3 장 스테이트리스 처리

스트림 처리의 가장 간단한 형태는 이전에 보았던 이벤트에 대한 기억을 필요로 하지 않는다. 각 이벤트는 소비되고, 처리되며[1], 추후 잊혀진다. 이 패러다임을 스테이트리스 처리라고 부르는데 카프카 스트림즈는 스테이트리스 방식으로 데이터 작업을 수행하기 위해 풍부한 일련의 연산자를 포함하고 있다.

이 장에서는 카프카 스트림즈에 포함되어 있는 스테이트리스 연산자를 논의할 것이며 그러면서 가장 일반적인 스트림 처리 태스크의 일부가 어떻게 쉽게 해결될 수 있는지를 볼 것이다. 논의할 주제는 다음과 같다:

- 레코드 필터링
- 필드 추가 및 삭제
- 레코드 키 재생성 (rekeying)
- 스트림 분기
- 스트림 병합
- 레코드를 하나 이상의 레코드로 변환
- 한번에 하나씩 레코드 보강

이러한 개념을 소개하기 위해 튜토리얼 기반 방법을 취할 것이다. 구체적으로 트윗으로부터 암호화폐에 대한 데이터를 스트리밍하고 원시 데이터를 의미있는 투자 신호로 변환하기 위해 연산자를 적용할 것이다. 이 장 말미에는 원시 데이터 보강 및 변환을 위해 카프카 스트림즈의 스테이트리스 연산자 사용 방법을 이해할 것이며 이는 추후 장에서 논의할 보다 고급 개념에 대해 여러분을 준비시킬 것이다.

튜토리얼로 들어가기 전에 스테이트리스 처리가 무엇인지를 스테이트풀 처리와 비교함으로써 더 나은 참조 프레임을 얻어보자.

스테이트리스 대 스테이트풀 처리

카프카 스트림즈 애플리케이션을 구축할 때 고려해야 할 가장 중요한 사실 중 하나는 애플리케이션이 스테이트풀 처리를 필요로 하는지 여부이다. 다음은 스테이트리스 및 스테이트풀 스트림 처리 간 차이를 설명한다:

- 스테이트리스 애플리케이션에서 카프카 스트림즈 애플리케이션이 다루는 각 이벤트는 다른 이벤트에 상관없이 처리되며 애플리케이션은 단지 스트림 뷰만 필요하다 (스트림과 테이블

[1] 처리된다는 것은 적재된다는 말이다. 여기서는 좀 더 넓은 의미로 단어를 사용하며 보강, 변환, 반응 및 선택적으로 처리 데이터의 출력 토픽 쓰기 프로세스와 관련이 있다.

참조). 다른 말로 애플리케이션은 각 이벤트를 독립적인 insert로 처리하며 이전에 보았던 이벤트에 대한 기억을 필요로 하지 않는다.

- 반면 스테이트풀 애플리케이션은 보통 이벤트 스트림 집계, 윈도우 또는 조인을 위해 프로세서 토폴로지의 하나 이상의 단계에서 이전에 보았던 이벤트에 대한 정보를 기억해야 한다. 이 애플리케이션은 추가적인 데이터 또는 상태를 추적해야 하기 때문에 내부적으로 보다 복잡하다.

고수준 DSL에서 최종 구축 스트림 처리 애플리케이션의 유형은 토폴로지에서 사용된 개별 연산자에 따라 달라진다[2]. 연산자는 이벤트가 토폴로지를 통해 흐를 때 이벤트에 적용되는 스트림 처리 함수(예, filter, map, flatMap, join 등)이다. Filter와 같은 일부 연산자는 액션을 수행하기 위해 현재 레코드만 보면 되기 때문에 스테이트리스다 (이 경우 filter는 레코드가 다운스트림 프로세서로 전달되어야 하는지 여부를 결정하기 위해 각 레코드를 개별적으로 본다). Count와 같은 연산자는 이전 이벤트를 알아야 하기 때문에 스테이트풀이다 (count는 메시지 수를 추적하기 위해 지금까지 얼마나 많은 이벤트를 보았는지 알 필요가 있다).

카프카 스트림즈 애플리케이션이 단지 스테이트리스 연산자만 필요로 한다면 (따라서 이전에 보았던 이벤트에 대한 어떤 기억도 유지할 필요가 없다면) 애플리케이션은 스테이트리스로 간주된다. 그러나 (다음 장에서 배울 것인) 하나 이상의 스테이트풀 연산자가 도입된다면 애플리케이션이 스테이트리스 연산자를 사용하는지 여부에 상관없이 애플리케이션은 스테이트풀로 간주된다. 스테이트풀 애플리케이션에 더해지는 복잡성은 유지보수, 확장성 및 내고장성과 관련하여 추가적인 고려사항을 필요로 하며 따라서 다음 장에서 별도로 이 유형의 스트림 처리를 다룰 것이다.

이 모든 것이 추상적으로 들릴 수 있지만 걱정하지 말기 바란다. 다음 절에서 스테이트리스 카프카 스트림즈 애플리케이션을 구축하고 스테이트리스 연산자를 직접 경험해 봄으로써 이러한 개념을 보여줄 것이다. 더 이상 고민하지 말고 이 장의 튜토리얼을 소개해보자.

튜토리얼 소개: 트윗 스트림 처리

이 튜토리얼에서는 알고리즘 트레이딩이라는 유스케이스를 논의할 것이다. 종종 고빈도 트레이딩 (high-frequency trading, HFT)으로 불리는 이 수익성을 위한 사례는 많은 유형의 시장 신호를 최소한의 지연으로 처리 및 대응함으로써 자동적으로 주식을 평가 및 구매하기 위한 소프트웨어 구축을 포함한다.

가상의 트레이딩 소프트웨어를 지원하기 위해 다양한 유형의 암호화폐 (비트코인, 이더리움, 리플 등) 관련 시장 감성을 측정하고 맞춤형 트레이딩 알고리즘[3]에서 투자/회수 신호로 이러한 감성 점수를 사

---

[2] 이 장에서는 DSL에 집중을 할 것이지만 7 장에서 프로세서 API를 통한 스테이트리스 및 스테이트풀 처리를 다룰 것이다.

[3] 이 튜토리얼에서 완전한 트레이딩 알고리즘을 개발하지는 않을 것이다. 왜냐하면 이상적으로 트레

용하는 스트림 처리 애플리케이션을 구축할 것이다. 수백만명의 사람들이 암호화폐와 다른 주제에 대한 생각을 공유하기 위해 트윗을 사용하고 있기 때문에 애플리케이션 소스 데이터로 트윗을 사용할 것이다.

시작하기 전에 스트림 처리 애플리케이션 구축에 필요한 단계들을 살펴보자. 이후 스테이트리스 카프카 스트림즈 애플리케이션을 구축함에 따라 유용한 가이드가 될 프로세서 토폴로지를 설계하기 위해 이러한 요건들을 사용할 것이다. 각 단계의 주요 개념은 기울임꼴로 표시하였다:

1. 특정 디지털 화폐를 언급하는 트윗 (#bitcoin, #ethereum)이 tweets 소스 토픽으로부터 소비되어야 한다:
    - 각 레코드가 JSON으로 인코딩되어 있기 때문에 이들 레코드를 고수준 데이터 클래스로 적절히 *역직렬화*하는 방법을 알아야 한다.
    - 코드 단순화를 위해 불필요한 필드는 역직렬화 프로세스 동안 제거되어야 한다. 작업할 필드의 단지 일부만 선택하는 것은 *투영(projection)*으로 간주되며 스트림 처리에서 가장 일반적인 태스크 중 하나이다.
2. 리트윗은 처리에서 배제되어야 한다. 이는 일종의 데이터 *필터링*을 포함할 것이다.
3. 영어가 아닌 트윗은 번역을 위해 별도의 스트림으로 *분기*되어야 한다.
4. 비영어 트윗은 영어로 번역되어야 한다. 이는 (비영어 트윗인) 하나의 입력 값을 (영어 번역 트윗인) 새로운 출력 값으로의 *매핑*을 포함한다.
5. 새롭게 번역된 트윗은 하나의 통합된 스트림을 생성하기 위해 영어 트윗과 *병합*되어야 한다.
6. 각 트윗은 사용자가 특정 암호화폐를 논의할 때 긍정 또는 부정 감정을 전달하는지를 나타내는 감성 점수로 *보강*되어야 한다. 단일 트윗이 복수의 암호화폐를 언급할 수 있기 때문에 flatMap 연산자를 사용하여 각 입력 (트윗)을 가변 수의 출력으로 변환하는 방법을 배울 것이다.
7. 보강된 트윗은 Avro를 사용하여 직렬화되고 crypto-sentiment 출력 토픽으로 써져야 한다. 가공의 트레이딩 알고리즘은 이 토픽을 읽고 신호 기반으로 투자 의사결정을 할 것이다.

이제 요건이 수집되었고 프로세서 토폴로지를 설계할 수 있다. 그림 3-1은 이 장에서 무엇을 구축할지와 데이터가 카프카 스트림즈를 통해 어떻게 흐를 것인지를 보여준다.

이딩 알고리즘은 시장 감성만이 아닌 다양한 유형의 신호를 포함할 것이기 때문이다 (돈을 잃기를 원하지 않는다면).

