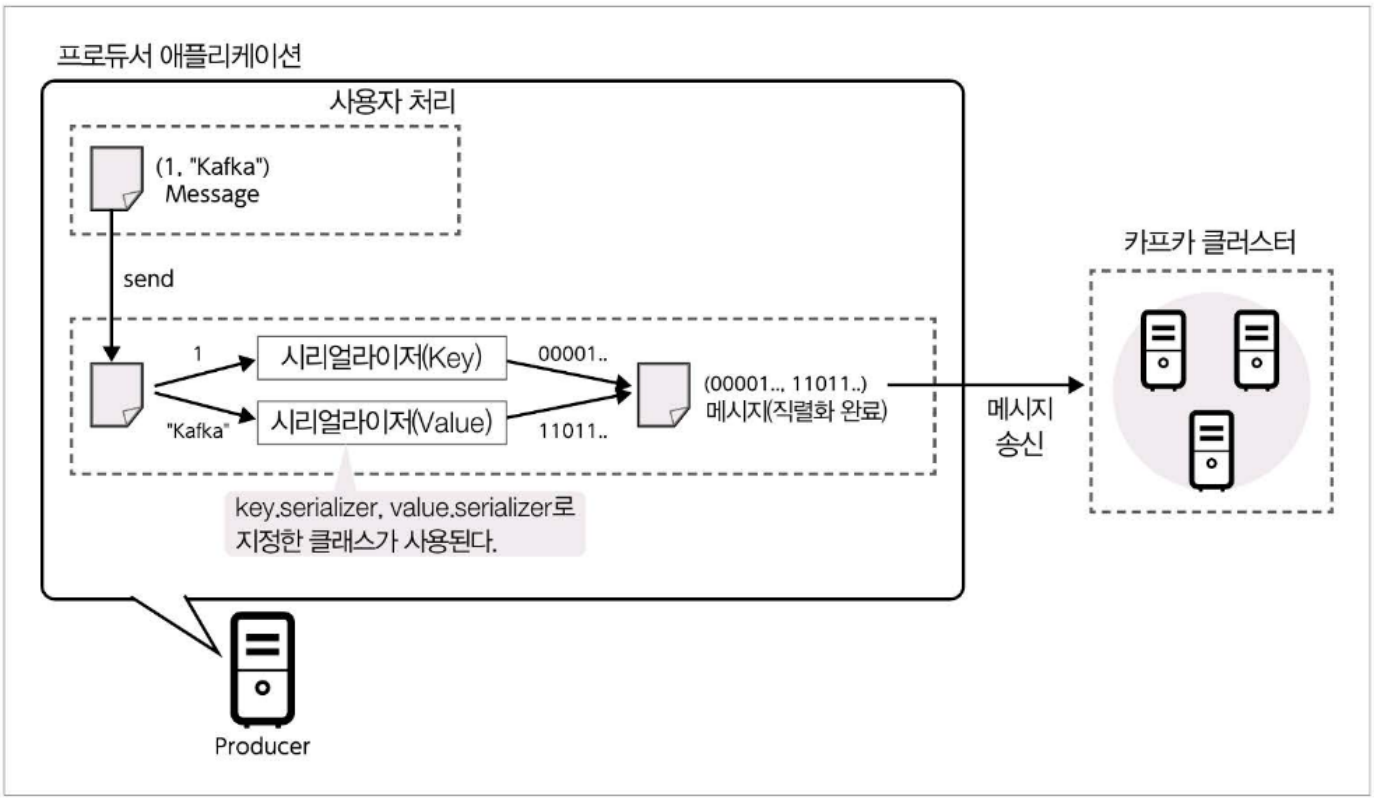
프로듀서

```
package com.kafka;
import org.apache.kafka.clients.producer.*;
import java.util.Properties;

public class FirstAppProducer {
    private static String topicName = "en0605";
    public static void main(String[] args) {
        // 1. KafkaProducer에 필요한 설정
        Properties conf = new Properties();
        conf.setProperty("bootstrap.servers",
"broker1:9092,broker2:9092,broker3:9092");
        conf.setProperty("key.serializer",
"org.apache.kafka.common.serialization.IntegerSerializer");
        conf.setProperty("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");

        // 2. Kafka 클러스터에 메시지를 송신(produce)하는 객체를 생성
        Producer<Integer, String> producer = new KafkaProducer<>(conf);
        int key;
        String value;
        for(int i = 1; i <= 100; i++) {
            key = i;
            value = String.valueOf(i);

            // 3. 송신할 메시지를 생성
            ProducerRecord<Integer, String> record = new ProducerRecord<>(topicName,
key, value);

            // 4. 메시지를 송신하고, Ack을 받았을 때에 실행할 작업(Callback)을 등록한다
            producer.send(record, new Callback() {
                @Override
                public void onCompletion(RecordMetadata metadata, Exception e) {
                    if( metadata != null) {
                        // 송신에 성공한 경우의 처리
                        String infoString = String.format("Success partition:%d,
offset:%d", metadata.partition(), metadata.offset());
                        System.out.println(infoString);
                    } else {
                        // 송신에 실패한 경우의 처리
                        String infoString = String.format("Failed:%s",
e.getMessage());
                        System.err.println(infoString);
                    }
                }
            });
```

```
        }

        // 5. KafkaProducer를 클로즈하여 종료
        producer.close();
    }
}
```

