LAG 확인

Session timeout 시간설정

Heartbeat 시간 설정

--------------------------------------------------------------------------------------------------------------

리눅스 설치시 java설치 + kafka설치

java -version

프로퍼티 세팅

--------------------------------------------------------------------------------------------------------------

디렉토리 config로 이동 ( ex) C:\dev\kafka\_2.11-1.1.0\config) ->

server.properties 파일 텍스트 편집기로 오픈 ->

log.dirs=/tmp/kafka-logs 를 카프카 설치 경로로 변경 ->

( ex ) log.dirs=C:\\dev\\kafka\_2.11-1.1.0\\logs)

--------------------------------------------------------------------------------------------------------------

path: bin/windows

kafka-server-start

kafka-server-stop

zookeeper-server-start

zookeeper-server-stop

path: config

zookeeper와 kafka의 properties config파일

- server

- zookeeper

--------------------------------------------------------------------------------------------------------------

1) zookeeper 실행하기

# zookeeper 실행 명령어

bin\windows\zookeeper-server-start.bat config\zookeeper.properties

2) kafka 서버 실행

bin\windows\kafka-server-start.bat config\server.properties

--------------------------------------------------------------------------------------------------------------

netstat -na | findstr "2181"

1) zookeeper

TCP 0.0.0.0:2181 DESKTOP-DSOFM80:0 LISTENING

2) kafka default port

TCP 0.0.0.0:9092 DESKTOP-DSOFM80:0 LISTENING

--------------------------------------------------------------------------------------------------------------

[Kafka Test]

1. topic 생성(broker)

//.\kafka-topics.bat --create --bootstrap-server localhost:9092 --topic oingdaddy

bin\windows\kafka-topics.bat --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic test20190715

2. 생성한 topic에 message 보내기 (producer)

////.\kafka-console-producer.bat --broker-list localhost:9092 --topic oingdaddy

bin\windows\kafka-console-producer.bat --broker-list localhost:9092 --topic test20190715 --from-beginning

bin\windows\kafka-console-producer.bat --bootstrap-server localhost:9092 --topic test20190715

3. topic에 있는 message 가져오기 (consumer)

bin\windows\kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic test20190715

--------------------------------------------------------------------------------------------------------------

카프카 토픽 리스트 조회

bin\windows\kafka-topics.bat --list --zookeeper localhost:2181

.\bin\windows\kafka-topics.bat --bootstrap-server localhost:9092 --list

--------------------------------------------------------------------------------------------------------------

리스트 조회 상세

.\bin\windows\kafka-topics.bat --describe --topic quickstart-events --bootstrap-server localhost:9092

--------------------------------------------------------------------------------------------------------------

실행중인 내역 port 확인

#zookeeper 정상 실행 여부 확인

netstat -na | findstr "2181"

#kafka 정상 실행 여부 확인

netstat -na | findstr "9092"

---------------------------------------------------------------------------------------------------------------------------------------

error: kafka.common.InconsistentClusterIdException: The Cluster ID doesn't match stored clusterId Some in meta.properties. The broker is trying to join the wrong cluster. Configured zookeeper.connect may be wrong.)

카프카 설정파일(카프카 설치 디렉토리/config/server.properties)의 log.dirs 항목에서 설정되어 있는 카프카 로그 path로 가보면 meta.properties라는 파일이 있는데

해당 파일을 지워주고 카프카를 재시작해주면 해결된다.

카프카를 구성하는 주요 요소로는 아래와 같습니다.

• 주키퍼(Zookeeper): 아파치 프로젝트 애플리케이션으로 카프카의 메타데이터(metadata) 관리 및 브로커의 정상상태 점검(health check) 을 담당 합니다.

• 카프카(Kafka) 또는 카프카 클러스터(Kafka cluster) : 아파치 프로젝트 애플리케이션으로 여러대의 브로커를 구성한 클러스터를 의미 합니다.

• 브로커(broker) : 카프카 애플리케이션이 설치된 서버 또는 노드를 의미 합니다.

• 프로듀서(producer): 카프카로 메세지를 보내는 역할을 하는 클라이언트로 총칭 합니다.

• 컨슈머(consumer) : 카프카에서 메세지를 꺼내가는 역할을 하는 클라이언트를 총칭 합니다.

• 토픽(topic) : 카프카는 메시지 피드들을 토픽으로 구분하고, 각 토픽의 이름은 카프카 내에서 고유 합니다.

• 파티션(partition) : 병렬 처리 및 고성능을 얻기 위해 하나의 토픽을 여러개로 나눈 것을 의미 합니다

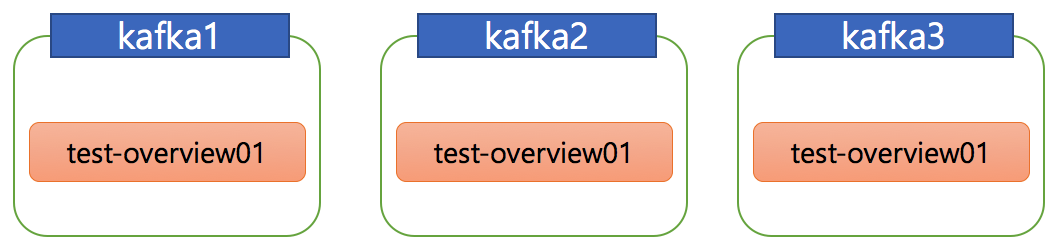
• 세그먼트(segment) : 프로듀서가 전송한 실제 메세지가 브로커의 로컬 디스크에 저장되는 파일을 말합니다.

• 메세지(message) 또는 레코드(record): 프로듀서가 브로커로 전송하거나 컨슈머가 읽어가는 데이터 조각을 말합니다.

--------------------------------------------------------------------------------------------------------------

리플리케이션

카프카에서 리플리케이션(replication) 이란 각 메세지들을 여러 개로 복제해서 카프카 클러스터내 브로커들에 분산시키는 동작을 의미 합니다.  
  
이러한 리플리케이션 동작 덕분에 하나의 브로커가 종료 되더라도 카프카는 안정성을 유지 할 수 있습니다.  
  
카프카 클러스터 설치 완료후 토픽을 생성하였을 때 --partition 1, --replication-factor 3 이라는 옵션을 이용하여 생성하였습니다.  
  
토픽 명령어 중 replication-factor 옵션은 카프카내에서 몇개의 리플리케이션을 유지하는지를 설정하는 옵션 입니다.  
  
토픽 생성시 --replication-factor 3 으로 지정하였기 때문에 원본은 포함하여 총 3개의 replication 이 있음을 의미 합니다.

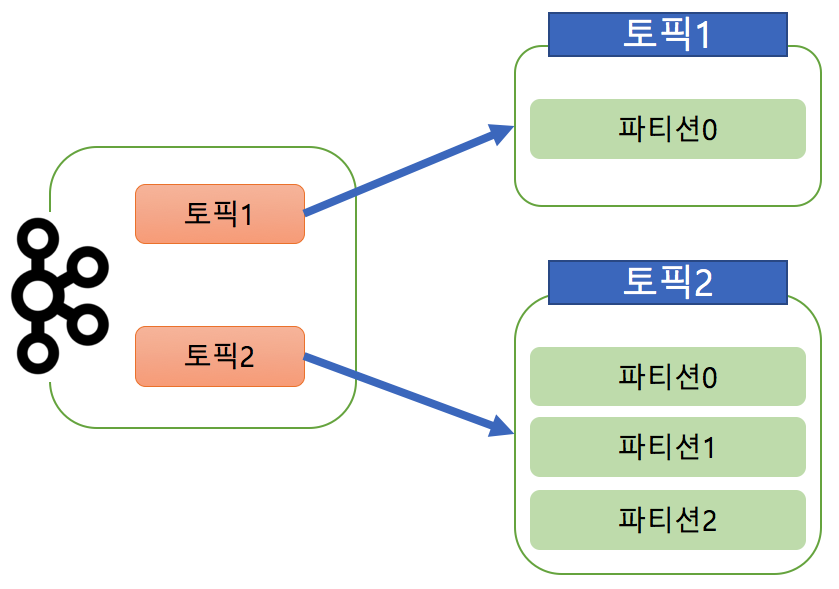
[](https://hoing.io/storage/2022/04/topic_replication_01.png)

위의 그림에서 test-overview01 토픽은 설정된 --replication-factor 옵션 값에 의해서 각 브로커에 배치된 상태를 보여주는 것 입니다.  
  
