3장

카프카 브로커

카프카 브로커

- 카프카 클라이언트와 데이터를 주고받는 주체
- 프로듀서가 보낸 데이터를 파티션에 안전하게 분산 저장하고 복제하는 역할을 수행
- 프로듀서로부터 전달된 데이터는 파일 시스템에 저장되지만, 페이지 캐시를 활용하여 파일 입출력의 성능을 향상시켜 빠르게 동작함
- 컨슈머가 데이터를 요청하면 파티션에 저장된 데이터를 전달
- 카프카의 데이터 복제(replication)는 파티션 단위로 이루어짐
- 토픽을 생성할 때 파티션이 복제 계수(replication factor)가 같이 설정됨 (1 ~ 브로커 개수)

컨트롤러

- 클러스터의 브로커 중 한대가 컨트롤러 역할을 수행
- 다른 브로커들의 상태를 체크하고, 한 브로커가 클러스터에서 빠지는 경우 해당 브로커에 존재하는 리더 파티션을 재분배

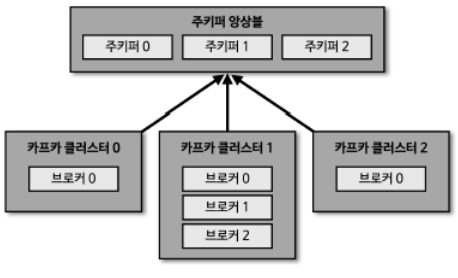
데이터 삭제

- 다른 메시징 플랫폼과 달리 카프카는 컨슈머가 데이터를 가져가도 토픽의 데이터는 삭제되지 않는다
- 컨슈머나 프로듀서는 데이터 삭제를 요청할 수 없으며, 오직 브로커만이 데이터를 삭제할 수 있다
- 데이터 삭제는 파일 단위로 이루어지며, 이 단위를 '로그 세그먼트(log segment)'라고 부른다
- 세그먼트 파일은 log로 시작하는 옵션값에 따라 열리고 닫히며 삭제되는 등 자동으로 관리된다
- 데이터를 삭제하지 않고 메시지 키를 기준으로 오래된 데이터를 압축하는 정책도 가능하다

코디네이터

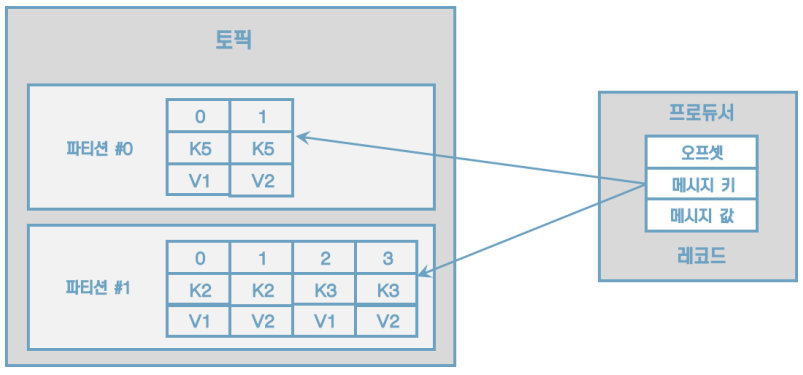
- 클러스터의 브로커 중 한대가 코디네이터 역할을 수행
- 컨슈머 그룹의 상태를 체크하고 파티션을 컨슈머와 매칭되도록 분배
- rebalance : 컨슈머가 그룹에서 빠지면 매칭되지 않은 파티션을 다른 컨슈머에게 재할당하는 과정

주키퍼



- 카프카 클러스터를 운영하기 위해 사용되는 메타 데이터를 관리
- 카프카 클러스터로 묶인 브로커들은 동일한 경로의 주키퍼 경로로 선언해야 같은 카프카 브로커 묶음이 된다
- 만약 클러스터를 여러 개로 운영한다면 한 개의 주키퍼에 다수의 카프카 클러스터를 연결해서 사용할 수도 있다
  - 이 경우 root node가 아닌 서로 다른 znode에 카프카 클러스터들을 설정하면 된다
    - znode : 주키퍼에서 사용하는 데이터 저장 단위 (tree 구조)

