ГЛАВА 2

Начало работы с Kafka Streams

Kafka Streams — это легкая, но мощная библиотека Java для обогащения, преобразования и обработки потоков данных в реальном времени. В этой главе я расскажу о ней в общих чертах. Мой рассказ можно сравнить с первым свиданием, на котором вы немного узнаете об истории библиотеки Kafka Streams и познакомитесь с ее функционалом.

К концу этого свидания, ну-у... то есть этой *главы*, вы будете владеть следующей информацией.

- Место библиотеки Kafka Streams в экосистеме Kafka.
- Зачем была создана библиотека Kafka Streams.
- Каким функционалом и рабочими характеристиками она обладает.
- Кому она нужна.
- Как она выглядит относительно других решений для обработки потоковых данных.
- Как создать и запустить базовое приложение Kafka Streams.

Впрочем, довольно предисловий. Начнем наше метафорическое свидание с простого вопроса: *где вы живете* (в экосистеме Kafka)?

Экосистема Kafka

Библиотека Kafka Streams «живет» в группе технологий, совокупность которых называется *экосистемой Kafka*. В главе 1 вы узнали, что в основе Apache Kafka лежит распределенный журнал, обновляемый только путем добавления в него новых записей. Мы можем добавлять в него сообщения и читать их оттуда. В кодовую базу Kafka входят также API-интерфейсы для взаимодействия с этим

журналом (который разделен на категории, называемые *темами*). Три API экосистемы Kafka, связанные с перемещением данных, представлены в табл. 2.1.

Таблица 2.1. API для перемещения данных в брокер Kafka и из него

API	Взаимодействие с темой	Примеры
API производителей	*Запись* сообщений в темы Kafka	• Filebeat. • Rsyslog. • Пользовательские варианты производителей
API потребителей	*Чтение* сообщение из тем Kafka	• Logstash. • Kafkacat. • Пользовательские варианты потребителей
API коннекторов	*Соединение* внешних хранилищ данных, API и файловых систем с темами Kafka. Отвечает как за *чтение* из тем (коннекторы-приемники), так и за *запись* в темы (коннекторы-источники)	• Коннектор JDBC-источника. • Коннектор Elasticsearch-приемника. • Пользовательские варианты коннекторов

Перемещение данных через брокер сообщений Kafka, безусловно, важно для создания конвейеров, но немало задач требует мгновенной *реакции* на поступающую информацию. Речь идет о так называемой *потоковой обработке* (stream processing). Создавать приложения потоковой обработки с помощью Kafka можно разными способами. Но для начала я хочу вам рассказать, как такие приложения создавались до появления Kafka Streams и как в экосистеме Kafka, наряду с другими API, появилась выделенная библиотека потоковой обработки.

До появления Kafka Streams

Пока не появилась Kafka Streams, в экосистеме Kafka была пустота[1]. Это была совсем не та пустота, которую можно почувствовать во время утренней медитации и потом ощутить себя освеженными и просветленными. Нет, я имею в виду пробел, сильно затруднявший создание приложений для обработки потоковой информации. На уровне библиотек обработка данных в темах Kafka просто не поддерживалась.

[1] Здесь я имею в виду официальную экосистему, включающую в себя все компоненты, поддерживаемые в рамках проекта Apache Kafka.

В ранних версиях экосистемы Kafka существовало два основных способа создания приложений для обработки потоков. Вы могли:

- напрямую использовать API потребителей и API производителей;
- обратиться к другим фреймворкам, реализующим обработку потоков (например, Apache Spark Streaming или Apache Flink).

Имея API потребителей и производителей, можно напрямую читать из потока и записывать в поток, а также реализовывать любые логические схемы обработки данных. Главное — знать какой-то язык программирования (Python, Java, Go, C/C ++, Node.js и т. п.) и быть готовыми с нуля писать много кода. Это очень простые API. Они лишены множества базисных элементов, которые позволили бы им считаться API для обработки потоков. В частности, в них отсутствуют:

- локальное и отказоустойчивое состояние[1];
- многие операторы для преобразования потоков данных;
- более совершенные варианты представления потоков[2];
- продуманное управление временем[3].

