СБЕР УНИВЕРСИТЕТ

Школа Java Middle Developer

Kafka

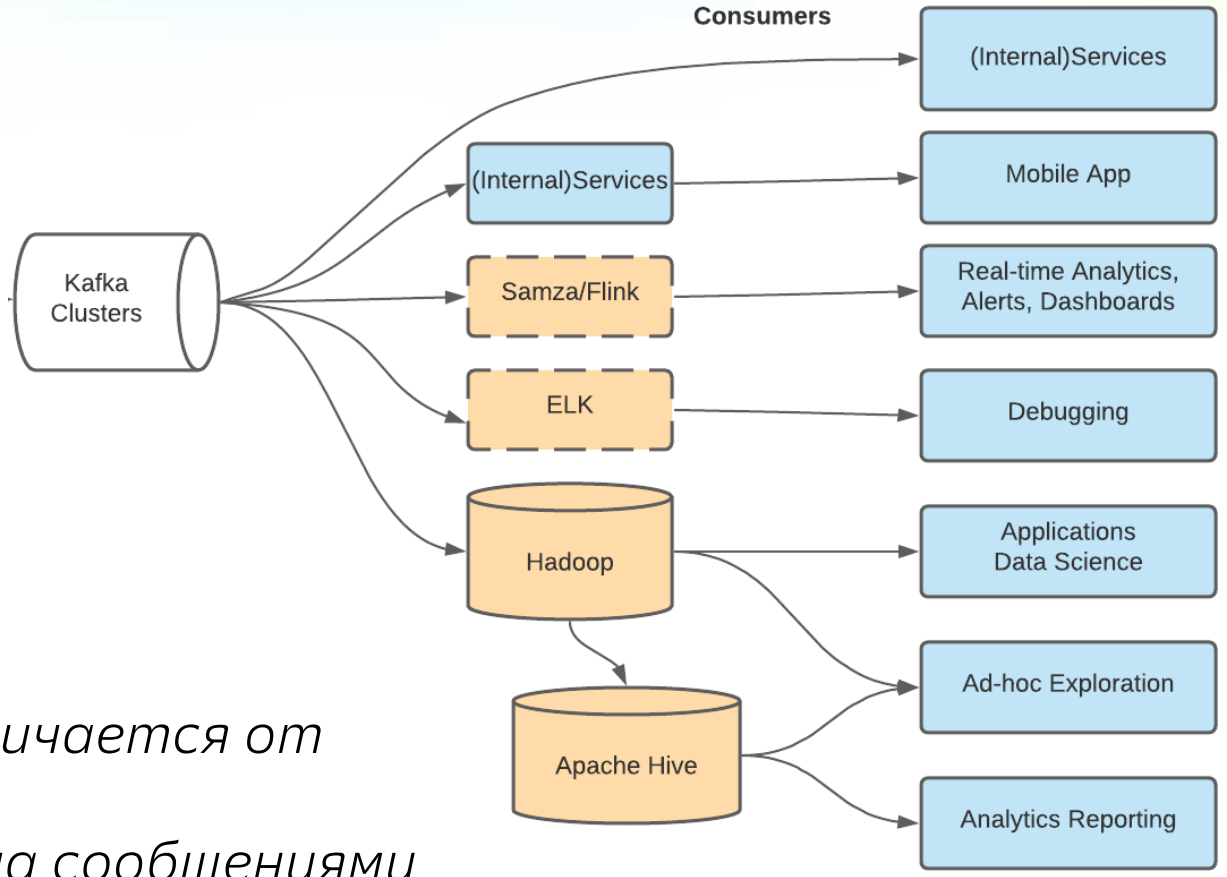
Потребители

Содержание

Чтение данных из Kafka

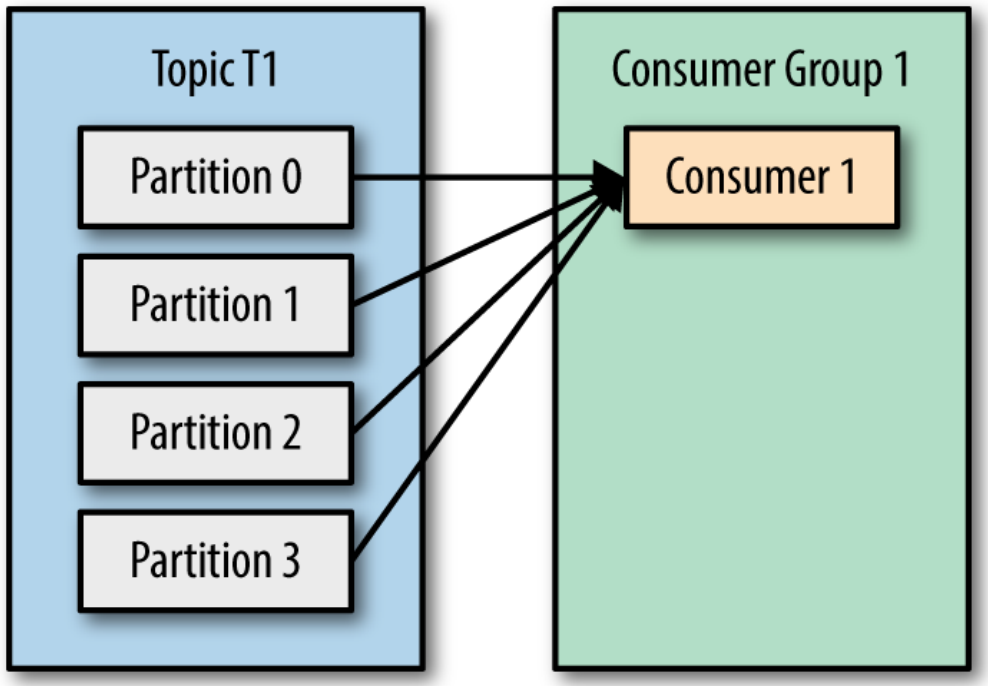
Чтение данных из Kafka

- ✓ *Приложения, читающие данные из Kafka, используют объект KafkaConsumer*
- ✓ *Чтение данных из Kafka несколько отличается от чтения данных из других систем обмена сообщениями*

