

**129.53**

Рейтинг

Слёрм

Учебный центр для тех, кто работает в IT

[Подписаться](#)**Polina_Averina** 7 апр 2021 в 09:18

Apache Kafka: основы технологии



9 мин



408K

Блог компании Слёрм, Системное администрирование*, Программирование*, IT-инфраструктура*, Apache*

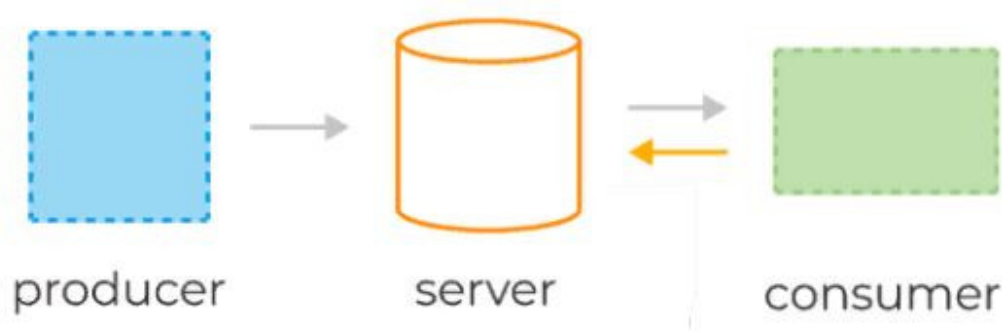
У Kafka есть множество способов применения, и у каждого способа есть свои особенности. В этой статье разберём, чем Kafka отличается от популярных систем обмена сообщениями; рассмотрим, как Kafka хранит данные и обеспечивает гарантию сохранности; поймём, как записываются и читаются данные.

Статья подготовлена на основе открытого занятия из видеокурса по Apache Kafka. Авторы — Анатолий Солдатов, Lead Engineer в Авито, и Александр Миронов, Infrastructure Engineer в Stripe. Базовые темы курса доступны на Youtube .

Для первого погружения в технологию сравним Kafka и классические сервисы очередей, такие как RabbitMQ и Amazon SQS.

Системы очередей обычно состоят из трёх базовых компонентов:

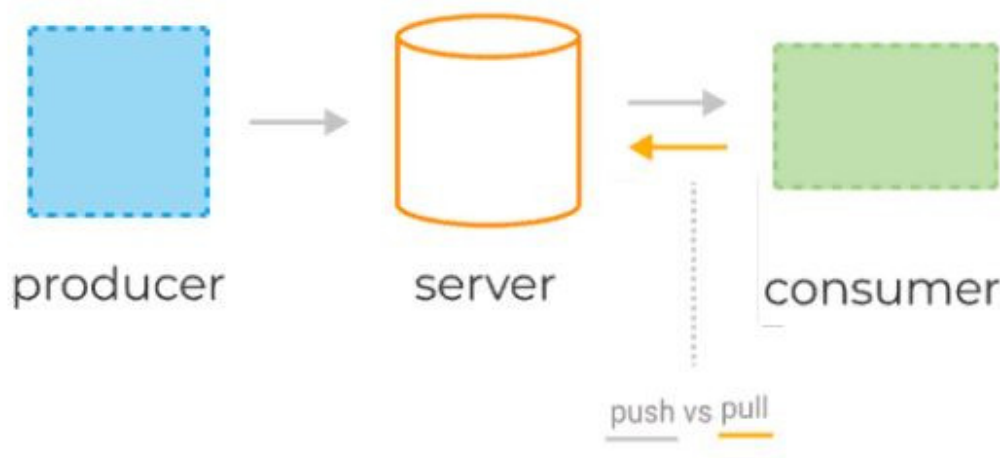
- 1) сервер,
- 2) продюсеры, которые отправляют сообщения в некую именованную очередь, заранее сконфигурированную администратором на сервере,
- 3) консьюмеры, которые считывают те же самые сообщения по мере их появления.



Базовые компоненты классической системы очередей

В веб-приложениях очереди часто используются для отложенной обработки событий или в качестве временного буфера между другими сервисами, тем самым защищая их от всплесков нагрузки.