토픽과 파티션



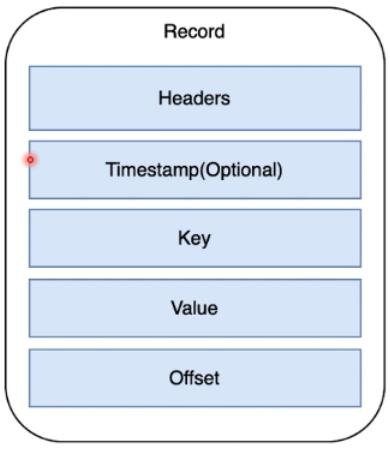
토픽

- 카프카에서 데이터를 구분하기 위해 사용하는 단위
- 토픽은 1개 이상의 파티션을 소유
- 토픽 이름은 변경할 수 없기 때문에 의미를 잘 부여해야 한다

파티션

- 레코드(프로듀서가 보낸 데이터)를 저장하는 곳
- 한 그룹에 속한 컨슈머들은 각 파티션과 매칭되어 레코드를 병렬로 처리
- FIFO 구조
- 파티션의 레코드는 컨슈머가 가져가는 것과 별개로 관리됨 (삭제 x)

레코드



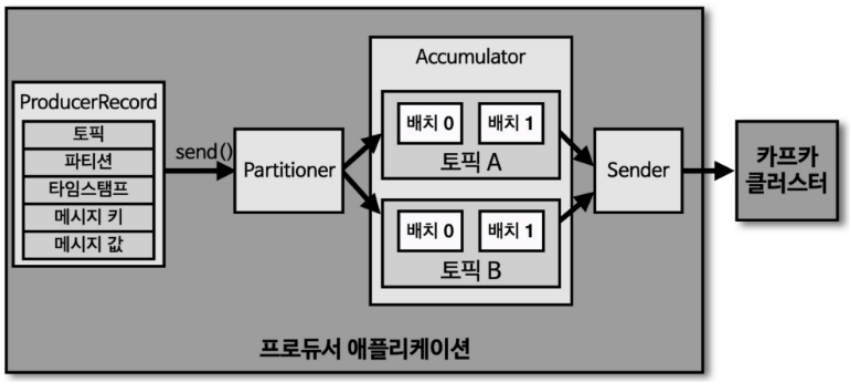
- 한번 저장된 레코드는 **수정 불가**
- 로그 리텐션 기간 또는 용량에 따라서만 삭제됨
- 메시지 키의 해시값을 토대로 파티션을 지정하는데 사용됨 (nullable)
  - 메시지 키와 메시지 값은 직렬화되어 전송됨 (컨슈머 측은 동일한 형태로 역직렬화 필요)
- 오프셋 : 0 이상의 숫자, 직접 지정 불가, 브로커에 저장될 때 이전 레코드 오프셋 + 1 로 설정됨
- 헤더 : 레코드의 추가 정보를 담는 메타데이터 저장소 용도로 사용, 키/값 형태

카프카 클라이언트

프로듀서 API

```
public class SimpleProducer {
    private final static Logger logger = LoggerFactory.getLogger(SimpleProducer.class);
    private final static String TOPIC_NAME = "test";
    private final static String BOOTSTRAP_SERVERS = "my-kafka:9092";

    public static void main(String[] args) {

        Properties configs = new Properties();
        configs.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
        configs.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
        configs.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());

        KafkaProducer<String, String> producer = new KafkaProducer<>(configs);

        String messageValue = "testMessage";
        ProducerRecord<String, String> record = new ProducerRecord<>(TOPIC_NAME, messageValue);
        producer.send(record);
        logger.info("{}", record);
        producer.flush();
        producer.close();
    }
}
```

- 브로커의 특정 토픽의 파티션에 레코드를 전송
- 프로듀서는 데이터를 직렬화하여 리더 파티션을 가지고 있는 카프카 브로커와 직접 통신
- KafkaProducer
  - `send()` : 즉시 전송이 아닌 **배치 전송**
  - `flush()` : 통해 레코드 배치를 브로커로 전송
  - `close()` : producer 인스턴스는 재활용 불가능?