설정

- boostrap.servers
  - Producer가 접속하는 브로커의 호스트명과 포트번호를 지정
  - <호스트명>:<포트번호>의 형식으로 저장하고 여러 브로커를 지정할 때는 쉼표로 연결
- key.serializer, value.serilizer
  - 모든 메시지가 직렬화된 상태로 전송
  - 이 직렬화 처리에 이용되는 시리얼라이저 클래스를 지정
  - 메시지의 key를 직렬화하는데 사용되고, value.serializer는 메시지의 value를 직렬화하는데 사용



- 카프카에 있는 시리얼라이저

데이터 형	카프카에서 제공되고 있는 시리얼라이저
Short	org.apache.kafka.common.serialization.ShortSerializer
Integer	org.apache.kafka.common.serialization.IntegerSerializer
Long	org.apache.kafka.common.serialization.LongSerializer
Float	org.apache.kafka.common.serialization.FloatSerializer
Double	org.apache.kafka.common.serialization.DoubleSerializer
String	org.apache.kafka.common.serialization.StringSerializer
Byte배열	org.apache.kafka.common.serialization.ByteArraySerializer
ByteBuffer	org.apache.kafka.common.serialization.ByteBuffer
Bytes[5]	org.apache.kafka.common.serialization.BytesSerializer

```
 Producer<Integer, String> producer = new KafkaProducer<>(conf);
```

- 프로듀서 인터페이스를 구현하기 위해 변수의 형을 Producer로 설정
- KafkaProducer의 객체를 작성할 때 형 파라미터를 지정
- 이것은 각각 송신하는 메시지의 Key와 Value의 형을 나타냄
- 메시지의 Key를 정수형(Integer), Value를 문자열형(String)으로 하고 있음
- 여기에서 지정한 형은 먼저 지정한 시리얼라이저와 대응하고 있음