Консьюмеры получают данные с сервера, используя две разные модели запросов: pull или push.

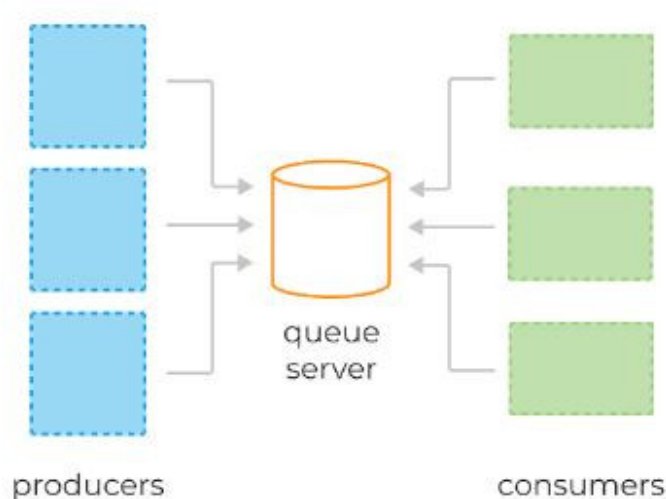


pull-модель — консьюмеры сами отправляют запрос раз в n секунд на сервер для получения новой порции сообщений. При таком подходе клиенты могут эффективно контролировать

собственную нагрузку. Кроме того, pull-модель позволяет группировать сообщения в батчи, таким образом достигая лучшей пропускной способности. К минусам модели можно отнести потенциальную разбалансированность нагрузки между разными консьюмерами, а также более высокую задержку обработки данных.

push-модель — сервер делает запрос к клиенту, посылая ему новую порцию данных. По такой модели, например, работает RabbitMQ. Она снижает задержку обработки сообщений и позволяет эффективно балансировать распределение сообщений по консьюмерам. Но для предотвращения перегрузки консьюмеров в случае с RabbitMQ клиентам приходится использовать функционал QS, выставляя лимиты.

Как правило, приложение пишет и читает из очереди с помощью нескольких инстансов продюсеров и консьюмеров. Это позволяет эффективно распределить нагрузку.



Типичный жизненный цикл сообщений в системах очередей:

1. Продюсер отправляет сообщение на сервер.
2. Консьюмер фетчит (от англ. fetch — принести) сообщение и его уникальный идентификатор сервера.
3. Сервер помечает сообщение как in-flight. Сообщения в таком состоянии всё ещё хранятся на сервере, но временно не доставляются другим консьюмерам. Таймаут этого состояния контролируется специальной настройкой.
4. Консьюмер обрабатывает сообщение, следуя бизнес-логике. Затем отправляет ask или nack-запрос обратно на сервер, используя уникальный идентификатор, полученный ранее — тем самым либо подтверждая успешную обработку сообщения, либо сигнализируя об ошибке.
5. В случае успеха сообщение удаляется с сервера навсегда. В случае ошибки или таймаута состояния in-flight сообщение доставляется консьюмеру для повторной обработки.



Типичный жизненный цикл сообщений в системах очередей

С базовыми принципами работы очередей разобрались, теперь перейдём к Kafka. Рассмотрим её фундаментальные отличия.

Как и сервисы обработки очередей, Kafka условно состоит из трёх компонентов:

- 1) сервер (по-другому ещё называется брокер),
- 2) продюсеры — они отправляют сообщения брокеру,
- 3) консьюмеры — считывают эти сообщения, используя модель pull.



Базовые компоненты Kafka

Пожалуй, фундаментальное отличие Kafka от очередей состоит в том, как сообщения хранятся на брокере и как потребляются консьюмерами.

- Сообщения в Kafka **не удаляются** брокерами по мере их обработки консьюмерами — данные в Kafka могут храниться днями, неделями, годами.
- Благодаря этому одно и то же сообщение может быть обработано **сколько угодно раз** разными консьюмерами и в разных контекстах.

В этом кроется главная мощь и главное отличие Kafka от традиционных систем обмена сообщениями.

Теперь давайте посмотрим, как Kafka и системы очередей решают одну и ту же задачу. Начнём с системы очередей.