그리고 조금더 정확하게 말해서 카프카에서 토픽이 리플리케이션이 된 것이 아니라 토픽의 파티션이 리플리케이션(복제)가 된 것이라고 표현할 수 있습니다.  
  
replication-factor의 값이 높아지면 안정성이 높아지지만, 그만큼 브로커의 리소스 사용이 많아지며,복제에 대한 오버헤드가 발생될 수 있으므로 적절한 팩터수를 설정해서 사용을 해야 합니다.  
  
테스트나 개발환경 : 리플리케이션 팩터 수 1  
운영(로그성 메세지로 약간의 유실 허용가능) : 리플레키이션 팩터 수 2  
운영(유실 허용안됨) : 리플레키이션 팩터 수 3

**파티션**

하나의 토픽이 한번에 처리할 수 있는 단계를 높이기 위해서 토픽을 여러 개로 나눠 병렬 처리가 가능하게 만든 것을 파티션(partition) 이라고 합니다.

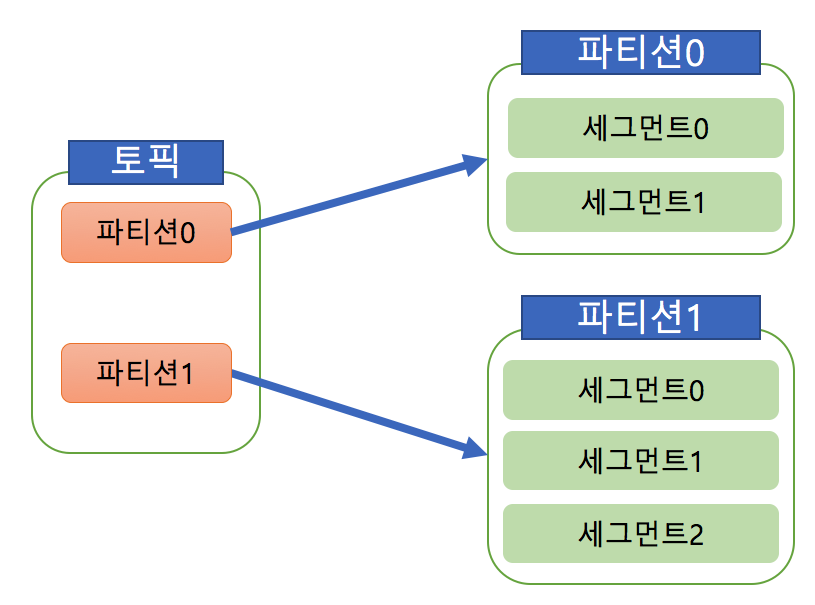
이렇게 하나를 여러개 로 나누면(파티션 하면) 분산 처리가 가능해지며, 파티션 수 만큼 컨슈머를 연결할 수 있게 됩니다.

[](https://hoing.io/storage/2022/04/topic_parittion.png)

위의 그림에서 카프카 클러스터에 있는 토픽을 파티션으로 나눈 그림 입니다.  
  
토픽1은 파티션1개로 구성 되어있고, 토픽3는 파티션0, 파티션1, 파티션2 의 총 3개의 파티션으로 구성되어 있습니다  
  
파티션 수도 토픽을 생성할 때 옵션으로 설정하게 되며, 사실 파티션 수를 정하는 기준은 다소 모호한 경우가 많습니다. 인터넷에서는 적절한 파티션 수를 구하는 공식도 간혹 있으나 각 메세지 크기나 초당 메세지 건수 등에 따라서 달라짐으로 정확한 예측은 사실 어렵습니다.  
  
특히 파티션수는 초기 생성 후 언제든지 늘릴 수 있지만, 반대로 한번 늘린 파티션 수는 절대로 줄일 수 없습니다.  
  
따라서 초기에 토픽 생성시에는 파티션수를 작게 적절하게(2또는 4정도) 생성 한 후, 메세지 처리량인 컨슈머의 LAG 등을 모니터링 하면서 조금씩 늘려가는 방법이 가장 좋습니다.  
  
여기서 컨슈머의 LAG 이란 '프로듀서가 보낸 메세지수(카프카에 남아 있는 메세지 수) - 컨슈머가 가져간 메세지 수' 를 의미 합니다.  
  
컨슈머의 지연이 없다면(모든 메세지를 가져갔다면) **5-5=0** 이 되게 됩니다

**세그먼트**

카프카에서 전송한 메세지는 어디에 저장이 될까요?  
프로듀서를 이용해 보낸 메세지는 토픽의 파티션 0에 저장되어 있습니다  
  
이처럼 프로듀서에 의해 브로커로 전송된 메세지는 토픽의 파티션에 저장되며, 각 메세지들은 세그먼트(segment) 라는 로그 파일의 형태로 브로커의 로컬 디스크에 저장되게 됩니다.

[](https://hoing.io/storage/2022/04/partition_segment.png)

위의 그림은 각 파티션별로 세그먼트를 나눠본 그림입니다. 각 파티션 마다 N 개의 세그먼트 로그 파일들이 존재 합니다.  
  
카프카가 설치된 디렉토리의 logs 디렉토리로 이동하여 살펴보면 생성한 토픽 이름으로 디렉토리가 있는 것을 확인 할 수 있습니다. 해당 디렉토리 이름 끝에 "-0" 는 0번 파티션을 의미하게 됩니다.  
  
그리고 그 디렉토리 아래에는 여러가지 파일이 있습니다

[root@kafka1]# ls -al | grep overview

drwxr-xr-x. 2 root root 4096 Apr 20 14:57 test-overview01-0

[root@kafka1]# cd test-overview01-0

[root@kafka1]# ls -alrt

total 16

-rw-r--r--. 1 root root 10485760 Apr 20 14:52 00000000000000000000.index

-rw-r--r--. 1 root root 10485756 Apr 20 14:52 00000000000000000000.timeindex

-rw-r--r--. 1 root root 8 Apr 20 14:57 leader-epoch-checkpoint

drwxr-xr-x. 2 root root 4096 Apr 20 14:57 .

-rw-r--r--. 1 root root 81 Apr 20 14:57 00000000000000000000.log

drwxr-xr-x. 20 root root 4096 Apr 20 17:41 ..

hexdump 를 보기 위해서 xxd 명령어나 strings 명령어를 이용하면 이전 테스트에서 전송한 First message 라는 메세지를 확인 할 수 있으며, 브로커의 로컬디스크에 안전하게 저장된 것을 확인할 수 있습니다.

[root@kafka1]# xxd 00000000000000000000.log

0000000: 0000 0000 0000 0000 0000 0045 0000 0000 ...........E....

0000010: 02d5 73b5 9400 0000 0000 0000 0001 8045 ..s............E

0000020: 8bb7 1f00 0001 8045 8bb7 1fff ffff ffff .......E........

0000030: ffff ffff ffff ffff ff00 0000 0126 0000 .............&..

0000040: 0001 1a46 6972 7374 206d 6573 7361 6765 ...First message

0000050: 00

[root@kafka1]# strings 00000000000000000000.log

First message

메세지 전송 과정을 다시 한번 정리해보면 아래와 같습니다.  
  
1. 프로듀서는 카프카의 test-overview01 토픽으로 메세지를 전송합니다.  
2. test-overview01 토픽은 파티션이 하나뿐임으로, 프로듀서로 부터 받은 메세지를 파티션0의 세그먼트 로그 파일에 저장합니다.  
3. 브로커의 세그먼트 로그 파일에 저장된 메세지는 컨슈머가 읽어 가게 됩니다.

