Platform V Corax

(KFK)

Детальная архитектура

Термины и определения

Термин/аббревиатура	Определение
ACL	Access Control List, список управления доступом
API	Application Programming Interface, программный интерфейс приложения
GSSAPI	Generic Security Services API, общий программный интерфейс сервисов безопасности
HTTP	HyperText Transfer Protocol, протокол передачи гипертекста
HTTPS	Расширение протокола HTTP для поддержки шифрования в целях повышения безопасности
JVM	Java Virtual Machine, виртуальная машина Java
POST	Метод запроса, поддерживаемый HTTP-протоколом, при котором веб-сервер принимает данные, заключенные в тело сообщения, для хранения
SASL	Simple Authentication and Security Layer, простой уровень аутентификации и безопасности
TCP	Transmission Control Protocol, протокол управления передачей
TLS	Transport Layer Security, протокол защиты транспортного уровня

Назначение

Цель создания

Platform V Corax — программный брокер сообщений, представляющий собой распределенную, отказоустойчивую, реплицированную и легко масштабируемую систему передачи сообщений, рассчитанную на высокую пропускную способность. Спроектирован специально для работы с большими объемами данных. Работает по принципу «публикация — доставка». Доставка сообщений осуществляется от так называемого Производителя сообщений (Producer) его Подписчикам (Consumer). Таким образом, чтобы подписчик смог прочитать сообщение, он должен предварительно подписаться на некоторый канал, в который производитель «публикует» сообщения.

Высокая пропускная способность Platform V Corax обеспечивается как со стороны источников, так и для систем подписчиков. Подписчики могут объединяться в группы, а данные могут временно храниться для последующей пакетной обработки. Все сообщения Platform V Corax хранит на диске.

Platform V Corax основан на open-source-решении Apache Kafka — брокере сообщений, реализованном на языке Scala. По сути, Platform V Corax является standalone-приложением, работающим на JVM.

Основные функции

- **публикация** (запись) и **чтение** (подписка) на события, включая постоянный обмен данными с другими системами;
- надежное **хранение** сообщений;
- **обработка** сообщений при их появлении, либо задним числом.

Сценарии использования

- обмен сообщениями;
- отслеживание активности веб-страниц в реальном времени;
- сбор статистики по метрикам с распределенных приложений с целью мониторинга;
- журналирование;
- обработка потоков;
- порождение событий (журналирование изменений состояния приложений);
- внешний commit-log для распределенных сообщений.

Руководство прикладного разработчика

Термины и определения

Термин/аббревиатура	Определение
I/O	Input/Output, ввод-вывод
Pub-sub	Publisher-subscriber, издатель-подписчик — поведенческий шаблон проектирования передачи сообщений, в котором отправители сообщений, именуемые издателями, напрямую не привязаны программным кодом отправки сообщений к подписчикам. Вместо этого сообщения делятся на классы и не содержат сведений о своих подписчиках, если таковые есть
TCP	Transmission Control Protocol, протокол управления передачей

Системные требования

Наименование	Версия	Разрядность	Тип	Назначение
Альт СП	8	64	Операционная система	Операционная система
Open JDK	8	64	Программная платформа	Среда выполнения Java-приложений

Подключение и конфигурирование

Подключение и конфигурирование программного продукта описывается в документе «Руководство по системному администрированию».

Установка продукта описывается в документе «Руководство по установке».

Использование программного продукта

Для работы Platform V Corax использует два класса:

- `KafkaProducer` — клиент, публикующий записи в кластере Platform V Corax (производитель);
- `KafkaConsumer` — клиент, читающий записи в кластере Platform V Corax (потребитель).

KafkaProducer

Функцией `KafkaProducer` является публикация записей в кластере Platform V Corax.

Класс `KafkaProducer` отсылает один экземпляр производителя по всем потокам; это позволяет значительно повысить быстродействие по сравнению с отправкой отдельных экземпляров для каждого потока.

Примеры использования

Ниже приведен простой пример использования производителя для отсылки записей со строками, содержащими последовательности чисел в качестве пар «ключ-значение»:

```
Properties props = new Properties();
 props.put("bootstrap.servers", "localhost:9092");
 props.put("acks", "all");
 props.put("retries", 0);
 props.put("linger.ms", 1);
 props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerial
izer");
 props.put("value.serializer", "org.apache.kafka.common.serialization.StringSeri
alizer");

 Producer<String, String> producer = new KafkaProducer<>(props);
 for (int i = 0; i < 100; i++)
     producer.send(new ProducerRecord<String, String>("my-topic", Integer.toStri
ng(i), Integer.toString(i)));

 producer.close();
```

Производитель состоит из пула буферного пространства, содержащего в себе записи, которые еще не были переданы на сервер, а также из дополнительного потока I/O, ответственного за конвертацию этих записей в запросы и их передачу в кластер.

> **Внимание**
>
> Ошибка выполнения метода `producer.close()` после использования производителя приведет к утечке ресурсов.

Метод `send()` — асинхронный. Он добавляет запись в буфер неотправленных записей и сразу же возвращается. Это позволяет производителю собирать отдельные записи в одном месте, что повышает его эффективность.