Представим, что есть некий сайт, на котором происходит регистрация пользователя. Для каждой регистрации мы должны:

- 1) отправить письмо пользователю,
- 2) пересчитать дневную статистику регистраций.

В случае с RabbitMQ или Amazon SQS функционал может помочь нам доставить сообщения всем сервисам одновременно. Но при необходимости подключения нового сервиса придётся конфигурировать новую очередь.



Kafka упрощает задачу. Достаточно послать сообщения всего один раз, а консьюмеры сервиса отправки сообщений и консьюмеры статистики сами считают его по мере необходимости.



Kafka также позволяет тривиально подключать новые сервисы к стриму регистрации. Например, сервис архивирования всех регистраций в S3 для последующей обработки с помощью Spark или Redshift можно добавить без дополнительного конфигурирования сервера или создания дополнительных очередей.

Кроме того, раз Kafka не удаляет данные после обработки консьюмерами, эти данные могут обрабатываться заново, как бы отматывая время назад сколько угодно раз. Это оказывается невероятно полезно для восстановления после сбоев и, например, верификации кода новых консьюмеров. В случае с RabbitMQ пришлось бы записывать все данные заново, при этом, скорее всего, в отдельную очередь, чтобы не сломать уже имеющихся клиентов.

Структура данных

Наверняка возникает вопрос: «Раз сообщения не удаляются, то как тогда гарантировать, что консьюмер не будет читать одни и те же сообщения (например, при перезапуске)?».

Для ответа на этот вопрос разберёмся, какова внутренняя структура Kafka и как в ней хранятся сообщения.

Каждое сообщение (event или message) в Kafka состоит из ключа, значения, таймстампа и опционального набора метаданных (так называемых хедеров).

Например:

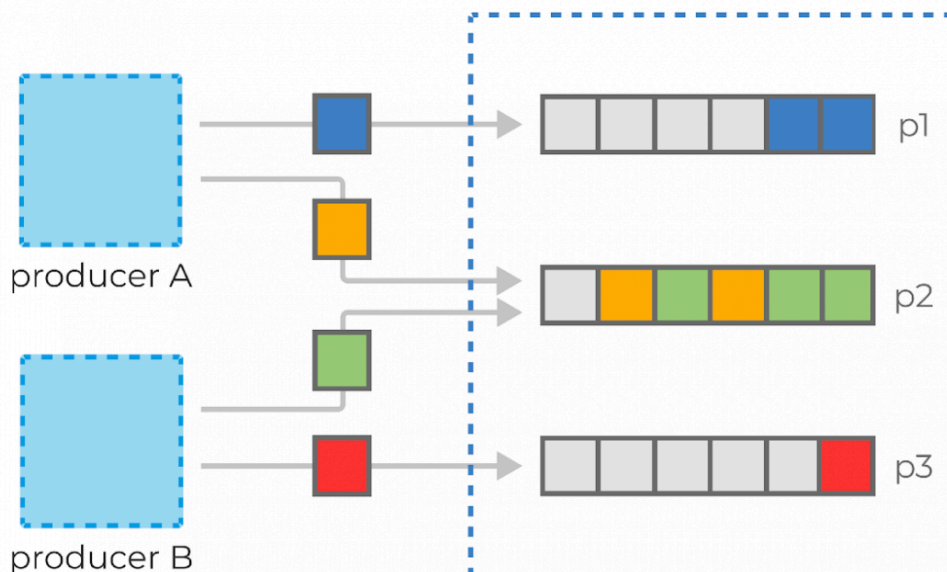


Сообщения в Kafka организованы и хранятся в именованных топиках (Topics), каждый топик состоит из одной и более партиций (Partition), распределённых между брокерами внутри одного кластера. Подобная распределённость важна для горизонтального масштабирования кластера, так как она позволяет клиентам писать и читать сообщения с нескольких брокеров одновременно.

Когда новое сообщение добавляется в топик, на самом деле оно записывается в одну из партиций этого топика. Сообщения с одинаковыми ключами всегда записываются в одну и ту же партицию, тем самым гарантируя очередность или порядок записи и чтения.