핵심기능

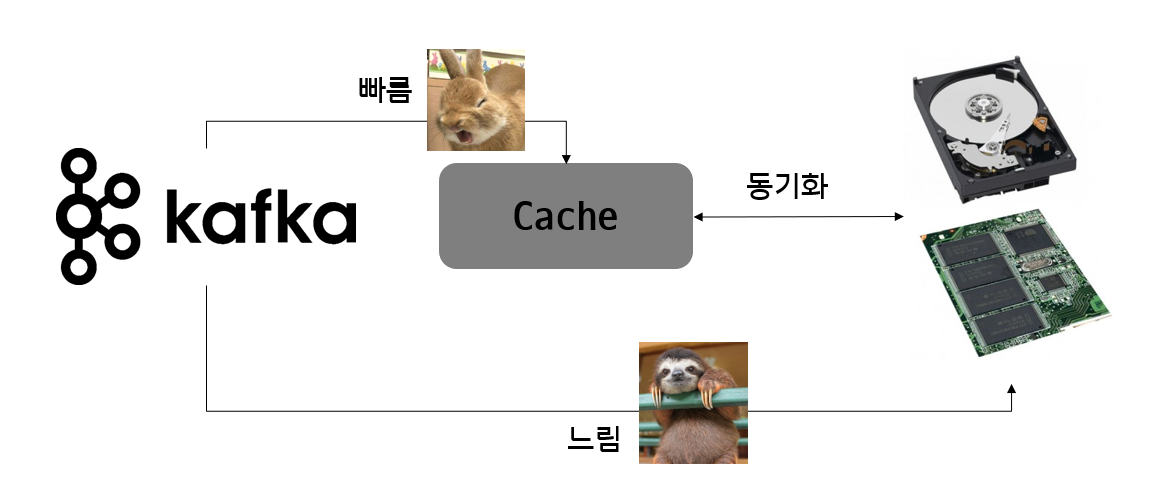
카프카는 여러 우수한 기업들이 서둘러서 도입하고 많은 개발자가 좋아하는 이유로는 앞에서 설명한 내용처럼 높은 처리량, 빠른 응답 속도, 안정성 때문 입니다.  
  
이런 카프카가 어떻게 좋은 지 어떻게 높은 처리량을 갖게 되었는지에 대해서 카프카의 특성과 핵심 기능에 대해서 살펴보도록 하겠습니다.

### **분산 시스템**

분산 시스템은 네트워크 상에서 연결된 컴퓨터들의 그룹을 말하며, 단일 시스템이 갖지 못한 높은 성능을 목표로 구성하게 됩니다. 이러한 분산 시스템은 성능이 높다는 장점 이외에도 하나의 서버 또는 노드 등에 장애가 발생할 때 다른 서버 또는 노드가 대신 처리하므로 장애 대응에 좋으며, 부하가 높을 경우에는 확장이 용이 하다는 장점도 있습니다.  
  
카프카도 이와 같은 의미로 분산 시스템이므로 최초 구성한 클러스터에서 브로커를 추가 하는 방식으로 확장이 가능 하며 카프카 브로커 추가는 온라인 상태에서 매우 간단하게 추가 할 수 있습니다.  
  
이처럼 확장이 용이 하다는 점이 카프카의 매우 큰 장점 입니다.

### **페이지 캐시**

카프카는 높은 처리량을 얻기 위해서 여러가지 추가된 기능 중에 대표적으로 페이지 캐시 의 이용 입니다.  
운영체제는 성능을 높이기 위해 계속 진화하고 개선되고 있으며, 그중 대표적인것이 페이지 캐시의 활용 입니다.

[](https://hoing.io/storage/2022/04/file_page_cache.png)

[[**https://needjarvis.tistory.com/602**](https://needjarvis.tistory.com/602)]

카프카 역시 OS의 페이지 캐시를 활용하는 방식으로 설계가 되어있으며 페이지 캐시는 애플리케이션이 사용하지 않는 잔여 메모리를 활용하며, 이러한 페이지 캐시를 이용하면 디스크 I/O에 대한 접근이 줄어 성능을 높일 수 있습니다.

### **배치 전송 처리**

카프카는 프로듀서 , 컨슈머 클라이언트들과 서로 통신하며, 이들 사이에서 수많은 메세지를 주고 받습니다.  
  
이때 발생하는 수많은 통신을 묶어서 처리할 수 있다면 네트워크 오버헤드를 줄일 수 있으며 더욱 빠르고 효율적으로 처리 할 수 있습니다.  
  
예를 들어, 온라인 상품 구매 프로세스에서 상품의 재고 수량 업데이트 작업과 구매 로그를 저장소로 보내는 작업을 한번 생각해봅시다.  
  
상품의 재고 수량 업데이트 작업은 지연 없이 실시간으로 처리 되어야 하지만, 구매 로그를 저장소로 보내는 작업은 이미 로그가 서버에 기록되어 있기 때문에 실시간 처리 보다는 배치 처리를 이용하는 것이 더욱 효율적일 것 입니다.  
  
카프카에서는 이와 같은 장점을 지닌 배치 전송을 권장하고 있습니다.

### 압축 전송

카프카는 메세지 전송시 좀 더 성능이 높은 압축 전송을 사용하는 것을 권장 합니다.  
  
카프카에서 지원하는 압축 타입은 gzip , snappy , lz4 , zstd 등 입니다.  
  
압축만으로도 네트워크 대역폭이나 회선 비용을 줄일수 있으며 앞에 설명한 배치 전송과 결합해 사용한다면 더욱 높은 효과를 얻게 됩니다.  
  
파일 하나를 압축하는 것보다 비슷한 파일 10개, 혹은 20개를 압축하는 쪽의 효율이 더욱 좋기 때문 입니다.  
  
압축 타입에 따라서 조금씩 특색이 있으며, 일반적으로 높은 압축률이 필요한 경우면 gzip 이나 zstd 를 권장하고, 빠른 응답속도가 필요하다면 lz4 나 snappy 를 권장 하고 있습니다.

### 토픽, 파티션, 오프셋

카프카는 토픽(topic) 이라는 곳에 데이터를 저장하는데, 이는 우리가 흔히 사용하는 메일 전송 시스템에서 이메일 주소 정도의 개념으로 이해하면 쉽습니다.  
  
토픽은 병렬 처리를 위해서 여러개의 파티션 이라는 단위로 다시 나뉘게 됩니다.  
  
카프카에서는 이와 같은 파티셔닝을 통해 단 하나의 토픽이라도 높은 처리량을 수행할 수 있도록 되어 있습니다.  
  
파티션의 메세지가 저장되는 위치를 오프셋(offset) 이라고 부르며, 오프셋은 순차적으로 증가하는 숫자(64비트 정수) 형태로 되어 있습니다.

[](https://hoing.io/storage/2022/04/kafka-log-anatomy.png)

[[**https://sookocheff.com/post/kafka/kafka-in-a-nutshell**](https://sookocheff.com/post/kafka/kafka-in-a-nutshell/)]

하나의 토픽이 총 3개의 파티션으로 되어있고, 프로듀서로부터 전송된 메세지들은 쓰기 동작이 각 파티션 별로 이루어짐을 볼 수 있으며, 각 파티션마다 순차적으로 증가 하는 숫자들이 오프셋 입니다.  
  
각 파티션에서의 오프셋은 고유한 숫자이며 오프셋을 통해 메세지의 순서를 보장하고 컨슈머에서는 마지막 읽은 위치를 알수 있게 됩니다.

### 고가용성 보장

카프카는 앞서 설명한 것 처럼 분산 시스템이기 때문에 하나의 서버나 노드가 다운 되어도 다른 서버 또는 노드가 대신하여 안정적인 서비스가 가능하게 합니다.  
  
이러한 고가용성을 보장하기 위해서 카프카는 리플리케이션(복제) 기능을 제공 하고 있으며, 토픽 자체를 복제하는 것이 아닌 토픽의 파티션을 복제를 하는 방식으로 합니다.  
  
토픽을 생성할 때 옵션으로 리플리케이션 팩터 수를 지정할 수 있으며, 이 숫자에 따라서 리플리케이션들이 존재하게 됩니다.  
  
원본과 리플리케이션을 구분하기 위해서 카프카에서는 리더(leader) 와 팔로워(follower) 라고 부릅니다.

| 리플리케이션 팩터수 | 리더 수 | 팔로워 수 |

|----------------|-------|---------|

| 2 | 1 | 1 |

| 3 | 1 | 2 |

| 4 | 1 | 3 |

위의 표는 리플리케이션 팩터 수에 따른 리더와 팔로우 수의 관계를 보여주고 있습니다.  
  
표에서와 같이 리더의 숫자는 1을 유지 하게 되고 리플리케이션 팩터의 수에 따라서 팔로워 수만 증가하게 됩니다.  
  