Конфигурационный параметр `acks` контролирует критерии, по которым определяется завершенность запросов. Настройка `all` означает блокировку при полной фиксации записи. Это самая медленная, но наиболее надежная настройка.

Если запрос окажется неудачным, производитель может повторить его. Однако в примере параметр `retries` равен 0, и запросы повторяться не будут.

> **Примечание**
>
> Включение параметра `retries` может привести к задвоению данных.

Производитель обрабатывает буферы необработанных записей для каждой партиции. Размер этих буферов определяется конфигурационным параметром `batch.size`. Увеличение размера буферов может увеличить количество записей, которое будет в них содержаться, однако это потребует больше памяти, поскольку на каждую активную партицию приходится по одному из таких буферов.

По умолчанию буфер доступен для отправки сразу, даже если он заполнен не до конца. Однако если нужно сократить количество запросов, можно установить параметр `linger.ms` в значение больше `0`. Этот параметр заставит производителя ждать дополнительных записей в течение заданного количества миллисекунд. Это похоже на алгоритм Нейгла для протокола TCP. Например, в блоке кода выше 100 записей были бы отосланы одновременно, так как параметр `linger.ms` равен 1 мс. Однако это добавило бы

задержку в 1 мс к ожиданию запросом дополнительных записей при не до конца заполненном буфере.

Внимание

Записи, появляющиеся близко друг к другу по времени, будут записываться в один и тот же пакет даже при `linger.ms=0`, и при высокой нагрузке запись в пакет будет осуществляться независимо от значения параметра `linger.ms`. Однако изменение значения данного параметра в большую сторону может привести к менее частым и более эффективным запросам при невысокой нагрузке за счет малой задержки.

Параметр `buffer.memory` контролирует количество памяти, доступное производителю для буферизации. Если скорость отсылки записей выше скорости передачи их на сервер, это приведет к переполнению буферного пространства. При переполнении буфера дальнейшие вызовы отправки будут заблокированы. Пороговое значение для времени блокировки определяется параметром `max.block.ms`. По истечении этого времени выдается исключение `TimeoutException`.

Параметры `key.serializer` и `value.serializer` отвечают за сериализацию объектов «ключ-значение», предоставляемых пользователем через `ProducerRecord`.

Примечание

Для простых строковых или байтовых типов можно использовать `ByteArraySerializer` и `StringSerializer`.

Начиная с версии Platform V Corax 0.11, `KafkaProducer` поддерживает два дополнительных режима:

- идемпотентный производитель — усиливает семантику доставки сообщений с `at least once` до `exactly once`. Это означает, что повторы запросов производителем не будут вызывать задвоения данных;
- транзакционный производитель — позволяет приложению отсылать сообщения в несколько партиций и топиков атомарно.

Для включения идемпотентности необходимо установить значение `true` для параметра `enable.idempotence`. В этом случае параметр `retries` примет значение `Integer.MAX_VALUE`, а параметр `acks` примет значение `all`.

Примечания

- Идемпотентность производителя не изменяет API, таким образом, имеющиеся приложения также не требуют изменений для корректной работы функции.
- Для корректной работы идемпотентного режима необходимо избегать повторных отправок на уровне приложений, поскольку задвоение отменить невозможно. Для этого необходимо оставить параметр `retries` без значения; оно автоматически станет равным `Integer.MAX_VALUE`.
- Если метод `send(ProducerRecord)` возвращает ошибку даже при бесконечном значении параметра `retries` (например, при сгорании сообщения в буфере до отправки), рекомендуется отключить производитель и проверить содержимое последнего созданного сообщения на предмет задвоения.

- Производитель может гарантировать идемпотентность только для сообщений, отправленных в одной сессии.

Чтобы использовать транзакционный производитель и сопутствующие API:

1. Установите значение параметра `transactional.id`. Когда этот параметр будет задан, идемпотентность включится автоматически вместе с конфигурационными параметрами производителя, от которых зависит идемпотентность.
2. Для повышения надежности сконфигурируйте топики, включенные в транзакции. Для этого значение параметра `replication.factor` должно быть не меньше `3`, а значение параметра `min.insync.replicas` для таких топиков должно быть равно `2`.
3. Для полной реализации гарантий транзакционности настройте потребители на чтение только завершенных сообщений.

Цель `transactional.id` — обеспечить возможность восстановления транзакций в нескольких сессиях одного экземпляра производителя. Обычно он происходит из идентификатора сегмента в партиционированном приложении с базой данных.

Все новые транзакционные API блокируются и при ошибке выдают исключения. Пример ниже показывает, каким образом можно использовать новые API. Он похож на пример выше. Различие состоит в том, что все 100 сообщений являются частью одной транзакции:

```
Properties props = new Properties();
 props.put("bootstrap.servers", "localhost:9092");
 props.put("transactional.id", "my-transactional-id");
 Producer<String, String> producer = new KafkaProducer<>(props, new StringSerial
izer(), new StringSerializer());

 producer.initTransactions();

 try {
     producer.beginTransaction();
     for (int i = 0; i < 100; i++)
         producer.send(new ProducerRecord<>("my-topic", Integer.toString(i), Int
eger.toString(i)));
     producer.commitTransaction();
 } catch (ProducerFencedException | OutOfOrderSequenceException | AuthorizationE
xception e) {
     // We can't recover from these exceptions, so our only option is to close t
he producer and exit.
     producer.close();
 } catch (KafkaException e) {
     // For all other exceptions, just abort the transaction and try again.
     producer.abortTransaction();
 }
 producer.close();
```

Как показано на примере, на один производитель может приходиться лишь одна открытая транзакция. Все сообщения, отсылаемые между методами `beginTransaction()` и `commitTransaction()`, будут частью одной транзакции. Когда параметр

`transactional.id` установлен, все сообщения, отосланные производителем, должны быть частью транзакции.