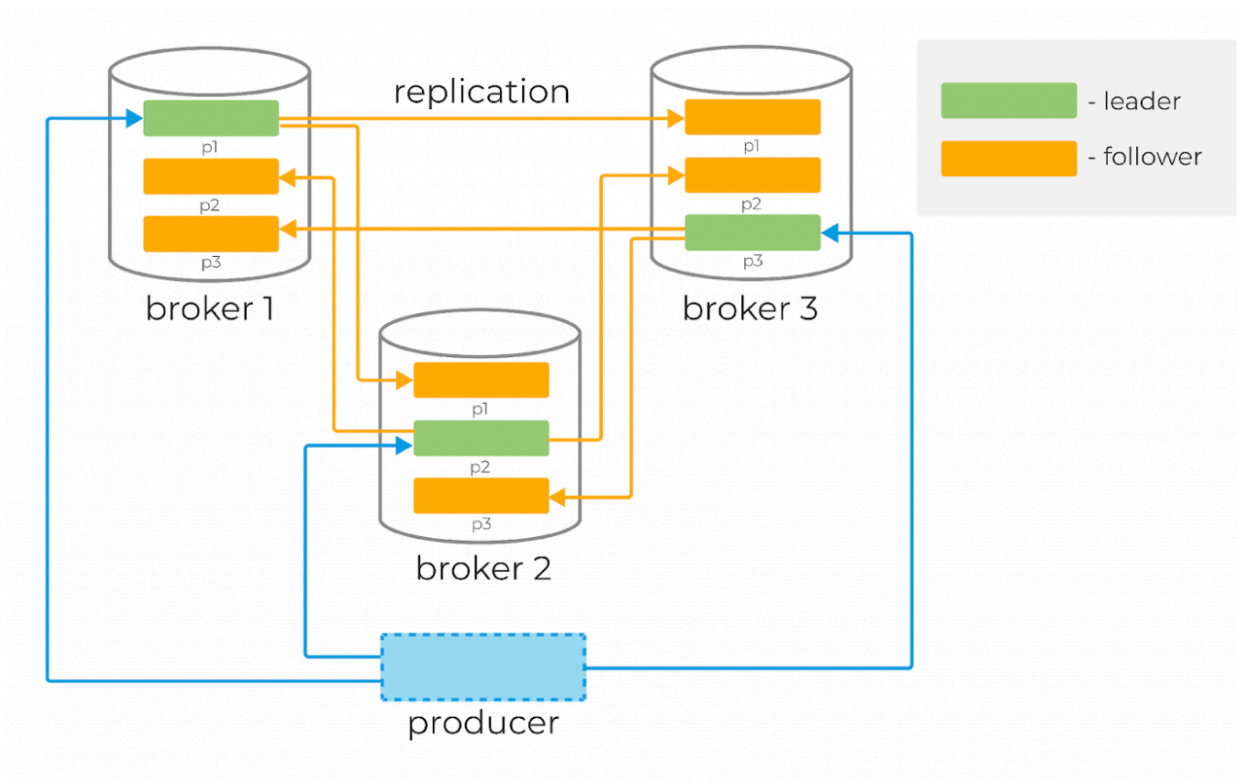
Для гарантии сохранности данных каждая партиция в Kafka может быть реплицирована n раз, где n — replication factor. Таким образом гарантируется наличие нескольких копий сообщения, хранящихся на разных брокерах.

registrations topic



У каждой партии есть «лидер» (Leader) — брокер, который работает с клиентами. Именно лидер работает с продюсерами и в общем случае отдаёт сообщения консьюмерам. К лидеру осуществляют запросы фолловеры (Follower) — брокеры, которые хранят реплику всех данных партий. Сообщения всегда отправляются лидеру и, в общем случае, читаются с лидера.

Чтобы понять, кто является лидером партии, перед записью и чтением клиенты делают запрос метаданных от брокера. Причём они могут подключаться к любому брокеру в кластере.



Основная структура данных в Kafka — это распределённый, реплицируемый лог. Каждая партия — это и есть тот самый реплицируемый лог, который хранится на диске. Каждое новое сообщение, отправленное продюсером в партию, сохраняется в «голову» этого лога и получает

свой уникальный, монотонно возрастающий offset (64-битное число, которое назначается самим брокером).