팔로워 수가 많다고 딱히 좋은 것은 아닙니다 왜냐하면 팔로워의 수만큼 결국 브로커의 디스크 공간도 소비되기 때문 입니다. 그래서 적절한(이상적인) 리플리케이션 팩터 수를 유지해야하며 일반적으로는 3으로 구성하도록 권장되고 있습니다.  
  
리더(leader) 는 프로듀서, 컨슈머로 부터 오는 모든 읽기 와 쓰기 요청을 처리하며, 팔로워는 오직 리더로 부터 복제를 하게 됩니다.

### 주키퍼의 의존성

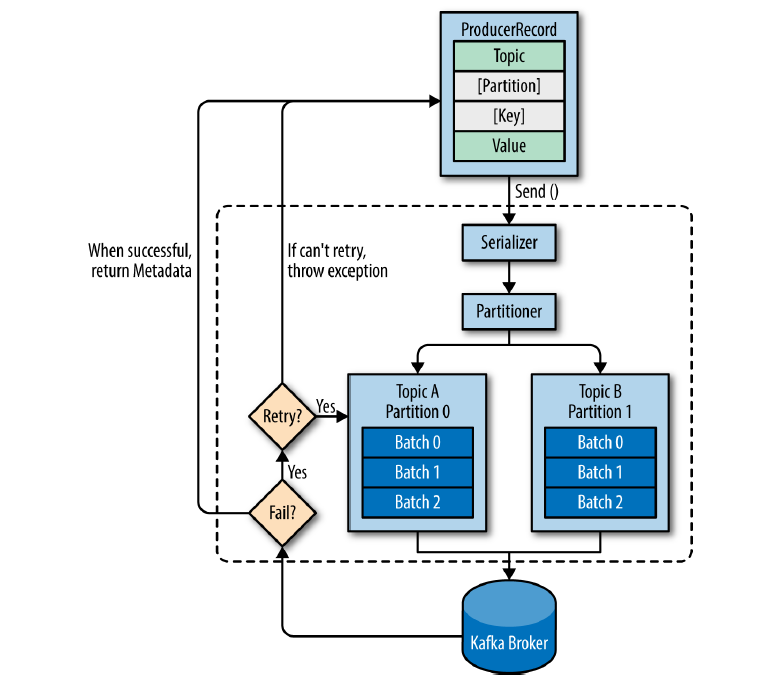
카프카에서는 빼놓을 수 없는 부분이 바로 주키퍼(ZooKeeper) 입니다. 주키퍼는 하둡의 서브 프로젝트 중 하나로 출발해 2011년에 아파치의 탑 레벨 프로젝트로 승격되었습니다.  
  
주키퍼는 오늘날에 이르러 카프카를 비롯해 아파치 산하의 프로젝트 인 하둡, 나이파이(NiFi), 에이치베이스(HBase) 등 많은 분산 애플리케이션에서 코디네이터 역할을 하는 애플리케이션으로 사용 되고 있습니다.  
  
주키퍼는 여러 대의 서버를 앙상블(Ensemble,클러스터) 로 구성하고, 살아 있는 노드 수가 과반수 이상 유지 된다면 지속적인 서비스가 가능한 구조 입니다.  
  
따라서 주키퍼는 반드시 홀수로 구성해야 하며, 책의 실습 과정에서의 구성시 홀수인 3대로 구성하였습니다.  
  
지노드(znode) 를 이용해 카프카의 메타 정보가 주키퍼에 기록되며, 주키퍼는 이러한 지노드를 이용해 브로커의 노드 관리, 토픽 관리 ,컨트롤러 관리 등 매우 중요한 역할을 하고 있습니다.  
  
최근 들어 카프카를 사용하면서 주키퍼의 사용과정에서의 한계성이 드러나기 시작하였고, 그에 따라 주키퍼의 의존성을 벗어나고자 카프카에서는 주키퍼에 대한 의존성을 제거 하려는 움직임이 진행되고 있었습니다.  
  
Apache Kafka 2.8 버전부터 주키퍼 대신 kraft 를 사용할 수 있으나 아직은 개발 단계로 문서에서도 표기 되어 있습니다.

***Note*** *KRaft is in early access and should be used in development only. It is not suitable for production.*[*https://developer.confluent.io/learn/kraft/*](https://developer.confluent.io/learn/kraft/)

**프로듀서의 기본 동작**

프로듀서는 카프카의 토픽으로 메세지를 전송하는 역할을 담당합니다 프로듀서는 여러 옵션을 제공하며, 원하는 형태에 따라 옵션을 변경 하면서 다양한 방법으로 카프카로 메세지를 전송할 수 있습니다.

### 프로듀서 디자인

[](https://hoing.io/storage/2022/04/producer.png)

[[**https://dzone.com/articles/take-a-deep-dive-into-kafka-producer-api**](https://dzone.com/articles/take-a-deep-dive-into-kafka-producer-api)]

**ProducerRecord** 라고 표시된 부분은 카프카로 전송하기 위한 실제 데이터이며, 레코드는 토픽, 파티션 , 키 , 밸류(value) 로 구성되어 있습니다.  
  
프로듀서가 카프카로 레코드를 전송할 때, 특정 토픽으로 메세지를 전송하며, 레코드에서 토픽과 밸류(메세지 내용)은 필수이고, 특정 파티션을 지정하기 위한 레코드의 파티션 과 특정 파티션에 레코드들을 정렬하기 위한 레코드의 키는 선택사항 (옵션) 입니다.  
  
다음으로 각 레코드들은 프로듀서의 send() 메소드를 통해서 시리얼라이저(serializer, 직렬화) 그리고 파티셔너를 거치게 됩니다.  
  
만약 프로듀서 레코드의 선택사항인 파티션을 지정 하였다면 파티셔너는 아무 동작도 하지 않고 지정된 파티션으로 레코드를 전달 합니다.  
  
파티션을 지정하지 않은 경우에는 키를 가지고 파티션을 선택해 레코드를 전달하는데 기본적으로 라운드 로빈(round robin) 방식으로 동작 합니다.  
  
send() 메소드 동작 이후 레코드들을 파티션 별로 잠시 모아두게 됩니다 레코드를 모아두는 이유는 프로듀서가 카프카로 전송하기 전에 **배치 전송을 하기 위해서** 입니다.  
  
전송이 실패되면 재시도 동작이 이루어지게 되고 지정한 횟수만큼 재시도가 실패하면 최종 실패로 전달하게 되고, 전송이 성공되면 성공에 대한 메타데이터를 리턴 하게 됩니다.

**컨슈머의 기본 동작**

프로듀서가 카프카의 토픽으로 메세지를 전송하면 해당 메세지들은 브로커들의 로컬 디스크에 저장 되게 됩니다.  
  
그리고 컨슈머를 이용해 토픽에 저장된 메세지를 가져올 수 있습니다.  
  
컨슈머 그룹은 하나이상의 컨슈머들이 모여 있는 그룹을 의미하고 컨슈머는 반드시 컨슈머 그룹에 속하게 됩니다.  
  
그리고 컨슈머 그룹은 각 파티션의 리더에게 카프카 토픽에 저장된 메세지를 가져오기 위한 요청을 보냅니다.  
  
이때 파티션 수와 컨슈머(하나의 컨슈머 그룹 안에 있는 컨슈머 수)는 일대일로 매핑 해야하는 것은 아니지만, 파티션 수보다 컨슈머 수가 많게 구현되는 것은 바람직한 구성은 아닙니다.  
  
컨슈머 수가 파티션 수보다 많다면 더 많은 수의 컨슈머들은 그냥 대기 상태로 존재하기 때문에 더 빠르게 메세지를 가져오거나 처리량을 늘어나지 않습니다.  
  
컨슈머 그룹내에서 리밸런싱 동작을 통해 장애가 발생한 컨슈머의 역할을 동일한 그룹에 있는 다른 컨슈머가 그 역할을 대신 수행 하므로 **굳이 장애 대비를 위한 추가 컨슈머 리소스를 할당하지 않아도 됩니다.**

### 컨슈머 그룹의 이해

컨슈머는 컨슈머 그룹 안에 속한 것이 일반적인 구조로, 하나의 컨슈머 그룹안에 여러개의 컨슈머가 구성될 수 있습니다. 그리고 아래의 그림 처럼 토픽의 파티션과 일대일로 매핑 되어 메세지를 가져오게 됩니다.