- KafkaProducer 인스턴스가 `send()`를 호출하면 레코드는 파티셔너(**Partitioner**)에서 토픽의 어느 파티션으로 전송될 것인지 정해진다
- 파티셔너에 의해 구분된 레코드는 **Accumulator** 내부의 버퍼로 쌓이다가 배치로 묶여 브로커로 전송된다 → 프로듀서 처리량 향상
- 프로듀서 API는 **UniformStickyPartitioner**, **RoundRobinPartitioner** 2개의 파티션을 제공한다.
  - 메시지 키가 있을 때는 메시지 키의 해쉬값과 파티션을 매칭하여 데이터를 전송한다.
  - UniformStickyPartitioner은 메시지 키가 없을때 파티션에 최대한 동일하게 분배하는 로직이 들어있다.
  - UniformStickyPartitioner는 프로듀서 동작에 특화되어 높은 처리량과 낮은 리소스 사용률을 가진다.
  - RoundRobinPartitioner은 ProducerRecord가 들어오는 대로 파티션을 순회하면서 전송하기 때문에 배치로 묶이는 빈도가 적다.
  - 될 수 있으면 많은 데이터가 배치로 묶여 전송되어야 성능 향상이 되므로 UniformStickyPartitioner가 기본 파티셔너로 설정되었다.
  - UniformStickyPartitioner는 어큐뮬레이터에서 데이터가 배치로 모두 묶일때까지 기다렸다가 배치로 묶인 데이터는 모두 동일한 파티션에 전송함으로써 성능이 더 향상되었다.
- send() 메서드는 Future 객체를 반환한다. 사용자 정의 Callback 클래스를 생성하여 비동기로 프로듀서의 전송 결과를 처리할 수 있다.

```
public class ProducerCallback implements Callback {
    private final static Logger logger = LoggerFactory.getLogger(ProducerCallback.class);
```

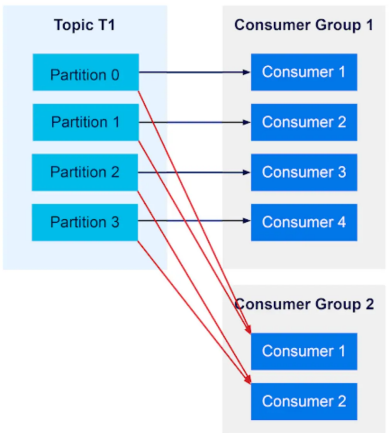
```
    @Override
    public void onCompletion(RecordMetadata recordMetadata, Exception e) {
        if (e != null)
            logger.error(e.getMessage(), e);
        else
            logger.info(recordMetadata.toString());
    }
}
```

```
public class ProducerWithAsyncCallback {
    private final static String TOPIC_NAME = "test";
    private final static String BOOTSTRAP_SERVERS = "my-kafka:9092";

    public static void main(String[] args) {

        Properties configs = new Properties();
        configs.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
        configs.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
        configs.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());

        KafkaProducer<String, String> producer = new KafkaProducer<>(configs);

        ProducerRecord<String, String> record = new ProducerRecord<>(TOPIC_NAME, "Pangyo", "Pangyo");
        producer.send(record, new ProducerCallback());

        producer.flush();
        producer.close();
    }
}
```

