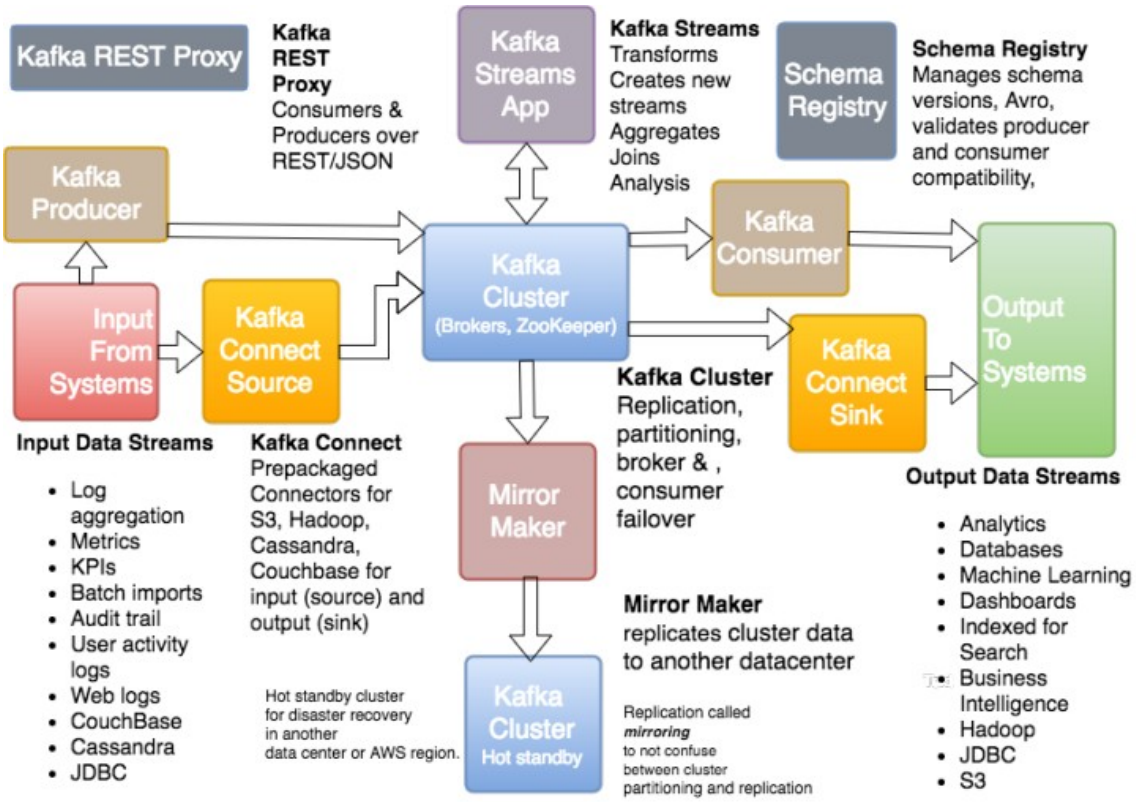
kafka streams & connect 이론 및 실습

<목차>

kafka streams 이론

kafka connect 이론

kafka streams 실습



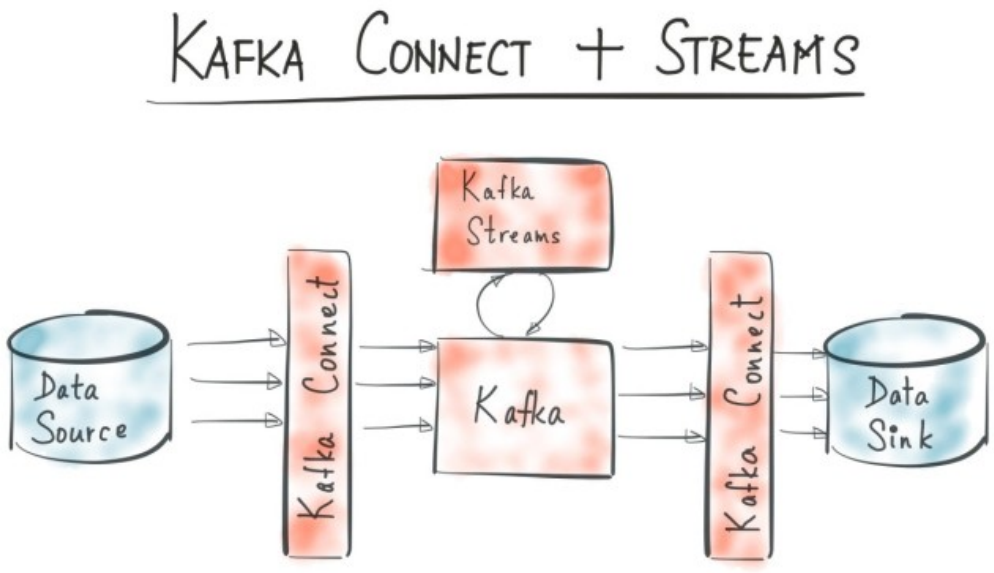
▼ Kafka Streams 이론

1. Kafka Streams란?

> 💡 Kafka Streams
>
> : 토픽에 적재된 데이터를 실시간으로 처리(데이터 변환 및 분석)하여 다른 토픽에 적재하는 라이브러리.

→ 데이터 처리?

ex) 데이터를 변환, 필터링, 집계 등…



- 유사 애플리케이션

카프카의 스트림 데이터 처리를 위해 아파치 스파크(Apache Spark), 아파치 플링크(Apache Flink), 플루언트디(Fluentd)와 같은 다양한 오픈소스 애플리케이션이 존재.

2. 왜 Kafka Streams를 사용하는가?

유사 애플리케이션 다수 존재,

그런데도 Kafka Streams를 사용하는 이유는?

Kafka Streams는 카프카에서 '공식적'으로 지원하는 라이브러리.

따라서 매번 kafka 버전이 출시될때마다 streams도 함께 업데이트.

아래와 같은 장점을 가짐.