송신

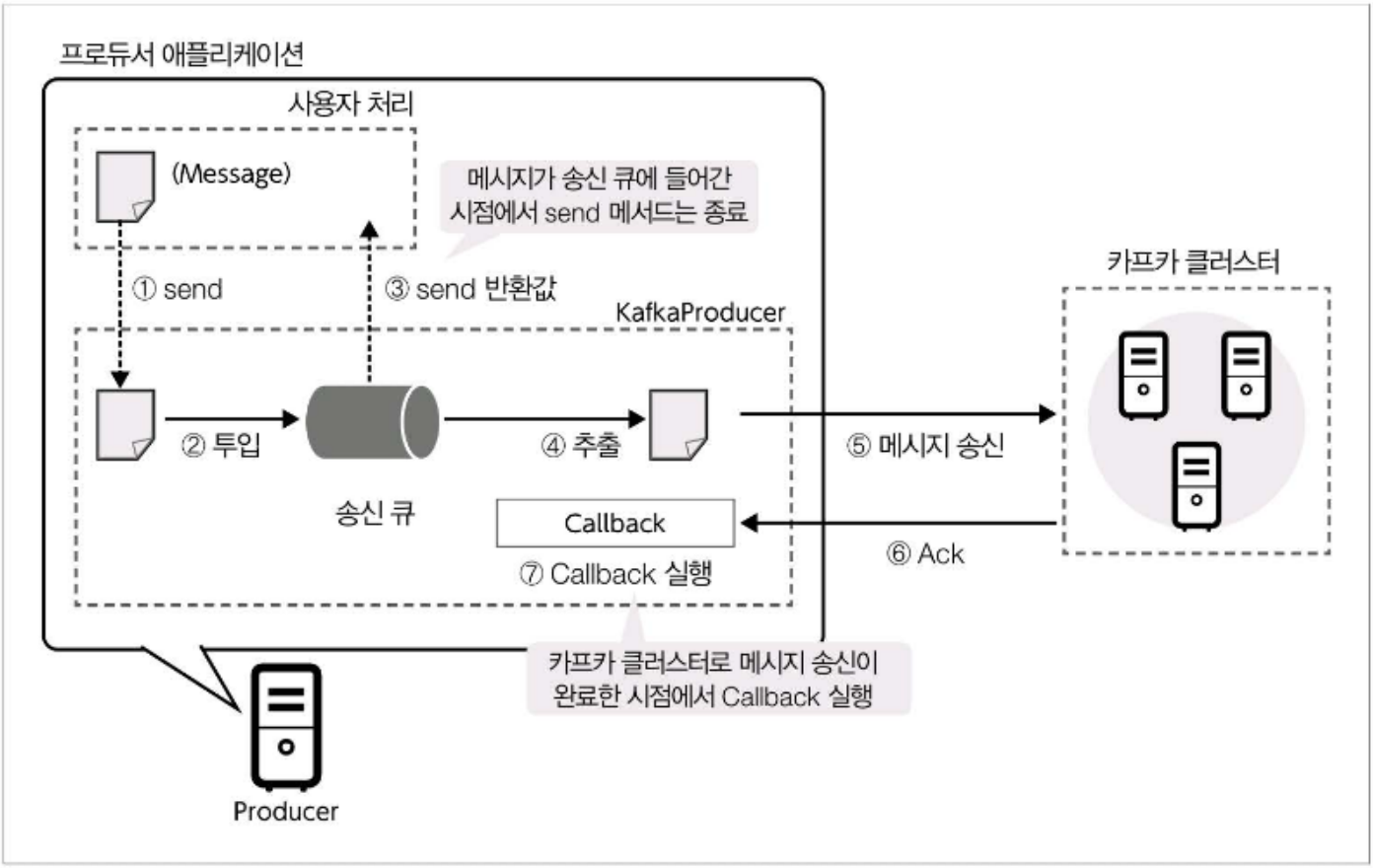
```
 ProducerRecord<Integer, String> record = new ProducerRecord<>(topicName, key,
value);
```

- KafkaProducer를 이용하여 메시지를 보낼 때는 송신 메시지를 ProducerRecord라는 객체에 저장
- 이 때 메시지의 Key, Value 외에 송신처의 토픽도 함께 등록
- ProducerRecord에도 형 파라미터가 있으므로 KafkaProducer의 객체를 만들 때와 동일하게 지정

```
// 4. 메시지를 송신하고, Ack을 받았을 때에 실행할 작업(Callback)을 등록한다
producer.send(record, new Callback()
                        {
    @Override
    public void onCompletion(RecordMetadata metadata, Exception e) {
        if( metadata != null) {
           // 송신에 성공한 경우의 처리
           String infoString = String.format("Success partition:%d, offset:%d",
metadata.partition(), metadata.offset());
           System.out.println(infoString);
         } else {
```

```
            // 송신에 실패한 경우의 처리
            String infoString = String.format("Failed:%s", e.getMessage());
            System.err.println(infoString);
            }
    }
});
```

- ProducerRecord 객체뿐만 아니라 Callback 클래스를 구현하여 KafkaProducer에 전달
- Callback 클래스에서 구현하고 있는 onCompletion 메서드에서는 송신을 완료 했을 때 실행되어야 할 처리를 하고 있다
- KafkaProducer의 송신 처리는 비동기적으로 이루어지기 때문에 send 메서드를 호출했을 때 발생하지 않는다.
- Send 메서드의 처리는 KafkaProducer의 송신 큐에 메시지를 넣을 뿐이다
- 송신 큐에 넣은 메시지는 사용자의 애플리케이션과는 다른 별도의 스레드에서 순차적으로 송신된다.
- 메시지가 송신된 경우 카프카 클러스트에서 Ack가 반환
- Callback 클래스의 메서드는 그 Ack를 수신했을 때 처리