- 데이터의 순서가 중요한 경우 동기로 전송 결과를 받는 것이 적절하다

컨슈머 API

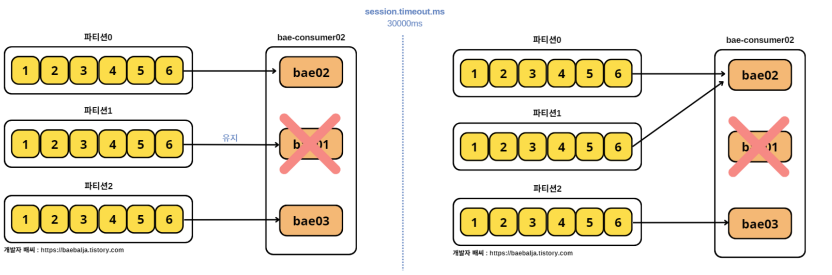
```
public class SimpleProducer {
    private final static Logger logger = LoggerFactory.getLogger(SimpleProducer.class);
    private final static String TOPIC_NAME = "test";
    private final static String BOOTSTRAP_SERVERS = "my-kafka:9092";

    public static void main(String[] args) {

        Properties configs = new Properties();
        configs.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
        configs.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
        configs.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());

        KafkaProducer<String, String> producer = new KafkaProducer<>(configs);

        String messageValue = "testMessage";
        ProducerRecord<String, String> record = new ProducerRecord<>(TOPIC_NAME, messageValue);
        producer.send(record);
        logger.info("{}", record);
        producer.flush();
        producer.close();
    }
}
```



컨슈머 그룹

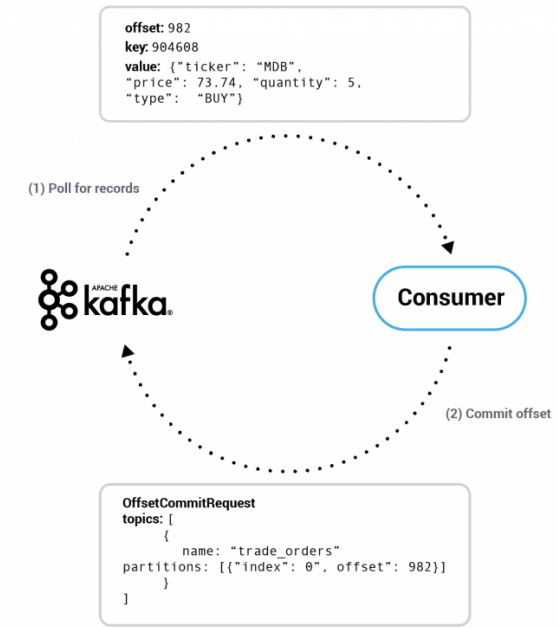
- 컨슈머 그룹은 토픽을 구독한다
- 컨슈머 그룹은 다른 컨슈머 그룹과 격리되는 특징을 가지고 있다. 토픽의 데이터가 어디에 적재되는지, 어떻게 처리되는지 파악하고 컨슈머 그룹으로 따로 나눌 수 있는 것은 최대한 나누는 것이 좋다.
- 1개의 파티션은 최대 1개의 컨슈머에 할당 가능하다
- 1개의 컨슈머는 여러 파티션에 할당될 수 있다
- 이로 인해, 컨슈머 그룹에 파티션의 개수보다 많은 컨슈머가 존재할 경우 스레드만 차지하게 된다

리밸런싱



- 컨슈머 그룹에 속한 일부 컨슈머에 장애가 발생하면 리밸런싱, 장애가 발생한 컨슈머에 할당된 파티션은 장애가 발생하지 않은 컨슈머에 소유권이 넘어간다. 이를 **리밸런싱(rebalancing)** 이라고 부른다
  - 리밸런싱 발생 상황 : 신규 컨슈머 추가, 기존 컨슈머 제외
  - 리밸런싱은 컨슈머가 데이터를 처리하는 도중에 언제든지 발생할 수 있으므로 데이터 처리 중 발생한 리밸런싱에 대응하는 코드를 작성해야 한다
- 리밸런싱이 진행되는 동안 해당 컨슈머 그룹의 컨슈머들은 토픽의 데이터를 읽을 수 없다.
- **그룹 조정자(group coordinator)**는 리밸런싱을 발동시키는 역할을 하는데 컨슈머 그룹의 컨슈머가 추가되고 삭제될 때를 감지한다. 카프카 브로커 중 한대가 그룹 조정자의 역할을 수행한다.