- 카프카 스트림즈는 카프카 클러스터와 완벽하게 호환.
- 스트림 처리에 필요한 편리한 기능들(신규 토픽 생성, 상태 저장, 데이터 조인 등)을 제공함.
- 장애 허용 시스템을 가짐(스트림즈 애플리케이션 또는 카프카 브로커에 장애가 발생하더라도 데이터 처리를 정확히 한번만 할 수 있도록 하는 기능)

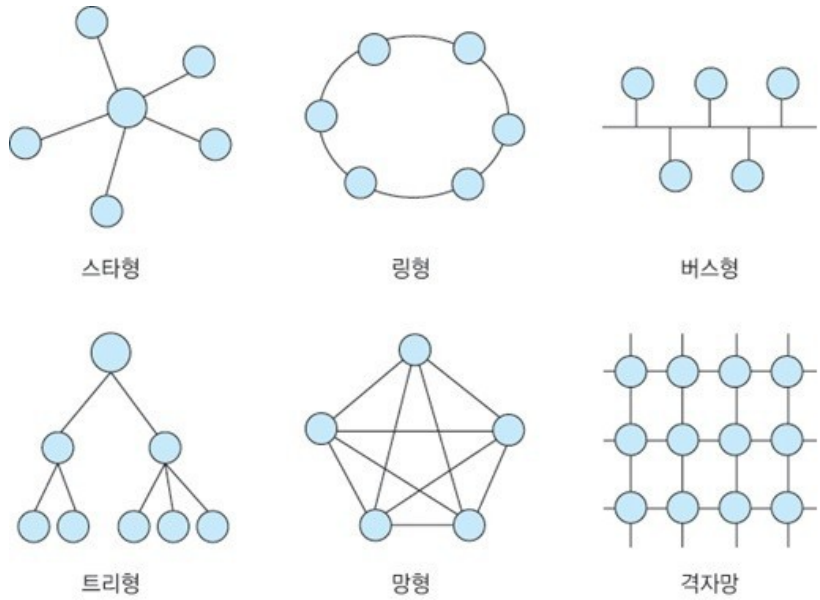
3. 토폴로지란?

카프카 스트림즈의 구조와 사용방법을 알기 위해서는 토폴로지와 관련된 개념을 알아야 한다.

토폴로지(Topology)

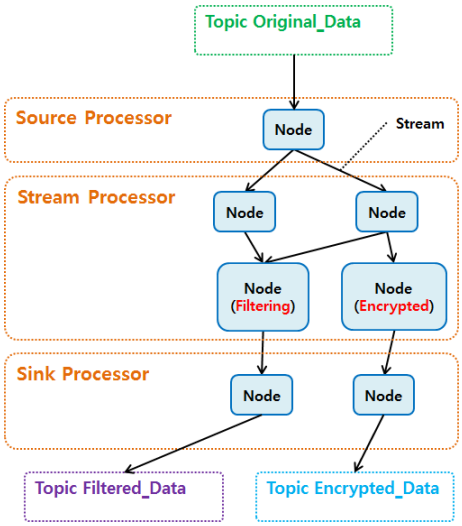
: 데이터의 흐름을 표현하는 2개 이상의 **노드**들과 **선**으로 이루어진 집합.

- **토폴로지의 종류**(링형, 트리형, 성형 등…)



→ 스트림즈에서는 **트리형**과 유사함.

- **토폴로지의 구성요소**



- **노드** : 프로세서(Processor)를 의미 → 프로세서는 3가지 종류로 나뉨
  - 소스 프로세서 (Source Processor)

    데이터를 처리하기 위해 최초로 선언해야 하는 노드.

    하나 이상의 토픽에서 **데이터를 가져오는 역할**을 한다.
  - 스트림 프로세서 (Stream Processor)

    다른 프로세서가 반환한 **데이터를 처리하는 역할**을 한다.
  - 싱크 프로세서 (Sink Processor)

    데이터 전달을 위해 마지막에 선언해야 하는 노드.

데이터를 특정 카프카 토픽으로 저장하는 역할을 한다.

streams로 처리된 데이터의 최종 종착지.