Отсутствие перечисленного затрудняет выполнение любых нестандартных действий, таких как агрегирование записей, объединение событий, произошедших в один период времени, или узкоспециализированные запросы к потоку. Абстракций, которые помогли бы в решении таких задач, в API потребителей и производителей нет, так что весь код приходится писать самостоятельно.

Второй вариант, то есть привлечение сторонних фреймворков, например Apache Spark или Apache Flink, привносит в систему ненужную сложность. Подробно о недостатках этого подхода мы поговорим в разделе *«Сравнение с другими системами»*, а пока я замечу только, что решение, которое предоставляет функционал потоковой обработки, одновременно создавая дополнительную нагрузку на обрабатывающий кластер, нельзя считать простым и оптимальным. Кроме того, нам требуется лучшая интеграция с Kafka, особенно при работе с промежуточными представлениями данных за пределами тем, играющих роли источника и приемника.

---

[1] Один из первых разработчиков Apache Kafka Джей Крепс (Jay Kreps) подробно обсуждал это в блоге O'Reilly еще в 2014 году (https://oreil.ly/vzRH-).

[2] Речь идет об агрегированных потоках/таблицах, о которых я расскажу чуть попозже.

[3] Вопросам, связанным со временем, я посвятил целую главу, но все равно рекомендую посмотреть отличную презентацию Маттиаса Джей Сакса (Matthias J. Sax) с конференции Kafka Summit в 2019 году (https://oreil.ly/wr123).

К счастью, сообщество Kafka осознало, насколько в экосистеме Kafka не хватает API потоковой обработки, и решило создать его[1].

Рождение Kafka Streams

В 2016 году после появления первой версии Kafka Streams (она же *Streams API*) экосистема Kafka навсегда изменилась. Многочисленные приложения обработки потоков, в которых многое предлагалось делать вручную, уступили место более продвинутым приложениям, использующим разработанные сообществом шаблоны и абстракции для обработки потоков данных в реальном времени.

В отличие от API производителей, потребителей и коннекторов, библиотека Kafka Streams призвана помочь не только в *перемещении* данных через брокер Kafka, но и в *обработке* их потоков в режиме реального времени[2]. Благодаря богатому набору операторов и примитивов обработки потоков упрощается преобразование потоковых данных и при необходимости запись их новых представлений обратно в Kafka (когда преобразованные или обогащенные события нужно сделать доступными для следующих систем в конвейере).

Рисунок 2.1 демонстрирует, какое место в экосистеме Kafka занимают упоминавшиеся выше API. Библиотека Kafka Streams функционирует в ней на уровне обработки потоков.

Из диаграммы видно, что библиотека Kafka Streams работает на сильно нагруженном уровне экосистемы Kafka, в месте, куда сходятся данные из многих источников. Именно здесь реализуются сложные варианты *обогащения*, *преобразования* и *обработки* данных. Именно на этом уровне до появления Kafka Streams (когда приходилось пользоваться API потребителей/производителей) кропотливо создавались собственные абстракции обработки потоков или преодолевались сложности, возникшие из-за применения стороннего фреймворка. Поэтому давайте знакомиться с функционалом Kafka Streams, позволяющим легко и эффективно работать на этом уровне.

---

[1] Большое спасибо Гочжен Ванну (Guozhang Wang), который отправил в Kafka Improvement Proposal предложение, послужившее основанием для создания Kafka Streams. См. https://oreil.ly/l2wbc.

[2] После того как к компоненту Kafka Connect добавили функцию преобразования отдельных сообщений, он тоже стал пригоден для обработки событий, но все равно в плане функционала он не выдерживает никакого сравнения с библиотекой Kafka Streams.