Транзакционный производитель использует исключения для сообщения ошибок. Таким образом, передавать функцию обратного вызова для метода `producer.send()` или вызывать метод `.get()` на возвращенном `Future` не требуется: `KafkaException` появится, если любой из вызовов метода `producer.send()` или транзакционных вызовов завершится неисправимой ошибкой во время транзакции.

При вызове `producer.abortTransaction()` при получении `KafkaException` можно убедиться в том, что все успешно завершенные записи будут помечены как отмененные, сохраняя таким образом гарантии транзакционности.

Ограничения

Клиент может сообщаться с брокерами версий 0.10.0 и новее. Более старые или более новые версии брокеров могут не поддерживать определенные функции клиента; например, транзакционные API требуют брокер версий 0.11.0 и новее. При вызове API, которая не поддерживает запущенный брокер, появится исключение `UnsupportedVersionException`.

KafkaConsumer

Класс `KafkaConsumer` — это клиент, который принимает записи из кластера Platform V Corax.

Этот клиент прозрачно обрабатывает сбои брокеров Platform V Corax и прозрачно адаптируется по мере миграции разделов топиков в пределах кластера. Он также взаимодействует с брокерами таким образом, чтобы группы потребителей могли балансировать нагрузку с помощью группировки потребителей.

Потребитель поддерживает TCP-соединения с необходимыми брокерами для получения данных. Если не закрыть соединение после того, как потребитель закончил его использовать, то произойдет утечка этого соединения.

> **Внимание**
>
> Потребитель не имеет средств безопасности для потоков.

Совместимость версий

Клиент может взаимодействовать с брокерами версии 0.10.0 или новее. Более старые или более новые брокеры могут не поддерживать определенные функции. Например, брокеры версии 0.10.0 не поддерживают функцию `offsetsForTimes`, поскольку она была добавлена в версии 0.10.1. При вызове API, недоступного в запущенной версии брокера, появится исключение `UnsupportedVersionException`.

Смещение и позиция потребителя

Platform V Corax хранит числовое смещение для каждой записи в разделе. Это смещение действует как уникальный идентификатор записи в этом разделе, а также обозначает положение потребителя в разделе.

Например, потребитель, находящийся в позиции 5, потребил записи со смещениями от 0 до 4 и затем получил запись со смещением 5. На самом деле для потребителя существуют два понятия позиции:

1. Позиция (`position`) потребителя определяет смещение следующей выдаваемой записи. Она будет на единицу больше, чем максимальное смещение, которое потребитель наблюдал в этом разделе. Позиция автоматически повышается каждый раз, когда потребитель получает сообщения при вызове `poll(Duration)`.

2. Зафиксированная позиция (`committed position`) — это последнее смещение, которое было надежно сохранено. В случае сбоя процесса и его перезапуска это будет то смещение, до которого восстановится потребитель. Потребитель может либо периодически автоматически фиксировать смещения, либо контролировать зафиксированную позицию вручную с помощью API-интерфейсов фиксации (например, используя `commitSync` и `commitAsync`).

Это различие дает потребителю контроль над тем, когда запись считается потребленной. Более подробное описание приведено ниже.

Группы потребителей и подписки на топики

Чтобы позволить пулу процессов разделить работу по потреблению и обработке записей, Platform V Corax использует концепцию групп потребителей. При этом процессы могут как выполняться на одной машине, так и быть распределены по нескольким, чтобы обеспечить масштабируемость и отказоустойчивость.

Все экземпляры потребителей, использующие один и тот же `group.id`, будут входить в одну и ту же группу потребителей.

Каждый потребитель в группе может динамически задавать список топиков, на которые он хочет подписаться, с помощью API подписки `subscribe`. Platform V Corax доставит каждое сообщение в топиках, на которые подписались потребители, одному процессу в каждой группе потребителей. Это достигается за счет балансировки разделов между всеми участниками группы потребителей таким образом, чтобы каждый раздел был назначен ровно одному потребителю в группе. Таким образом, если существует топик с четырьмя разделами и группа потребителей с двумя процессами, каждый процесс будет потреблять из двух партиций.

Членство в группе потребителей поддерживается динамически: в случае сбоя процесса назначенные ему разделы будут переназначены другим потребителям в той же группе. Аналогично, если новый потребитель присоединится к группе, разделы будут перенесены с существующих потребителей на новых. Это называется перебалансировкой группы и более подробно описывается ниже.