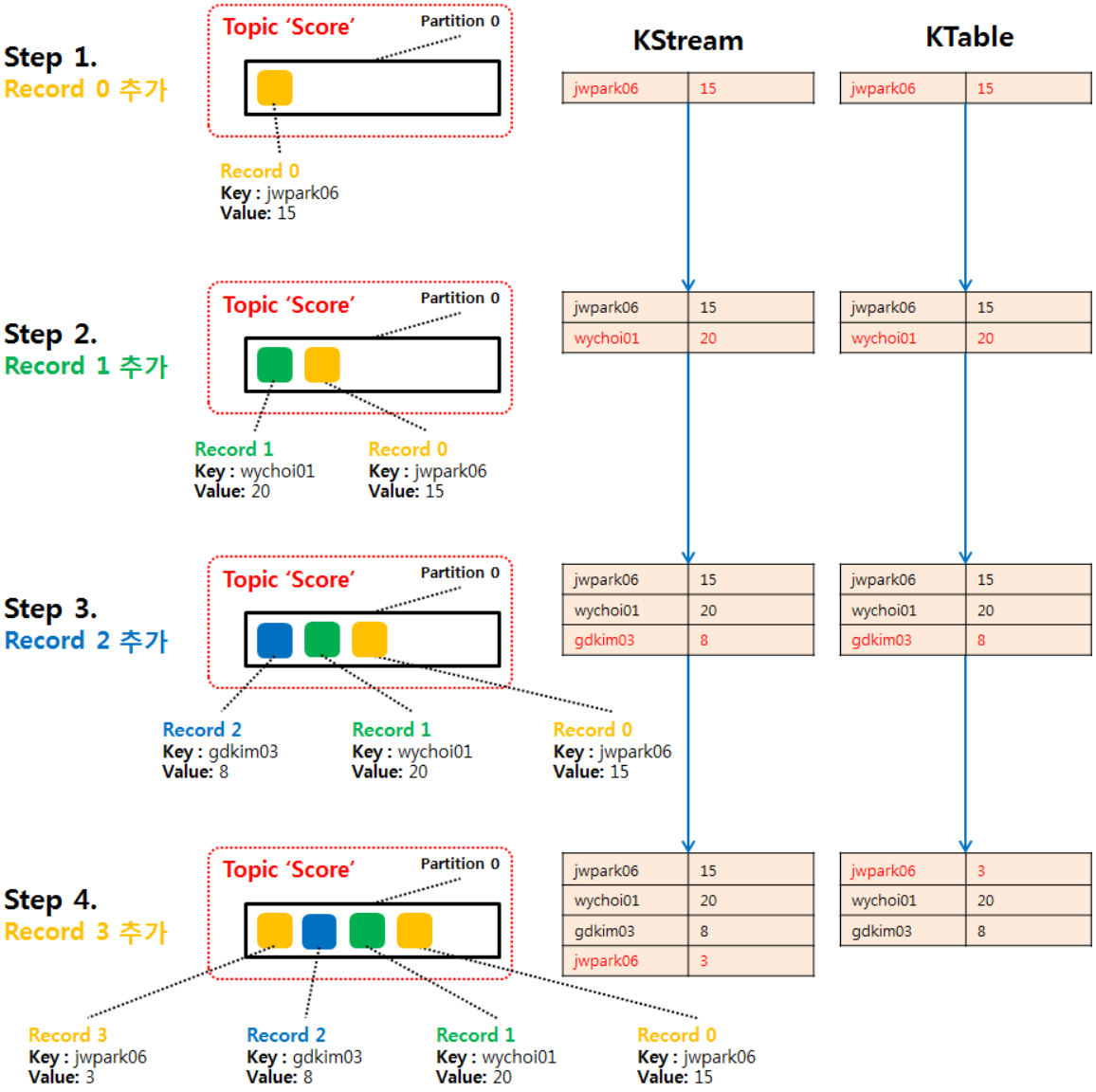
- **선** : 스트림(Stream)을 의미
  - 여기서 스트림은 '토픽의 데이터'를 뜻한다.

4. 데이터 처리 구현 방식 2가지

- **스트림즈DSL**(Domain Specific Language)과 **프로세서 API** 2가지 방법으로 개발 가능
  - 스트림즈 DSL은 스트림 프로세싱에 쓰일만한 다양한 기능들을 자체 API로 만들어 놓았기 때문에 대부분의 변환 로직을 어렵지 않게 개발 가능.

4-1. 스트림즈DSL로 구현

1) 레코드 흐름의 추상화 개념 3가지 (KStream, KTable, GlobalKTable)



- KStearm

  :레코드의 흐름을 표현한 것.

  메시지 키와 **메시지 값**으로 구성.

  →KStream으로 데이터를 조회하면, 토픽에 존재하는 **모든 레코드** 출력됨.

- KTable

  :메시지 키를 기준으로 묶어서 표현한 것.

  하나의 메시지 키에는 하나의 메시지 값만 존재.

  → KTable로 데이터를 조회하면, 하나의 메시지 키를 기준으로 **가장 최신에 추가된 레코드** 데이터가 출력.

- GlobalKTable
  - KTable과 공통점
    - KTable과 동일하게 메시지 키를 기준으로 묶어서 표현.
  - KTable과 차이점
    - KTable로 선언된 토픽은 1개 파티션이 1개 태스크에 할당되어 사용.

        - GlobalKTable로 선언된 토픽은 모든 파티션 데이터가 각 태스크에 할당되어 사용.
    - 사용 예시

      EX) **KStream과 KTable의 데이터 조인**

      KStream과 KTable은 반드시 코파티셔닝(co-partitoning)되어 있어야함.

      만약 조인을 수행하려는 토픽들이 코파티셔닝이 되어 있지 않을 경우 조인을 수행하려면

      → **리파티셔닝(repartitioning)**과정을 거치거나

      → 코파티셔닝 되지 않은 KStream과 KTable을 조인해서 사용하고싶다면, KTable을 **GlobalKTable로 선언**하여 사용하면 됨!

      > 💡 코파티셔닝, 리파티셔닝?
      >
      > - 코파티셔닝(co-partitioning) : 조인을 하는 2개 데이터의 파티션 개수와 파티셔닝 전략을 동일하게 맞추는 작업.
      > - 리파티셔닝(repartitioning) : 새로운 토픽에 새로운 메시지 키를 가지도록 재배열하는 과정.

2) 스트림즈DSL 주요 옵션

- 필수 옵션
    - **Bootstrap.servers** : 프로듀서가 데이터를 전송할 대상 카프카 클러스터에 속한 브로커의 **호스트 이름:포트**를 1개 이상 작성한다.
    - **application.id** : 스트림즈 애플리케이션을 구분하기 위한 고유한 아이디 설정.
- 선택 옵션
    - **default.key.serde** : 레코드의 메시지 키를 직렬화, 역직렬화하는 클래스를 지정. 기본값은 바이트 직렬화, 역직렬화 클래스.
    - **default.value.serde** : 레코드의 메시지 값을 직렬화, 역직렬와하는 클래스를 지정. 기본값은 바이트 직렬화, 역직렬화 클래스.
    - **num.stream.threads** : 스트림 프로세싱 실행 시 실행된 스레드 개수를 지정. 기본값은 1.
    - **state.dir** : 상태기반 데이터 처리를 할 때 데이터를 저장할 디렉토리를 지정. 기본값은 /tmp/kafka-streams.