커밋



- 컨슈머는 카프카 브로커로부터 데이터를 어디까지 가져갔는 지 **커밋(commit)**을 통해 기록한다.
- 특정 토픽의 파티션을 어떤 컨슈머 그룹이 몇 번째 가져갔는지 카프카 브로커 내부에서 사용되는 내부 토픽(__consumer_offsets)에 기록된다
- __consumer_offsets 토픽에 어느 레코드까지 읽어갔는 지 오프셋 커밋이 기록되지 못했다면 데이터 처리 중복이 발생할 수 있기 때문에, 컨슈머 애플리케이션은 오프셋 커밋이 정상적으로 처리됐는지 검증해야 한다.
- **명시적 오프셋 커밋**
  - 명시적으로 어피셋을 커밋하려면 데이터 처리를 완료하고 commitSync() 메서드를 호출하면 된다.
  - `commitSync()` : 인자를 전달하지 않을 경우 poll() 메서드 호출 시 반환된 레코드의 가장 마지막 오프셋을 기준으로 커밋을 수행한다. 인자를 전달하여 오프셋을 지정하여 커밋을 수행할 수 있다.커밋이 정상적으로 처리되었는지 응답을 기다리므로 컨슈머의 처리량에 영향을 끼친다.
  - `commitAsync()` : 비동기적으로 오프셋 커밋을 수행한다. 하지만 비동기 커밋은 커밋 요청이 실패했을 경우 현재 처리중인 데이터의 순서를 보장하지 않으며 데이터 중복 처리가 발생할 수 있다. OffsetCommitCallback 인터페이스 구현체를 전달하여 콜백함수를 전달할 수 있다.
- **비명시적 오프셋 커밋**
  - poll() 메서드가 수행될 때 일정 간격마다 자동으로 커밋하는 방식
  - 커밋 관련 코드를 작성할 필요가 없어 간편하지만, poll() 메서드 호출 이후 리밸런싱 혹은 강제 종료 발생 시 데이터가 중복 또는 유실될 수 있는 취약한 구조를 가지고 있다.
  - 따라서, 데이터 중복이나 유실을 허용하지 않는 서비스라면 자동 커밋을 사용해서는 안된다.

리밸런스 리스너

```
public class ConsumerWithRebalanceListener {
    private final static Logger logger = LoggerFactory.getLogger(ConsumerWithRebalanceListener.class);
    private final static String TOPIC_NAME = "test";
    private final static String BOOTSTRAP_SERVERS = "my-kafka:9092";
    private final static String GROUP_ID = "test-group";

    private static KafkaConsumer<String, String> consumer;
    private static Map<TopicPartition, OffsetAndMetadata> currentOffset;

    public static void main(String[] args) {
        Properties configs = new Properties();
        configs.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
        configs.put(ConsumerConfig.GROUP_ID_CONFIG, GROUP_ID);
        configs.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());
        configs.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());
        configs.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);

        consumer = new KafkaConsumer<>(configs);
        consumer.subscribe(Arrays.asList(TOPIC_NAME), new RebalanceListener());

        while (true) {
            ConsumerRecords<String, String> records = consumer.poll(Duration.ofSeconds(1));
            currentOffset = new HashMap<>();

            for (ConsumerRecord<String, String> record : records) {
                logger.info("{}", record);
                currentOffset.put(
                        new TopicPartition(record.topic(), record.partition()),
```

```
                        new OffsetAndMetadata(record.offset() + 1, null));
                consumer.commitSync(currentOffset);
            }
        }
    }

    private static class RebalanceListener implements ConsumerRebalanceListener {
            public void onPartitionsAssigned(Collection<TopicPartition> partitions) {
                logger.warn("Partitions are assigned : " + partitions.toString());
            }

            public void onPartitionsRevoked(Collection<TopicPartition> partitions) {
                logger.warn("Partitions are revoked : " + partitions.toString());
                consumer.commitSync(currentOffsets);
            }
    }
}
```