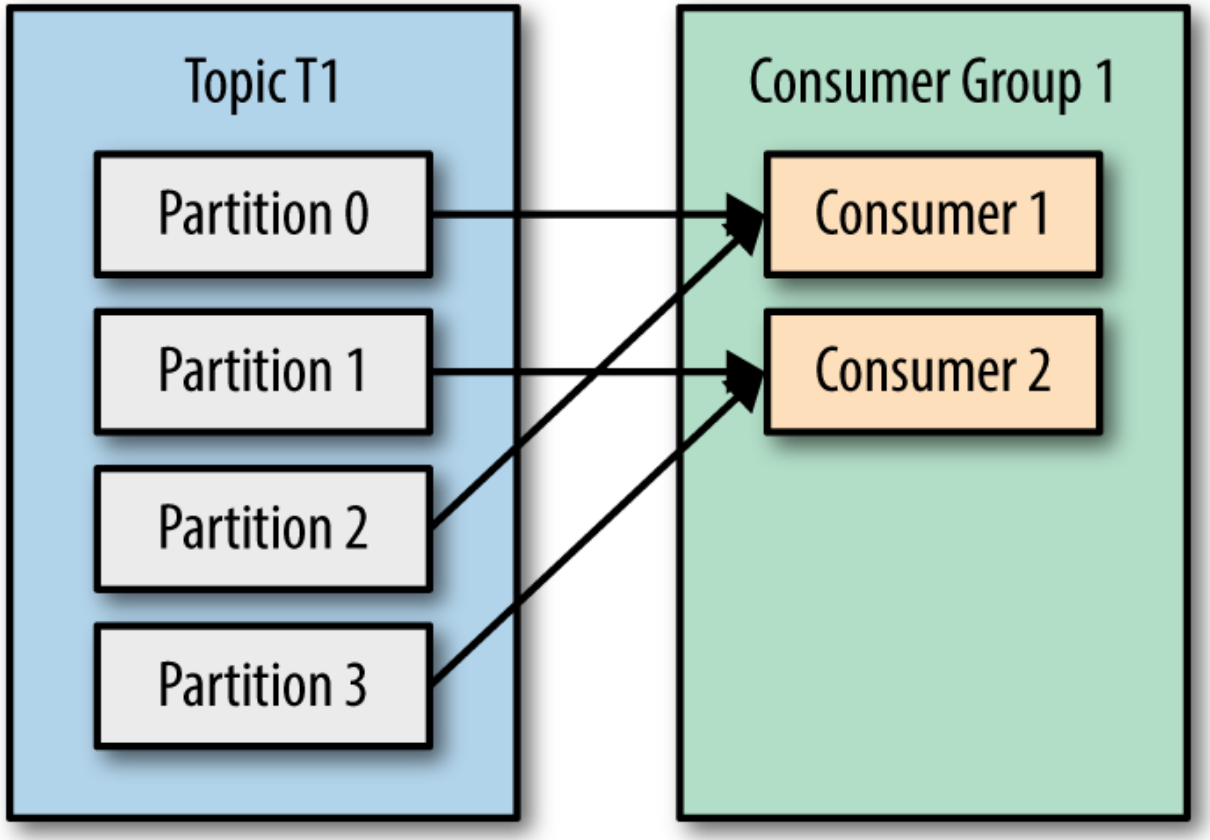


Потребители и группы потребителей

- *Потребители Kafka обычно состоят в группе потребителей;*
- *Потребители одной группы будут читать разные партиции;*
- *В группу стоит добавлять новые потребители при необходимости*