4-2. 프로세서 API로 구현

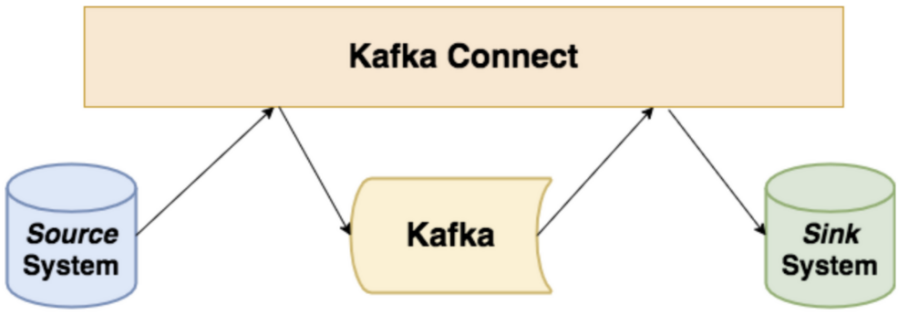
- 프로세서 API는 스트림즈 DSL 보다 투박한 코드를 가지지만 토폴로지를 기준으로 데이터를 처리한다는 관점에서 동일한 역할을 함.
- 스트림즈 DSL은 `데이터처리`, `분기`, `조인` 을 위한 다양한 메소드들을 제공하지만 추가적인 `상세 로직의 구현` 이 필요하다면 프로세서 API를 구현할 수 있음
- 프로세서 API에서는 다만 스트림즈 DSL에서 사용했던 `KStream`, `KTable`, `GlobalKTable` 의 개념이 없다.

▼ Kafka connect 이론

1. Kafka connect란?

kafka connect

: kafka connect는 카프카 오픈소스에 포함된 툴 중 하나로 데이터 파이프라인 생성 시 반복 작업을 줄이고 효율적인 전송을 이루기 위한 애플리케이션.



2. Kafka connect의 도입 배경

- 카프카(kafka)는 프로듀서(Producer)와 컨슈머(Consumer)를 통해 다양한 외부 시스템 데이터를 주고 받으며 메세지 파이프라인 아키텍쳐를 구성한다.
- 하지만 이러한 **파이프라인을 매번 구성하며 프로듀서와 컨슈머를 개발**하는 것은 쉽지 않다.
  - 반복적인 파이프라인 생성 작업이 있을 때 매번 프로듀서, 컨슈머 애플리케이션을 개발하고 배포, 운영해야 하기 때문
- 커넥트는 **특정한 작업 형태를 템플릿으로 만들어놓은 커넥터(connector)를 실행**함으로써 **반복 작업을 줄일 수 있다**.
  - 파이프라인 생성 시 자주 반복되는 값들(토픽 이름, 파일 이름, 테이블 이름 등)을 파라미터로 받는 커넥터를 코드로 작성하면 이후에 코드를 재작성할 필요가 없기 때문
- 따라서 카프카 메세지로 파이프라인 아키텍쳐를 보다 적은 비용으로 보다 쉽게 구현하게 도와주는 것이 카프카 커넥트(Kafka connect)이다.

3. kafka connect의 특징

kafka connect는 대표적으로 다음과 같은 5가지 특징을 가지고 있다.

- 데이터 중심 파이프라인

  : 카프카 커넥트를 이용해 카프카로 데이터를 보내거나, 카프카로 데이터를 가져옴
- 유연성과 확장성

  : 커넥트는 테스트를 위한 단독 모드(standalone mode)와 대규모 운영 환경을 위한 분산 모드(distributed mode)를 제공
- 재사용성과 기능 확장

  : 커넥트는 기존 커넥터를 활용할 수도 있고 운영 환경에서의 요구사항에 맞춰 확장이 가능
- 편리한 운영과 관리

: 카프카 커넥트가 제공하는 REST API로 빠르고 간단하게 커넥트 운영 가능

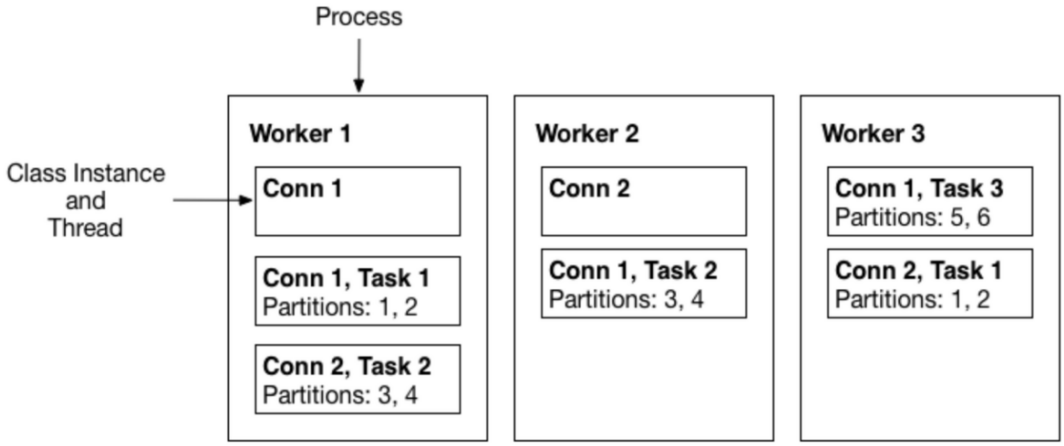
- 장애 및 복구

  : 카프카 커넥트를 분산 모드로 실행하면 워커 노드의 장애 상황에도 메타데이터를 백업함으로써 대응 가능하며 고가용성 보장

4. kafka connect 의 핵심 개념

카프카 커넥트는 내부적으로 **워커** 단위로 이루어져 있다.