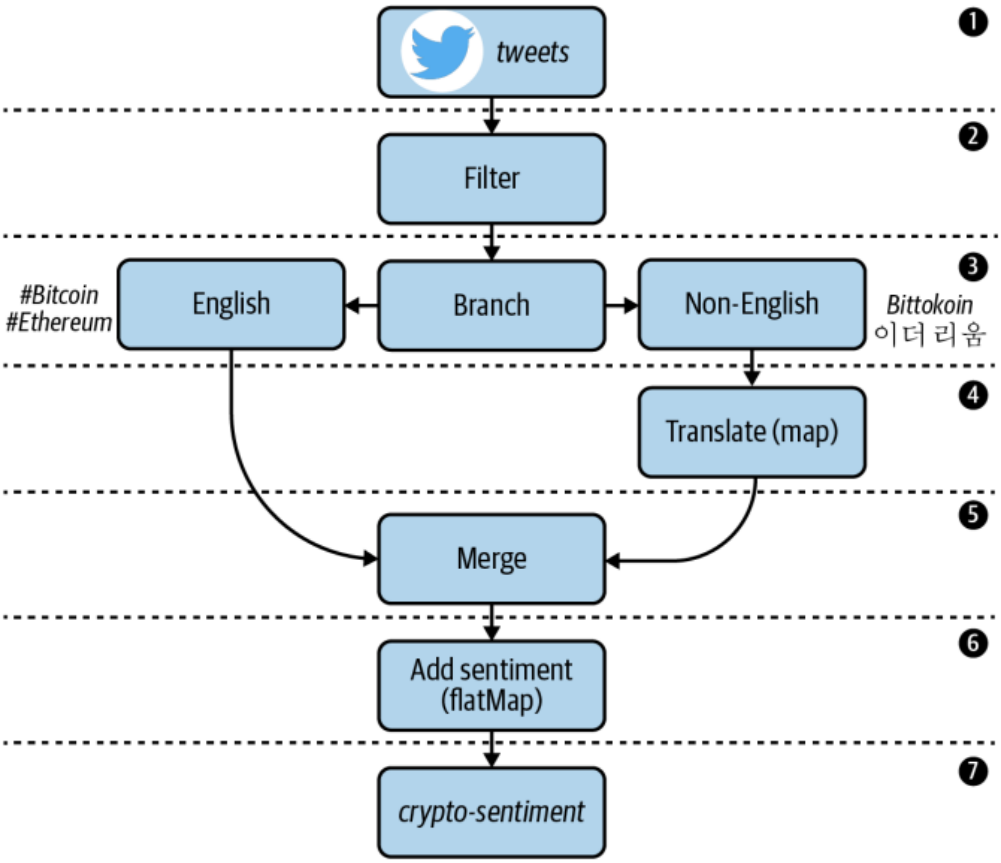

그림 3-1. 트윗 보강 애플리케이션을 위해 구현할 토폴로지

토폴로지 설계를 통해 그림 3-1의 처리 단계 각각을 수행함으로써 카프카 스트림즈 애플리케이션 구현을 시작할 수 있다. 프로젝트 환경 설정을 시작하고 토폴로지의 첫번째 단계를 진행해보자: 소스 토픽으로부터 트윗 스트리밍

프로젝트 환경설정

이 장의 코드는 https://github.com/mitch-seymour/mastering-kafka-streams-and-ksqldb.git에 위치해 있다.

각 토폴로지 단계를 통해 작업할 때 코드를 참조하고 싶다면 저장소를 복제하고 이 장 튜토리얼을 포함한 디렉토리로 이동한다. 다음 명령을 사용할 수 있다:

```
$ git clone git@github.com:mitch-seymour/mastering-kafka-streams-and-ksqldb.git

$ cd mastering-kafka-streams-and-ksqldb/chapter-03/crypto-sentiment
```

다음 명령을 실행함으로써 프로젝트를 언제라도 빌드할 수 있다.

```
$ ./gradlew build –info
```

(그림 3-1의 4 및 6 단계) 트윗 번역과 감성 분석의 경우 카프카 스트림즈에서 스테이트리스 연산자를 보여주는데 필요치 않기 때문에 구현 세부사항을 생략하였다. 그러나 Github의 소스 코드는 완

벽히 동작하는 예를 포함하고 있으며 따라서 구현 세부사항에 관심이 있다면 프로젝트의 README.md 파일을 참조하기 바란다.

프로젝트 환경 설정이 끝났고 카프카 스트림즈 애플리케이션 구현을 시작해보자.

KStream 소스 프로세서 추가하기

모든 카프카 스트림즈 애플리케이션은 하나 이상의 소스 토픽으로부터 데이터를 소비한다는 한 가지 공통점을 갖고 있다. 이 튜토리얼에는 tweets이라는 하나의 소스 토픽이 존재한다. 이 토픽은 트위터 스트리밍 API로부터 트윗을 스트리밍하여 JSON으로 인코딩된 트윗 레코드를 카프카에 쓰는 트위터 소스 커넥터로부터의 트윗으로 채워진다. 트윗 값[4] 예는 예제 3-1과 같다.

예제 3-1. 트윗 소스 토픽의 레코드 예

```
{
      "CreatedAt": 1602545767000,
       "Id": 1206079394583924736,
       "Text": "Anyone else buying the Bitcoin dip?",
       "Source": "",
       "User": {
            "Id": "123",
             "Name": "Mitch",
             "Description": "",
             "ScreenName": "timeflown",
             "URL": "https://twitter.com/timeflown",
             "FollowersCount": "1128",
             "FriendsCount": "1128"
     }
}
```

이제 데이터가 어떻게 생겼는지를 알았기 때문에 해결해야 할 첫 번째 단계는 소스 토픽으로부터 데이터를 카프카 스트림즈 애플리케이션으로 넣는 것이다. 이전 장에서 스테이트리스 레코드 스트림을 표현하기 위해 KStream 추상화를 사용할 수 있음을 배웠다. 다음 코드 블록에서 볼 수 있듯이 카프카 스트림즈에 KStream 소스 프로세서를 추가하는 것은 간단하며 단지 몇 줄의 코드만 필요하다:

```
StreamsBuilder builder = new StreamsBuilder(); ①

KStream<byte[], byte[]> stream = builder.stream("tweets"); ②
```

① 고수준 DSL을 사용할 때 프로세서 토폴로지는 StreamsBuilder 인터페이스를 사용하여 구축된다

② 토픽 이름을 StreamsBuilder.stream 메소드에 전달함으로써 KStream 인스턴스가 생성된다.

---

[4] 애플리케이션이 키 수준의 작업을 수행하지 않기 때문에 레코드 키에 대해서는 걱정할 필요가 없다.

Stream 메소드는 추후 절에서 논의할 추가적인 파라미터를 선택적으로 받아들일 수 있다.

주목해야 할 한 가지 사실은 생성한 KStream이 byte[] 타입의 파라미터를 갖는 다는 것이다:

```
KStream<byte[], byte[]>
```

이전 장에서 간략하게 다뤘지만 KStream 인터페이스는 2 개의 제네릭을 활용한다: 카프카 토픽에서 키 타입 (K)을 지정하는 제네릭과 값 타입 (V)을 지정하는 제네릭. 카프카 스트림즈 라이브러리의 내부를 들여다보면 이와 같이 생긴 인터페이스를 볼 것이다:

```
public interface KStream<K, V> {
  // omitted for brevity
}
```

따라서 KStream<byte[], byte[]>로 파라미터화된 KStream 인스턴스는 트윗 토픽으로 오는 레코드 키와 값이 바이트 배열(byte array)로 인코딩됨을 나타낸다. 그러나 트윗 레코드는 실제 (예제 3-1) 소스 커넥터에 의해 JSON 객체로 인코딩됨을 언급했다. 그렇다면 무엇을 주는가?

기본적으로 카프카 스트림즈는 애플리케이션을 통해 바이트 배열로 흐르는 데이터를 표현한다. 이는 카프카 자체가 데이터 원시 바이트 시퀀스로 저장하고 전송한다는 사실에 기인하며 따라서 데이터를 바이트 배열로 표현하는 것은 항상 동작할 것이다 (따라서 합리적인 기본값이다). 원시 바이트 저장 및 전송은 클라이언트 측에 어떠한 특정 데이터 포맷도 부과하지 않기 때문에 카프카를 유연하게 만들며 네트워크를 통해 원시 바이트 스트림을 전송하기 위해 브로커에서 더 적은 메모리와 CPU 사이클을 요구하기 때문에 빠르다[5]. 그러나 이는 카프카 스트림즈 애플리케이션을 포함한 카프카 클라이언트가 문자열 (구분(delimited) 또는 비구분(non-delimited), JSON, Avro, Protobuf 등을 포함하여 고수준 객체 및 포맷을 이용해 동작하기 위해 이러한 바이트 스트림의 직렬화/역직렬화를 담당해야 함을 의미한다[6].

트윗 객체를 고수준 객체로 역직렬화하는 문제를 다루기 전에 카프카 스트림즈 애플리케이션을 동작시키는데 필요한 추가적인 상용구 코드를 추가해보자. 테스트 목적으로 애플리케이션을 실제 동작하는 코드와 카프카 스트림즈 토폴로지를 구축하기 위한 로직을 분리하는 것이 유리하다. 따라서 상용구 코드는 2 개의 클래스를 포함할 것이다. 첫번째로 예제 3-2와 같이 카프카 스트림즈 토폴로지를 구축하기 위한 클래스를 정의할 것이다.

---

[5] 바이트 배열로 데이터를 저장 및 전송함으로써 zero-copy라는 것을 활용할 수 있도록 하며 이는 클라이언트 측에서 처리되기 때문에 직렬화/역직렬화 목적으로 데이터가 사용자-커널 공간 경계를 넘나들 필요가 없음을 의미한다. 이는 중요한 성능상 장점이다.

[6] 따라서 트위터 커넥터가 트윗을 JSON으로 인코딩한다고 할 때 카프카에서 트윗 레코드가 원시 JSON으로 저장된다는 것을 의미하지 않는다. 단순히 카프카 토픽에서 이 트윗을 표현하는 바이트가 역직렬화될 때 JSON 포맷임을 의미한다.

예제 3-2. 카프카 스트림즈 토폴로지를 정의하는 Java 클래스

```
class CryptoTopology {
    public static Topology build() {
        StreamsBuilder builder = new StreamsBuilder();
        KStream<byte[], byte[]> stream = builder.stream("tweets");
        stream.print(Printed.<byte[], byte[]>toSysOut().withLabel("tweets-stream")); ①
        return builder.build();
    }
}
```

① print 연산자를 통해 애플리케이션을 통해 데이터가 흐를 때 데이터를 쉽게 볼 수 있다. 이는 개발 목적으로만 권고된다.

App이라 부를 두번째 클래스는 예제 3-3과 같이 토폴로지를 인스턴스화하고 동작시킬 것이다.

예제 3-3. 카프카 스트림즈 애플리케이션을 동작시키는데 사용되는 별도의 Java 클래스

```
class App {
    public static void main(String[] args) {
        Topology topology = CryptoTopology.build();
        Properties config = new Properties(); ①
        config.put(StreamsConfig.APPLICATION_ID_CONFIG, "dev");
        config.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:29092");
        KafkaStreams streams = new KafkaStreams(topology, config); ②
        Runtime.getRuntime().addShutdownHook(new Thread(streams::close)); ③
        System.out.println("Starting Twitter streams");
        streams.start(); ④
    }
}
```

① 카프카 스트림즈는 (컨슈머 그룹에 해당하는) 애플리케이션 ID와 카프카 부트스트랩 서버를 포함하여 일부 기본 구성을 설정하도록 요구한다. Properties 객체를 사용하여 이들을 설정한다.

② 프로세서 토폴로지와 스트림 설정을 통해 KafkaStreams 객체를 인스턴스화한다.

③ 글로벌 셧다운 신호 수신 시 카프카 스트림즈 애플리케이션을 우아하게 중지시키기 위해 셧다운 훅을 추가한다.

④ 카프카 스트림즈 애플리케이션을 시작한다. Streams.start()가 블록 처리되지 않고 토폴로지가 백그라운드 처리 쓰레드를 사용하여 실행됨을 주목하기 바란다. 이것이 셧다운 훅이 필요한 이유이다.