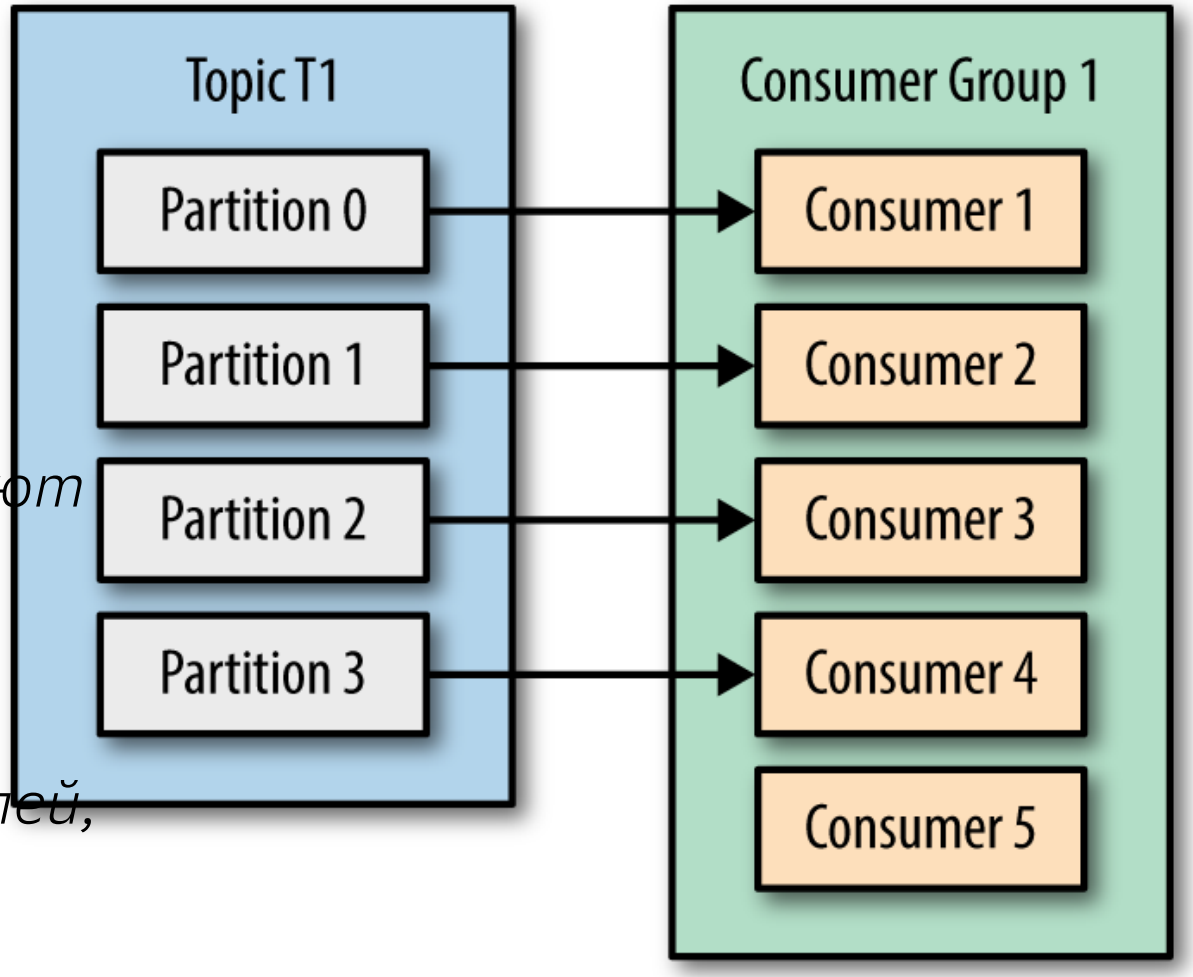


Потребители и группы потребителей



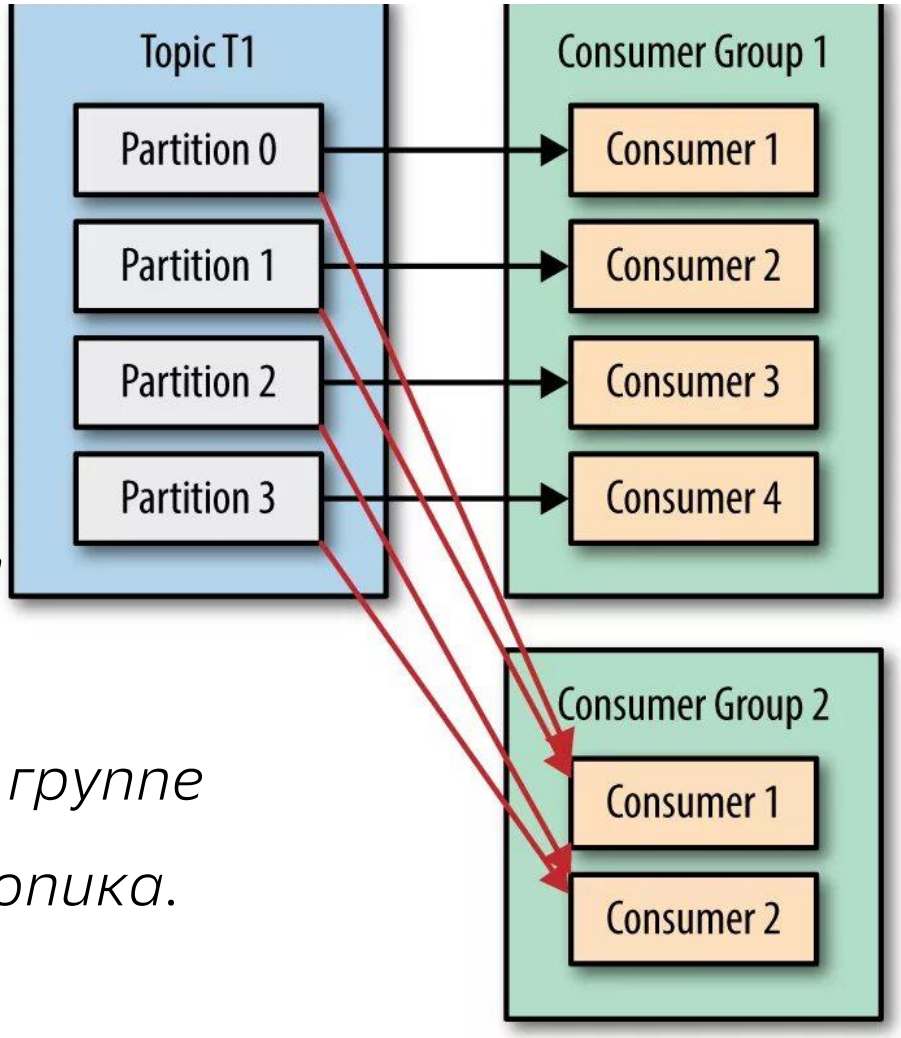
Потребители и группы потребителей

- *Основной способ масштабирования получения данных из Kafka – добавление новых потребителей в группу;*
- *Потребители Kafka часто выполняют операции с высоким значением задержки (например запись в БД);*
- *Нет смысла добавлять потребителей, больше, чем партиций в топике;*
- *Для масштабирования отдельного приложения широко распространено чтение несколькими приложениями данных из одного топика.*



Потребители и группы потребителей

- *Для каждого приложения, которому нужны все сообщения из одного топика или нескольких, необходимо создавать новую группу потребителей;*
- *Потребители добавляют в существующую группу при необходимости масштабирования чтения и обработки сообщений из топиков;*
- *До каждого дополнительного потребителя в группе доходит только подмножество сообщений топика.*