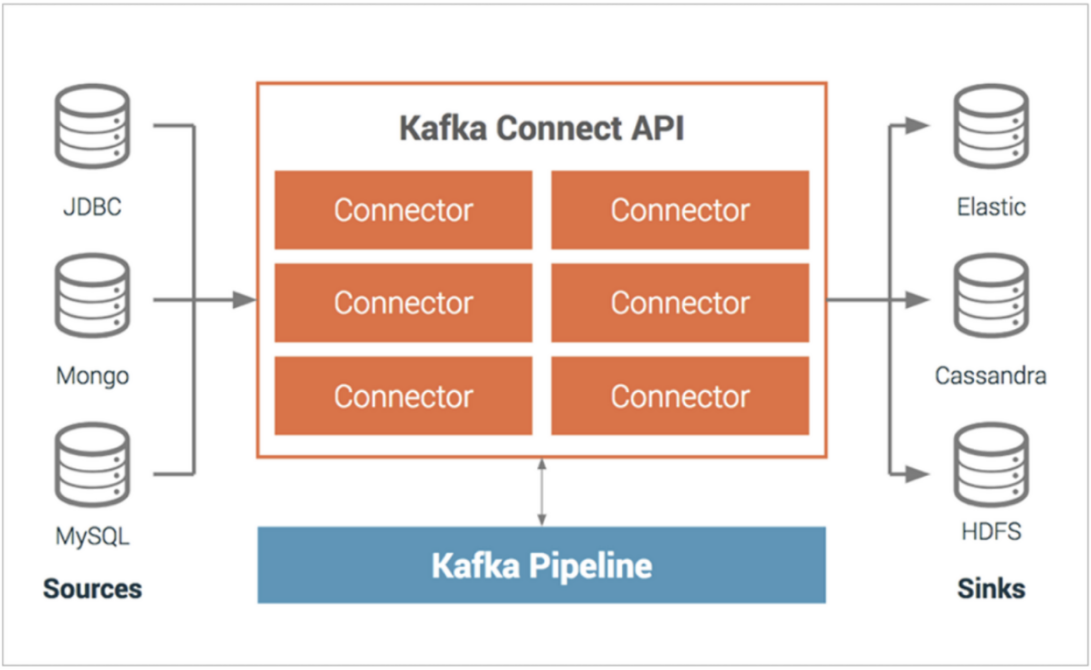
- **Worker(워커)** : 카프카 커넥트 프로세스가 실행되는 서버 또는 인스턴스. 워커의 내부는 **커넥터**와 **태스크**로 이루어짐.



사용자가 커넥트에 커넥터 생성 명령을 내리면 커넥트는 **워커** 내부에 **커넥터**와 **태스크**를 생성한다.

- **커넥터(Connector)**

  : 커넥트 내부에 존재하는 실제 메세지 파이프라인. 데이터를 어디에서 어디로 복사하는지의 작업을 정의하고 관리하는 역할을 한다.

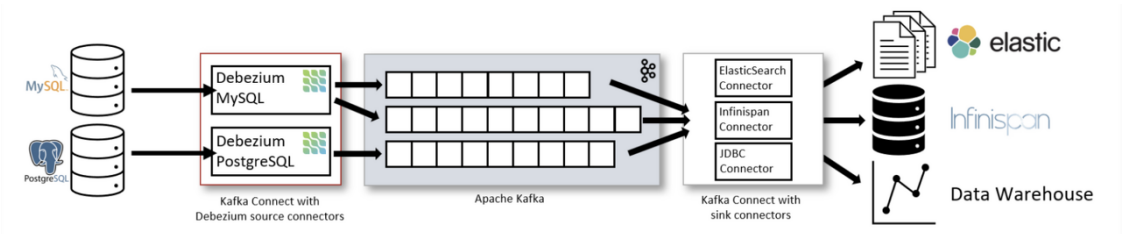


카프카 클러스터를 기준으로 양쪽에 위치하는데, 각각의 이름과 역할이 다름.

- **소스 커넥터** : 프로듀서 역할을 하는 커넥터(다른 시스템에서 데이터를 가져오는)
- **싱크 커넥터** : 컨슈머 역할을 하는 커넥터(다른 시스템으로 데이터를 내보내는)

  ex) MySQL로부터 데이터를 가져오는 커넥터 = MySQL 소스 커넥터

  ElasticSearch로 데이터를 내보내는 커넥터 = ElasticSearch 싱크 커넥터



<커넥터 사용방법>

- 커넥터 플러그인 (오픈 소스 커넥터) 가져와서 사용하기
- 직접 커넥터 플러그인을 만들어서 사용하기

→ 이미 굉장히 많은 커넥터들이 공개되어 있다. (https://www.confluent.io/hub/)에서 검색해서 사용하면 된다.

태스크는 커넥터에 종속되는 개념으로, 실질적인 데이터 처리를 함.

- **Task (태스크)**

  : 커넥터가 정의한 작업을 직접 수행.

사용자가 커넥터를 사용하여 파이프라인을 생성할 때 컨버터와 트랜스폼 기능을 '옵션'으로 추가할 수 있다.

- **Converter(컨버터)** : 데이터 처리를 하기 전에 스키마를 변경하도록 도와준다.

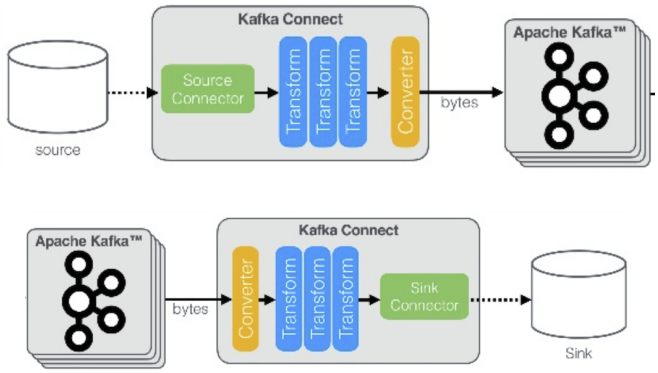
  JsonConverter, StringConverter, ByteArrayConverter 를 지원하고, 커스텀 컨버터를 작성해서 사용할 수도 있다.

- **Transform(트랜스폼)** : 데이터 처리 시 각 메시지 단위로 데이터를 간단하게 변환하기 위한 용도로 사용된다.

  예를들어, JSON 데이터를 커넥터에서 사용할 때 트랜스폼을 사용하면 특정 키를 삭제하거나 추가할 수 있다. 기본 제공 트랜스폼으로 Cast, Drop, ExtractField 등 존재.