Как мы уже выяснили, сообщения не удаляются из лога после передачи консьюмерам и могут быть вычитаны сколько угодно раз.

Время гарантированного хранения данных на брокере можно контролировать с помощью специальных настроек. Длительность хранения сообщений при этом не влияет на общую производительность системы. Поэтому совершенно нормально хранить сообщения в Kafka днями, неделями, месяцами или даже годами.

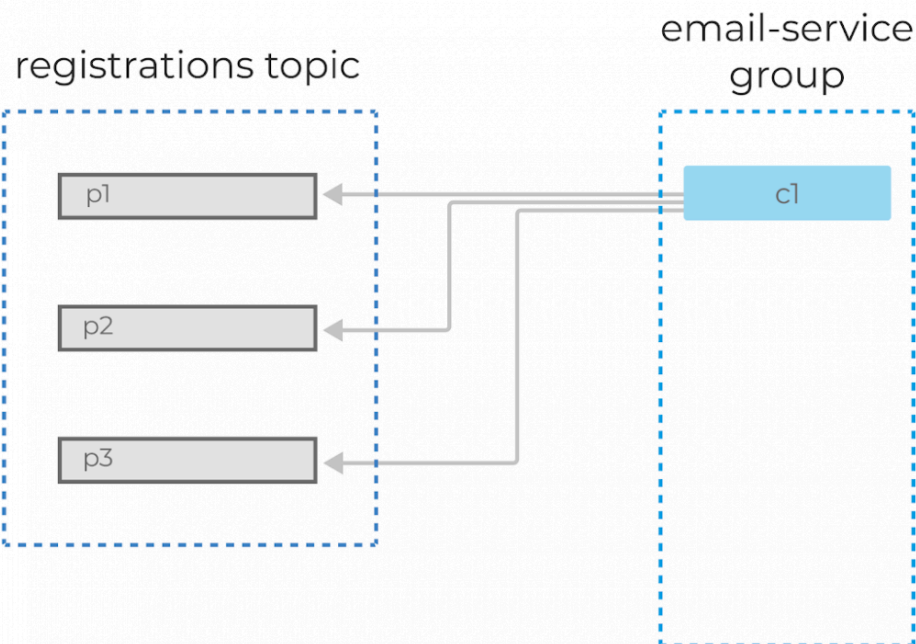


Consumer Groups

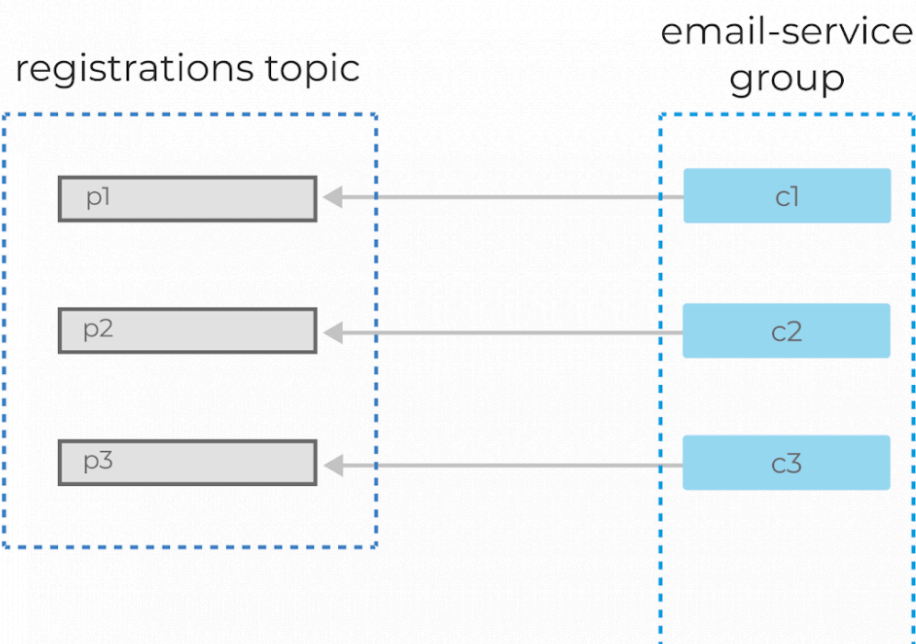
Теперь давайте перейдём к консьюмерам и рассмотрим их принципы работы в Kafka. Каждый консьюмер Kafka обычно является частью какой-нибудь консьюмер-группы.