[텍스트, 시계이(가) 표시된 사진

자동 생성된 설명](https://hoing.io/storage/2022/04/consumer_consumer_grp_2.png)

peter-01 이라는 토픽이 있고 파티션 0 ,1, 2 라는 총 3개의 파티션으로 구성되어 있습니다.  
  
우측에는 컨슈머 그룹01 이 있으며 그룹내에는 파티션수와 동일한 3개의 컨슈머가 속해 있습니다.  
  
이와 같이 컨슈머들은 하나의 컨슈머 그룹 안에 속해 있으며, 그룹 내의 컨슈머들은 서로의 정보를 고유 하게 됩니다.  
  
공유하는 정보로는 예를 들어 컨슈머01이 문제가 발생하여 종료되었다면, 컨슈머02 또는 컨슈머03은 컨슈머01이 하던일을 대신해서 peter-01 토픽의 파티션 0을 컨슘하게 됩니다.

## 카프카 리플리케이션

고가용성 분산 스트리밍 플렛폼 카프카는 무수히 많은 데이터 파이프라인의 정중앙에 위치하는 메인 허브 역할을 합니다.  
  
이렇게 중앙에서 메인 허브 역할을 하는 카프카 클러스터가 만약 하드웨어의 문제나 점검 등으로 인해 정상적으로 동작하지 못한다거나, 카프카와 연결된 전체 데이터 파이프라인에 영향을 영향을 미친다면 이는 매우 심각한 문제가 아닐수 없습니다.   
  
따라서 카프카는 초기 설계 단계에서부터 이러한 일시적인 하드웨어 이슈 등으로 대비를 해야 합니다.

### 리플리케이션 동작 개요

카프카는 브로커의 장애에도 불구하고 연속적으로 안정적인 서비스 제공함으로써 데이터 유실을 방지하며 유연성을 제공합니다.  
  
카프카의 리플리케이션 동작을 위해 토픽 생성시 필수로 replication factor 라는 옵션을 설정해야 합니다.  
  
아래와 같이 토픽을 생성하면서 리플리케이션을 설정하도록 하겠습니다.

kafka-topics.sh --bootstrap-server kafka1:9092 \

--create --topic jade-test01 --partitions 1 \

--replication-factor 3

Created topic jade-test01.

토픽이름은 jade-test-01 으로 생성 하였으며, kafka-topic.sh 를 명령어를 이용해 토픽을 생성하였으며, 파티션 수 1개,리플리케이션 팩터 수는 3으로 설정 하였습니다.  
  
토픽 생성 된후 describe 옵션을 이용해서 토픽에 대한 상세한 정보를 출력해보도록 하겠습니다.

$ kafka-topics.sh --bootstrap-server kafka1:9092 \

--topic jade-test01 --describe

결과)

Topic: jade-test01 PartitionCount: 1 ReplicationFactor: 3 Configs: segment.bytes=1073741824

Topic: jade-test01 Partition: 0 Leader: 1 Replicas: 1,2,3 Isr: 1,2,3

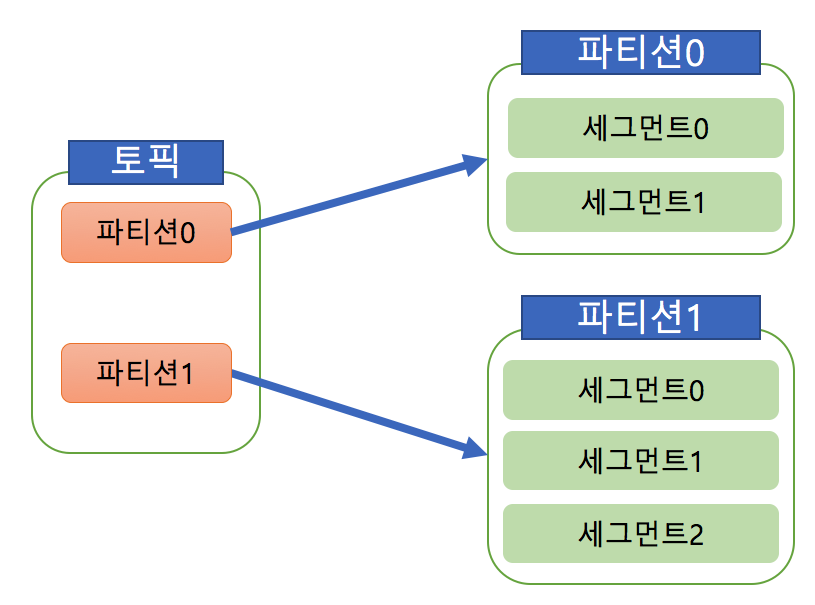
토픽 파티션의 상세 내용으로 파티션 개수는 1개이며 리더는 브로커 1(Leader : 1) 으로 되어 있는 것을 확인 할 수 있으며, 리플리케이션들은 브로커가 3개(1,2,3)가 있다는 것을 알 수 있습니다.  
  
이전에도 언급 드린 내용처럼 리플리케이션 되는 것은 토픽이 아니라 토픽을 구성하는 각각의 파티션들을 의미하고, 위의 예제에서는 파티션이 1개 인 토픽 입니다.  
  
  
이제 콘솔에서 프로듀서를 이용해서 테스트 메세지를 jade-test01 토픽으로 전송을 해보도록 하겠으며, kafka-console-producer.sh 명령어를 사용해서 진행 할 수 있습니다.

$ kafka-console-producer.sh --bootstrap-server \

kafka1:9092 --topic jade-test01

>test message1

메세지를 보내고 난 후 세그먼트 파일에 저장되어 있는지 확인 해봅니다.  
  
참고로 프로듀서에 의해 브로커로 전송된 메세지는 토픽의 파티션에 저장 되며, 각 메세지들은 세그먼트(segment) 라는 로그 파일의 형태로 브로커의 로컬 디스크에 저장되게 됩니다.

[](https://hoing.io/storage/2022/04/partition_segment.png)

kafka-dump-log.sh 를 통해서 세그먼트를 확인 합니다.

$ kafka-dump-log.sh --print-data-log \

--files /usr/local/kafka/logs/jade-test01-0/00000000000000000000.log

결과)

Dumping /usr/local/kafka/logs/jade-test01-0/00000000000000000000.log

Starting offset: 0

baseOffset: 0 lastOffset: 0 count: 1 baseSequence: -1

lastSequence: -1 producerId: -1 producerEpoch: -1

partitionLeaderEpoch: 0 isTransactional: false isControl: false

position: 0 CreateTime: 1652886146674 size: 81

magic: 2 compresscodec: NONE crc: 3267944857 isvalid: true

| offset: 0 CreateTime: 1652886146674 keysize: -1

valuesize: 13 sequence: -1 headerKeys: [] payload: test message1

\* 가로길이에 따른 개행이 되어 있습니다.  
  
위의 Dumping 결과에서 아래와 같은 정보를 확인 할 수 있습니다.  
• Starting offset : 시작 오프셋 위치는 0 임을 알 수 있습니다.  
• count : 1 - 메세지 카운트가 1 인 것을 알 수 있습니다.  
• payload: test message 1 - 프로듀서를 통해 보낸 메세지는 test message1 라는 것을 알 수 있습니다.  
  
프로듀서를 실행한 서버는 첫번째 브로커인 kafka1 에서 실행하였으나, 그외 브로커인 kafka2 나 kafka3 에서 위의 명령어를 실행하면 동일한 결과(즉 동일한 메세지가 있는)가 확인되는 것을 알 수 있습니다.  
  
직접 디렉토리에서도 봤을때 토픽에 해당하는 관련 디렉토리도 있으며 파일이 있으며 strings 로 봐도 동일한 메세지가 존재함을 알 수 있습니다.

[root]# hostname

kafka3

[root]# ls -arlt

total 24

-rw-r--r--. 1 root root 10485756 May 18 23:09 00000000000000000000.timeindex

-rw-r--r--. 1 root root 10485760 May 18 23:09 00000000000000000000.index

-rw-r--r--. 1 root root 8 May 19 00:02 leader-epoch-checkpoint

drwxr-xr-x. 2 root root 4096 May 19 00:02 .