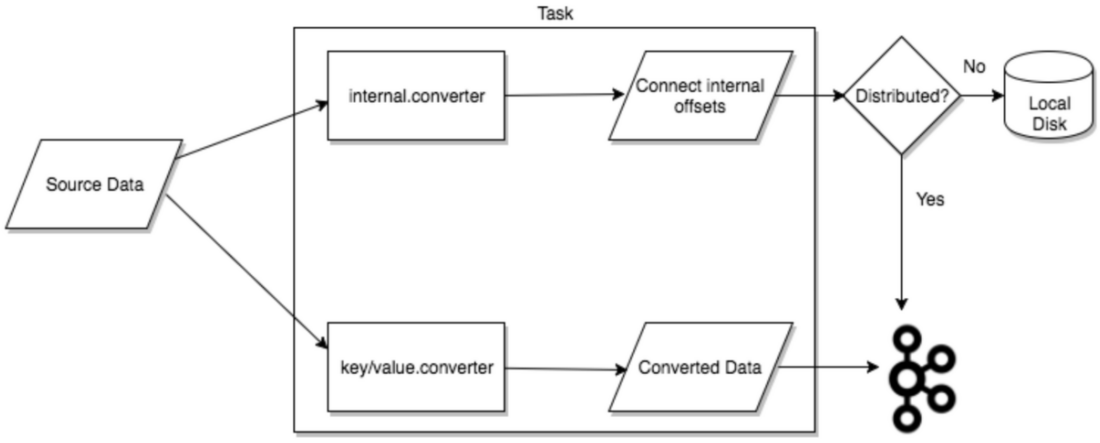
<정리>

- 커넥트의 내부 구성
  - 워커
    - 커넥터
      - 소스 커넥터
      - 싱크 커넥터
    - 테스크
    - 컨버터
    - 트랜스폼



5. kafka connect 내부 동작

데이터 처리가 호출되면 워커 내부의 커넥터들에 의해 테스크들이 생성되고, 파이프라인이 구동된다. 그 후 테스크 내부에서는 외부 시스템의 메시지를 변화시키는 컨버터 처리가 발생한 후 전송이 일어나는 원리이다.



<소스 커넥터 실행시 동작을 표현한 그림>

6. kafka connect 2가지 모드

카프카 커넥트는 앞서 보았던 '워커의 개수'에 따라 모드가 정해짐

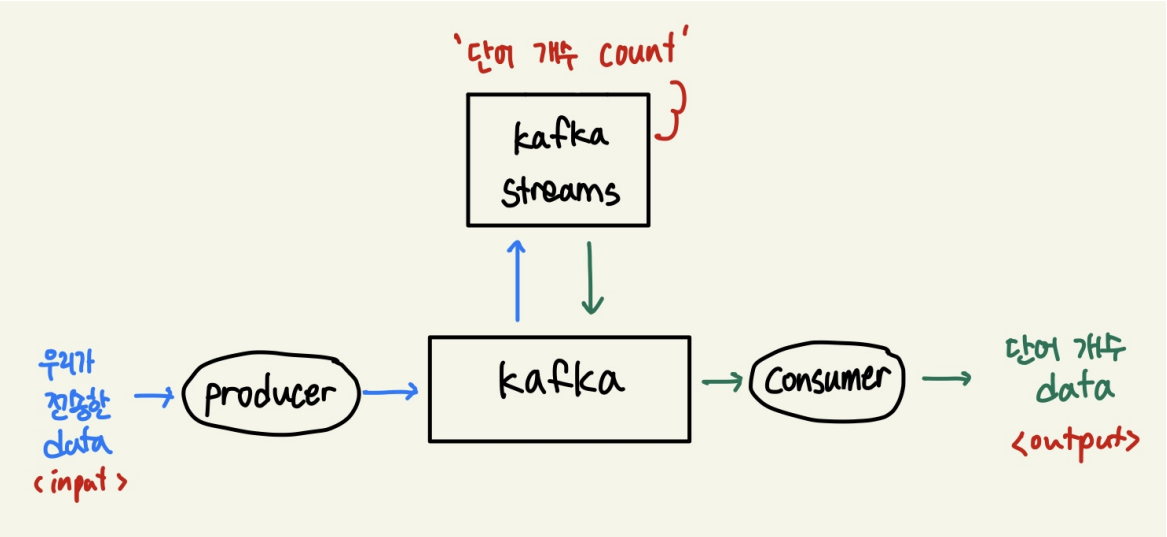
- **단일모드 커넥트** : 단일 워커로 이루어진 카프카 커넥트
  - 테스트 및 일회성 환경에 적합.
- **분산모드 커넥트** : 다수의 워커로 이루어진 카프카 커넥트
  - 실제 운영 환경에 적합.
  - REST API를 사용할 수 있음.
  - 커넥트의 메타 데이터 정보를 저장함에 있어 카프카 클러스트를 이용할 수 있기에 장애에 유연하게 대처가 가능함.

7. kafka connect가 제공하는 REST API

API 옵션	설명
GET /	커넥트 버전과 클러스터 ID 확인
GET /connectors	커넥터 리스트 확인
GET /connectors/커넥터 이름	커넥터 상세 내용 확인
GET /connectors/커넥터 이름/config	커넥터 config 정보 확인
GET /connectors/커넥터 이름/status	커넥터 상태 확인
PUT /connectors/커넥터 이름/config	커넥터 config 설정
PUT /connectors/커넥터 이름/pause	커넥터 일시 중지
PUT /connectors/커넥터 이름/resume	커넥터 다시 시작
DELETE /connectors/커넥터 이름	커넥터 삭제
GET /connectors/커넥터 이름/tasks	커넥터의 태스크 정보 확인
GET /connectors/커넥터 이름/tasks/태스크 ID/status	커넥터의 특정 태스크 상태 확인
POST /connectors/커넥터 이름/tasks/태스크 ID/restart	커넥터의 특정 태스크 재시작

▼ Kafka Streams 실습

[실습 소개]



<실습 실행용 - colab>

🔗colab : https://colab.research.google.com/drive/1G3SkUpm-k4_Ad_lRzSSGyPuQGexK_Yyi?usp=sharing

→터미널 총 5개 필요 (데몬으로 실행 시 3개 필요)

- 주키퍼 실행 (터미널 1)
- kafka 실행 (터미널 2)
- 데모 어플리케이션 실행 (터미널 3(1))
- 프로듀서 실행 (터미널 4(2))
- 컨슈머 실행 (터미널 5(3))

[실습 코드]

step 0) aws ec2 인스턴스에 접속 (모든 터미널에서 수행)

"이미 접속된 상태 or kafka를 로컬에 설치 후 실행하는 경우 step 1 부터 시행."