- Callback 클래스의 메서드는 메시지 송신에 성공했을 때와 실패했을 때 동일한 내용이 호출
- 성공했을 때는 metadata가 null이 아닌 객체
- 실패했을 때는 null이 된다.
- 그렇기 때문에 if문을 사용해서 성공 및 실패 로직 작성

컨슈머

```java
package com.kafka;

import org.apache.kafka.clients.consumer.*;
import org.apache.kafka.common.TopicPartition;
import java.util.*;

public class FirstAppConsumer {

    private static String topicName = "en0605";

    public static void main( String[] args ) {

        // 1. KafkaConsumer에 필요한 설정
        Properties conf = new Properties();
        conf.setProperty("bootstrap.servers",
"broker1:9092,broker2:9092,kafka3:9092");
        conf.setProperty("group.id", "FirstAppConsumerGroup");
        conf.setProperty("enable.auto.commit", "false");
        conf.setProperty("key.deserializer",
"org.apache.kafka.common.serialization.IntegerDeserializer");
        conf.setProperty("value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");

        // 2. Kafka클러스터에서 Message를 수신(Consume)하는 객체를 생성
        Consumer<Integer, String> consumer = new KafkaConsumer<>(conf);

        // 3. 수신(subscribe)하는 Topic을 등록
        consumer.subscribe(Collections.singletonList(topicName));

        for(int count = 0; count < 300; count++) {
            // 4. Message를 수신하여, 콘솔에 표시한다
            ConsumerRecords<Integer, String> records = consumer.poll(1);
            for(ConsumerRecord<Integer, String> record: records) {
                String msgString = String.format("key:%d, value:%s", record.key(),
record.value());
                System.out.println(msgString);

                // 5. 처리가 완료한 Message의 Offset을 Commit한다
                TopicPartition tp = new TopicPartition(record.topic(),
record.partition());
                OffsetAndMetadata oam = new OffsetAndMetadata(record.offset() + 1);
                Map<TopicPartition, OffsetAndMetadata> commitInfo =
Collections.singletonMap(tp, oam);
                consumer.commitSync(commitInfo);
            }
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ex ) {
                ex.printStackTrace();
```

```
            }
        }

        // 6. KafkaConsumer를 클로즈하여 종료
        consumer.close();
    }
}
```

설정

- boostrap.servers
  - 접속할 브로커의 호스트명과 포트 번호
- group.id
  - 작성할 KafkaConsumer가 속한 Consumer Group을 지정
- enable.auto.commit
  - 오프셋 커밋을 자동을 실행할지의 여부 지정
- key.deserializer, value.deserializer
  - 카프카에 송신되는 모든 메시지가 직렬화되는데 이 데이터를 다시 역질렬화 처리에 이용
  - 프로듀서에서 지정한 시리얼라이저에 해당하는 것

```
Consumer<Integer, String> consumer = new KafkaConsumer<>(conf);
```

수신

- 메시지를 수신하는 토픽을 구독할 필요가 있음

```
consumer.subscribe(Collections.singletonList(topicName));
```