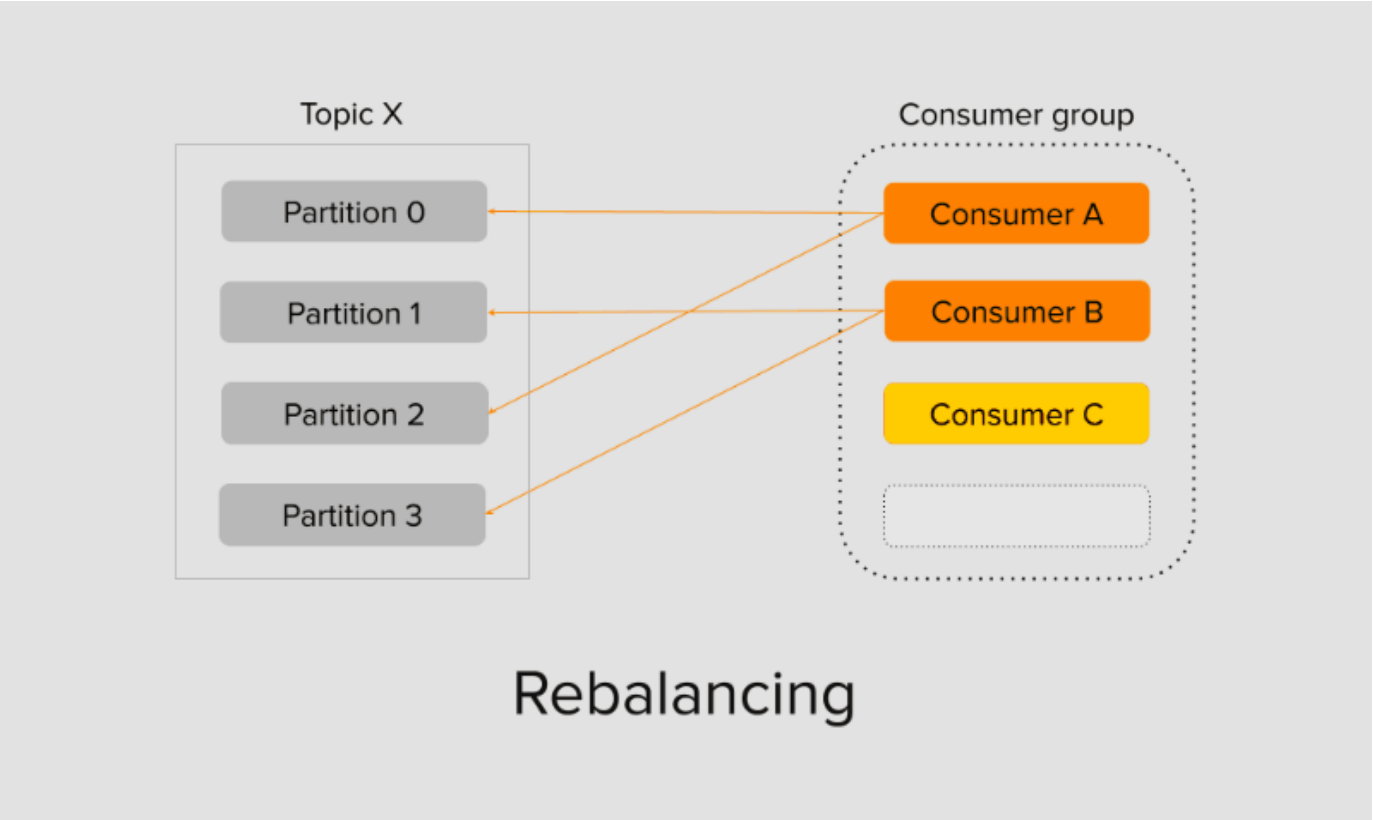


Группы потребителей и перебалансировка партиций

- *Перебалансировка происходит при добавлении потребителя в группу или при остановке потребителя;*
- *Переназначение партиций потребителям происходит также при изменении топиков, которые читает группа;*
- *Перебалансировка - передача партиции от одного потребителя другому;*
- *При обычных обстоятельствах перебалансировка нежелательна.*

Группы потребителей и перебалансировка партиций

- *Во время перебалансировки потребители не могут получать сообщения;*
- *Если потребитель использует кэш, после перебалансировки его нужно обновить;*



Как избежать нежелательной перебалансировки

- *Потребители поддерживают членство в группе и принадлежность партиций за счёт отправки периодических контрольных сигналов (heartbeats);*
- *Если потребитель на длительное время прекращает отправку контрольных сигналов инициируется перебалансировка;*
- *Если потребитель завершает работу штатным образом, он извещает координатора группы об этом, перебалансировка инициируется немедленно.*

Как происходит распределение разделов по брокерам

- *Когда потребитель хочет присоединиться к группе, он отправляет координатору группы запрос JoinGroup;*
- *Первый присоединившийся к группе потребитель становится ведущим потребителем группы;*
- *Реализация PartitionAssignor используется для распределения партиций по потребителям;*
- *Каждый потребитель знает только о назначенных ему партициях;*
- *Процедура распределения повторяется при*

Создание потребителя Kafka

- *Первый шаг к получению записей из Kafka - создание экземпляра класса KafkaConsumer;*
- *Для KafkaConsumer требуется экземпляр Properties, содержащий свойства, необходимые потребителю;*
- *Три обязательных свойства: bootstrap.servers, key.deserializer и value.deserializer;*
- *Необязательное свойство group.id – создаёт группу потребителей, к которой относится создаваемый KafkaConsumer.*