Перебалансировка групп также применяется при добавлении новых разделов в один из топиков, у которого есть подписчики, или при создании нового топика, соответствующего регулярному выражению подписки. Группа автоматически обнаружит новые разделы с помощью периодического обновления метаданных и назначит их участникам группы.

Группу потребителей можно представлять как одного логического подписчика, который состоит из нескольких процессов. Как система с несколькими подписчиками, Platform V Corax поддерживает наличие любого количества групп потребителей для любого топика без дублирования данных (дополнительные потребители на самом деле не требуют большого количества ресурсов).

Это обобщение функциональности, распространенной в системах обмена сообщениями. Чтобы получить семантику, аналогичную очереди в традиционной системе обмена сообщениями, все процессы могут входить в одну группу потребителей, и, следовательно, доставка записей будет сбалансирована по группе, как в очереди. Однако, в отличие от традиционной системы обмена сообщениями, у вас может быть несколько таких групп. Чтобы получить семантику, аналогичную pub-sub в традиционной системе обмена сообщениями, можно выделить каждому процессу свою собственную группу потребителей, и тогда каждый процесс будет подписываться на все записи, опубликованные в топике.

Кроме того, когда происходит автоматическое переназначение группы, потребители могут получать уведомления через слушателя `ConsumerRebalanceListener`, что позволяет им завершить необходимые операции на уровне приложения (например, очистку состояния, фиксацию смещения вручную и т. д.).

Потребитель также может вручную назначить определенные разделы (аналогично более старому «простому» потребителю) с помощью метода `assign(Collection)`. В этом случае динамическое назначение разделов и координация групп потребителей будут отключены.

Обнаружение сбоев у потребителей

После подписки на набор топиков потребитель автоматически присоединится к группе при вызове опроса `poll(Duration)`. API опроса разработан для поддержания активности потребителей. Пока опросы вызываются, потребитель будет оставаться в группе и продолжать получать сообщения из назначенных ему разделов. При этом потребитель периодически посылает на сервер сигнал, что с ним всё в порядке. Если потребитель выходит из строя или не отправляет сигнал в течение времени, заданного конфигурационным параметром `session.timeout.ms`, то он будет считаться отключенным и его разделы будут переназначены.

Иногда случается ситуация «живой блокировки»: потребитель продолжает посылать сигнал «всё в порядке», но в реальности никакой деятельности не выполняет. Чтобы в этом случае потребитель не мог бесконечно удерживать свои разделы, применяется механизм обнаружения активности с использованием параметра `max.poll.interval.ms`.

Таким образом, если опрос не вызывается по крайней мере с частотой заданного максимального интервала, то клиент сам покинет группу, чтобы другой потребитель мог занять его разделы. Когда это происходит, возможен сбой фиксации смещения (в этом случае будет выдано исключение `CommitFailedException` в результате вызова метода `commitSync()`). Это механизм безопасности, который гарантирует, что возможность фиксировать смещения будет только у активных членов группы. Поэтому, чтобы остаться в группе, нужно продолжать вызывать опросы.

Потребитель предоставляет два параметра конфигурации для управления поведением цикла опроса:

- `max.poll.interval.ms` — увеличивая интервал между ожидаемыми опросами, можно предоставить потребителю больше времени для обработки пакета записей, возвращенных из `poll(Duration)`. Недостаток данного способа заключается в том, что увеличение этого значения может задержать перебалансировку группы, поскольку потребитель присоединится к перебалансировке только внутри вызова опроса. С помощью этого параметра можно ограничить время для завершения перебалансировки, однако есть риск замедлить прогресс, если потребитель на самом деле не может достаточно часто вызывать `poll`.
- `max.poll.records` — позволяет ограничить максимальное количество записей, возвращаемых за один вызов опроса. Это может упростить прогнозирование максимума, который должен быть обработан в течение каждого интервала опроса. Настроив это значение, можно сократить интервал опроса, что уменьшит влияние перебалансировки групп.

В случаях, когда время обработки сообщений непредсказуемо варьируется, ни один из этих вариантов не сможет помочь наверняка. В таких случаях рекомендуется перенести обработку сообщений в другой поток, что позволит потребителю продолжать вызывать `poll`, пока процессор все еще работает. При этом необходимо строго следить, чтобы зафиксированные смещения не опережали фактическую позицию. Как правило, следует отключить автоматическую фиксацию и вручную фиксировать обработанные смещения для записей только после того, как поток завершит их обработку (в зависимости от необходимой семантики доставки).

Внимание

Чтобы новые записи из опроса не поступали до тех пор, пока поток не завершит обработку ранее возвращенных записей, приостановите раздел.

Примеры использования

API потребителя предоставляет определенную гибкость для большого количества сценариев использования. В подразделах приведены примеры таких кейсов.