-rw-r--r--. 1 root root 81 May 19 00:02 00000000000000000000.log

drwxr-xr-x. 20 root root 12288 May 19 02:35 ..

[root]# strings 00000000000000000000.log

test message1

위와 같이 확인한 것처럼 모든 브로커가 동일한 메세지를 가지고 있음을 확인 할 수 있습니다.  
  
토픽 생성시 지정한 리플리케이션 팩터 수에 따라서 복제를 유지하기 때문에 브로커 장애가 발생하더라도 메세지 손실 없이 안정적으로 메세지를 주고 받을 수 있습니다.  
  
위의 예제에서는 3개의 리플리케이션이 있으므로 이중에서 2대 브로커의 장애까지는 문제가 없이 메세지를 처리할 수 있습니다.

### 리더 와 팔로워

토픽 상세보기 명령어를 실행해보면 출력 내용 중 **파티션의 리더(leader)** 라는 부분이 있습니다.  
  
모두 동일한 리플리케이션이라고 하더라도 리더만의 역할이 따로 있으며, 카프카에서 리더를 특별히 강조해서 표시가 됩니다. 카프카는 내부적으로 모두 동일한 리플리케이션들을 리더와 팔로워로 구분을 하며, 각자의 역할이 분리가 되어있습니다.

카프카의 모든 읽기와 쓰기는 리더를 통해서만 가능 합니다. 프로듀서는 모든 리플리케이션에 메세지를 보내는 것이 아닌 리더에게만 메세지를 전송하게 됩니다.  
  
또한 메세지를 가져오는 컨슈머도 리더로부터 메세지를 가져오게 됩니다.

[텍스트, 시계이(가) 표시된 사진

자동 생성된 설명](https://hoing.io/storage/2022/05/leader_follower.png)

위의 그림은 프로듀서와 컨슈머 그리고 리더 와 팔로워의 관계를 그림으로 표현 한 것 입니다.  
  
토픽의 파티션은 1개 이며, 리플리케이션 팩터 수는 3 입니다. 리플리케이션은 토픽 단위가 아니라 파티션 단위로 수행 됩니다.  
  
프로듀서는 토픽으로 메세지를 전송하면 파티션의 리더에 쓰기가 이루어지게 되고 컨슈머도 동일하게 리더를 통해서 메세지를 가져오게 됩니다.  
  
팔로워는 리더에 문제가 발생할 경우를 대비해서 언제든지 새로운 리더가 될 준비를 해야 하고 그에 따라서 컨슈머가 토픽의 메세지의 읽어 가는 것과 비슷한 동작으로 지속적으로 리더가 새로운 메세지를 받았는지를 받았는지 확인하고, 새로운 메세지가 있다면 리더로 부터 복제를 하게 됩니다.

### 복제 유지 와 커밋

리더와 팔로워는 **ISR(In Sync Replica)** 이라는 논리적으로 그룹으로 묶여 있습니다.  
  
이렇게 리더와 팔로워를 별도의 그룹으로 나누는 이유는 기본적으로 해당 그룹안에 속한 팔로워들만이 새로운 리더의 자격 대상이 될수 있기 때문 입니다.  
그래서 다른 부분으로 설명하면 ISR 그룹에 속하지 못한 팔로워는 새로운 리더의 자격을 가질수 없습니다.  
  
ISR 내의 팔로워들은 리더와 데이터 일치를 유지하기 위해서 지속적으로 리더의 데이터를 가져와서 복제를 하게 되고, 리더는 ISR 내 모든 팔로워가 메세지를 받을 때까지 기다립니다.  
  
하지만 팔로워가 네트워크 오류, 브로커 장애 등으로 리더로 부터 복제를 할 수 없을 경우가 있을 것 입니다.  
이렇게 뒤쳐진 팔로워는 이미 리더와의 데이터가 불일치한 상태가 될 것 입니다.  
  
만약 이런 상황에서 이 팔로워에게 새로운 리더를 넘겨준다면 데이터의 정합성 문제가 발생하거나 메세지 손실 등의 문제가 발생할 수 있습니다.  
  
따라서 파티션의 리더는 팔로워들이 리플리케이션 동작을 잘하고 있는지 체크(감시)하게 됩니다.  
  
리더의 메세지가 상태를 뒤쳐지지 않고 복제를 잘 따라오고 있는 팔로워들만이 ISR 그룹에 속하게 됩니다.  
그림으로 보면 아래와 같이 표현할 수 있을 것 같습니다.

[테이블이(가) 표시된 사진

자동 생성된 설명](https://hoing.io/storage/2022/05/in-sync-replication-1.png)

쉽게 이해하는 방법으로 Replication Group 개념으로 볼 수도 있습니다.  
  
위와 같은 ISR 내에서 리더가 장애가 발생할 경우 새로운 리더가 될 수 있는 자격을 얻을 수 있게 되는 것 입니다.  
  
  
리더는 읽고 쓰고의 기본 동작이외에 팔로워가 리플리케이션 동작을 잘하고 있는지도 체크 하고 판단합니다. 만약 팔로워가 특정 주기의 시간만큼 복제 요청을 하지 않는다면 리더는 해당 팔로워의 복제 동작에 문제가 있다고 판단하여 ISR 그룹에서 추방하게 됩니다. 그리고 추방되게 되면 리더가 될 자격이 없게 되는 것 입니다.  
  
토픽 상세보기(describe) 명령어를 통해서 ISR 목록이 표시되는 것을 확인할 수 있으며 현재 ISR 상태를 확인 함에 따라서 토픽의 상태가 양호한지 문제가 있는지를 확인 할 수 있습니다.

$ kafka-topics.sh --bootstrap-server kafka1:9092 \

--topic jade-test01 --describe

결과)

Topic: jade-test01 PartitionCount: 1

ReplicationFactor: 3 Configs: segment.bytes=1073741824

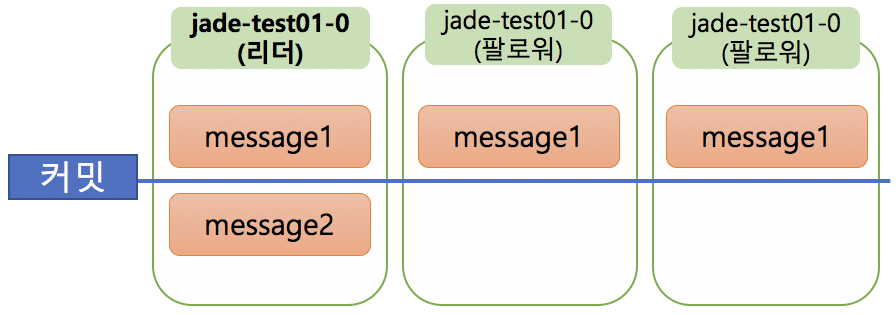
Topic: jade-test01 Partition: 0 Leader: 1

Replicas: 1,2,3

Isr: 1,2,3

<<<----ISR

\* 가로 길이에 따라서 개행 되어 있습니다  
  
위의 describe 조회 결과처럼 ISR 의 정보를 확인 할 수 있습니다.  
  
ISR 내 모든 팔로워가 복제가 완료 되었다면, 리더는 내부적으로 커밋 되었다는 표시를 하게 됩니다.  
마지막 커밋 오프셋 위치를 하이워터마크(high-water mark) 라고 부릅니다.  
  
커밋이 되었다는 것은 리플리케이션 팩터수의 팔로워가 메세지를 전부 저장하였음을 의미하며 이렇게 커밋된 메세지만이 컨슈머가 읽어갈수 있습니다.  
  
카프카에서는 커밋되지 않은 메세지는 컨슈머가 읽을 수 없게 한 이유는 **메세지의 일관성** 때문입니다.

[](https://hoing.io/storage/2022/05/commit_message_3.png)

**jade-test01 이라는 토픽**을 그림으로 표현한 그림으로 토픽내 파티션 1개, 3개의 리플리케이션 팩터로 생성(설정) 되었습니다.  
  
프로듀서가 첫 번째 test message1 이라는 메세지를 jade-test01 토픽으로 보내게 되고, 모든 팔로워가 리플리케이션 동작을 모두 완료 하여 모두 저장하고 있으며, 커밋이 완료된 상태 입니다.  
  