Каждая группа имеет уникальное название и регистрируется брокерами в кластере Kafka. Данные из одного и того же топика могут считываться множеством консьюмер-групп одновременно. Когда несколько консьюмеров читают данные из Kafka и являются членами одной и той же группы, то каждый из них получает сообщения из разных партиций топика, таким образом распределяя нагрузку.

Вернёмся к нашему примеру с топиком сервиса регистрации и представим, что у сервиса отправки писем есть своя собственная консьюмер-группа с одним консьюмером `c1` внутри. Значит, этот консьюмер будет получать сообщения из всех партиций топика.



Если мы добавим ещё одного консьюмера в группу, то партии автоматически распределятся между ними, и *c1* теперь будет читать сообщения из первой и второй партии, а *c2* — из третьей. Добавив ещё одного консьюмера (*c3*), мы добъёмся идеального распределения нагрузки, и каждый из консьюмеров в этой группе будет читать данные из одной партии.



А вот если мы добавим в группу ещё одного консьюмера (*c4*), то он не будет задействован в обработке сообщений вообще.

Важно понять: внутри одной консьюмер-группы партии назначаются консьюмерам уникально, чтобы избежать повторной обработки.

Если консьюмеры не справляются с текущим объёмом данных, то следует добавить новую партицию в топик. Только после этого консьюмер с4 начнёт свою работу.

Механизм партиционирования является нашим основным инструментом масштабирования Kafka. Группы являются инструментом отказоустойчивости.

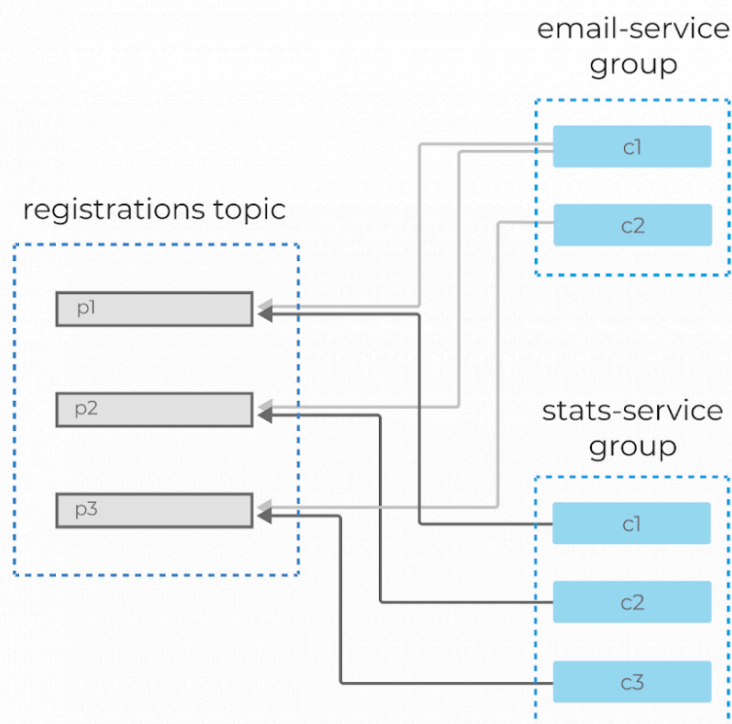
Кстати, как вы думаете, что будет, если один из консьюмеров в группе упадёт? Совершенно верно: партиции автоматически распределятся между оставшимися консьюмерами в этой группе.

Добавлять партиции в Kafka можно на лету, без перезапуска клиентов или брокеров. Клиенты автоматически обнаружат новую партицию благодаря встроенному механизму обновления метаданных. Однако, нужно помнить две важные вещи:

1. Гарантия очерёдности данных — если вы пишете сообщения с ключами и хешируете номер партиции для сообщений, исходя из общего числа, то при добавлении новой партиции вы можете просто сломать порядок этой записи.
2. Партиции невозможно удалить после их создания, можно удалить только весь топик целиком.