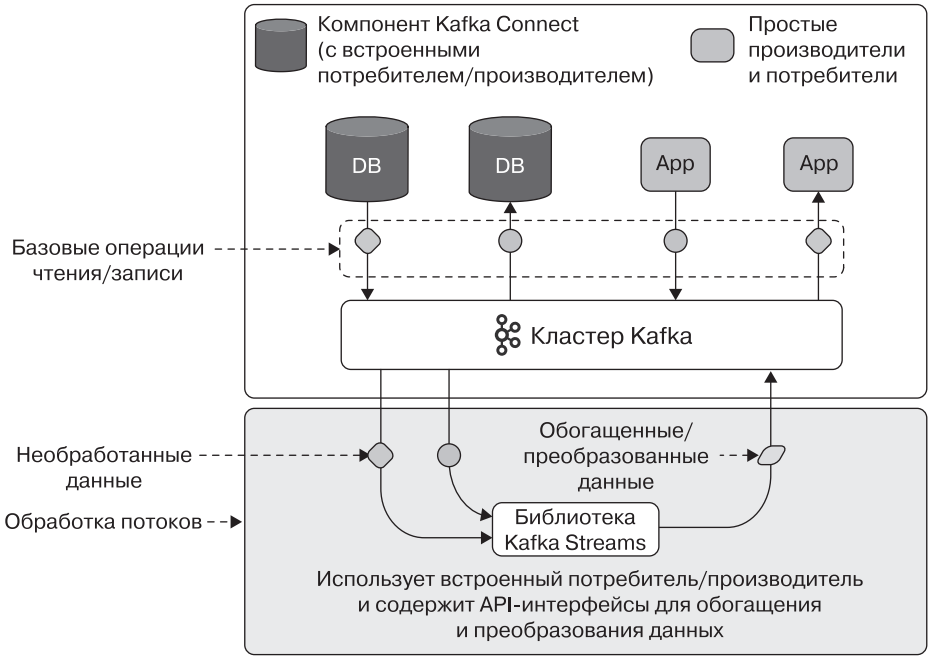

Рис. 2.1. Библиотека Kafka Streams — «мозг» экосистемы Kafka, потребляющий записи из потока событий, обрабатывающий данные и при необходимости записывающий расширенные или преобразованные записи обратно в Kafka

Обзор функционала

Благодаря своему огромному функционалу библиотека Kafka Streams — отличный выбор для современных приложений потоковой обработки. Вот основные элементы этого функционала.

- Высокоуровневый DSL, напоминающий API для работы с потоками на языке Java. Он обеспечивает плавный и функциональный подход к обработке потоков данных и отличается легкостью изучения и применения.
- Низкоуровневый Processor API, предоставляющий разработчикам детальный контроль над происходящим.
- Удобные абстракции для моделирования данных в виде потоков или таблиц.
- Возможность объединения потоков и таблиц, предназначенная для преобразования и обогащения данных.

- Операторы и утилиты для создания приложений потоковой обработки как без сохранения, так и с сохранением состояния.
- Поддержка операций, привязанных ко времени, в том числе оконных и периодических функций.
- Простая установка. Так как Kafka Streams — это просто библиотека, ее можно добавить в любое приложение Java[1].
- Масштабируемость, надежность, удобство сопровождения.

В процессе знакомства с этим функционалом вы быстро поймете, почему эта библиотека так широко используется и так любима. Как высокоуровневый DSL, так и низкоуровневый Processor API не только легки в освоении, но и чрезвычайно эффективны. Даже сложные задачи потоковой обработки (например, соединение движущихся потоков данных) решаются с помощью небольшого кода, что делает процесс разработки по-настоящему легким.

Последний пункт приведенного выше списка касается долговременной стабильности приложений для обработки потоковых данных. В конце концов, на этапе изучения многие технологии вызывают искренний интерес, но на самом деле важно, подходят ли они для решения реальных, как правило, куда более сложных, чем учебные, задач. Поэтому прежде чем погрузиться в освоение Kafka Streams, имеет смысл оценить, насколько эта технология жизнеспособна. Для этого рассмотрим ее эксплуатационные характеристики.

Эксплуатационные характеристики

В прекрасной книге Мартина Клеппмана «Высоконагруженные приложения»[2] для систем обработки данных выделены три важные метрики.

- Масштабируемость.
- Надежность.
- Удобство сопровождения.

Именно на их основе я собираюсь оценивать библиотеку Kafka Streams. Определим каждую из метрик и узнаем, насколько рассматриваемое средство отвечает заданным ими требованиям.

---

[1] Библиотека Kafka Streams работает и с другими языками, созданными для JVM, в том числе с языками Scala и Kotlin. Однако все примеры в книге написаны только на Java.

[2] *Клеппман М.* Высоконагруженные приложения. Программирование, масштабирование, поддержка. — Питер, 2018.