이제 애플리케이션이 동작할 준비가 되었다. 카프카 스트림즈 애플리케이션을 시작하여 일부 데이터를 트윗 토픽으로 생산하면 화면에 출력되는 원시 바이트 배열을 볼 것이다 (각 출력 행에서 콤마 후에 나타나는 비밀 값).

[tweets-stream]: null, [B@c52d992

[tweets-stream]: null, [B@a4ec036

[tweets-stream]: null, [B@3812c614

예상한바와 같이 바이트 배열의 저수준 특성은 이들을 이용하여 작업하기를 어렵게 만든다. 사실 소스 토픽에서 다른 데이터 표현 방법을 찾는다면 추가적인 스트림 처리 단계 구현은 더욱 쉬울 것이다. 여기서 데이터 직렬화 및 역직렬화 개념이 동작한다.

직렬화/역직렬화

카프카는 바이트가 들어오고 바이트가 나가는 스트림 처리 플랫폼이다. 이는 카프카 스트림즈와 같은 클라이언트가 소비하는 바이트 스트림을 보다 고수준 객체로 변환해야 한다는 것을 의미한다. 이 프로세스를 역직렬화라고 부른다. 비슷하게 클라이언트는 또한 카프카에 다시 쓰길 원하는 어떠한 데이터도 바이트 배열로 변환해야 한다. 이 프로세스는 직렬화라고 부른다. 그림 3-2에 이 프로세스들을 나타냈다.

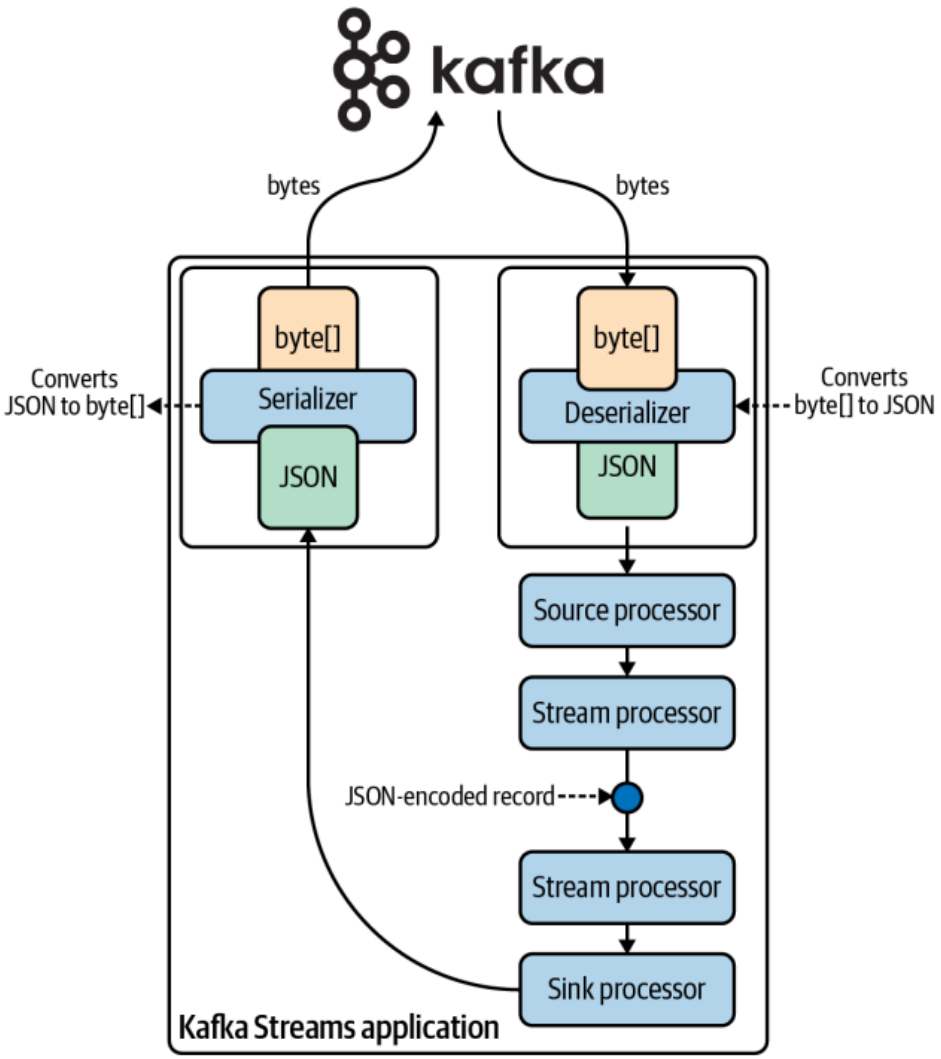

그림 3-2. 카프카 스트림즈 애플리케이션에서 직렬화 및 역직렬화 프로세스가 일어나는 곳에 대한 아키텍처 뷰

카프카 스트림즈에서 직렬화기(serializer)와 역직렬화기(deserializer) 클래스는 종종 Serdes 라는 하나의 클래스로 결합되며 표 3-1[7]과 같이 여러 구현과 함께 라이브러리가 제공된다. 예를 들어 String Serdes (Serdes.String() 메소드를 통해 액세스 가능)는 String 직렬화기와 역직렬화기 클래스 모두를 포함한다.

표 3-1. 카프카 스트림즈에서 사용가능한 기본 Serdes 구현

데이터 타입	Serdes 클래스
byte[]	Serdes.ByteArray(), Serdes.Bytes()
ByteBuffer	Serdes.ByteBuffer()
Double	Serdes.Double()
Integer	Serdes.Integer()
Long	Serdes.Long()
String	Serdes.String()
UUID	Serdes.UUID()
Void	Serdes.Void()

카프카 스트림즈에서 데이터를 직렬화/역직렬화할 필요가 있을 때마다 우선 내장 Serdes 클래스 중 하나가 요구에 부합하는지를 확인해야 한다. 그러나 알아챘듯이 카프카 스트림즈는 JSON[8], Avro 및 Protobuf 를 포함하여 일부 공통적인 포맷에 대해 Serdes 클래스를 제공하지 않는다. 그러나 필요한 경우 스스로 Serdes 를 구현할 수 있으며 트윗이 JSON 객체로 표현되기 때문에 이 포맷을 다루기 위해 맞춤형 Serdes 를 구축하는 법을 배울 것이다.

맞춤형 Serdes 구축하기

이전에 언급했듯이 소스 토픽의 트윗은 JSON 객체로 인코딩되어 있지만 카프카에는 단지 원시 바이트만 저장된다. 따라서 해야 할 첫번째는 스트림 처리 애플리케이션에서 데이터를 이용해 작업하기 쉽도록 트윗을 고수준 JSON 객체로 역직렬화하기 위한 코드를 작성하는 것이다. 직접 구현하는 대신 내장 String Serdes 인 Serdes.String()를 사용할 수 있지만 이는 트윗 객체 내 각 필드에 쉽게 액세스할 수 없기 때문에 트윗 데이터를 이용한 작업을 어렵게 만들 것이다[9].

데이터가 JSON 또는 Avro 와 같은 공통 포맷인 경우 스스로 저수준 JSON 직렬화/역직렬화 로직을 구현함으로써 처음부터 다시 만들 필요는 없을 것이다. JSON 직렬화 및 역직렬화를 위한 많은 Java 라이브러리가 존재하며 이 튜토리얼에서는 Gson 을 사용할 것이다. Gson 은 구글이 개발한 것으로

---

[7] 보다 많은 Serdes 클래스가 향후 버전에 도입될 것 같으며 사용가능한 Serdes 클래스의 전체 목록은 공식 문서를 참조하기 바란다.

[8] 예제 JSON Serdes가 카프카 소스 코드에 포함되어 있지만 이 책 작성 시점에는 공식적인 Serdes와 같은 Serdes 팩토리 클래스를 통해 노출되어 있지 않다.

[9] 이는 어려운 정규표현식의 사용을 필요로 하며 그 후에도 잘 작동하지 않을 것이다.

JSON 바이트를 Java 객체로 변환하기 위한 직관적인 API 를 갖고 있다. 다음 코드 블록은 Gson 을 이용하여 바이트 배열을 역직렬화하기 위한 기본 메소드를 보여주는데 카프카로부터 JSON 레코드를 읽을 필요가 있을 때마다 편리할 것이다:

```
Gson gson = new Gson();
byte[] bytes = ...; ①
Type type = ...; ②
gson.fromJson(new String(bytes), type); ③
```

① 역직렬화해야 할 원시 바이트

② 타입은 역직렬화된 레코드를 표현하기 위해 사용될 Java 클래스이다.

③ fromJson 메소드는 원시 바이트를 Java 클래스로 실제 변환한다.

또한 Gson 은 이 프로세스의 역도 지원하며 따라서 Java 객체를 원시 바이트 배열로 변환 (직렬화)할 수 있도록 해준다. 다음 코드는 Gson 으로 직렬화를 수행하는 방법을 보여준다:

```
Gson gson = new Gson();
gson.toJson(instance).getBytes(StandardCharsets.UTF_8);
```

Gson 라이브러리가 저수준에서 보다 복잡한 JSON 직렬화/역직렬화 태스크를 관리하기 때문에 맞춤형 Serdes 구현 시 단지 Gson 의 기능을 활용하기만 하면 된다. 첫번째 단계는 원시 바이트 배열이 역직렬화되는 데이터 클래스를 정의하는 것이다.

데이터 클래스 정의하기

Gson (그리고 Jackson 과 같은 다른 JSON 직렬화기)의 한 가지 특징은 JSON 바이트 배열을 Java 객체로 변환할 수 있도록 한다는 것이다. 이를 위해 소스 객체로부터 역직렬화하기 원하는 필드를 포함하는 데이터 클래스 또는 POJO (Plain Old Java Object)를 정의해야 한다.

예제 3-4 에서 볼 수 있듯이 카프카 스트림즈 애플리케이션에서 원시 트윗 레코드를 표현하기 위한 데이터 클래스는 매우 단순하다. 원시 트윗에서 수집하려는 각 필드에 대해 클래스 속성 (예, createdAt)과 각 속성에 액세스하기 위한 해당 getter/setter 메소드를 정의한다 (예, getCreateAt).

예제 3-4. 트윗 레코드를 역직렬화하기 위해 사용하는 데이터 클래스

```
public class Tweet {
    private Long createdAt;
    private Long id;
    private String lang;
    private Boolean retweet;
    private String text;
    // getters and setters omitted for brevity
}
```

소스 레코드에서 특정 필드를 수집하지 않으려면 데이터 클래스에서 단지 필드를 생략할 수 있으며 Gson 은 자동적으로 이를 드롭할 것이다. 스스로 역직렬화기를 생성할 때마다 역직렬화 프로세스 동안 소스 레코드에서 애플리케이션 또는 다운스트림 애플리케이션이 필요로 하지 않는 어떤 필드라도 드롭하는 것을 고려해야 한다. 사용가능한 필드를 보다 작은 서브셋으로 줄이는 프로세스를 투영이라고 부르며 이는 SQL 에서 단지 관심있는 칼럼만 선택하기 위해 SELECT 문을 사용하는 것과 유사하다.