И ещё неочевидный момент: если вы добавляете новую партицию на проде, то есть в тот момент, когда в топик пишут сообщения продюсеры, то важно помнить про настройку `auto.offset.reset=earliest` в консьюмере, иначе у вас есть шанс потерять или просто не обработать кусок данных, записавшихся в новую партицию до того, как консьюмеры обновили метаданные по топику и начали читать данные из этой партиции.

Помимо этого, механизм групп позволяет иметь несколько несвязанных между собой приложений, обрабатывающих сообщения.



Как мы обсуждали ранее, можно добавить новую группу консьюмеров к тому же самому топике, например, для обработки и статистики регистраций. Эти две группы будут читать одни и те же сообщения из топика тех самых ивентов регистраций — в своём темпе, со своей внутренней логикой.

А теперь, зная внутреннее устройство консьюмеров в Kafka, давайте вернёмся к изначальному вопросу: **«Каким образом мы можем обозначить сообщения в партиции, как обработанные?»**.

Для этого Kafka предоставляет механизм консьюмер-офсетов. Как мы помним, каждое сообщение партиции имеет свой собственный, уникальный, монотонно возрастающий офсет. Именно этот офсет и используется консьюмерами для сохранения партиций.

Консьюмер делает специальный запрос к брокеру, так называемый `offset-commit` с указанием своей группы, идентификатора топик-партиции и, собственно, офсета, который должен быть отмечен как обработанный. Брокер сохраняет эту информацию в своём собственном специальном топике. При рестарте консьюмер запрашивает у сервера последний закоммиченный офсет для нужной топик-партиции, и просто продолжает чтение сообщений с этой позиции.

В примере консьюмер в группе `email-service-group`, читающий партицию `p1` в топике `registrations`, успешно обработал три сообщения с офсетами 0, 1 и 2. Для сохранения позиций консьюмер делает запрос к брокеру, коммитя офсет 3. В случае рестарта консьюмер запросит свою последнюю закоммиченную позицию у брокера и получит в ответе 3. После чего начнёт читать данные с этого офсета.



Консьюмеры вольны коммитить совершенно любой офсет (валидный, который действительно существует в этой топик-партиции) и могут начинать читать данные с любого офсета, двигаясь вперёд и назад во времени, пропуская участки лога или обрабатывая их заново.

Ключевой для понимания факт: в момент времени может быть только один закоммиченный офсет для топик-партиции в консьюмер-группе. Иными словами, мы не можем закоммитить

несколько офсетов для одной и той же топик-партиции, эмулируя каким-то образом выборочный acknowledgment (как это делалось в системах очередей).

Представим, что обработка сообщения с офсетом 1 завершилась с ошибкой. Однако мы продолжили выполнение нашей программы в консьюмере и запроцессили сообщение с офсетом 2 успешно. В таком случае перед нами будет стоять выбор: какой офсет закоммитить — 1 или 3. В настоящей системе мы бы рекомендовали закоммитить офсет 3, добавив при этом функционал, отправляющий ошибочное сообщение в отдельный топик для повторной обработки (ручной или автоматической). Подобные алгоритмы называются Dead letter queue.

Разумеется, консьюмеры, находящиеся в разных группах, могут иметь совершенно разные закоммиченные офсеты для одной и той же топик-партиции.

Apache ZooKeeper

В заключение нужно упомянуть об ещё одном важном компоненте кластера Kafka — Apache ZooKeeper.

ZooKeeper выполняет роль консистентного хранилища метаданных и распределённого сервиса логов. Именно он способен сказать, живы ли ваши брокеры, какой из брокеров является контроллером (то есть брокером, отвечающим за выбор лидеров партиций), и в каком состоянии находятся лидеры партиций и их реплики.

В случае падения брокера именно в ZooKeeper контроллером будет записана информация о новых лидерах партиций. Причём с версии 1.1.0 это будет сделано асинхронно, и это важно с точки зрения скорости восстановления кластера. Самый простой способ превратить данные в тыкву — потеря информации в ZooKeeper. Тогда понять, что и откуда нужно читать, будет очень сложно.