Масштабируемость

Система считается *масштабируемой* (scalable), если она остается работоспособной при увеличении нагрузки. В главе 1 вы узнали, что масштабирование тем Kafka сводится к добавлению дополнительных разделов и при необходимости дополнительных брокеров (последнее необходимо при выходе темы за пределы емкости имеющегося кластера Kafka).

В Kafka Streams единицей работы тоже считается одна тема-раздел, и Kafka автоматически распределяет работу между группами потребителей[1].

Это имеет два важных последствия.

- Поскольку расширение тем происходит путем добавления разделов, масштабировать объем работы, которую выполняет приложение Kafka Streams, можно, увеличивая количество разделов в исходных темах[2].
- Благодаря группам потребителей общий объем работы, выполняемой приложением Kafka Streams, может быть распределен между несколькими взаимодействующими экземплярами приложения.

Немного проиллюстрирую второй пункт списка. Развертывание приложения Kafka Streams почти всегда означает развертывание нескольких экземпляров, каждый из которых будет отвечать за свой фрагмент работы. Например, для темы из 32 разделов можно развернуть четыре экземпляра приложения. В этом случае на каждый из них придется восемь разделов ($4 \times 8 = 32$). Если же развернуть 16 экземпляров приложения, каждый из них будет обрабатывать два раздела ($16 \times 2 = 32$).

Итак, способность Kafka Streams справляться с повышенной нагрузкой (то есть *масштабируемость*) достигается добавлением дополнительных разделов (единиц работы) и экземпляров приложений (работников).

Кроме того, Kafka Streams *адаптивна*, что позволяет без особых усилий (хотя и вручную) уменьшать и увеличивать число экземпляров приложения. Впрочем, горизонтальное масштабирование равно количеству задач, созданных для конкретной топологии. Более подробно этот вопрос будет рассматриваться в подразделе «Задачи и потоки выполнения» этой главы.

---

[1] С одной темой может работать одновременно несколько групп потребителей, при этом каждая группа обрабатывает сообщения независимо от других.

[2] Хотя разделы можно добавлять в существующие темы, рекомендуется создать тему с желаемым количеством разделов и перенести туда все существующие задачи.

Надежность

Такая характеристика систем данных, как надежность, важна не только для обслуживающего персонала (никто не хочет, чтобы его будили в три часа ночи из-за неисправности), но и для клиентов (которых не устраивает выход системы из строя и уж тем более потеря или повреждение данных). В Kafka Streams есть различный устойчивый к сбоям функционал[1], наиболее очевидный вариант которого был описан в разделе «Группы потребителей» предыдущей главы.

Если один экземпляр приложения Kafka Streams выходит из строя (например, по причине аппаратного сбоя), брокер Kafka автоматически перераспределяет его задачи между остальными экземплярами. После устранения сбоя (или в более современных архитектурах, использующих систему оркестрации, таких как Kubernetes, — после перемещения приложения на работоспособный узел) Kafka возобновляет работу. Именно эта способность корректно обрабатывать ошибки обеспечивает *надежность* Kafka Streams.

Удобство сопровождения

> Общеизвестно, что большая часть стоимости программного обеспечения связана не с его разработкой, а с его текущим обслуживанием — исправлением ошибок, поддержанием работоспособности систем, анализом сбоев...
>
> *Мартин Клеппман*

Поскольку Kafka Streams — это библиотека Java, выявление неисправностей и исправление ошибок не должны вызывать затруднения. Мы работаем с автономными приложениями, а шаблоны как для устранения неполадок, так и для мониторинга приложений Java хорошо известны. Скорее всего, вам уже приходилось ими пользоваться. Это шаблоны ведения и анализа журнала приложения, сбора метрик приложения и JVM, профилирование и трассировка и т. п.

Кроме того, благодаря лаконичности и наглядности API Kafka Streams, обслуживание на уровне кода занимает меньше времени, чем в случае более сложных библиотек. Это достаточно простая процедура, доступная даже новичкам. Даже если приложение Kafka Streams никто не трогал много месяцев, скорее всего, вам не потребуется много времени, чтобы понять его код. По тем же причинам специалисты по сопровождению новых проектов обычно быстро осваивают приложения Kafka Streams, что делает сам процесс сопровождения еще более удобным.

---

[1] В эту категорию попадает и функционал, характерный для приложений с отслеживанием состояния, о которых пойдет речь в главе 4.

