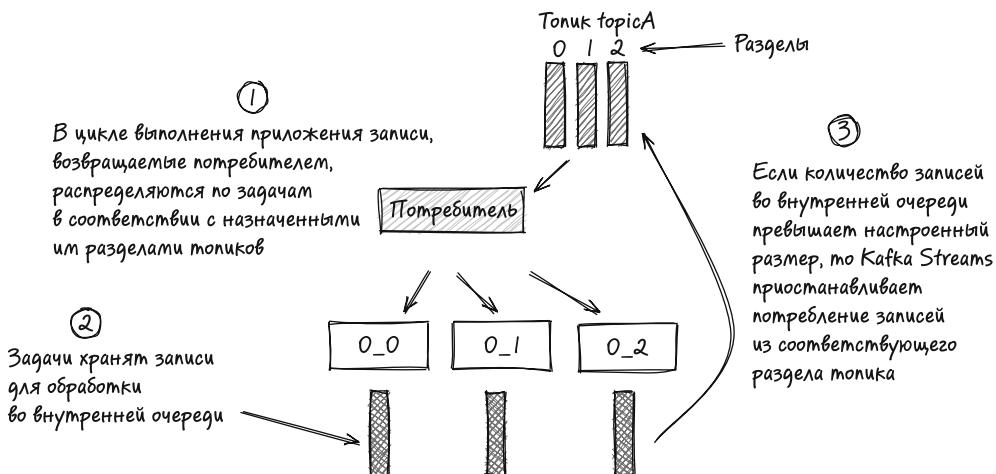
**Рис. Г.14.** Жизненный цикл StreamThread, выполняющегося в приложении

Итак, первый шаг — встроенный потребитель подписывается на топики, указанные в вызове метода `StreamBuilder#stream()`, независимо от того, указали ли вы полные имена или регулярное выражение.

Следующий шаг после подписки — извлечение записей для обработки в топологии. Записи, полученные вызовом метода `poll`, необходимо распределить по задачам. Процесс распределения включает извлечение задачи, соответствующей разделу топика. Затем извлеченные записи добавляются в очередь задачи для обработки. Здесь есть один момент, который мы должны обсудить, для чего воспользуемся диаграммой на рис. Г.15.

Как показано на рис. Г.15, у каждой задачи есть очередь для хранения записей, которые она будет обрабатывать. Если количество записей в очереди превышает настроенный размер (`StreamsConfig.BUFFERED_RECORDS_PER_PARTITION_CONFIG`), который по умолчанию равен 1000, то встроенный потребитель приостанавливает потребление из заданного раздела топика. Как только размер очереди опустится ниже порогового значения, потребление из этого раздела топика возобновится.

На этом этапе все задачи добавили записи в свои очереди, но перед началом обработки проверяется, нужно ли восстанавливать локальное состояние из топика журнализирования изменений. Мы предполагаем, что в нашем приложении нет задач с состоянием, поэтому восстановление не требуется. Процесс восстановления обсуждался в главе 7, где рассказывалось об операциях с состоянием в Kafka Streams.



**Рис. Г.15.** Приостановка раздела топика, если количество записей превышает настроенный максимальный размер буфера

После выяснения, что восстановление состояния не требуется, начинается обработка записей. Пользуясь рис. Г.16, обсудим, как Kafka Streams выбирает следующую запись для обработки в каждой задаче.



**Рис. Г.16.** Сначала выбираются записи из очереди с наименьшей отметкой времени в первой записи

Kafka Streams перебирает задачи и из очереди каждой выбирает записи с наименьшей отметкой времени. После того как все задачи обработают некоторые записи, выполняются служебные действия.

#### ПРИМЕЧАНИЕ

Восстановление и обработка не завершаются за один шаг. В ходе этого циклического процесса каждая задача продвинется вперед, а затем вернется к исходному состоянию. Соответственно, каждый шаг в этом цикле получит шанс на выполнение.