그 다음으로 message2 라는 메세지를 전송 하게 됩니다. 이 메세지는 리더에만 저장되어 있는 상태이고 팔로워들은 아직 해당 메세지에 대한 리플리케이션 동작을 하기 전 상태 입니다.  
  
만약 이러한 상태에서 여기서 커밋되기 전 메세지를 컨슈머가 읽을 수 있다고 한다면 아래와 같은 문제가 발생될 수 있게 됩니다.  
  
1) 컨슈머 A 는 jade-test01 토픽을 컨슘ㅇ르 합니다.  
2) 컨슈머 A는 jade-test01 이 토픽의 파티션 리더로부터 메세지를 읽어갑니다. 읽어간 메세지는 test message1,2 입니다.  
3) jade-test01 토픽의 파티션 리더가 있는 브로커에 문제가 발생할 팔로워 중 하나가 새로운 리더가 됩니다.  
  
4) 프로듀서가 보낸 test message2 메세지는 아직 팔로워들에게 리플리케이션이 되지 않은 상태에서 새로운 리더로 변경 됐으므로, 새로운 리더는 test message1 메세지만 갖고 있습니다.  
5) 새로운 컨슈머 B가 jade-test01 토픽 컨슘합니다.  
6) 새로운 리더로 부터 메세지를 읽어가고, 읽어간 메세지는 오직 test message1 입니다.  
  
이런식의 시나리오 일 경우 컨슈머 A 와 B는 jade-test01 이라는 동일한 토픽의 파티션을 읽었지만, 컨슈머 A는 test message1,2 를 가져왔고, 컨슈머 B는 test message1 만 가져오게 됩니다.  
  
이 처럼 커밋되지 않는 메세지를 읽게 된다면 같은 토픽을 읽었음에도 메세지가 일치하지 현상이 발생할 수 있게 됩니다.  
따라서 카프카에서는 이러한 메세지 불일치 현상을 방지하고자 커밋된 메세지만 컨슈머가 읽어 갈수 있도록 구현되어 있습니다.  
  
  
이러한 설계에 따라서 카프카에서는 커밋된 위치도 매우 중요한 요소가 되게 됩니다.  
  
카프카의 브로커는 재시작 될 때 커밋된 메세지를 유지하기 위해서 로컬 디스크의 replication-offset-checkpoint 라는 파일에 마지막 커밋 오프셋 위치를 기록/저장하게 됩니다.  
  
**replication-offset-checkpoint** 파일은 서버의 카프카에서 설정한 로그 디렉토리내에 위치해 있습니다.

[root]# cd /usr/local/kafka/logs/

[root]# ls -alrt | grep replication-offset-checkpoint

-rw-r--r--. 1 root root 446 May 19 17:52 replication-offset-checkpoint

[root]# cat replication-offset-checkpoint

<중략>

jade-test01 0 1

<중략>

체크포인트 파일에 기록 된 내용)  
jade-test01 은 토픽명을 의미하고, 0은 파티션 번호, 1은 커밋된 오프셋 번호를 의미하게 됩니다.  
  
  
프로듀서에서 다시 test message2 라는 메세지를 하나더 입력 후 다시 체크포인트 파일을 살펴보면 오프셋 번호가 증가 한 것을 확인할 수 있습니다.

[root@acs2 logs]# cat replication-offset-checkpoint

jade-test01 0 2

만약 특정 토픽 또는 파티션에 복제가 되지 않거나 문제가 있다고 판단되는 경우, replication-offset-checkpoint 라는 파일의 내용을 확인하고 리플리케이션 되고 있는 다른 브로커들과 비교해 살펴보면, 어떤 브로커, 토픽, 파티션에 문제가 있는지를 파악할 수 있습니다.

### 리더와 팔로워의 단계별 리플리케이션 동작

메세지의 읽고 쓰기를 처리하는 리더는 매우 바쁘게 동작을 합니다.  
  
이렇게 바쁜 리더가 리플리케이션 동작을 위해 팔로워들과 많은 통신을 주고 받거나 리플리케이션 동작에 많은 관여를 한다면  그 결과는 리더의 성능은 떨어지고 카프카의 장점의 빠른 성능을 내기도 어려울 것 입니다.  
  
그래서 카프카는 리더와 팔로워간의 리플리케이션 동작을 처리할때 서로 통신을 최소화 할 수 있도록 설계함으로 리더의 불하를 줄였습니다.  
  
  
**리더와 팔로워 간의 리플리케이션 동작**  
  
1) 프로듀서가 토픽으로 메세지를 전송  
2) 리더만 메세지를 수신 받아 저장(**시나리오에서는 지금 메세지는 0번 오프셋에 위치함을 가정**)  
3) 팔로워들은 리더에게 메세지 가져오기(fetch) 요청을 보낸 후 새로운 메세지 가 있다는 사실을 인지하고 메세지를 리플리케이션 함  
4) 리더는 모든 팔로워가 리더에게 메세지를 리플리케이션하기 위한 요청을 보냈다는 사실을 알고 있음  
5) 하지만 리더는 팔로워들이 리플리케이션 동작으르 성공했는지 실패했는지 여부는 알지 못함  
(카프카에서 리더와 팔로워간의 ACK 를 주고 받는 통신이 없으며, ACK 를 제거함으로 성능이 더욱 좋아짐)  
  
(아래 부터 ACK 통신없이 안정적인 리플리케이션 처리에 대한 내용이 있음)  
6) 리더는 프로듀서로 부터 다음 메세지를 수신 하게 되면 저장하게 됩니다(**여기서 수신받은 메세지는 1번 오프셋이라고 가정**)  
7) 팔로워들은 리더에게 새로운 메세지인 1번 오프셋에 대한 리플리케이션을 요청 합니다.  
8) 팔로워들로 부터 1번 오프셋에 대한 리플리케이션 요청을 받은 리더는 팔로워들의 0번 오프셋에 대한 리플리케이션 동작이 성공했음을 인지하고 오프셋 0번 대해서 커밋 표시를 한 후에 하이워터마크를 증가 시키게 됩니다.  
9) 팔로워들로 부터 1번 오프셋 메세지에 대한 리플리케이션 요청을 받은 리더는 응답시 0번 오프셋의 메세지가 커밋되었다는 내용도 함께 전달하게 됩니다.  
10) 리더의 응답을 받은 모든 팔로워는 0번 오프셋 메세지가 커밋이 된 내용을 인지를 하게 되고, 리더와 동일하게 커밋이 된것을 표시 하게 되며, 그 다음 1번 오프셋 메세지를 리플리케이션 하게 합니다.  
  
  
여기까지가 리더와 팔로워간의 리플리케이션 동작에 대한 내용입니다.  
  
위의 시나리오상에서 팔로워가 0번 오프셋에 대한 복제가 성공하지 못했다면 팔로워는 1번 오프셋에 대한 리플리케이션 요청이 아닌 0번 오프셋에 대한 리플리케이션 요청을 보내게 됩니다.  
  
따라서 리더는 팔로워들이 보내는 리플리케이션 요청의 오프셋을 보고 팔로워들이 어느 위치의 오프셋까지 리플리케이션을 성공했는지를 인지 할 수 있게 됩니다.  
  
다른 유사한 메세징 시스템에서는 ACK 통신을 통해서 메세지를 잘 받고 하였는지를 체크하지만, 카프카에서는 ACK 통신을 제거하였다는 부분이 중요한 차이점 입니다.  
(대량의 메세지를 처리할 경우 ACK 통신을 주고 받는 것도 부하 이면서 성능저하의 요소가 됨)  
  
그래서 카프카의 특징이 ACK 통신을 제외하였음에도 불구하고 팔로워와 리더간의 리플리케이션 동작이 매우 빠르면서도 신뢰할 수 있다는 점 입니다.  
  
카프카에서 리더와 팔로워들의 리플리케이션 동작 방식은 리더가 푸시(push) 하는 방식이 아니라, 팔로워가 풀(pull) 하는 방식으로 동작하며, 리플리케이션에서 리더의부하를 줄여주기 위해서 팔로워 풀(pull) 형태로 구현되었습니다.