Создание потребителя Kafka

```
Properties props = new Properties();
props.put ("bootstrap.servers", "broker1:9092, broker2:9092");
props.put ("group.id", "CountryCounter");
props.put ("key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
props.put ("value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");

KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
```

Подписка на топики

```
consumer.subscribe(Collections.singletonList("customerCountries"));
```

```
consumer.subscribe(Pattern.compile("test.*"));
```

Цикл опроса

```
try {
    while (true) { // 1

        ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));// 2
        for (ConsumerRecord<String, String> record : records) {// 3

            log.debug("topic = %s, partition = %d, offset = %d, customer = % s, country = %s\n",
                    record.topic(), record.partition(), record.offset(), record.key(),
                    record.value());
            int updatedCount = 1;
            if (custCountryMap.countainsValue(record.value())) {
                updatedCount = custCountryMap.get(record.value()) + 1;
            }
        }
        custCountryMap.put(record.value(), updatedCount);
        JSONObject json = new JSONObject(custCountryMap);
        System.out.println(json.toString(4));// 4
    }
}
finally {
    consumer.close();
}
```

Потокобезопасность

- *Несколько потребителей, относящихся к одной группе, не могут работать в одном потоке;*
- *Несколько потоков не могут безопасно использовать один потребитель;*
- *Железное правило: один потребитель на один поток;*
- *Рекомендуется обернуть логику потребителя в отдельный объект и воспользоваться объектом типа ExecutorService языка Java для запуска нескольких потоков, каждый – со своим потребителем.*

Настройка потребителей

- *fetch.min.bytes - позволяет потребителю задавать минимальный объём данных, получаемых от брокера при извлечении записей;*
- *fetch.max.wait.ms - позволяет управлять тем, сколько именно ждать. По умолчанию Kafka ждет 500 мс;*
- *max.partition.fetch.bytes - определяет максимальное число байт, возвращаемых сервером из расчета на одну партицию;*
- *session.timeout.ms – если потребитель не отправляет контрольный сигнал координатору группы в течение промежутка времени, большего, чем определено этим параметром, он считается отказавшим и инициируется перебалансировка.*

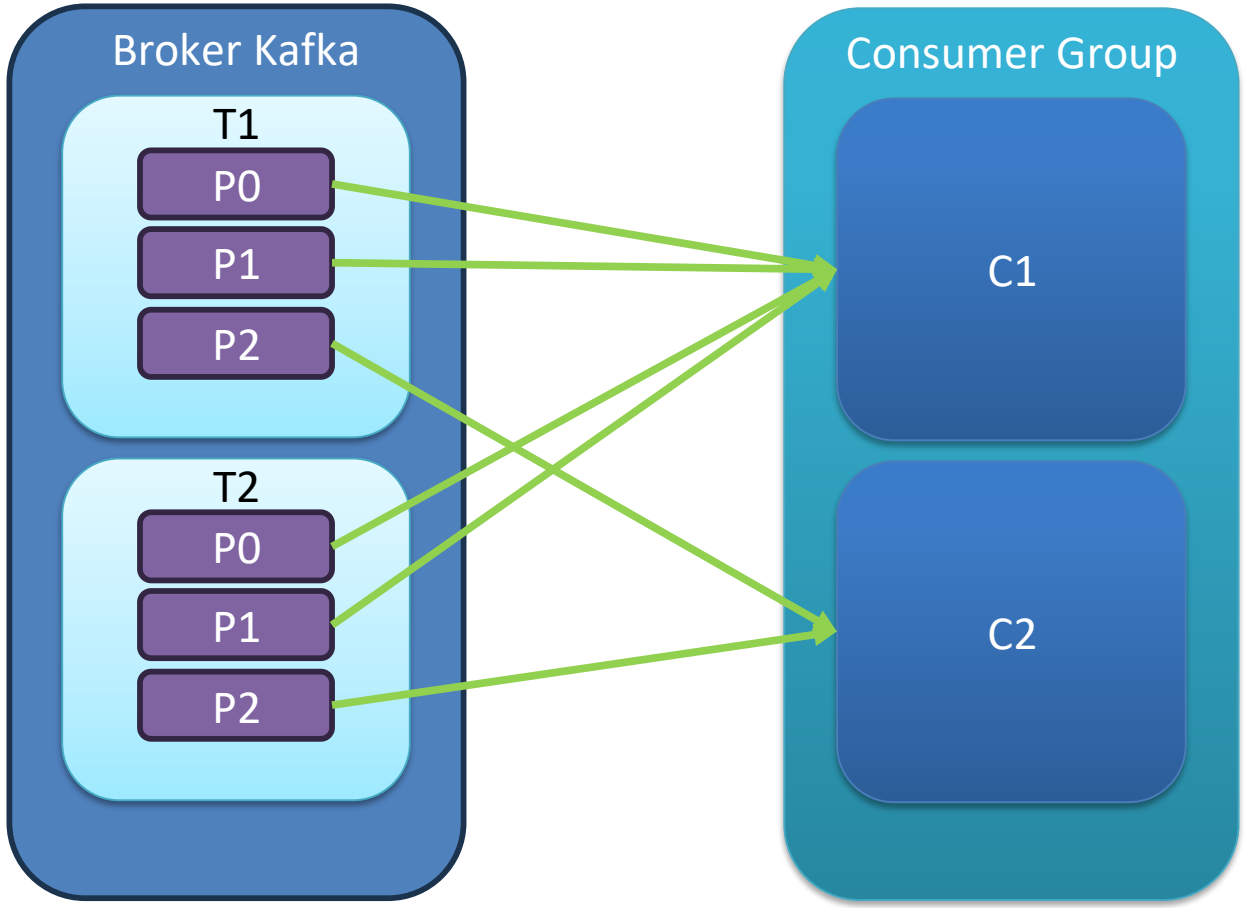
Настройка потребителей

- *auto.offset.reset - определяет поведение потребителя при начале чтения партиции, для которого у него зафиксированное смещение отсутствует или стало некорректным;*
- *enable.auto.commit - определяет, будет ли потребитель фиксировать смещения автоматически (по умолчанию true);*
- *partition.assignment.strategy – выбор стратегии распределения партиций по потребителям.*