Первое служебное действие — выполнение всех требуемых пунктуаций. Пунктуация — это произвольное действие, которое можно запланировать с помощью API узлов-обработчиков (Processor API). Мы рассматривали пунктуации в главе 10, посвященной API узлов-обработчиков. После проверки пунктуаций определяется, пришло ли время для фиксации. Напомню, что под фиксацией подразумевается сохранение потребителем смещения (+1) последней успешно обработанной записи. Kafka Streams использует встроенный потребитель, который следует той же процедуре.

По умолчанию интервал фиксации в Kafka Streams составляет 30 секунд. То есть считается, что фиксация необходима, если прошло не менее 30 секунд (или другого настроенного интервала фиксации) с момента предыдущей фиксации. Для расчетов используется системное время из окружения приложения Kafka Streams.

На последнем шаге в конце цикла обработки потенциально могут быть выполнены еще два действия: корректировка количества записей, которые будет обрабатывать каждая задача, и/или проверка, пришло ли время выйти из цикла обработки, вернуться к потребителю и запросить новые записи.

Конечно, важно обработать как можно больше записей, но не менее важно обеспечить своевременное завершение цикла обработки для вызова метода `poll` и опроса входных топиков. В противном случае приложение Kafka Streams будет исключено из группы потребителей, что приведет к перебалансировке, а затем к еще одной перебалансировке при повторном присоединении к группе. Рассмотрим иллюстрацию на рис. Г.17, демонстрирующую проверки, которые Kafka Streams выполняет для максимизации обработки и обеспечения бесперебойной работы.

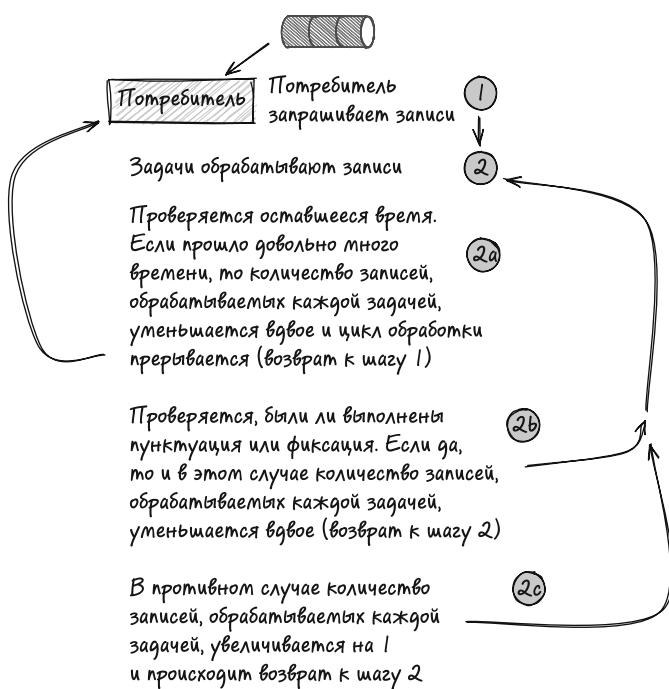


Рис. Г.17. В конце цикла обработки Kafka Streams выполняет некоторые проверки

Kafka Streams проверяет, сколько времени прошло с момента последнего вызова `poll`, и если прошло времени больше половины настроенного максимального интервала опроса, то количество записей, обрабатываемых каждой задачей, уменьшается на половину, после чего цикл обработки прерывается. Если времени для продолжения цикла обработки достаточно, но выполнялись пунктуации или фиксация, то в этот момент количество записей, обрабатываемых каждой задачей, также уменьшается вдвое. В противном случае максимальное количество записей, которое может обработать каждая задача, увеличивается на 1.

И прежде, чем закончить это приложение, упомяну еще один важный момент. Несмотря на то что Kafka Streams принимает все меры, чтобы обеспечить бесперебойную работу, разработчик все равно несет ответственность за то, чтобы узлы-обработчики выполняли свою работу максимально быстро, потому что перебалансировки, возникающие из-за пропуска вызова `poll`, отрицательно влияют на пропускную способность.

*Билл Беджек*

**Kafka Streams в действии. Приложения и микросервисы,  
управляемые событиями**

2-е издание