Автоматическая фиксация смещения (offset)

Пример ниже показывает, как работает API потребителя Platform V Corax при автоматической фиксации смещения:

```
Properties props = new Properties();
     props.setProperty("bootstrap.servers", "localhost:9092");
     props.setProperty("group.id", "test");
     props.setProperty("enable.auto.commit", "true");
     props.setProperty("auto.commit.interval.ms", "1000");
     props.setProperty("key.deserializer", "org.apache.kafka.common.serializatio
n.StringDeserializer");
     props.setProperty("value.deserializer", "org.apache.kafka.common.serializat
ion.StringDeserializer");
     KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
     consumer.subscribe(Arrays.asList("foo", "bar"));
```

```
    while (true) {
        ConsumerRecords<String, String> records = consumer.poll(Duration.ofMill
is(100));
        for (ConsumerRecord<String, String> record : records)
            System.out.printf("offset = %d, key = %s, value = %s%n", record.off
set(), record.key(), record.value());
    }
```

Соединение с кластером инициализируется при помощи указания списка из одного или нескольких брокеров в конфигурационном параметре `bootstrap.servers`. Данный список используется для обнаружения других брокеров в кластере. Он не должен быть исчерпывающим списком серверных узлов в кластере, однако есть возможность указать больше одного серверного узла на случай, если при подключении клиента к кластеру некоторые серверы будут оффлайн.

Настройка `enable.auto.commit` означает, что смещения фиксируются автоматически с частотой, которая контролируется конфигурационным параметром `auto.commit.interval.ms`.

В данном примере потребитель подписывается на топики `foo` и `bar` как часть группы потребителей, имеющей название `test`, согласно параметру `group.id`.

Настройки десериализатора определяют, каким образом необходимо трансформировать байты в объекты. Например, при передаче строковых десериализаторов пары «ключ-значение» станут простыми строками.

Ручная фиксация смещений

Пользователь может самостоятельно контролировать потребление записей и фиксировать их смещение. Это может быть полезным в случаях, когда потребление сообщений объединено с логикой обработки и, таким образом, сообщение не может считаться потребленным, пока не завершена его обработка:

```
Properties props = new Properties();
    props.setProperty("bootstrap.servers", "localhost:9092");
    props.setProperty("group.id", "test");
    props.setProperty("enable.auto.commit", "false");
    props.setProperty("key.deserializer", "org.apache.kafka.common.serializatio
n.StringDeserializer");
    props.setProperty("value.deserializer", "org.apache.kafka.common.serializat
ion.StringDeserializer");
    KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
    consumer.subscribe(Arrays.asList("foo", "bar"));
    final int minBatchSize = 200;
    List<ConsumerRecord<String, String>> buffer = new ArrayList<>();
    while (true) {
        ConsumerRecords<String, String> records = consumer.poll(Duration.ofMill
is(100));
        for (ConsumerRecord<String, String> record : records) {
            buffer.add(record);
        }
        if (buffer.size() >= minBatchSize) {
```

```
            insertIntoDb(buffer);
            consumer.commitSync();
            buffer.clear();
        }
    }
```

В данном примере потребляется определенный набор записей и перемещается в память. Как только в памяти соберется достаточно большой набор записей, они будут перемещены в БД. Если бы фиксация происходила автоматически, как в предыдущем примере, записи бы считались потребленными после после их возвращения пользователю в `poll`. Это потенциально может привести к сбоям после сбора записей в один набор, но до отправки их в БД.

Чтобы избежать этого, нужно вручную зафиксировать смещения только после отправки соответствующих записей в БД. Это позволит четко проконтролировать, когда запись стала считаться потребленной. Однако это может привести к появлению противоположной возможности: сбой может произойти в промежутке, когда данные уже были отправлены в БД, но смещения еще не были зафиксированы (даже если такой промежуток составил всего несколько миллисекунд, возможность возникновения сбоев сохраняется). В этом случае потребляющий процесс будет потреблять сообщения из последнего зафиксированного смещения и повторять вставку последнего набора данных.

В таком режиме Platform V Corax предоставляет гарантии доставки сообщений `at-least-once`, что означает, что сообщение, скорее всего, будет доставлено, но при ошибке возможно его задвоение.

> **Внимание**
>
> Использование автоматической фиксации смещений также может предоставлять гарантии доставки сообщений `at-least-once`, но только при потреблении всех данных, возвращенных от каждого вызова методу `poll(Duration)` до начала любого последующего вызова, либо до закрытия (`close`) потребителя. Если какое-то из этих условий не будет выполнено, то зафиксированное смещение вполне может опередить позицию потребленных сообщений, что может привести к потере записей.
>
> Преимуществом ручной фиксации является непосредственный контроль над тем, когда запись может считаться «потребленной».

Приведенный выше пример использует метод `commitSync` для маркировки всех полученных записей как «зафиксированных». В некоторых случаях может понадобиться еще более четкий контроль того, какие записи фиксируются. Этого можно добиться, явно установив значение смещения. Пример ниже показывает, как зафиксировать смещение после окончания обработки записей в каждой партиции:

```
try {
        while(running) {
            ConsumerRecords<String, String> records = consumer.poll(Duration.of
Millis(Long.MAX_VALUE));
            for (TopicPartition partition : records.partitions()) {
                List<ConsumerRecord<String, String>> partitionRecords = records
```

```
.records(partition);
                for (ConsumerRecord<String, String> record : partitionRecords)
{
                    System.out.println(record.offset() + ": " + record.value())
;
                }
                long lastOffset = partitionRecords.get(partitionRecords.size()
- 1).offset();
                consumer.commitSync(Collections.singletonMap(partition, new Off
setAndMetadata(lastOffset + 1)));
            }
        }
    } finally {
      consumer.close();
    }
```

Внимание

Зафиксировано всегда должно быть смещение следующего сообщения, которое будет прочитано вашим приложением. То есть при вызове метода `commitSync(offsets)` необходимо добавить к смещению последнего обработанного сообщения 1.