- poll() 메서드를 통해 반환받은 데이터를 모두 처리하기 전에 리밸런스가 발생하면 데이터가 중복 처리될 수 있다. 커밋되지 않았기 때문이다.
- 리밸런스 발생 시 데이터를 중복 처리하지 않게 하기 위해서는 리밸런스 발생 시 처리한 데이터를 기준으로 커밋을 시도해야 한다.
- 카프카 라이브러리는 리밸런스 발생을 감지하는 `ConsumerRebalanceListener` 인터페이스를 지원한다.
  - `onPartitionAssigned()` : 리밸런스가 끝난 뒤 파티션 할당이 완료되면 호출되는 메서드
  - `onPartitionRevoked()` : 리밸런스가 시작되기 직전에 호출되는 메서드

파티션 할당 컨슈머

```
public class ConsumerWithExactPartition {
    private final static Logger logger = LoggerFactory.getLogger(ConsumerWithExactPartition.class);
    private final static String TOPIC_NAME = "test";
    private final static int PARTITION_NUMBER  = 0;
    private final static String BOOTSTRAP_SERVERS = "my-kafka:9092";

    public static void main(String[] args) {
        Properties configs = new Properties();
        configs.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
        configs.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());
        configs.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());

        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(configs);
        consumer.assign(Collections.singleton(new TopicPartition(TOPIC_NAME, PARTITION_NUMBER)));
        while (true) {
            ConsumerRecords<String, String> records = consumer.poll(Duration.ofSeconds(1));
            for (ConsumerRecord<String, String> record : records) {
                logger.info("record:{}", record);
            }
        }
    }
}
```

- 컨슈머를 운영할 때 subscribe() 메서드를 사용하여 토픽을 구독하는 형태 외에, 직접 파티션을 컨슈머에 명시적으로 할당하여 운영할 때는 assign() 메서드를 사용할 수 있다.
- subscribe() 메서드를 사용할 때와 다르게 직접 컨슈머가 특정 토픽의 특정 파티션에 할당되므로 리밸런싱하는 과정이 없다.

컨슈머 내부 구조

- 컨슈머 애플리케이션이 실행되면 내부에서 **Fetcher** 인스턴스가 생성되어 poll() 메서드가 호출되기 전에 미리 레코드들을 내부 큐로 가져온다.
- 이후 사용자가 명시적으로 poll() 메서드를 호출하면 컨슈머는 내부 큐에 있는 레코드들을 반환받아 처리를 수행한다.

컨슈머의 안전한 종료

```
try {
        while (true) {
            ConsumerRecords<String, String> records = consumer.poll(Duration.ofSeconds(1));
            for (ConsumerRecord<String, String> record : records) {
                logger.info("{}", record);
            }
            consumer.commitSync();
        }
} catch (WakeupException e) {
    logger.warn("Wakeup consumer");
} finally {
    logger.warn("Consumer close");
    consumer.close();
}
```

- 정상적으로 종료되지 않은 컨슈머는 세션 타임아웃이 발생할때까지 컨슈머 그룹에 남게 된다.
- 실제로는 종료되었지만 더는 동작하지 않는 컨슈머가 존재하기 때문에 파티션의 데이터는 소모되지 못하고 컨슈머 랙이 늘어나며 데이터 처리 지연이 발생하게 된다.
- 컨슈머를 안전하게 종료하기 위해 KafkaConsumer 클래스는 `wakeup()` 메서드를 지원한다.
- wakeup() 메서드가 실행된 이후 poll() 메서드가 호출되면 `WakeupException` 예외가 발생한다.
- WakeupException 예외를 받은 뒤에는 데이터 처리를 위해 사용한 자원들을 해제하면 된다.
- 마지막에 `close()` 메서드가 호출되면 카프카 클러스터에 해당 컨슈머는 더는 동작하지 않는다는 것을 명시적으로 알려주므로 컨슈머 그룹에서 이탈되고 나머지 컨슈머들이 파티션을 할당받게 된다.
- wakeup() 메서드는 셧다운 훅에서 호출하는 방식으로 활용할 수 있다.