이제 데이터 클래스를 생성했으며 tweets 토픽의 원시 바이트 배열을 (보다 작업을 수행하기 쉬운) 고수준의 Tweet Java 객체로 변환하기 위한 카프카 스트림즈 역직렬화기를 구현할 수 있다.

맞춤형 역직렬화기 구현하기

맞춤형 역직렬화기 구현에 필요한 코드는 바이트 배열 역직렬화의 복잡성 모두를 숨기는 라이브러리를 사용할 때 아주 최소한이다 (Gson 의 경우와 같이). 단순히 카프카 클라이언트 라이브러리로 Deserializer 인터페이스를 구현하여 deserialize 가 호출될 때 역직렬화 로직을 호출하기만 하면 된다. 다음 코드 블록은 Tweet 역직렬화기를 구현하는 법을 보여준다.

```
public class TweetDeserializer implements Deserializer<Tweet> {
    private Gson gson =
            new GsonBuilder()
                    .setFieldNamingPolicy(FieldNamingPolicy.UPPER_CAMEL_CASE) ①
                    .create();
    @Override
    public Tweet deserialize(String topic, byte[] bytes) {②
            if (bytes == null) return null; ③
             return gson.fromJson(
                    new String(bytes, StandardCharsets.UTF_8), Tweet.class); ④
     }
}
```

① Gson 은 JSON 필드 이름에 몇몇 다른 포맷을 지원한다. 트위터 카프카 커넥터가 필드 이름에 upper camel case 를 사용하기 때문에 JSON 객체가 정확히 역직렬화되는 것을 보장하기 위해 적절한 필드 명명 정책을 설정한다. 이는 매우 구현 특정적인 것으로 스스로 역직렬화기를 작성할 때 바이트 배열 역직렬화의 어려운 작업에 도움이 되도록 여기서 수행한 것과 같이 제 3 자 라이브러리와 맞춤형 설정을 자유롭게 활용하기 바란다.

② tweets 토픽의 레코드를 역직렬화하기 위해 고유 로직을 갖는 deserialize 메소드를 재정의한다. 이 메소드는 데이터 클래스 (Tweet)의 인스턴스를 반환한다.

③ Null 이면 바이트 배열을 역직렬화하려고 시도하지 말기 바란다.

④ 바이트 배열을 Tweet 객체로 역직렬화하기 위해 Gson 라이브러리를 사용한다.

이제 직렬화기를 구현할 준비를 마쳤다.

맞춤형 직렬화기 구현하기

맞춤형 직렬화기를 구현하는 코드는 매우 쉽다. 이제 카프카 클라이언트 라이브러리의 Serializer 인터페이스의 serialize 메소드를 구현하면 된다. 다시 Gson 의 직렬화 기능을 활용하여 어려운 작업의 대부분을 수행할 것이다. 다음 코드 블록은 이 장에서 사용할 Tweet 직렬화기를 보여준다.

```
class TweetSerializer implements Serializer<Tweet> {
    private Gson gson = new Gson();
    @Override
    public byte[] serialize(String topic, Tweet tweet) {
        if (tweet == null) return null; ①
        return gson.toJson(tweet).getBytes(StandardCharsets.UTF_8); ②
    }
}
```

① Null 인 Tweet 객체를 역직렬화라려고 시도하지 말기 바란다.

② Tweet 객체가 null 이 아닌 경우 객체를 바이트 배열로 변환하기 위해 Gson 을 사용한다.

Tweet 역직렬화기와 직렬화기를 완료하였고 이제 이들 클래스를 Serdes 로 결합할 수 있다.

Tweet Serdes 구축하기

지금까지 직렬화기와 역직렬화기 구현은 코드 상 매우 간단했다. 비슷하게 유일한 목적이 카프카 스트림즈가 사용할 역직렬화기와 직렬화기를 편리한 랩퍼 클래스로 결합하는 맞춤형 Serdes 클래스도 또한 아주 최소한이다. 다음 코드 블록은 맞춤형 Serdes 클래스 구현이 얼마나 쉬운지를 보여준다.

```
public class TweetSerdes implements Serde<Tweet> {
    @Override
    public Serializer<Tweet> serializer() {
        return new TweetSerializer();
    }
    @Override
    public Deserializer<Tweet> deserializer() {
        return new TweetDeserializer();
    }
}
```

이제 맞춤형 Serdes 가 완료되었고 예제 3-2 의 코드를 약간 수정할 수 있다. 다음 코드를

```
KStream<byte[], byte[]> stream = builder.stream("tweets");
stream.print(Printed.<byte[], byte[]>toSysOut().withLabel("tweets-stream"));
```

이 코드로 변경시켜보자.

```
KStream<byte[], Tweet> stream = ①
 builder.stream(
```

```
        "tweets",
         Consumed.with(Serdes.ByteArray(), new TweetSerdes())); ②
stream.print(Printed.<byte[], Tweet>toSysOut().withLabel("tweets-stream"));
```

① 값 타입이 byte[]에서 Tweet 로 변경되었음을 주목하기 바란다. Tweet 인스턴스를 사용하는 것은 다음 절에서 보듯이 작업을 보다 쉽게 만들 것이다.

② 카프카 스트림즈의 Consumed 헬퍼를 사용하여 KStream 에 대해 사용할 키와 값 Serdes 를 명시적으로 설정한다. 값 Serdes 를 new TweetSerdes()로 설정함으로써 스트림은 바이트 배열 대신 Tweet 객체로 채워질 것이다. 키 Serdes 는 변경되지 않고 유지된다.

키가 아직까지 바이트 배열로 직렬화되고 있음을 주목하기 바란다. 이는 토폴로지가 레코드 키를 갖고 아무 것도 할 것을 요구하지 않기 때문으로 따라서 이를 역직렬화할 필요가 없다[10]. 이제 카프카 스트림즈 애플리케이션을 동작하고 일부 데이터를 소스 토픽에 생산한다면 원래 JSON 필드에 액세스하기 위해 추후 스트림 처리 단계에서 활용할 유용한 getter 메소드를 포함하여 Tweet 객체로 지금 수행하는 것을 볼 것이다.

```
[tweets-stream]: null, Tweet@182040a6
[tweets-stream]: null, Tweet@46efe0cb
[tweets-stream]: null, Tweet@176ef3
```

이제 Tweet 데이터 클래스를 활용하는 KStream 을 생성했고 토폴로지의 나머지 구현을 시작할 수 있다. 다음 단계는 트위터 스트림으로부터 어떤 리트윗이라도 걸러내는 것이다.

카프카 스트림즈에서 역직렬화 에러 다루기

애플리케이션이 역직렬화 에러를 다루는 방식을 항상 명시해야 한다. 그렇지 않은 경우 형식에 맞지 않는 레코드가 소스 토픽에 존재한다면 원치않게 놀랄 수 있다. 카프카 스트림즈는 직렬화 예외 핸들러를 지정하는데 사용될 수 있는 DEFAULT_DESERIALIZATION_EXCEPTION_HANDLER_CLASS_CONFIG 설정을 갖고 있다. 스스로 (에러를 기록하고 이후 카프카 스트림즈에 셧다운 신호를 보내는) 예외 핸들러를 구현하거나 LogAndFailExceptionHandler 또는 (에러를 기록하고 처리를 계속하는) LogAndContinueExceptionhandler 를 포함하여 내장 기본 핸들러 중 하나를 사용할 수 있다.

데이터 필터링

스트림 처리 애플리케이션에서 가장 일반적인 스테이트리스 태스크 중 하나는 데이터 필터링이다. 필터링은 레코드의 일부만을 선택하여 처리하고 나머지는 무시한다. 그림 3-3 은 필터링의 기본 아이디어를 보여준다.

---

[10] 불필요한 직렬화/역직렬화는 어떤 애플리케이션에서 성능에 악영향을 미칠 수 있다.

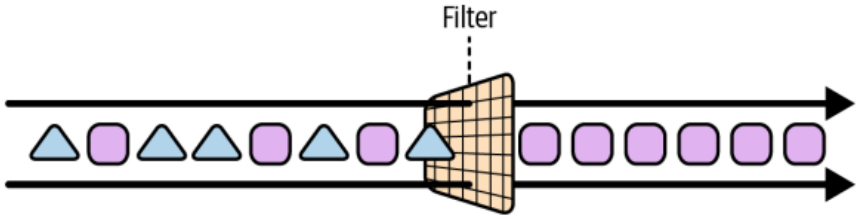

그림 3-3. 데이터 필터링은 이벤트 스트림에서 단지 데이터의 일부만 처리할 수 있도록 한다.

카프카 스트림즈에서 필터링에 사용되는 2 가지 기본 연산자[11]는 다음과 같다:

- filter
- filterNot

filte 연산자를 논의하면서 시작할 것이다. Filter 는 메시지가 유지되어야 하는지를 결정하기 위해 서술부(Predicate)라는 부울 표현식을 전달하면 된다. 서술부가 true 을 반환한다면 이벤트는 다운스트림 프로세서로 전달될 것이다. False 를 반환한다면 레코드는 추가 처리에서 배제될 것이다. 여기서는 리트윗을 걸러낼 필요가 있으며 따라서 서술부는 Tweet.isRetweet() 메소드를 활용할 것이다.

서술부는 함수 인터페이스[12]이기 때문에 filter 연산자에 람다 표현식을 사용할 수 있다. 예제 3-5 는 이 메소드를 사용하여 리트윗을 걸러내는 법을 보여준다.

예제 3-5. DSL 의 filter 연산자 사용법 예

```
KStream<byte[], Tweet> filtered =
  stream.filter(
        (key, tweet) -> {
              return !tweet.isRetweet();
        });
```

filterNot 연산자는 부울 로직이 역 (true 이면 레코드가 드롭되며 filter 의 반대)인 것을 제외하면 filter 와 비슷하다. 예제 3-5 에서 볼 수 있듯이 필터링 조건이 무효화된다: !tweet.isRetweet(). 예제 3-6 의 코드를 사용하여 연산자 수준에서 쉽게 무효화할 수 있다.

예제 3-6. DSL 의 filterNot 연산자 사용법 예

```
KStream<byte[], Tweet> filtered =
```

---

[11] 뒤에서 볼 것인데 flatMap과 flatMapValues 또한 필터링에 사용될 수 있다. 그러나 명확함을 위해 필터링 단계가 하나 이상의 레코드를 생산할 필요가 없다면 filter 또는 filterNot을 사용하는 것이 좋다.