В настоящее время ведутся активные работы по избавлению Kafka от зависимости в виде ZooKeeper, но пока он всё ещё с нами (если интересно, посмотрите на [Kafka improvement proposal 500](#), там подробно расписан план избавления от ZooKeeper).

Важно помнить, что ZooKeeper по факту является ещё одной распределённой системой хранения данных, за которой необходимо следить, поддерживать и обновлять по мере необходимости.

Традиционно ZooKeeper раскатывается отдельно от брокеров Kafka, чтобы разделить границы возможных отказов. Помните, что падение ZooKeeper — это практически падение всего кластера Kafka. К счастью, нагрузка на ZooKeeper при нормальной работе кластера минимальна. Клиенты Kafka никогда не коннектятся к ZooKeeper напрямую.

→ [Дополнительные материалы к уроку](#)


→ [Полный курс по Kafka, релиз 7 апреля 2021](#)

Теги: kafka, apache kafka, topic, consumer, producer, очередь сообщений

Редакторский дайджест

Присылаем лучшие статьи раз в месяц


Электронпочта



Слёрм

Учебный центр для тех, кто работает в IT

Сайт



76

0


Карма

Рейтинг

@Polina_Averina

Редактор


Подписаться



 Комментарии 10

Публикации


ЛУЧШИЕ ЗА СУТКИ ПОХОЖИЕ


- 


123skipper123

23 часа назад


3D рендер с редактором карт в Консоли


 Средний



 12 мин

 3.1K

Из песочницы

 +40


 35


 6 +6
- 


RationalAnswer


13 часов назад


Доверие и честность в инвестициях, или два открытых вопроса Андрею Мовчану и Елене Чирковой по фонду GEIST

 7 мин

 5.5K

 +30

 5

 14 +14



El_Gato_Grande 10 часов назад

Не «Ctrl+C»/«Ctrl+V» едиными. История клавиш-модификаторов



10 мин



5.9K



+26



19



5 +5



ru_vds 10 часов назад

GPU для дата-центров



Простой



6 мин



1.2K

Обзор



+24



9



0



habr_career 7 часов назад

Зарплаты разработчиков в первом полугодии 2024: языки и квалификации



5 мин



9.9K



+22



11



17 +17



ru_vds 6 часов назад

Советы по программированию, которые бы я дал себе 15 лет назад



Средний



8 мин



2.8K

Обзор

Перевод



+21



31



1 +1



CyberPaul 5 часов назад

Джойбаблз. Удивительная история самого известного фрикера планеты



Простой



7 мин



940

Ретроспектива



+20



6



0



Cad1L 8 часов назад

От десятков до сотен тысяч RPS: как мы создали API, который развивается 10 лет без дропа обратной совместимости



Средний



10 мин



1.6K

+20

19

6 +6



meloman47 12 часов назад

Как я собрал настоящую Hi-Fi аудиосистему за 125 тыс. рублей

Простой

10 мин

10K

Тutorial

+19

33

24 +24



slavanikolsky 20 часов назад

Конец августа 2024. YouTube после замедления, про Rutube, Дзен и VK видео

5 мин

11K

+17

11

6 +6

Показать еще

ИНФОРМАЦИЯ

Ваш аккаунт

Войти

Регистрация

Разделы

Статьи

Новости

Хабы

Компании

Авторы

Песочница

Информация

Устройство сайта

Для авторов

Для компаний

Документы

Соглашение

Конфиденциальность

Услуги

Корпоративный блог

Медийная реклама

Нативные проекты

Образовательные

программы

Стартапам



Настройка языка

Техническая поддержка



ВИДЖЕТ



БЛОГ НА ХАБРЕ

6 часов назад

Системные сервисы Linux: плюсы, минусы и особенности

874

1 +1




13 авг в 18:59

Решаем задачи на Go без внешних зависимостей

 2.4K  0

2 авг в 15:11

Высокая доступность в Kubernetes

 2K  1 

27 апр в 12:02

Как управлять Kubernetes с помощью Ansible

 14K  10 

15 апр в 15:31

Chaos engineering: проверяем устойчивость Vault с помощью Gremlin

 2.2K  0