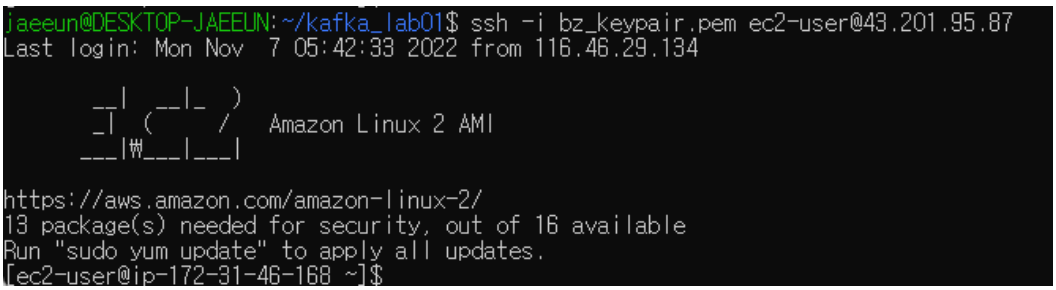
- 접속 방법
  - 키페어 다운로드
  - 키페어 저장된 위치로 이동

```
$ cd kafka_lab01
```

  - 해당 위치에서 다음 명령어로 인스턴스 접속

```
$ ssh -i bz_keypair.pem ec2-user@43.201.95.87
```

  - 접속 성공시 화면



step 1) kafka 디렉터리로 이동 (모든 터미널에서 수행)

```
$ cd kafka_2.12-3.3.1
```

step 2) 주키퍼 & kafka 서버 실행

(터미널 1)

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

(터미널 1)

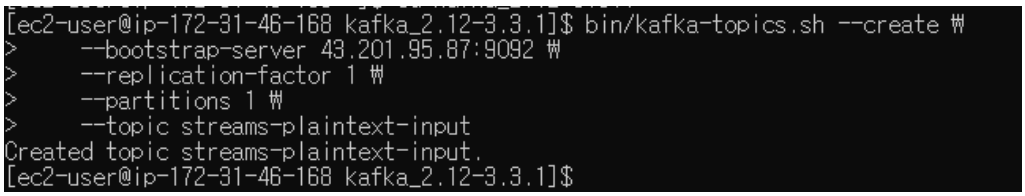
```
$ bin/kafka-server-start.sh -daemon config/server.properties
```

step 3-1) 2개의 토픽 생성

- streams-plaintext-input : input 토픽

  (터미널 1)

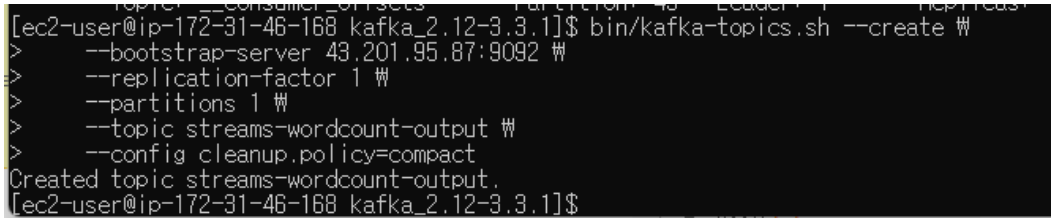
  ```
  $ bin/kafka-topics.sh --create \
      --bootstrap-server 43.201.95.87:9092 \
      --replication-factor 1 \
      --partitions 1 \
      --topic streams-plaintext-input
  ```

- streams-wordcount-output : output 토픽

  (터미널 1)

  ```
  $ bin/kafka-topics.sh --create \
      --bootstrap-server 43.201.95.87:9092 \
      --replication-factor 1 \
      --partitions 1 \
      --topic streams-wordcount-output \
      --config cleanup.policy=compact
  ```

step 3-2) 생성한 토픽이 제대로 생성되었는지 확인

(터미널 1)

- 토픽의 자세한 정보 조회

```
$ bin/kafka-topics.sh --bootstrap-server 43.201.95.87:9092 --describe
```

- 생성된 토픽 리스트 조회

```
$ bin/kafka-topics.sh --list --bootstrap-server 43.201.95.87:9092
```