Ручное назначение партиций

В примерах выше выполнена подписка на интересующие топики и позволено Platform V Corax динамически назначить каждому топику соответствующие партиции, основываясь на количестве активных потребителей в группе. Однако в некоторых случаях может понадобиться более четкий контроль над определенными партициями, если, например:

- процесс обслуживает какое-то локальное состояние, ассоциируемое с этой партицией (например, локальное дисковое хранилище пар «ключ-значение»), и тогда он должен получать только записи для партиции диска, которую он обслуживает;
- процесс сам по себе полностью доступен и при сбое восстанавливается (например, при помощи фреймворков управления кластером — YARN, Mesos или AWS либо как часть фреймворка обработки потока). В этом случае Platform V Corax не нужно обнаруживать сбой и перераспределять партиции, поскольку процесс-потребитель будет перезапущен на другой машине.

Для использования данного режима вместо подписки на топик с использованием метода `subscribe` можно просто вызвать метод `assign(Collection)` с полным списком партиций, которые необходимо потребить:

```
String topic = "foo";
     TopicPartition partition0 = new TopicPartition(topic, 0);
     TopicPartition partition1 = new TopicPartition(topic, 1);
     consumer.assign(Arrays.asList(partition0, partition1));
```

После назначения партиций можно сделать петлю с вызовом метода `poll`, как было описано в примерах выше. Группа, которую определяет потребитель, все еще

используется для фиксации смещений, но теперь набор партиций будет изменяться только при следующем обращении к методу `assign`. Ручное назначение партиций не использует координацию групп, поэтому сбои потребителя не приведут к перебалансировке назначенных партиций. Каждый потребитель действует независимо, даже если имеет общий `groupId` с другим потребителем. Для избежания конфликтов фиксации смещений необходимо всегда проверять уникальность `groupId` для каждого экземпляра потребителя.

> **Внимание**
>
> Комбинировать использование ручного назначения партиций (например, с использованием метода `assign`) с динамическим (например, с использованием метода `subscribe`) невозможно.

Сохранение смещений вне Platform V Corax

Приложение-потребитель не обязательно должно использовать встроенное хранилище смещений, оно может сохранять смещения в любом другом месте. Основным сценарием использования для этого является использование приложением для хранения смещения и результатов потребления в одной и той же системе таким образом, чтобы и результаты потребления, и смещения сохранялись атомарно. Это не всегда возможно, но это делает потребление более атомарным и позволяет получить семантику доставки сообщений `exactly once`, которая сильнее семантики `at-least-once`, которую пользователь получает, благодаря функциональности фиксирования смещений.

Ниже приведены примерные сценарии такого использования:

- Если результаты потребления сохраняются в реляционной БД, сохранение в ней смещения может дать возможность фиксировать результаты потребления и смещение в одной транзакции. Следовательно, либо транзакция окажется успешной и смещение будет обновлено, основываясь на том, какие записи были потреблены, либо результат не будет сохранен и смещение не будет обновлено.
- Если результаты сохраняются в локальное хранилище, то возможно будет сохранять в нем и смещение. Например, индекс поиска может быть настроен при подписке на определенную партицию и одновременном сохранении и смещения, и индексированных данных. При атомарном сохранении часто возникает такая возможность, и даже при возникновении сбоев с потерей несохраненных данных, все, что осталось после сбоя, будет иметь сохраненное смещение. Это означает, что в этом случае процесс индексации, вернувшийся с потерей последних обновлений, просто продолжает индексирование с того момента, где обновления не были потеряны.

Каждая запись имеет собственное смещение, поэтому для управления смещением необходимо сделать следующее:

- сконфигурировать параметр следующим образом: `enable.auto.commit=false`;
- использовать смещение, предоставляемое каждым экземпляром класса `ConsumerRecord` для сохранения текущей позиции;
- восстановить позицию потребителя, используя метод `seek(TopicPartition, long)`.

Такой тип использования является наиболее простым при ручном назначении партиций. Если же назначение партиций происходит автоматически, необходимо уделить дополнительное внимание перебалансировке партиций. Это можно сделать, передав экземпляр слушателя `ConsumerRebalanceListener` в методы `subscribe(Collection, ConsumerRebalanceListener)` и `subscribe(Pattern, ConsumerRebalanceListener)`. Например, если партиции берутся из потребителя, он будет пытаться зафиксировать свое смещение для этих партиций, применяя метод `ConsumerRebalanceListener.onPartitionsRevoked(Collection)`. При назначении партиций на потребителя он будет искать смещение для этих новых партиций и корректным образом инициализировать потребителя на эту позицию при помощи метода `ConsumerRebalanceListener.onPartitionsAssigned(Collection)`.

Другим распространенным случаем применения слушателя `ConsumerRebalanceListener` является освобождение обслуживаемых приложением кешей для перемещенных куда-либо партиций.