[12] Java에서 함수 인터페이스는 단일 추상 메소드를 갖는다. Predicate 클래스만 test라는 하나의 추상 메소드를 포함한다.

```
stream.filterNot(
        (key, tweet) -> {
                return tweet.isRetweet();
        });
```

이 두 방법은 기능적으로 동일하다. 그러나 필터링 로직이 무효화를 포함할 때 가독성 개선을 위해 filterNot 사용이 선호된다. 이 튜토리얼의 경우 리트윗을 추후 처리에서 배제하기 위해 예제 3-6 에 보여지는 filterNot 구현을 사용할 것이다.

> 애플리케이션이 필터링 조건을 필요로 한다면 가능한 초기에 필터링해야 한다. 추후 단계에서 특히 불필요한 이벤트 처리 로직이 계산적으로 비싸다면 단순히 버려질 데이터를 변환하거나 보강할 필요는 없다.

(그림 3-1 의) 프로세서 토폴로지에서 1 및 2 단계를 구현하였다. 이제 3 단계로 이동할 준비를 마쳤다: 필터링된 스트림을 트윗의 소스 언어에 기반하여 서브 스트림으로 분리하기

데이터 분기하기

이전 절에서 스트림을 필터링하기 위해 서술부라는 부울 조건을 사용하는 법을 배웠다. 카프카 스트림즈는 또한 스트림을 분리 (또는 분기)하기 위해 서술부를 사용할 수 있도록 한다. 분기는 일반적으로 이벤트 자체의 속성에 기반하여 이벤트가 다른 스트림 처리 단계 또는 출력 토픽으로 전달될 필요가 있을 때 필요하다.

요건 목록을 다시 살펴본다면 트윗이 소스 토픽에서 여러 언어로 나타날 수 있음을 볼 수 있다. 이는 스트림 내 레코드 서브셋 (비영어 트윗)이 여분의 처리 단계를 필요로 하기 때문에 분기에 대한 훌륭한 유스케이스이다: 영어로 번역되어야 한다. 그림 3-4 의 다이어그램은 애플리케이션에 구현해야 하는 분기 동작을 설명한다.

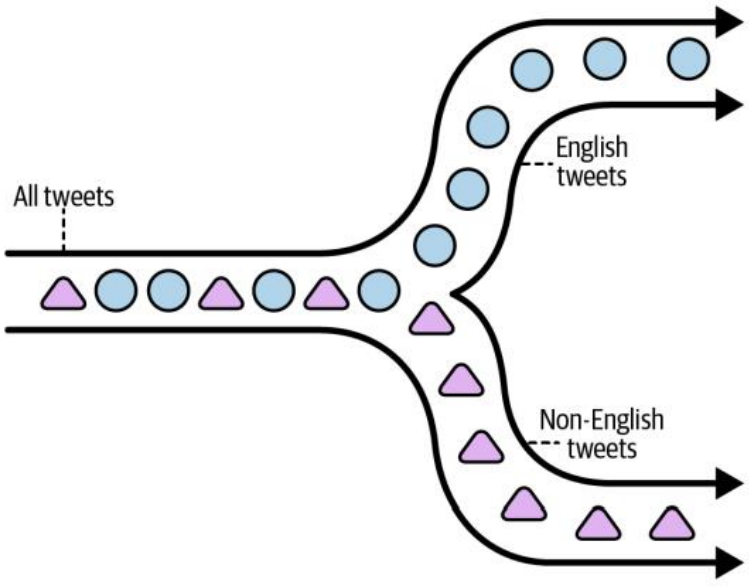

그림 3-4. 분기 연산은 단일 스트림을 복수의 출력 스트림으로 분리한다.

영어 트윗과 나머지를 수집하기 위한 2 개의 람다 함수를 생성해보자. 초반부 역직렬화 논의 덕택에 분기 로직을 구현하기 위해 데이터 클래스 내에서 다른 getter 메소드를 활용할 수 있다. 다음 코드 블록은 스트림 분기에 사용할 predicate 을 보여준다:

```
Predicate<byte[], Tweet> englishTweets =
        (key, tweet) -> tweet.getLang().equals("en"); ①
Predicate<byte[], Tweet> nonEnglishTweets =
        (key, tweet) -> !tweet.getLang().equals("en"); ②
```

① 이 서술부는 모든 영어 트윗과 일치시킨다.

② 이 서술부는 첫번째의 역으로 모든 비영어 트윗을 수집한다.

이제 분기 조건을 정의하였고 하나 이상의 서술부를 받아들여 각 서술부에 해당하는 출력 스트림 목록을 반환하는 카프카 스트림즈의 branch 연산자를 활용할 수 있다. 각 서술부가 순서대로 평가되고 레코드는 단지 하나의 분기에 추가될 수 있음을 주목하기 바란다. 레코드가 어떤 서술부에도 일치하지 않는다면 드롭될 것이다:

```
KStream<byte[], Tweet>[] branches =
        filtered.branch(englishTweets, nonEnglishTweets); ①
KStream<byte[], Tweet> englishStream = branches[0]; ②
KStream<byte[], Tweet> nonEnglishStream = branches[1]; ③
```

① 트윗의 소스 언어를 평가함으로써 스트림에 대해 2 개의 분기를 생성한다.

② englishTweets 서술부를 먼저 통과하기 때문에 (0 인덱스) 목록의 첫번째 KStream 은 영어 출력 스트림을 포함한다.

③ nonEnglishTweets 서술부가 마지막 분기 조건으로 사용되었기 때문에 번역이 필요한 트윗은 (인덱스 위치 1) 목록의 마지막 KStream 일 것이다.

> 스트림 분기 방법은 향후 카프카 스트림즈 버전에서 변경될 수도 있다. 2.7.0 이상의 카프카 스트림즈를 사용하고 있다면 잠재적 API 변경에 대해 "KIP-418: A methodchaining way to branch KStream"를 확인할 수 있다. 후방 호환성이 예상되며 현재 구현은 이 책 작성 시점의 최신 버전인 2.7.0 에서 동작한다.

이제 (englishStream 과 nonEnglishStream) 2 개의 서브 스트림을 생성했고 각각에 다른 처리 로직을 적용할 수 있다. 이는 (그림 3-1 의) 토폴로지 프로세서의 3 단계를 다루며 따라서 다음 단계로 이동할 준비를 마쳤다: 비영어 트윗을 영어로 번역하기

트윗 번역하기

이 시점에서 2 개의 레코드 스트림인 영어로 쓰여진 트윗 (englishStream)과 다른 언어로 쓰여진 트윗 (nonEnglishStream)을 갖고 있다. 각 트윗에 대해 감성 분석을 수행하려고 하지만 이 분석 수행에

사용할 API 는 단지 몇 개의 언어만 지원한다 (그 중 하나는 영어). 따라서 nonEnglishStream 내 각 트윗을 영어로 번역해야 한다.

하지만 비즈니스 문제로 이에 대해 이야기하고 있는데 스트림 처리 관점에서 요건을 살펴보자. 실제 필요한 것은 하나의 입력 레코드를 정확히 하나의 출력 레코드 (키와 값이 입력 레코드와 동일할 수도 아닐 수도 있음)로 변환하는 방법이다. 운좋게도 카프카 스트림즈는 요건에 맞는 2 개의 연산자를 갖고 있다.

- map
- mapValues

매핑 연산의 시각화를 그림 3-5 에 나타냈다.

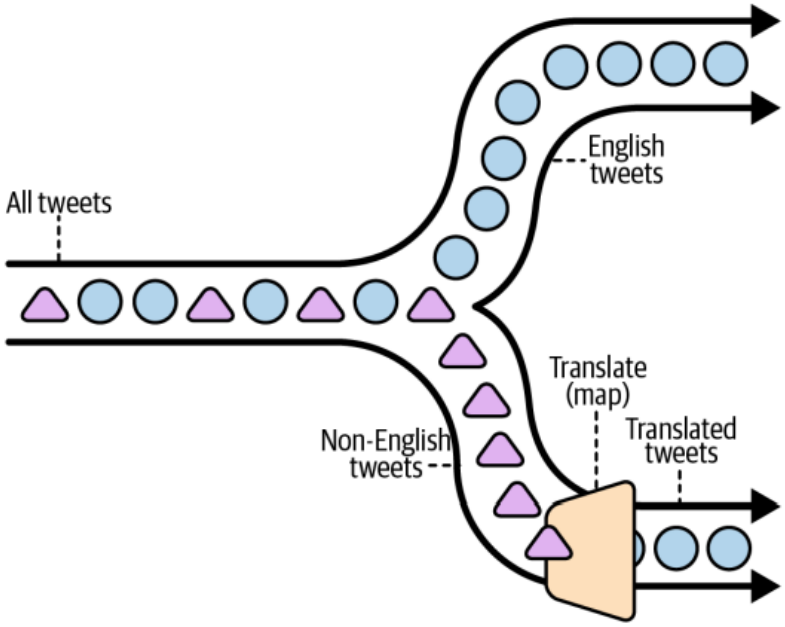

그림 3-5. 매핑 연산은 레코드의 1:1 변환을 수행할 수 있도록 한다.

Map 과 mapValues 연산자는 매우 유사하며 (둘 모두 입력과 출력 레코드 간 1:1 매핑을 한다) 둘 모두 이 유스케이스에 대해 동작할 수 있다. 단지 차이는 map 이 새로운 레코드 값과 키를 지정할 것을 요구하는 반면 mapValues 는 단지 새로운 값만을 설정할 것을 요구한다.

Map 을 통해 이를 어떻게 구현할 수 있는지를 우선 살펴보자. 트윗 텍스트를 번역할 뿐만 아니라 주어진 트윗에 대해 트위터 사용자 명으로 각 레코드의 키를 재생성하길 원한다고 가정해보자. 다음 코드 블록과 같이 트윗 번역을 구현할 수 있다:

```
KStream<byte[], Tweet> translatedStream =
 nonEnglishStream.map(
        (key, tweet) -> { ①
                byte[] newKey = tweet.getUsername().getBytes();
```

```
                Tweet translatedTweet = languageClient.translate(tweet, "en");
                return KeyValue.pair(newKey, translatedTweet); ②
        });
```

① 현재 레코드 키와 레코드 값으로 매핑 함수가 호출된다 (tweet 이라는)

② 매핑 함수는 카프카 스트림즈의 KeyValue 클래스를 사용하여 표현되는 새로운 레코드 키와 값을 반환하는데 필요하다. 여기서 새로운 키는 트위터 사용자 이름으로 설정되며 새로운 값은 번역된 트윗이다. 텍스트를 번역하는 실제 로직은 이 튜토리얼의 범위를 벗어나며 구현 세부 사항은 소스 코드에서 확인할 수 있다.

그러나 요건은 새로운 레코드 키와 값을 반환할 것을 요구하지는 않는다. 다음 장에서 볼 것인데 레코드의 키 재성성(rekeying)은 스테이트풀 연산이 추후 스트림 처리 단계에서 수행될 때 종종 사용된다[13]. 따라서 이전 코드가 동작하지만 대신 레코드 값 변환에만 관련된 mapValues 연산자를 사용하여 구현을 단순화할 수 있다.