Стратегии распределения партиций по топикам

Диапазонная (Range)

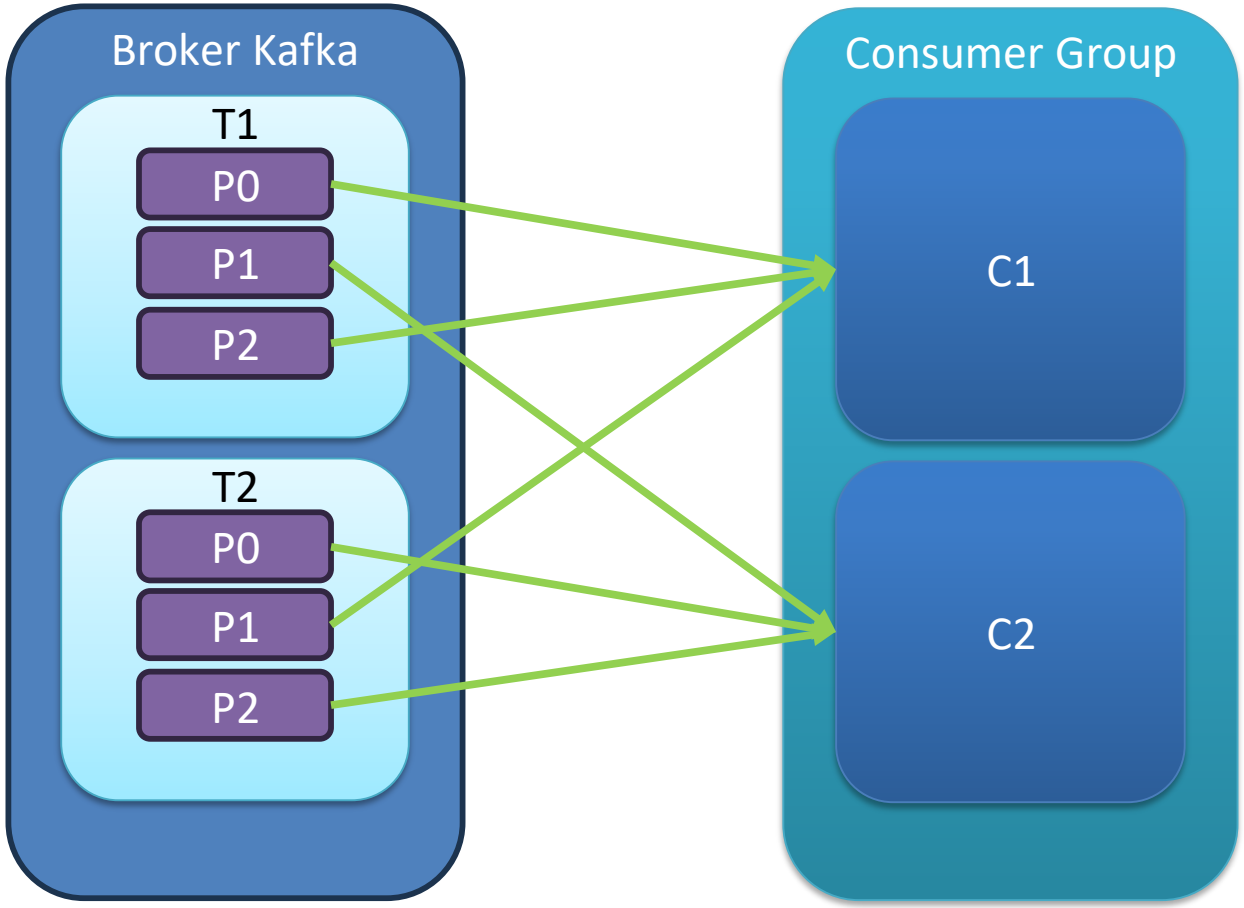
Поскольку в каждом из топиков нечетное количество партиций, а распределение партиций по потребителям выполняется для каждого топика отдельно, у первого потребителя окажется больше партиций, чем у второго. Подобное происходит всегда, когда используется диапазонная стратегия распределения, а количество потребителей не делится нацело на количество



Стратегии распределения партиций по топикам

Циклическая (RoundRobin)

Когда все потребители подписаны на одни и те же топики (очень распространенный сценарий), циклическое распределение дает одинаковое количество партиций у всех потребителей (или, в крайнем случае, различие в 1 партицию).



Настройка потребителей

- *client.id - используют брокеры для идентификации отправленных клиентом сообщений;*
- *max.poll.records - задаёт максимальное число записей, возвращаемое при одном вызове метода poll();*
- *receive.buffer.bytes и send.buffer.bytes - размеры TCP-буферов отправки и получения, используемых сокетами при записи и чтении данных. Если значение этих параметров равно -1, будут использоваться значения по умолчанию операционной системы.*

Фиксация смещения

- *Метод poll() при вызове возвращает записанные в Kafka данные, ещё не прочитанные потребителями из группы;*
- *Действие по обновлению текущей позиции потребителя в партиции называется фиксацией (commit);*
- *Потребители фиксируют смещение потём отправки в специальный топик _consumer_offsets сообщения, содержащего смещение для каждой из партиций;*
- *Если зафиксированное смещение меньше смещения последнего обработанного клиентом сообщения, то расположенные между ними сообщения будут обработаны дважды;*
- *Если зафиксированное смещение превышает смещение последнего фактически обработанного клиентом события, расположенные в этом промежутке сообщения будут пропущены группой потребителей.*