Многопоточная обработка

> **Внимание**
>
> Потребитель Platform V Corахне имеет средств безопасности для потоков. Весь сетевой I/O происходит в потоке вызывающего приложения. Обеспечение корректной синхронизации нескольких потоков является ответственностью пользователя. Несинхронизированный доступ приведет к появлению исключения `ConcurrentModificationException`.

Единственным исключением для вышеописанного правила является метод `wakeup()`, который может использоваться из внешнего потока для безопасного прерывания активной операции. В этом случае поток, блокирующий операцию, выдаст исключение `WakeupException`. Такой способ может использоваться для отключения потребителя из других потоков. Код ниже отражает типичный случай применения:

```
public class KafkaConsumerRunner implements Runnable {
    private final AtomicBoolean closed = new AtomicBoolean(false);
    private final KafkaConsumer consumer;

    public KafkaConsumerRunner(KafkaConsumer consumer) {
      this.consumer = consumer;
    }

    @Override
    public void run() {
        try {
            consumer.subscribe(Arrays.asList("topic"));
            while (!closed.get()) {
                ConsumerRecords records = consumer.poll(Duration.ofMillis(10000
));
                // Handle new records
            }
        } catch (WakeupException e) {
            // Ignore exception if closing
            if (!closed.get()) throw e;
```

```
        } finally {
            consumer.close();
        }
    }

    // Shutdown hook which can be called from a separate thread
    public void shutdown() {
        closed.set(true);
        consumer.wakeup();
    }
}
```

Затем, в отдельном потоке, потребитель может быть отключен с помощью добавления атрибута `closed` и вызова метода `wakeup()` для потребителя:

```
closed.set(true);
consumer.wakeup();
```

> **Примечание**
>
> Поскольку существует возможность использовать прерывания потока вместо использования метода `wakeup()` для прерывания блокирующей операции (в этом случае появится исключение `InterruptException`), использовать их не рекомендуется, так как это может привести к прерыванию «чистого» отключения потребителя. Прерывания можно использовать для случаев, в которых использование метода `wakeup()` не представляется возможным, например, когда поток потребителя управляется кодом, который не «знает» о наличии клиента Platform V Corax.

Такая модель поточности для обработки не была внедрена специально, так как это оставляет несколько возможностей для применения многопоточной обработки записей:

- **Один потребитель — один поток**.

  Это простая возможность давать каждому потоку свой экземпляр потребителя.

  Преимущества:

  - Легкость реализации.
  - Часто такой подход является наиболее быстрым и не требует внутрипоточной координации.
  - Такой подход облегчает реализацию порядковой обработки на основе порядка партиций (каждый поток просто обрабатывает сообщения по порядку их получения).

  Недостатки:

  - Большее количество потребителей означает большее количество TCP-подключений к кластеру (по одному на поток). Однако Platform V Corax эффективно управляет подключениями, поэтому это не сильно повлияет на работу.

  - Большее количество потребителей означает большее количество запросов, отправляемых на сервер, и немного меньшее количество пакетов с данными, что может привести к падению пропускной способности I/O.
  - Общее количество потоков по всем процессам будет ограничено общим количеством партиций.

- **Разделение потребления и обработки записей**.

  Другой альтернативой является наличие одного или нескольких потоков-потребителей, которые потребляют все данные и передают экземпляры `ConsumerRecords` в блокирующую очередь, потребленную пулом потоков-обработчиков, которые и выполняют обработку записей.

  Преимущество: позволяет масштабировать количество потребителей и обработчиков независимо друг от друга. Это дает возможность иметь один потребитель, который обслуживает несколько потоков-обработчиков, что снимает ограничение по партициям.

  Недостатки:

  - Гарантии порядка обработчиков требуют определенного внимания, так как потоки будут работать независимо друг от друга, и более ранний пакет данных может быть обработан после более позднего просто из-за недостатка времени работы потока.

    Примечание

    Для обработки без порядковых требований это не считается недостатком.

  - Ручное фиксирование позиции усложнится, поскольку будет требовать координации всех потоков для обеспечения успешного завершения обработки партиции.

Такой подход имеет множество вариаций. Например, каждый поток может иметь собственную очередь, а потоки-потребители могут хешироваться в эти очереди, используя метод `TopicPartition` для обеспечения порядкового потребления и облегчения процесса фиксирования.

Руководство по установке

Термины и определения

Термин/аббревиатура	Определение
PID	Process Identifier, идентификатор процесса
SSL	Secure Sockets Layer, криптографический протокол, предназначенный для защиты обмена данными в сети
СПО	Системное программное обеспечение

Системные требования

Описание комплекса технических средств и системного программного обеспечения (СПО), необходимых для работы Platform V Corax

В таблице ниже содержится перечень СПО, требуемого для функционирования Platform V Corax.

Наименование	Версия	Разрядность	Тип	Назначение
Альт СП	8	64	Операционная система	Операционная система
Open JDK	8	64	Программная платформа	Среда выполнения Java-приложений

В данном документе описывается развертывание на Linux.