어드민 API

```
public class KafkaAdminClient {
    private final static Logger logger = LoggerFactory.getLogger(KafkaAdminClient.class);
    private final static String BOOTSTRAP_SERVERS = "my-kafka:9092";

    public static void main(String[] args) throws Exception {

        Properties configs = new Properties();
```

```
        configs.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "my-kafka:9092");
        AdminClient admin = AdminClient.create(configs);
        logger.info("== Get broker information");
        for (Node node : admin.describeCluster().nodes().get()) {
            logger.info("node : {}", node);
            ConfigResource cr = new ConfigResource(ConfigResource.Type.BROKER, node.idString());
            DescribeConfigsResult describeConfigs = admin.describeConfigs(Collections.singleton(cr));
            describeConfigs.all().get().forEach((broker, config) -> {
                config.entries().forEach(configEntry -> logger.info(configEntry.name() + "= " + configEntry.value()));
            });
        }

        logger.info("== Get default num.partitions");
        for (Node node : admin.describeCluster().nodes().get()) {
            ConfigResource cr = new ConfigResource(ConfigResource.Type.BROKER, node.idString());
            DescribeConfigsResult describeConfigs = admin.describeConfigs(Collections.singleton(cr));
            Config config = describeConfigs.all().get().get(cr);
            Optional<ConfigEntry> optionalConfigEntry = config.entries().stream().filter(v -> v.name().equals("num.partitions")).findFirst();
            ConfigEntry numPartitionConfig = optionalConfigEntry.orElseThrow(Exception::new);
            logger.info("{}", numPartitionConfig.value());
        }

        logger.info("== Topic list");
        for (TopicListing topicListing : admin.listTopics().listings().get()) {
            logger.info("{}", topicListing.toString());
        }

        logger.info("== test topic information");
        Map<String, TopicDescription> topicInformation = admin.describeTopics(Collections.singletonList("test")).all().get();
        logger.info("{}", topicInformation);

        logger.info("== Consumer group list");
        ListConsumerGroupsResult listConsumerGroups = admin.listConsumerGroups();
        listConsumerGroups.all().get().forEach(v -> {
            logger.info("{}", v);
        });

        admin.close();
    }
}
```

- 카프카 클라이언트에서는 내부 옵션들을 설정하거나 조회하기 위해 AdminClient 클래스를 제공한다.
- AdminClient 클래스를 활용하면 클러스터의 옵션과 관련된 부분을 자동화할 수 있다.
  - 구독하는 토픽의 파티션 개수만큼 스레드를 생성하고 싶을 때
  - 특정 토픽의 데이터양이 늘어남을 감지하고 해당 토픽의 파티션을 늘릴 때
- 어드민 API를 활용할 때 클러스터의 버전과 클라이언트의 버전을 맞춰서 사용해야 한다.

카프카 스트림즈

- 토픽에 적재된 상태기반(Stateful) 또는 비상태기반(Stateless)으로 실시간 변환하여 다른 토픽에 적재하는 라이브러리
- 대안 : Apache Spark, Apache Flink, Apache Storm, FluentD
- 카프카 스트림즈를 사용해야 하는 이유
  - 카프카에서 공식적으로 지원하는 라이브러리
  - 카프카 버전이 오를 때마다 스트림즈 자바 라이브러리도 같이 릴리즈됨
  - 카프카 클러스터와 완벽하게 호환되면서 편리 기능을 제공
  - 스트림즈 애플리케이션 또는 브로커의 장애가 발생하더라도 Exactly Once를 보장하는 Fault Tolerant System을 가지고 있어 안정성이 매우 뛰어남
  - 결론 : **카프카 클러스터를 운영하면서 실시간 스트림 처리를 해야하는 필요성이 있는 경우 1순위로 고려**하는 것이 좋음