Сравнение с другими системами

К этому моменту свидания с Kafka Streams вы, скорее всего, уже ответили себе на вопрос, стоит ли начинать с этой библиотекой долгосрочные отношения. Тем не менее посмотреть на существующие альтернативы все-таки стоит.

И в самом деле, как оценить, насколько хороша технология, без ее сравнения с конкурентами? Поэтому сопоставим Kafka Streams с некоторыми популярными технологиями в области обработки потоковых данных[1]. И начнем рассмотрение с модели развертывания.

Модель развертывания

Фреймворк Apache Flink и расширение Spark Streaming для фреймворка Apache Spark требуют для программы потоковой обработки выделенного кластера. Это увеличивает как сложность системы, так и затраты вычислительных ресурсов. Даже опытные инженеры из солидных компаний признают, что вычислительный кластер обходится недешево. Например, Нитин Шарма из компании Netflix в интервью рассказывал, что, когда они создавали на базе фреймворка Apache Flink приложение и кластер, адаптация к его нюансам заняла около шести месяцев.

С другой стороны, Kafka Streams реализована как *библиотека* Java. Диспетчер кластера в этом случае не нужен, достаточно добавить в приложение Java зависимость от этой библиотеки. Создаваемые таким способом приложения для обработки потоковой информации представляют собой отдельные программы, что дает большую свободу при мониторинге, упаковке и развертывании кода. Например, на платформе Mailchimp приложения Kafka Streams развертываются с помощью тех же шаблонов и инструментов, что и другие внутренние Java-приложения. Фактически все это дает нам такое огромное преимущество, как возможность немедленной интеграции приложений Kafka Streams в любую систему.

Теперь сравним модель обработки данных у Kafka Streams и ее конкурентов.

Модель обработки

Еще одним ключевым отличием Kafka Streams от конкурентов стала *модель обработки по отдельным событиям* (event-at-a-time processing). Каждое событие обрабатывается в момент его поступления. Это считается настоящей

[1] Провести полное сравнение невозможно из-за постоянного появления новых решений для обработки потоковых данных, поэтому я рассмотрю только самые популярные и давно существующие варианты систем-аналогов, доступные на момент написания этого текста.

потоковой передачей и обеспечивает меньшую задержку, чем альтернативный подход, в котором поток интерпретируется как непрерывная последовательность микропакетов данных. Речь идет о так называемом *микропакетировании* (microbatching). Пакеты создаются через регулярные интервалы времени (например, каждые 500 миллисекунд) и отправляются на обработку. Разницу между двумя подходами иллюстрирует рис. 2.2.

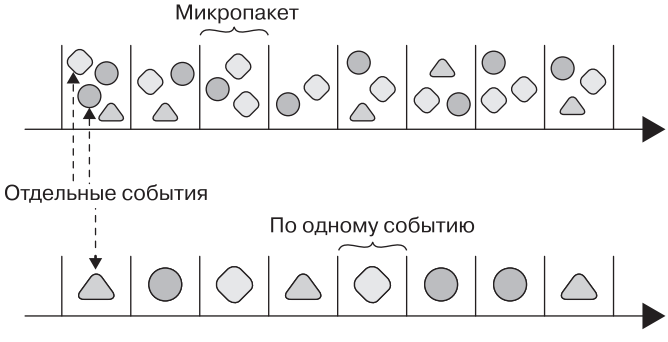

Рис. 2.2. При микропакетировании записи через регулярные интервалы времени группируются в небольшие партии и передаются обработчикам; обработка каждого события в момент его наступления позволяет не ждать, пока сформируется пакет

В фреймворках на базе микропакетной архитектуры для увеличения *пропускной способности* часто начинают увеличивать *задержку*. В Kafka Streams за счет распределения данных по разделам можно добиться чрезвычайно низкой задержки при сохранении высокой пропускной способности.

Наконец, разберем архитектуру обработки данных в Kafka Streams и посмотрим, чем ее ориентация на потоковую передачу отличается от других систем.