다음 코드 블록은 mapValues 연산자 사용법 예를 보여준다:

```
KStream<byte[], Tweet> translatedStream =
 nonEnglishStream.mapValues(
        (tweet) -> { ①
                return languageClient.translate(tweet, "en"); ②
        });
```

① 단지 레코드 값만으로 mapValues 함수가 호출된다.

② mapValues 연산자를 사용할 때 새로운 레코드만 반환하면 된다. 이 경우 값은 번역된 트윗이다.

여기서는 어떤 레코드의 키도 재생성할 필요가 없기 때문에 mapValues 구현을 고수할 것이다.

> 가능한 map 대신 mapValues 사용이 권고된다. 왜냐하면 카프카 스트림즈가 아마도 프로그램을 보다 효율적으로 실행시킬 수 있도록 하기 때문이다.

모든 비영어 트윗을 번역한 후 영어 트윗을 포함하는 2 개의 KStream 을 갖고 잇다:

- englishStream, 영어 트윗의 원래 스트림
- translatedStream, 이제 영어인 새롭게 번역된 트윗

이제 (그림 3-1 의) 프로세서 토폴로지의 4 단계를 완료했다. 그러나 애플리케이션의 목표는 모든 영어 트윗에 대해 감성 분석을 수행하는 것이다. 우리는 두 스트림 englishStream 과 translatedSream 에

---

[13] 키 재생성은 관련 데이터가 동일 스트림 태스크에 함께 배치됨을 보장하는데 유용하며 이는 데이터 집계 및 조인에 중요하다. 다음 장에서 이를 세부적으로 논의할 것이다.

대해 이 로직을 복제할 수 있다. 그러나 불필요한 코드 복제 대신 이들 두 스트림을 병합하는 것이 더 나을 것이다. 다음 절에 스트림 병합하는 법을 논의할 것이다.

스트림 병합하기

카프카 스트림즈는 복수의 스트림을 단일 스트림으로 결합하는 쉬운 메소드를 제공한다. 스트림 병합은 분기 연산의 반대로 간주될 수 있으며 애플리케이션에서 동일 처리 로직이 하나 이상의 스트림에 적용될 수 있을 때 일반적으로 사용된다.

SQL 에서 merge 연산자에 해당하는 것은 union 쿼리이다. 예를 들어

```
SELECT column_name(s) FROM table1
UNION
SELECT column_name(s) FROM table2;
```

이제 비영어 스트림 내 모든 트윗을 번역하였고 감성 분석을 수행해야 할 2 개의 별도 스트림이 존재한다: englishStream 과 translatedStream. 더구나 감성 점수로 보강할 모든 트윗은 하나의 출력 토픽인 crypto-sentiment 로 써져야 할 것이다. 이는 이러한 별도의 데이터 스트림에 대해 동일한 스트림/싱크 프로세서를 재사용할 수 있기 때문에 병합에 대한 이상적인 유스케이스이다.

그림 3-6 의 다이어그램은 무엇을 달성하려고 하는지를 보여준다.

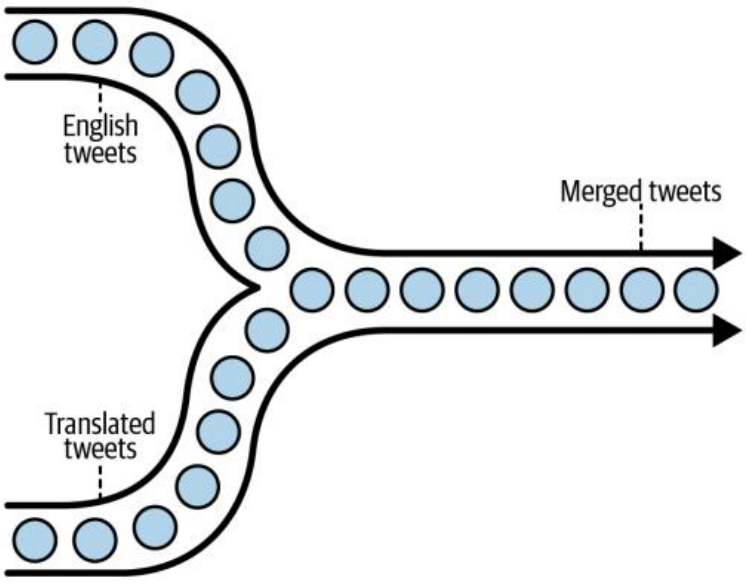

그림 3-6. 병합 연산은 복수의 스트림을 단일 스트림으로 결합한다.

스트림 병합 코드는 매우 간단하다. 다음과 같이 merge 연산자에 결합하려고 하는 스트림들을 전달하기만 하면 된다:

```
KStream merged = englishStream.merge(translatedStream);
```

스트림이 결합되었고 이제 다음 단계로 이동할 준비를 마쳤다.

트윗 보강하기

이제 감성 점수로 각 트윗을 보강할 수 있다는 목표에 가까워지고 있다. 그러나 알아챘듯이 현재 데이터 클래스인 Tweet 은 소스 토픽 내 원시 트윗의 구조를 나타낸다. 출력 토픽인 crypto-sentiment 에 쓸 보강된 레코드를 표현하기 위해 새로운 데이터 클래스가 필요하다. 이 시점에서 JSON 을 사용하여 데이터를 직렬화하는 대신 Avro 데이터 직렬화 포맷을 사용할 것이다. Avro 에 대해 깊이 살펴보고 카프카 스트림즈 애플리케이션에서 보강된 레코드를 표현하기 위해 Avro 데이터 클래스를 생성하는 법을 배워보자.

Avro 데이터 클래스

Avro 는 (높은 처리량을 요하는 애플리케이션에 유용한) 간결한 바이트 표현, 레코드 스키마의 기본 지원[14] 과 카프카 스트림즈와 함께 잘 동작하고 시작부터 Avro 를 강력하게 지원하는 스키마 레지스트리라는 스키마 관리 툴[15]로 인해 카프카 커뮤니티에서 인기가 높은 포맷이다. 또한 다른 장점도 갖고 있다. 예를 들어 카프카 커넥터가 다운스트림 데이터 저장소의 테이블 구조를 자동적으로 추론하기 위해 Avro 스키마를 사용할 수 있으며 따라서 이 포맷으로 출력 레코드를 인코딩하는 것은 데이터 통합 다운스트림에 도움이 될 수 있다.

Avro 로 작업을 수행할 때 일반 레코드 또는 특수 레코드를 사용할 수 있다. 일반 레코드는 레코드 스키마가 런타임 시 알려져 있지 않을 때 적당하다. 이는 일반 getter 과 setter 을 사용하여 필드 이름에 액세스할 수 있도록 한다. 예를 들어 GenericRecord.get(String key) 및 GenericRecord.put(String key, Object value).

반면 특수 레코드는 Avro 스키마 파일로부터 생성되는 Java 클래스이다. 이들은 레코드 데이터 액세스에 대해 보다 우수한 인터페이스를 제공한다. 예를 들어 EntitySentiment 라는 특수 레코드 클래스를 생성한다면 각 필드 이름에 대해 전용 getter/setter 을 사용하여 필드에 액세스할 수 있다. 예를 들어 entitySentiment.getSentimentScore()[16].

애플리케이션이 출력 레코드 포맷을 정의하고 있기 때문에 (따라서 스키마가 빌드 타임에 알려져 있기 때문에) 특수 레코드를 생성하기 위해 Avro 를 사용할 것이다 (이제부터 데이터 클래스로 간주). Avro 클래스에 대해 스키마 정의를 추가할 좋은 장소는 카프카 스트림즈 프로젝트의 src/main/avro

---

[14] 레코드 스키마는 레코드에 대한 필드 이름과 타입을 정의한다. 이를 통해 다른 애플리케이션과 서비스 간 데이터 포맷에 대해 강력한 계약을 제공할 수 있다.

[15] 컨플루언트 플랫폼 5.5 이후 Protobuf와 JSON 스키마 모두 지원된다. https://oreil.ly/4hsQh 참조.

[16] 여기서 entitySentiment는 Avro가 생성한 EntitySentiment 클래스의 인스턴스로 이 절의 후반부에 생성할 것이다.

디렉토리이다. 예제 3-7 은 감성으로 보강된 출력 레코드에 대해 사용할 Avro 스키마 정의를 보여준다. 이 스키마를 src/main/avro 디렉토리 내에 entity_sentiment.avsc 파일로 저장하기 바란다.

예제 3-7. 보강된 트윗에 대한 Avro 스키마

```
{
        "namespace": "com.magicalpipelines.model", ①
        "name": "EntitySentiment", ②
        "type": "record",
        "fields": [
        {
                "name": "created_at",
                "type": "long"
        },
        {
                "name": "id",
                "type:" "long"
        },
        {
                "name": "entity",
                "type": "string"
        },
        {
                "name": "text",
                "type": "string"
        },
        {
                "name": "sentiment_score",
                "type": "double"
        },
        {
                "name": "sentiment_magnitude",
                "type": "double"
        },
        {
                "name": "salience",
                "type": "double"
        }]
}
```

① 데이터 클래스에 대해 원하는 패키지 이름

② Avro 기반 데이터 모델을 포함할 Java 클래스 이름. 추후 스트림 처리 단계에서 이 클래스가 사용될 것이다.

이제 스키마를 정의하였고 이 정의로부터 데이터 클래스를 생성해야 한다. 이를 위해 프로젝트에 일부 의존성을 추가해야 한다. 이는 프로젝트의 build.gradle 파일에 다음 라인을 추가함으로써 달성될 수 있다:

```
plugins {
    id 'com.commercehub.gradle.plugin.avro' version '0.9.1' ①
}
   dependencies {
    implementation 'org.apache.avro:avro:1.8.2' ②
}
```

① 이 Gradle 플러그인이 예제 3-7 에서 생성했던 것과 같이 Avro 스키마 정의로부터 Java 클래스를 자동생성하기 위해 사용된다.

② 이 의존성은 Avro 를 통해 작업하기 위해 핵심 클래스를 포함한다.

이제 프로젝트를 빌드할 때 EntitySentiment 라는 새로운 데이터 클래스가 자동적으로 생성될 것이다[17]. 이 생성된 데이터 클래스는 트윗 감성 저장을 위한 새로운 일련의 필드 (sentiment_score, sentiment_magnitude 및 salience)와 해당 getter/setter 메소드를 포함한다. 데이터 클래스가 준비되었고 트윗에 감성 점수를 추가하여 진행할 수 있다. 이는 데이터 변환에 매우 유용한 새로운 일련의 DSL 연산자를 도입할 것이다.