step 4) WordCount 데모 어플리케이션 시작하기

(터미널 1) : 미리 작성된 kafka streams 코드 실행

→ input으로 들어온 data의 단어 개수를 집계하여 output으로 내보내는 작업 수행하는 코드

```
package org.apache.kafka.streams.examples.wordcount;

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.streams.KafkaStreams;
import org.apache.kafka.streams.StreamsBuilder;
import org.apache.kafka.streams.StreamsConfig;
import org.apache.kafka.streams.kstream.KStream;
import org.apache.kafka.streams.kstream.KTable;
import org.apache.kafka.streams.kstream.Produced;

import java.util.Arrays;
import java.util.Locale;
import java.util.Properties;
import java.util.concurrent.CountDownLatch;


public final class WordCountDemo {

    public static void main(final String[] args) {
        //프로퍼티 설정
        final Properties props = new Properties();
        props.put(StreamsConfig.APPLICATION_ID_CONFIG, "streams-wordcount");
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(StreamsConfig.CACHE_MAX_BYTES_BUFFERING_CONFIG, 0);
        props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass().getName());
        props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.String().getClass().getName());
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");

        //스트림 토폴로지를 정의
        final StreamsBuilder builder = new StreamsBuilder();

        //streams-plaintext-input 토픽을 담을 KStream 객체를 만들기 위해 stream() 메소드 사용
        final KStream<String, String> source = builder.stream("streams-plaintext-input");

        //단어의 개수를 저장하는 KTable 객체 만들기
        final KTable<String, Long> counts = source
            .flatMapValues(value -> Arrays.asList(value.toLowerCase(Locale.getDefault()).split(" ")))
            .groupBy((key, value) -> value)
            .count();

        //결과값을 streams-wordcount-output 토픽으로 전송하기 위해 to()메소드 사용
        counts.toStream().to("streams-wordcount-output", Produced.with(Serdes.String(), Serdes.Long()));

        //SteamBuilder로 정의한 토폴로지에 대한 정보와 스트림즈 실행을 위한 기본 옵션을 파라미터로 KafkaStreams 인스턴스 생성
        final KafkaStreams streams = new KafkaStreams(builder.build(), props);
        final CountDownLatch latch = new CountDownLatch(1);

        //ctrl-c를 감지하기 위한 shutdown handler 코드
        Runtime.getRuntime().addShutdownHook(new Thread("streams-wordcount-shutdown-hook") {
            @Override
            public void run() {
```

```
                streams.close();
                latch.countDown();
            }
        });

        try {
            streams.start();  //KafkaStreams 인스턴스를 실행하려면 start() 메소드 사용.
            latch.await();
        } catch (final Throwable e) {
            System.exit(1);
        }
        System.exit(0);
    }
}
```

아래 명령으로 위의 코드를 cli환경에서 실행시킬 수 있음.

```
$ bin/kafka-run-class.sh org.apache.kafka.streams.examples.wordcount.WordCountDemo
```

* 재실행시 아래 명령 수행 후 다시 실행 *

```
$ bin/kafka-streams-application-reset.sh --application-id streams-wordcount
```

step 5) data processing

(터미널 2) : producer 실행시켜서 input 토픽에 data 넣기

```
$ bin/kafka-console-producer.sh --broker-list 43.201.95.87:9092 --topic streams-plaintext-input
```

> 입력
>
> >all streams lead to kafka
>
> >hello kafka streams
>
> >join kafka summit

(터미널 3) : consumer 실행시켜서 output 토픽의 내용 가져와 출력하기

```
bin/kafka-console-consumer.sh --bootstrap-server 43.201.95.87:9092 \
    --topic streams-wordcount-output \
    --from-beginning \
    --formatter kafka.tools.DefaultMessageFormatter \
    --property print.key=true \
    --property print.value=true \
    --property key.deserializer=org.apache.kafka.common.serialization.StringDeserializer \
    --property value.deserializer=org.apache.kafka.common.serialization.LongDeserializer
```

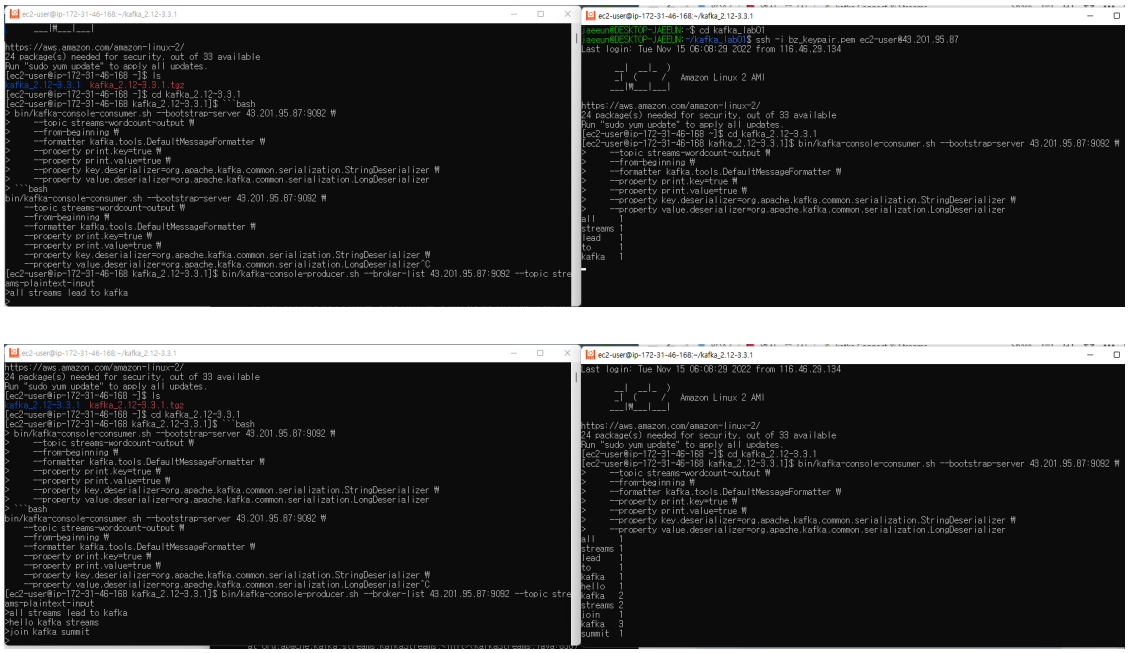
> 출력

```
all     1
streams 1
lead    1
to      1
kafka   1
```

```
all     1
streams 1
lead    1
to      1
kafka   1
hello   1
kafka   2
streams 2
```

```
all     1
streams 1
lead    1
to      1
kafka   1
hello   1
kafka   2
streams 2
join    1
kafka   3
summit  1
```

▼ <실행결과>



→ input 토픽으로 들어간 data가 **streams에 의해 실시간 처리**되어 output 토픽에 적재되는 것을 확인할 수 있다.

▼ 참고 링크

https://kimseunghyun76.tistory.com/464 - windows

https://kafka.apache.org/22/documentation/streams/quickstart#anchor-changelog-output - mac/linux