Каппа-архитектура

Выбирая между Kafka Streams и альтернативными решениями, важно учитывать, требует ли система, которую вы строите, поддержки как пакетной, так и потоковой обработки. На момент написания этого текста Kafka Streams фокусировалась исключительно на сценариях использования потоковой передачи[1] (так называемая каппа-архитектура), в то время как в фреймворках Apache Flink и Apache Spark поддерживается как пакетная, так и потоковая

[1] Имеется устаревшее, но еще открытое предложение добавить в Kafka Streams поддержку пакетной обработки (https://oreil.ly/v3DbO).

обработка (это *лямбда-архитектура*). К сожалению, архитектуры, в которых поддерживаются оба варианта, не лишены недостатков. О слабых местах гибридной системы почти за два года до появления Kafka Streams высказывался Джей Крепс (https://oreil.ly/RwkNi):

> Операционная нагрузка по запуску и отладке двух систем очень высока. И любая новая абстракция может предоставить только функционал, который одновременно поддерживается обеими системами.

Эти проблемы не помешали росту популярности проекта Apache Beam, который представляет собой унифицированную модель программирования для пакетной и потоковой обработки. Впрочем, Apache Beam, в отличие от Apache Flink, напрямую сравнить с Kafka Streams не получится, так как он находится на другом уровне абстракции. Большая часть работы отдается на откуп механизму выполнения, в качестве которого может выступать, например, Apache Flink или Apache Spark. Поэтому при сравнении с Kafka Streams нужно рассматривать не только Beam API, но и механизм выполнения, которым он пользуется.

Кроме того, конвейерам на базе Apache Beam не хватает важного функционала, который присутствует в Kafka Streams. Вот что пишет об этом Роберт Йокота (https://oreil.ly/24zG9), который создал экспериментальную программу Kafka Streams Beam Runner (https://oreil.ly/24zG9) и поддерживает несколько инновационных проектов в экосистеме Kafka[1].

Приведем возможный перечень различий между двумя системами.

- Платформа Kafka Streams позволяет обрабатывать *как потоки, так и отношения*.
- Платформа Apache Beam предназначена только для работы с *потоками*.

При этом платформа потоково-реляционной обработки имеет перечисленные ниже дополнительные возможности.

- Отношения (или таблицы) — полноправные сущности, так как каждое из них уникально.
- Отношения можно преобразовать в другие отношения.
- Отношения допускают произвольные запросы.

[1] К этим проектам относятся в числе прочих поддерживаемая брокером Kafka реляционная база данных KarelDB, библиотека для анализа графов на базе Kafka Streams и многое другое. См. https://yokota.blog.

Все эти особенности будут демонстрироваться в следующих главах, пока же я только замечу, что многое из наиболее мощного функционала Kafka Streams (включая запросы состояния потока) недоступно в Apache Beam и в других универсальных фреймворках[1]. Кроме того, каппа-архитектура предлагает более простой и более специализированный подход к работе с потоками данных, позволяющий усовершенствовать процесс разработки и упростить эксплуатацию и обслуживание программного обеспечения. Так что, если сценарии использования не требуют пакетной обработки, гибридная система только внесет ненужную сложность.

Надеюсь, этот небольшой обзор дал вам представление о том, чем Kafka Streams отличается от конкурентов. Теперь давайте посмотрим, как применяется эта библиотека.

Сценарии использования

Библиотека Kafka Streams оптимизирована для быстрой и эффективной обработки бесконечных наборов данных. Следовательно, она отлично подходит для областей, в которых важна немедленная обработка информации и низкая задержка. Например:

- обработка финансовых данных (компания Flipkart https://oreil.ly/dAcbY), мониторинг купли-продажи, обнаружение мошенничества;
- алгоритмическая торговля;
- мониторинг фондового рынка/криптобиржи;
- отслеживание и пополнение ресурсов в реальном времени (компания Walmart https://oreil.ly/VoF76);
- бронирование мероприятий, выбор мест (компания Ticketmaster https://oreil.ly/V4t1h);
- отслеживание доставки электронной почты (платформа Mailchimp);

[1] На момент написания этого текста в Apache Flink появилась бета-версия запросов состояния. При этом про соответствующий API в официальной документации сообщалось следующее: «Клиентский интерфейс для запросов состояния в настоящее время дорабатывается, и поэтому его стабильность не гарантирована. Вероятно, в следующих версиях Flink в него будут внесены критические изменения». Так что тщательно продуманный и работоспособный API для запросов состояния пока имеется только в Kafka Streams.