감성 분석

우리는 이미 map 과 mapValues 를 사용함으로써 트윗 번역 단계 동안 레코드를 변환하는 법을 봤다. 그러나 map 과 mapValues 모두 수신하는 각 입력 레코드에 대해 정확히 하나의 출력 레코드를 산출한다. 어떤 경우에는 하나의 입력 레코드에 대해 0, 하나 또는 복수의 레코드를 산출하길 원할 수도 있다.

복수의 암호화폐를 언급하는 트윗 예를 고려해보자:

```
#bitcoin is looking super strong. #ethereum has me worried though
```

이 가공 트윗은 2 개의 암호화폐 또는 엔티티 (비트코인과 이더리움)를 명시적으로 언급하고 있으며 2 개의 별도 감정을 포함하고 있다 (비트코인에 대해서는 긍정적, 이더리움에 대해서는 부정적인 감정). 현대 자연어 처리 (Natural Language Processing, NLP) 라이브러리와 서비스는 텍스트 내 각 엔티티에

---

[17] Gradle Avro 플러그인은 src/main/avro 디렉토리를 자동으로 탐색하여 찾은 스키마 파일을 Avro 컴파일러에 전달할 것이다.

대한 감성을 계산하는데 충분하며 따라서 위와 같은 하나의 입력 문자열이 복수의 출력 레코드를 야기할 수 있다. 예를 들어,

```
{"entity": "bitcoin", "sentiment_score": 0.80}
{"entity": "ethereum", "sentiment_score": -0.20}
```

다시 한번 비즈니스 문제를 벗어나 이 요건을 고려해보자. 여기서는 매우 일반적인 스트림 처리 태스크가 존재한다: 하나의 입력 레코드를 가변 수의 출력 레코드로 변환해야 한다. 운좋게도 카프카 스트림즈는 이 유스케이스에 도움이 될 수 있는 2 개의 연산자를 포함하고 있다.

- flatMap
- flatMapValues

flatMap 과 flatMapValues 모두 호출될 때마다 0, 하나 또는 복수의 출력 레코드를 산출할 수 있다. 그림 3-7 은 입력과 출력 레코드의 1:N 매핑을 시각화하고 있다.

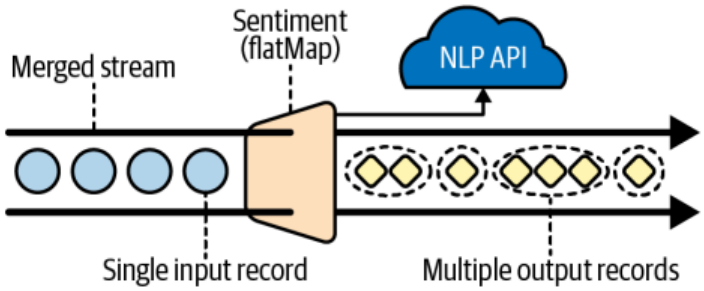

그림 3-7. flatMap 연산은 하나의 입력 레코드를 0 이상의 출력 레코드로 변환할 수 있도록 한다.

앞에서 보았던 map 연산과 유사하게 flatMap 은 새로운 레코드 키와 값을 설정하도록 요구한다. 반면 flatMapValues 는 새로운 값만 지정할 것을 요구한다. 이 예에서는 레코드 키를 처리할 필요가 없기 대문에 다음 코드 블록과 같이 트윗에 대한 엔티티 수준 감성 분석을 수행하기 위해 flatMapValues 를 사용할 것이다 (Avro 기반 데이터 클래스 EntityStentiment 를 사용하고 있음을 주목하기 바란다):

```
KStream<byte[], EntitySentiment> enriched =
 merged.flatMapValues(
        (tweet) -> {
                List<EntitySentiment> results =
                        languageClient.getEntitySentiment(tweet); ①
                results.removeIf(
                        entitySentiment -> !currencies.contains(
                                entitySentiment.getEntity())); ②
                return results; ③
        });
```

① 트윗 내 각 엔티티에 대한 감성 점수 목록을 얻는다.

② 추적하고 있는 암호화폐 중 하나와 일치하지 않는 모든 엔티티를 제거한다. 이들을 모두 제거한 후 최종 리스트 크기는 가변 (flatMap 과 flatMapValues 반환 값의 주요 특성)적으로 0 이상의 항목을 가질 수 있다.

③ flatMap 과 flatMapValues 연산자가 어떤 수의 레코드도 반환할 수 있기 때문에 출력 스트림에 추가되어야 하는 모든 레코드를 포함하는 리스트를 반환할 것이다. 카프카 스트림즈는 반환하는 모음의 flattening 을 처리할 것이다 (리스트 내 각 요소를 스트림 내의 별도 레코드로 나눌 것이다).

mapValues 사용에 대한 권고와 유사하게 가능한 flatMap 대신 flatMapValues 사용이 권고된다. 왜냐하면 카프카 스트림즈가 아마도 프로그램을 보다 효율적으로 실행시킬 수 있도록 하기 때문이다.

이제 마지막 단계를 해결할 준비를 마쳤다: 보강 데이터 (감성 점수)를 새로운 출력 토픽에 쓰기. 이를 위해 생성했던 Avro 인코딩된 EntitySentiment 레코드를 직렬화하기 위해 사용될 수 있는 Avro Serdes 를 구축해야 한다.

Avro 데이터 직렬화하기

초기에 언급했듯이 카프카는 바이트가 들어오고 바이트가 나가는 스트림 처리 플랫폼이다. 따라서 EntitySentiment 레코드를 출력 토픽에 쓰기 위해서는 이러한 Avro 레코드를 바이트 배열로 직렬화해야 한다.

Avro 를 사용하여 데이터를 직렬화할 때 2 가지 선택이 존재한다:

- 각 레코드에 Avro 스키마를 포함한다.
- 컨플루언트 스키마 레지스트리에 Avro 스키마를 저장하고 전체 스키마 대신 각 레코드에 보다 작은 스키마 ID 만을 포함함으로써 보다 간결한 포맷을 사용한다.

그림 3-8 에서 보듯이 첫번째 방법의 장점은 카프카 스트림즈 애플리케이션과 함께 별도의 서비스를 설정하고 실행시킬 필요가 없다는 것이다. 컨플루언트 스키마 레지스트리는 Avro, Protobuf 및 JSON 스키마를 생성 및 검색하기 위한 REST 서비스이기 때문에 별도의 배치가 필요하며 따라서 유지보수 비용과 추가적인 실패지점을 도입한다. 그러나 첫번째 방법의 경우 스키마가 포함되기 때문에 메시지 크기가 보다 크다.

그러나 카프카 스트림즈 애플리케이션 외부에서 작게나마 성능을 올리려고 한다면 컨플루언트 스키마 레지스트리가 가능케 하는 보다 작은 페이로드 크기가 요구될 수 있다. 더구나 레코드 스키마와 데이터

모델의 지속적인 진화를 예상한다면 스키마 레지스트리에 포함되어 있는 스키마 호환성 검사가 향후 스키마가 안전하고 깨지지 않는 방식으로 변경됨을 보장하는데 도움이 된다[18].

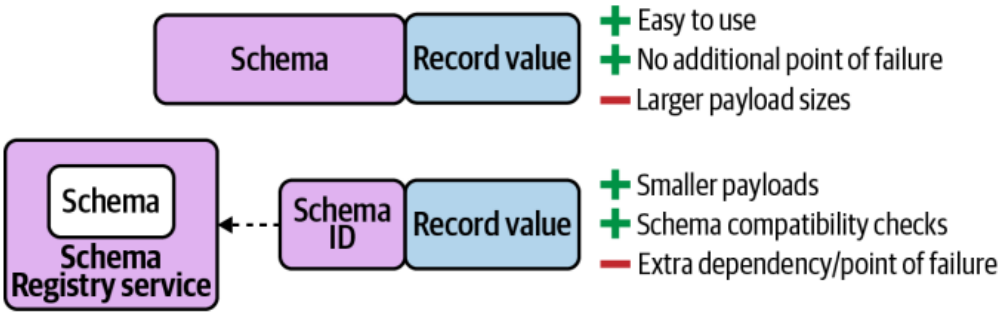

그림 3-8. 각 레코드에서 Avro 스키마를 포함할 때의 장점 및 단점

두 방법 모두에서 Avro Serdes 가 사용가능하기 때문에 Avro 데이터를 직렬화하기 위해 애플리케이션에 도입이 필요한 코드양은 최소한이다. 다음 절은 레지스트리가 없는 그리고 스키마 레지스트를 인식하는 Avro Serdes 모두를 어떻게 설정하는지를 보여준다.

레지스터리없는 Avro Serdes

레지스트리없는 Avro Serdes 를 여러분 스스로 구현할 수 있지만 com.mitchseymour:kafka-registryless-avro-serdes 패키지 내에 오픈소스로 공개하였다[19]. 여러분은 다음 코드를 통해 build.gradle 파일을 업데이트함으로써 이 패키지를 사용할 수 있다.

```
dependencies {
 implementation 'com.mitchseymour:kafka-registryless-avro-serdes:1.0.0'
}
```

카프카 스트림즈 애플리케이션에서 이 Serdes 를 사용할 필요가 있을 때마다 다음과 같이 AvroSerdes.get 메소드에 단지 Avro 생성 클래스를 제공하기 바란다:

```
AvroSerdes.get(EntitySentiment.class)
```

결과적으로 Serdes 는 여러분이 카프카의 내장 Serdes 중 하나를 일반적으로 사용하는 모든 곳에서 사용할 수 있다.

스키마 레지스트리 – Aware Avro Serdes

컨플루언트는 스키마 레지스트리 인식 Avro Serdes 를 배포하기 위한 패키지를 발표하였다. 컨플루언트의 스키마 레지스트리를 활용하고자 한다면 build.gradle 파일을 다음과 같이 업데이트하기 바란다:

---

[18] 스키마 호환성에 대한 더욱 자세한 정보는 Gwen Shapira의 2019년 기사를 확인하기 바란다.

[19] 레지스트리 없는 Avro Serdes의 코드 저장소는 https://oreil.ly/m1kk7에서 찾을 수 있다.

```
repositories {
        mavenCentral()
        maven {
                url https://packages.confluent.io/maven/ ①
        }
}
dependencies {
        implementation ('io.confluent:kafka-streams-avro-serde:6.0.1') { ②
                exclude group: 'org.apache.kafka', module: 'kafka-clients' ③
        }
}
```

① 컨플루언트 Maven 저장소를 추가한다. 이 저장소는 스키마 레지스트리 인식 Avro Serdes artifact 가 존재하는 곳이다.

② 스키마 레지스트리 인식 Avro Serdes 를 사용하기 위해 필요한 의존성을 추가한다.

③ 이 책 작성 시점에 kafka-streams-avro-serdes 에 포함된 비호환되는 일시적 의존성을 배제한다[20].