- subscribe 메서드에 전달하는 리스트에 여러 토픽 등록함으로써 여러 토픽을 구독할 수 있음

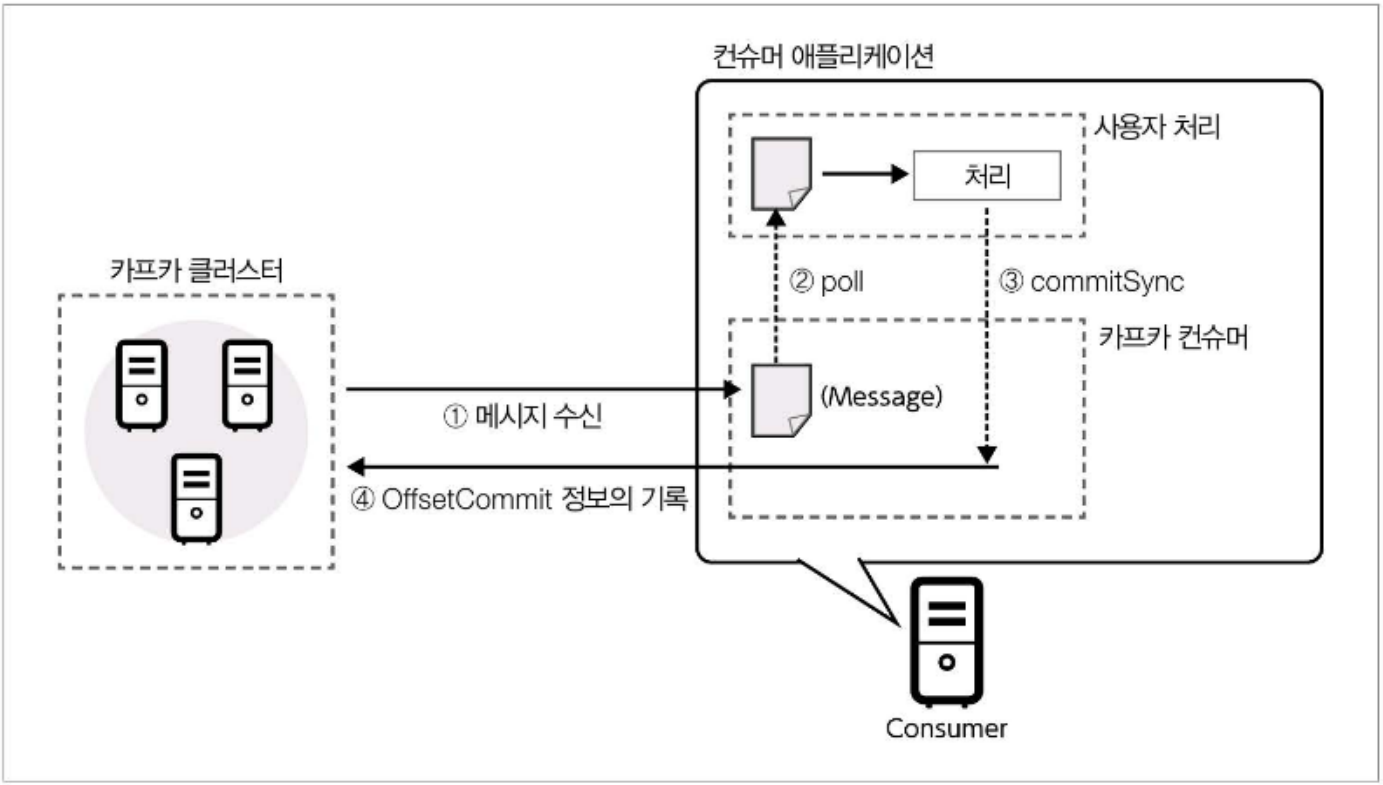
```
ConsumerRecords<Integer, String> records = consumer.poll(1);
```

- poll 메서드를 호출하여 토픽을 수신한 후에는 메시지를 받는다

```
// 5. 처리가 완료한 Message의 Offset을 Commit한다
TopicPartition tp = new TopicPartition(record.topic(),record.partition());
OffsetAndMetadata oam = new OffsetAndMetadata(record.offset() + 1);
                Map<TopicPartition, OffsetAndMetadata> commitInfo =
Collections.singletonMap(tp, oam);
                consumer.commitSync(commitInfo);
}
```

- ConsumerRecords라는 객체로 전달

- 이 객체에서 수신된 여러 메시지의 Key, Value, 타임스탬프등 메타 데이터가 포함되어 있음
- for 문으로 순서대로 처리해 콘솔에 출력
- 설정에서 commit를 False로 설정했기에 메시지 처리가 완료 될 때마다 오프셋을 커밋
- 여기에서는 오프셋 커밋 정보가 카프카 클러스터로 기록이 완료될 때까지 처리를 기다리는 commitSync 메서드를 이용하고 있음
- 비동기적으로 처리하고, 처리 완료를 기다리지 않고 다음 처리로 진행하는 commitAsync라는 메시지가 있음



용어

직렬화

- Serialization
- 객체를 직렬화하여 전송 가능한 형태로 만드는 것을 의미
- 객체들의 데이터를 특정 형식으로 변환하여 시스템이나 저장소에 효율적으로 저장 및 전송하는 역할을 하며, 이를 통해 네트워크를 통해 전송하거나 파일에 저장하기 용이하게 만듬

역직렬화

- Deserialization
- 직렬화된 데이터를 원래의 객체나 데이터 구조로 복원
- 다른 시스템에서 받은 데이터를 다시 사용 가능한 형태로 변환