Автоматическая фиксация

- *При значении true параметра enable.auto.commit потребитель каждые 5 секунд (auto.commit.interval.ms) будет автоматически фиксировать максимальное смещение, возвращенное клиенту методом poll();*
- *При включенной автофиксации вызов метода poll() всегда будет фиксировать последнее смещение, возвращенное предыдущим вызовом;*
- *Метод close() также автоматически фиксирует смещения;*
- *Автоматическая фиксация удобна, но она не позволяет разработчику управлять так, чтобы полностью избежать*

Фиксация текущего смещения

- *При задании параметра auto.commit.offset=false смещения будут фиксироваться только тогда, когда приложение потребует этого явным образом;*
- *Простейший и наиболее надежный API фиксации — commitSync();*
- *commitSync() фиксирует последнее возвращенное методом poll() смещение;*
- *При запуске перебалансировки все сообщения с начала самого недавнего пакета и до момента начала перебалансировки окажутся обработанными дважды;*

Фиксация текущего смещения

```
while (true) {

    ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
    for (ConsumerRecord<String, String> record : records) {

        log.debug("topic = %s, partition = %d, offset = %d, customer = % s, country = %s\n",
                record.topic(), record.partition(), record.offset(), record.key(), record.value());// 1

    }

    try {
        consumer.commitSync();// 2
    }
    catch (Exception e) {
        log.error("commit error", e);// 3
    }
}
```

Асинхронная фиксация

- *При ручной фиксации приложение оказывается заблокировано, пока брокер не ответит на запрос фиксации – это ограничивает пропускную способность;*
- *Снижение частоты фиксации повышает пропускную способность, но повышает число потенциальных дубликатов при перебалансировке;*
- *Для повышения пропускной способности можно применять API асинхронной фиксации.*

Асинхронная фиксация

```
while (true) {

    ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));

    for (ConsumerRecord<String, String> record : records) {

        log.debug("topic = %s, partition = %d, offset = %d, customer = % s, country = %s\n",
                record.topic(), record.partition(), record.offset(), record.key(), record.value());// 1

    }

    consumer.commitAsync();// 1
}
```

- *Недостаток: в то время как commitSync() будет повторять попытку фиксации до тех пор, пока она не завершится успешно или не возникнет ошибка, которую нельзя исправить путём повтора, commitAsync() повторять попытку не станет.*

Асинхронная фиксация

```
while (true) {

    ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
    for (ConsumerRecord<String, String> record : records) {

        log.debug("topic = %s, partition = %d, offset = %d, customer = % s, country = %s\n",
                record.topic(), record.partition(), record.offset(), record.key(), record.value());// 1

    }
    consumer.commitAsync(
            new OffsetCommitCallback() {
                @Override
                public void onComplete(
                        Map<TopicPartition, OffsetAndMetadata> offsets, Exception exception) {

                    if (exception != null) {
                        log.error("commit failed for offsets {}", offsets, exception);
                    }
                }
            }
    );
}
```

Сочетание асинхронной и синхронной фиксации

```
try {
    while (true) {

        ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
        for (ConsumerRecord<String, String> record : records) {

            log.debug("topic = %s, partition = %d, offset = %d, customer = % s, country = %s\n",
                    record.topic(), record.partition(), record.offset(), record.key(), record.value());

        }
        consumer.commitAsync();// 1
    }
}
catch (Exception e) {
    log.error("Unexpected error", e);
}
finally {
    try {
        consumer.commitSync();// 2
    }
    finally {
        consumer.close();
    }
}
```

Фиксация заданного смещения

- *Фиксация последнего смещения происходит только при завершении обработки пакетов;*
- *API потребителей предоставляет возможность вызывать методы commitAsync() или commitSync(), передавая им HashMap партиций и смещений, которые нужно зафиксировать.*

Фиксация заданного смещения

```
private Map<TopicPartition, OffsetAndMetadata> currentOffsets = new HashMap<>();// 1
int counter = 0;

…

while (true) {

ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
for (ConsumerRecord<String, String> record : records) {

    log.debug("topic = %s, partition = %d, offset = %d, customer = % s, country = %s\n",
            record.topic(), record.partition(), record.offset(), record.key(), record.value());// 2

    currentOffsets.put(new TopicPartition(record.topic(), record.partition()),
            new OffsetAndMetadata(record.offset() + 1, "no metadata"));// 3
    if (counter % 1000 == 0) {// 4
        consumer.commitAsync(currentOffsets, null);// 5
        counter++;
    }
}
```

Прослушивание на предмет перебалансировки

- *Потребителю необходимо выполнить определенную «чистку» перед завершением выполнения, а также перед перебалансировкой партиций;*
- *Если известно, что партиция вот-вот перестанет принадлежать данному потребителю, то желательно зафиксировать смещения последних обработанных событий;*
- *API потребителей позволяет выполнять ваш код во время смены (добавления/удаления) принадлежности партиций потребителю. Для этого необходимо передать объект ConsumerRebalanceListener в вызов метода subscribe().*

Прослушивание на предмет перебалансировки

У класса ConsumerRebalanceListener есть два доступных для реализации метода:

- *public void onPartitionsRevoked(Collection‹TopicPartition› partitions) - вызывается до начала перебалансировки и после завершения получения сообщений потребителем. Именно в этом методе необходимо фиксировать смещения, чтобы следующий потребитель, которому будет назначен этот раздел, знал, с какого места начинать;*
- *public void onPartitionAssigned (Collection‹TopicPartition> partitions) - вызывается после переназначения партиций потребителю, но до того, как он начнёт получать сообщения.*

Получение записей с заданными смещениями

- *Если вы хотите начать чтение всех сообщений с начала партиции или пропустить все сообщения до конца партиции и получать только новые сообщения, то можно применить специализированные API с методами seekToBeginning(TopicPartition tp) и seekToEnd(TopicPartition tp);*