스키마 레지스트리 인식 Avro Serdes 는 추가적인 설정을 필요로 하며 따라서 프로젝트 내 각 데이터 클래스에 대한 Serdes 인스턴스를 인스턴스화하기 위해 팩토리 클래스를 생성함으로써 코드 가독성을 개선할 수 있다. 예를 들어 다음 코드 블록은 TweetSentiment 클래스에 대해 레지스트리 인식 Avro Serdes 를 생성하는 법을 보여준다:

```
public class AvroSerdes {
    public static Serde<EntitySentiment> EntitySentiment(
        String url, boolean isKey) {
                Map<String, String> serdeConfig =
                        Collections.singletonMap("schema.registry.url", url); ①
                Serde<EntitySentiment> serde = new SpecificAvroSerde<>();
                serde.configure(serdeConfig, isKey);
                return serde;
        }
}
```

① 레지스트리 인식 Avro Serdes 는 스키마 레지스트리 엔드 포인트를 설정할 것을 요구한다.

이제 다음 코드를 통해 카프카 스트림즈 애플리케이션에 필요한 곳에서 Serde 를 인스턴스화할 수 있다.

---

[20] Kafka-streams-avro-serde 라이브러리의 향후 버전 사용 시에는 불필요할 수도 있으며 이 책 작성 시점에 kafka-streams-avro-serde 최신 버전 (6.0.1)은 카프카 스트림즈 2.7.0의 의존성과 충돌한다.

AvroSerdes.EntitySentiment("http://localhost:8081", false) ①

① 스키마 레지스트리 엔드 포인트를 적절한 값으로 업데이트한다.

스키마 등록, 수정 또는 목록화를 위해 직접 스키마 레지스트리와 상호작용할 수 있다. 그러나 카프카 스트림즈에서 레지스트리 인식 Avro Serdes 를 사용할 때 스키마는 자동으로 등록될 것이다. 더구나 성능을 향상시키기 위해 레지스트리 인식 Avro Serdes 는 스키마 ID 와 스키마를 로컬에서 캐싱함으로써 스키마 조회 수를 최소화한다.

이제 Avro Serdes 를 완료했고 싱크 프로세서를 생성할 수 있다.

싱크 프로세서 추가하기

마지막 단계는 보강 데이터를 출력 토익인 crypto-sentiment 에 쓰는 것이다.

이를 위해 몇 개의 연산자가 존재한다:

- to
- through
- repartition

추가적으로 연산자/스트림 처리 로직을 추가하기 위해 새로운 KStream 인스턴스를 반환하고자 한다면 repartition 또는 through 연산자를 사용해야 한다 (후자는 이 책 발표 전에 유지보수가 중단되었지만 아직까지 널리 사용되며 후방 호환성이 예상된다). 내부적으로 이 연산자는 builder.stream 을 호출하며 따라서 이들을 사용하는 것은 카프카 스트림즈에 의해 생성되는 추가적인 서브 토폴로지를 야기할 것이다 (서브 토폴로지 참조). 그러나 스트림에서 마지막 단계에 도달한다면 KStream 에 어떤 스트림 프로세서도 추가될 필요가 없기 때문에 void 를 반환하는 to 연산자를 사용해야 한다.

이 예에서는 프로세서 토폴로지의 최종 단계에 도달했기 때문에 to 연산자를 사용할 것이고 또한 스키마 진화에 대해 보다 나은 지원과 보다 작은 메시지 크기를 원하기 때문에 스키마 레지스트리 인식 Avro Serdes 를 사용할 것이다. 다음 코드는 싱크 프로세스를 추가하는 법을 보여준다:

```
enriched.to(
        "crypto-sentiment",
        Produced.with(
                Serdes.ByteArray(),
                AvroSerdes.EntitySentiment("http://localhost:8081", false)));
```

이제 (그림 3-1 의) 프로세서 토폴로지의 단계 각각을 구현하였다. 마지막 단계는 코드를 실행하여 예상한데로 동작하는지를 확인하는 것이다.

코드 실행하기

애플리케이션을 실행하기 위해 카프카 클러스터와 스키마 레지스트리 인스턴스를 구동해야 한다. 이 튜토리얼의 소스 코드에는 이의 달성을 돕기 위해 도커 컴포즈 환경을 포함하고 있다[21]. 카프카 클러스터와 스키마 레지스트리가 동작하고 있다면 다음 명령을 통해 카프카 스트림즈 애플리케이션을 시작할 수 있다:

```
./gradlew run --info
```

이제 테스트할 준비를 마쳤다. 다음 절에서는 애플리케이션이 예상한데로 동작하는지를 확인하는 방법을 보여준다.

경험적 확인

2 장에서 언급했듯이 애플리케이션이 예상한데로 동작하는지를 확인하는 가장 쉬운 방법 중 하나는 경험적인 확인을 통해서이다. 이는 로컬 카프카 클러스터에서 데이터를 생성하고 이후 출력 토픽에 쓰여진 데이터를 관찰하는 것을 포함한다. 이를 수행하는 가장 쉬운 방법은 텍스트 파일에 tweet 레코드 예를 저장하고 이들 레코드를 소스 토픽인 tweets 에 쓰기 위해 kafka-console-producer 를 사용하는 것이다.

소스 코드는 테스팅에 사용할 2 개의 레코드를 갖는 test.json 파일을 포함하고 있다. 비고: 예제 레코드는 test.json 내에 flattened 되어 있으며 예제 3-8 에는 가독성을 위해 각 레코드의 pretty 버전을 나타냈다.

예제 3-8. 카프카 스트림즈 애플리케이션 테스팅을 위한 2 개의 트윗 예

```
{
        "CreatedAt": 1577933872630,
        "Id": 10005,
        "Text": "Bitcoin has a lot of promise. I'm not too sure about #ethereum",
        "Lang": "en",
        "Retweet": false, ①
        "Source": "",
        "User": {
                "Id": "14377871",
                "Name": "MagicalPipelines",
                "Description": "Learn something magical today.",
                "ScreenName": "MagicalPipelines",
                "URL": "http://www.magicalpipelines.com",
                "FollowersCount": "248247",
                  "FriendsCount": "16417"
        }
```

[21] 카프카 클러스터와 스키마 레지스트리 작동 관련 지시는 https://oreil.ly/ DEoaJ에서 찾을 수 있다.

```
}
{
        "CreatedAt": 1577933871912,
        "Id": 10006,
        "Text": "RT Bitcoin has a lot of promise. I'm not too sure about #ethereum",
        "Lang": "en",
        "Retweet": true, ②
        "Source": "",
        "User": {
                "Id": "14377870",
                "Name": "Mitch",
                "Description": "",
                "ScreenName": "Mitch",
                "URL": "http://mitchseymour.com",
                "FollowersCount": "120",
                "FriendsCount": "120"

}
```

① 첫번째 트윗 (ID 10005)는 리트윗이 아니며 감성 점수로 보강될 것으로 예상된다.

② 두번째 트윗 (ID 10006)은 리트윗으로 이 레코드는 무시될 것으로 예상된다.

이제 다음 명령을 사용하여 로컬 카프카 클러스터에 예제 레코드를 생산해보자:

```
kafka-console-producer ₩
        --bootstrap-server kafka:9092 ₩
        --topic tweets < test.json
```

다은 탭에서 보강 레코드를 소비하기 위해 kafka-console-consumer 를 사용해보자. 다음 명령을 실행한다:

```
kafka-console-consumer ₩
        --bootstrap-server kafka:9092 ₩
        --topic crypto-sentiment ₩
        --from-beginning
```

많은 이상한 기호화 함께 일부 비밀스럽게 보이는 출력을 봐야 한다:

```
����[��|Bitcoin has a lot of promise.
I'm not too sure about #ethereumbitcoin`ff?`ff? -��?
����[��|Bitcoin has a lot of promise.
I'm not too sure about #ethereumethereum���ㄱ���ㄱ���?
```

이는 Avro 가 바이너리 포맷이기 때문이다. 스키마 레지스트리를 사용하고 있다면 Avro 데이터의 가독성을 개선시키는 컨플루언트 개발 특수 컨솔 스크립트를 사용할 수 있다. 다음과 같이 단순히 kafka-console-consumer 을 kafka-avro-console-consumer 로 변경하기 바란다:

```
kafka-avro-console-consumer ₩
        --bootstrap-server kafka:9092 ₩
        --topic crypto-sentiment ₩
        --from-beginning
```

마지막으로 다음과 비슷한 출력을 봐야 한다:

```
{
 "created_at": 1577933872630,
 "id": 10005,
 "text": "Bitcoin has a lot of promise. I'm not too sure about #ethereum",
 "entity": "bitcoin",
 "sentiment_score": 0.699999988079071,
 "sentiment_magnitude": 0.699999988079071,
 "salience": 0.47968605160713196
}
{
 "created_at": 1577933872630,
 "id": 10005,
 "text": "Bitcoin has a lot of promise. I'm not too sure about #ethereum",
 "entity": "ethereum",
 "sentiment_score": -0.20000000298023224,
 "sentiment_magnitude": -0.20000000298023224,
 "salience": 0.030233483761548996
}
```

트윗 ID 10006 은 출력에 나타나지 않음을 주목하기 바란다. 이는 리트윗으로 그림 3-1 의 2 단계에서 필터링되었다. 또한 트윗 ID 10005 가 2 개의 출력 레코드에 존재함을 주목하기 바란다. 이 트윗이 2 개의 별개 암호화폐를 언급했기 때문에 이는 예상된 바이며 flatMapValues 연산자가 예상한데로 동작했음을 확인한다 (그림 3-1 의 6 단계).

언어 번역 단계를 확인하고 싶다면 비영어 트윗을 갖는 예제 레코드를 test.json 에 자유롭게 업데이트 하기 바란다[22]. 독자를 위한 연습으로 남겨둘 것이다.

요약

[22] 추가적인 환경 설정 단계가 필요할 수 있으며 이 장 튜토리얼의 README를 참조하기 바란다.

지금까지 다음을 포함하여 카프카 스트림즈를 사용하여 스테이트리스 스트림 처리 애플리케이션을 구축하기 위한 일련의 새로운 기능을 배웠다:

- Filter 과 filterNot 을 통한 데이터 필터링
- Branch 연산자를 사용한 서브 스트림 생성
- Merge 연산자를 통한 스트림 결합
- Map 과 mapValues 를 사용한 1:1 레코드 변환 수행
- flatMap 과 flatMapValues 를 사용한 1:N 레코드 변환 수행
- to, through 및 repartition 을 사용하여 출력 토픽에 레코드 쓰기
- 맞춤형 직렬화기, 역직렬화기 및 Serdes 구현을 사용한 직렬화, 역직렬화 및 재직렬화

다음 장에서는 스트림 조인, 윈도우 및 집계를 포함하여 스테이트풀 태스크를 도입함으로써 보다 복잡한 스트림 처리 작업을 논의할 것이다.