Установка

Распакуйте архив дистрибутива `kafka_2.11-0.9.0.0.tgz` и зайдите в папку с помощью команд:

```
tar --xzf kafka_2.11-0.9.0.0.tgz
cd kafka_2.11-0.9.0.0
```

Запуск сервера Platform V Corax

Выполните запуск сервера Platform V Corax в следующем порядке:

- Запуск узла Platform V Corax:

  1. Запуск Zookeeper:

     ```
     > bin/zookeeper-server-start.sh config/zookeeper.properties
     ```

     ```
      [2015-03-09 15:01:37,495]  INFO Reading configuration from config/zoo
     keeper.properties (org.apache.zookeeper.server.quorum.QuorumPeerConfig
     )
      ...
     ```

     Где `config/zookeeper.properties` — файл конфигурации Zookeeper.

2. Запуск узла Platform V Corax:

```
> bin/kafka-server-start.sh config/server.properties

 [2015-03-09 15:02:37,495]  INFO Verifying properties (kafka.utils.Ver
ifiableProperties)
 [2015-03-09 15:02:37,495]  INFO Property socket.send.buffer.bytes is
overridden to 1048576 (kafka.utils.VerifiableProperties)
...
```

Где `config/server.properties` — файл конфигураций узла (брокера) Platform V Corax.

3. Запуск узлов Zookeeper и Platform V Corax в режиме демона:

```
> bin/zookeeper-start.sh -daemon config/zookeeper.properties
> bin/kafka-server-start.sh -daemon config/server.properties
```

Где `-daemon` — опция для запуска в режиме демона.

Запуск в режиме демона является рекомендуемым способом для эксплуатации.

Чтение журналов Zookeeper и Platform V Corax при запуске в режиме демона:

1. журнал приложения Zookeeper выводится в файл `logs/zookeeper.out`;
2. журнал приложения Platform V Corax выводится в файл `logs/server.log`.

Запуск узлов Zookeeper и Platform V Corax в режиме демона с выводом журнала из файла в консоль:

```
> bin/zookeeper-start.sh -daemon \
config/zookeeper.properties && tail -f logs/zookeeper.out

> bin/kafka-server-start.sh -daemon \
config/server.properties && tail -f logs/server.log
```

- Для остановки узлов Platform V Corax и Zookeeper выполните скрипты в следующем порядке:

```
> bin/kafka-server-stop.sh
> bin/zookeeper-server-stop.sh
```

- Для проверки состояния процессов узлов Zookeeper и Platform V Corax воспользуйтесь поиском PID этих процессов в оперативной памяти:

4. Проверка состояния узла Platform V Corax с помощью поиска PID его процесса:

```
> ps ax | grep -i 'kafka \. Kafka' | grep java | grep -v grep | awk '{
print $1}'

1234
```

В результате выведется PID найденного процесса или пустая строка в случае его отсутствия.

5. Проверка состояния узла Platform V Corax командой, запрашивающей список топиков:

```
> bin/kafka-topics.sh --list --zookeeper localhost:2181

...
__ consumer _ offsets
...
```

Где:

1. `--list` — опция, обозначающая режим вывода списка топиков;
2. `--zookeeper` — хост Zookeeper в формате `IP:Port`;
3. `localhost` — имя IP-адреса сервера, на котором запущен Zookeeper;
4. `2181` — порт `clientPort`, задаваемый в файле конфигураций `config/zookeeper.properties` для Zookeeper, через который осуществляется подключение клиентов.

В результате будет выведен список топиков, в котором, например, может содержаться топик `__ consumer _ offsets` (если к этому топику подключался Потребитель).

Примечания

- В текущей версии указанная команда будет работать только при отключенном SSL.
- Использование параметра `--zookeeper` для текущей версии считается устаревшим.

6. Проверка состояния узла Zookeeper с помощью поиска PID его процесса:

```
> ps ax | grep -i 'zookeeper' | grep -v grep | awk '{print $1}'

1234
```

В результате выведется PID найденного процесса или пустая строка в случае его отсутствия.

Обновление

Platform V Corax поддерживает обновление без прерывания сервиса в случае корректной конфигурации.

Системные требования, используемые для обновления:

3. Фактор репликации каждого топика не ниже `min_insync_replicas+1`.
4. Фактор репликации смещений не ниже 2.
5. Фактор репликации сведений о транзакциях не ниже 2.

Обновление без прерывания сервиса происходит путем последовательного перезапуска брокеров с обновлением исполняемых файлов и/или конфигурации. При смене версии протокола требуется задать дополнительные параметры конфигурации вида `inter.broker.protocol.version=X` и `log.message.format.version=X` в файле свойств брокера.

Общий принцип обновления с версии X на версию X+1:

- В конфигурационном файле `server.properties` задайте параметры `inter.broker.protocol.version=X` и `log.message.format.version=X`.
- Остановите работу одного из брокеров и замените исполняемые файлы.
- Запустите обновленный брокер и дождитесь окончания репликации (утилита `kafka-topics` не должна показывать ни одной не полностью реплицированной партиции).
- Повторите шаги 2 и 3 для каждого последующего брокера кластера.
- Уберите параметр `inter.broker.protocol.version=X` из файлов `server.properties` всех брокеров и выполните повторный перезапуск.
- После перехода клиентов на новую версию уберите параметр `log.message.format.version=X` и перезапустить кластер.