- *API Kafka даёт возможность переходить к конкретному смещению.*

Получение записей с заданными смещениями

```
while (true) {

    ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
    for (ConsumerRecord<String, String> record : records) {

        this.currentOffsets.put(new TopicPartition(record.topic(), record.partition()),
                new OffsetAndMetadata(record.offset() + 1, "no metadata"));

        processRecord(record);
        storeRecordInDB(record);
        consumer.commitAsync(this.currentOffsets, null);
    }
}
```

Получение записей с заданными смещениями

```
consumer.subscribe(Pattern.compile("test.*"), new ConsumerRebalanceListener() {
    @Override
    public void onPartitionsRevoked(Collection<TopicPartition> partitions) {
        commitDBTransaction();// 1
    }
    @Override
    public void onPartitionsAssigned(Collection<TopicPartition> partitions) {
        for (TopicPartition partition : partitions) {
            consumer.seek(partition, getOffsetFromDB(partition));// 2
        }
    }
});
consumer.poll(Duration.ofMillis(0));
for (TopicPartition partition : consumer.assignment()) {
    consumer.seek(partition, getOffsetFromDB(partition)); // 3
}
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
    for (ConsumerRecord<String, String> record : records) {
        processRecord(record);
        storeRecordInDB(record);
        storeOffsetInDB(record.topic(), record.partition(), record.offset());// 4
    }
    commitDBTransaction();
}
```

Выход из бесконечного цикла при вычитывании данных

- *Если вы решите выйти из цикла опроса, вам понадобится ещё один поток выполнения для вызова метода consumer.wakeup();*
- *Если цикл опроса выполняется в главном потоке, это можно сделать из потока ShutdownHook;*
- *Вызов метода wakeup() приведёт к завершению выполнения метода poll() с генерацией исключения WakeupException;*
- *Обрабатывать исключение WakeupException не требуется, но перед завершением выполнения потока нужно вызвать consumer.close().*

Десериализаторы

- *Для потребителей Kafka необходимы десериализаторы для преобразования полученных из Kafka байтовых массивов в объекты Java;*
- *Используемый для отправки событий в Kafka сериализатор должен соответствовать десериализатору, применяемому при их получении;*
- *AvroSerializer гарантирует, что все записываемые в конкретный топик данные совместимы со схемой топика, а значит, их можно будет десериализовать с помощью соответствующего десериализатора и схемы.*

Пользовательские сериализаторы

```
public class CustomerDeserializer implements Deserializer<Customer> {
    @Override
    public Customer deserialize(String topic, byte[] data) {
        int id;
        int nameSize;
        String name;
        try {
            if (data == null) {
                return null;
            }
            if (data.length < 8) {
                throw new SerializationException(
                        "Size of data received by IntegerDeserializer is shorter than expected");
            }
            ByteBuffer buffer = ByteBuffer.wrap(data);
            id = buffer.getInt();
            nameSize = buffer.getInt();
            byte[] nameBytes = new byte[nameSize];
            buffer.get(nameBytes);
            name = new String(nameBytes, StandardCharsets.UTF_8);
            return new Customer(id, name);// 2
        }
        catch (Exception e) {
            throw new SerializationException("Error when deserializing Customer to byte[] ", e);
        }
    }
}
```

Пользовательские сериализаторы

```
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092, broker2:9092");
props.put("group.id", "CountryCounter");
props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer", "org.apache.kafka.common.serialization.CustomDeserializer");

KafkaConsumer<String, Customer> consumer = new KafkaConsumer<>(props);

while (true) {

    ConsumerRecords<String, Customer> records = consumer.poll(Duration.ofMillis(100));
    for (ConsumerRecord<String, Customer> record : records) {

        log.debug("topic = %s, partition = %d, offset = %d, customer = % s, country = %s\n",
                record.topic(), record.partition(), record.offset(), record.key(), record.value());// 1

    }
}
```

Использование десериализации Avro в потребителе Kafka

```
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092, broker2:9092");
props.put("group.id", "CountryCounter");
props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer", "org.apache.kafka.common.serialization.KafkaAvroDeserializer");// 1
props.put("schema.registry.url", schemaUrl);// 2

String topic = "customerContracts";
KafkaConsumer<String, Customer> consumer = new KafkaConsumer<>(props);

consumer.subscribe(Collections.singletonList(topic));

while (true) {

    ConsumerRecords<String, Customer> records = consumer.poll(Duration.ofMillis(1000));// 3
    for (ConsumerRecord<String, Customer> consumerRecord : records) {

        System.out.println("Current customer name is: " + consumerRecord.value().getCustomerName());// 4

    }
    consumer.commitSync();
}
```

Автономный потребитель: зачем и как использовать потребитель без группы

```
String topic = "customerContracts";
   KafkaConsumer<String, Customer> consumer = new KafkaConsumer<>(props);
   List<PartitionInfo> partitionInfos = consumer.partitionsFor(topic);// 1

   if (partitionInfos != null) {
      consumer.assign(partitionInfos.stream()
            .map(partitionInfo -> new TopicPartition(partitionInfo.topic(), partitionInfo.partition()))
            .collect(Collectors.toList()));// 2
   }

   while (true) {

      ConsumerRecords<String, Customer> records = consumer.poll(Duration.ofMillis(1000));
      for (ConsumerRecord<String, Customer> consumerRecord : records) {

         log.debug("topic = %s, partition = %d, offset = %d, customer = % s, country = %s\n",
               consumerRecord.topic(), consumerRecord.partition(), consumerRecord.offset(), consumerRecord.key(), consumerRecord.value());

      }
      consumer.commitSync();
   }
}
```

Спасибо за внимание