### 리더에포크와 복구

**리더에포크(leader epoch)** 는 카프카의 파티션들이 복구 동작을 할 때 메세지의 일관성을 유지하기 위한 용도로 이용되며,  컨트롤러에 의해 관리되는 32비트의 숫자로 표현됩니다.  
  
리더에포크(leader epoch) 정보는 리플리케이션 프로토콜에 의해 전파 되며 리더가 변경 되면 변경 된 새로운 리더에 대한 정보를 팔로워에게 전달 합니다.  
  
리더에포크는 복구 동작 시 하이워터마크를 대체하는 수단으로 활용 됩니다.  
           

#### 리더에포크를 사용하지 않을 경우

먼저 리더에포크를 사용하지 않는 상황에서는 장애와 복구 과정을 확인해보도록 하겠습니다.

[텍스트, 시계이(가) 표시된 사진

자동 생성된 설명](https://hoing.io/storage/2022/05/leader-epoch-1.png)

**[복구관련 이미지 1]**

위의 이미지는 리더에포크를 사용하지 않는 장애 복구 과정을 의미 합니다.  
  
위의 jade-test01 토픽은 파티션수 1 , 리플리케이션 팩터 2 , min.insync.replicas 1 로 설정 이며 예제에서는 리더에포크가 없다는 가정으로 장애 복구 과정은 아래와 같이 진행됩니다.  
  
1) 리더는 프로듀서로 부터 message1 메세지를 받고 0번 오프셋에 저장, 팔로워는 리더에게 0 번 오프셋에 대한 가져오기 요청  
2) 가져오기 요청을 통해 팔로워는 message1 메세지를 리더로부터 리플리케이션함  
3) 리더는 하이워터마크를 1로 올림  
4) 리더는 프로듀서로 부터 다음 메세지인 message2를 받은 뒤 1번 오프셋에 저장  
5) 팔로워는 다음 메세지인 message2에 대해 리더에게 가져오기 요청을 보내고 응답으로 리더의 하이워터마크 변화를 감지하고 자신의 하이워터마크도 1로 올림  
6) 팔로워는 1번 오프셋의 message2 메세지를 리더로 부터 리플리케이션함  
7) 팔로워는 2번 오프셋에 대한 요청을 리더에게 보내고, 요청을 받은 리더는 하이워터마크를 2로 올림  
8) 팔로워는 2번 오프셋인 message2 메세지까지 리플리케이션을 완료하였지만 아직 리더로부터 하이워터마크를 2로 올리는 내용을 전달받지 못한 상태  
9) 이 상태에서 예상치 못한 장래로 팔로워가 다운됨  
(여기까지가 위의 "**복구관련 이미지 1**" 에 해당되는 내용)

[텍스트, 시계이(가) 표시된 사진

자동 생성된 설명](https://hoing.io/storage/2022/05/leader-epoch-2.png)

**[복구관련 이미지 2]**

위의 "복구관련 이미지2" 는 장애가 발생한 팔로워가 종료 된 후 장애 처리가 완료된 상태를 나타냅니다.  
  
장애에서 복구된 팔로워는 카프카 프로세스가 시작되면서 내부적으로 메세지 복구 동작을 시작하게 됩니다.  
  
1) 팔로워는 자신이 갖고 있는 메세지들 중에서 자신의 워터마크보다 높은 메세지들은 신뢰할 수 없는 메세지로 판단하고 삭제  
(따라서 1번 오프셋의 message2는 삭제됨)  
2) 팔로워는 리더에게 1번 오프셋의 새로운 메세지에 대한 가져오기 요청을 보냅니다.  
3) 이 순간 리더였던 브로커가 예상치 못한 장애로 다운되면서, 해당 파티션에 유일하게 남아 있던 팔로워가 새로운 리더로 승격 합니다.

[텍스트, 시계이(가) 표시된 사진

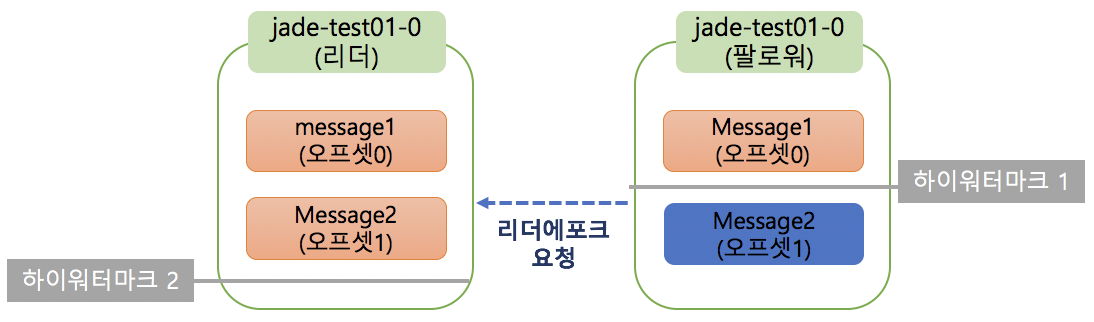
자동 생성된 설명](https://hoing.io/storage/2022/05/leader-epoch-3_2.png)

**[복구관련 이미지 3]**

위의 그림 "복구관련 이미지 3" 는 팔로워가 새로운 리더로 승격된 후의 상태를 나타내게 됩니다.  
  
그림에서 알 수 있듯이 기존의 리더는 1번 오프셋의 message2 라는 메세지를 가지고 있었지만, 팔로워는 message2 라는 메세지가 없이 새로운 리더로 승격이 되었습니다.  
  
결국은 새로운 리더는 message2 라는 메세지를 갖고 있지 않으며, 리더와 팔로워간의 리플리케이션이 작동하고 있었지만 리더가 변경되는 과정에서 message2 라는 메세지가 손실된 것 입니다.

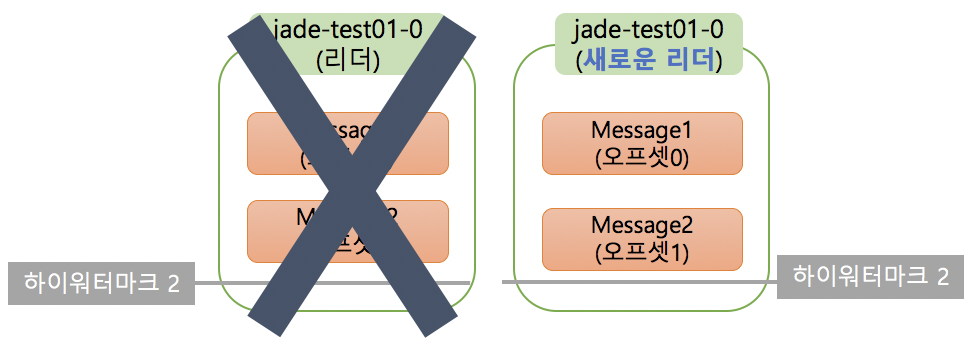
#### 리더에포크를 사용할 경우

이번에는 리더에포크를 사용 된다는 상황으로 장애 와 복구에 대한 내용을 확인 해보도록 하겠습니다.

[](https://hoing.io/storage/2022/05/leader-epoch-4.png)

**[복구관련 이미지 4]**

위의 그림 "복구관련 이미지4" 는 리더와 팔로워의 리플리케이션 동작 이후 그리고 팔로워가 장애로 종료 된 후 막 복구된 상태 이후의 과정을 나타내고 있습니다.  
  
앞에서의 동작은 카프카 프로세스가 시작되면서 복구 시 자신의 하이워터마크 보다 높은 메세지를 즉시 삭제 하였습니다.  
  
하지만 리더에포크를 사용하는 경우에는 하이워터마크 보다 앞에 있는 메세지를 무조건 삭제 하는 것 아닌 리더에게 리더에포크 요청을 보냅니다.  
  
1) 팔로워는 복구 동작을 하면서 리더에게 리더에포크 요청를 보냄  
2) 요청받은 리더는 리더에포크의 응답으로 "1번 오프셋의 message2까지" 라고 팔로워에게 보냄  
3) 팔로워는 자신의 하이워터마크보다 높은 1번 오프셋의 message2를 삭제하지 않고 리더의 응답을 화인 후에 오프셋2 인 message2 까지 자신의 하이워터마크를 상향 조정 합니다.

[](https://hoing.io/storage/2022/05/leader-epoch-5.png)

**[복구관련 이미지 5]**

위의 그림 "복구관련 이미지 5" 는 리더가 예상치 못한 장애로 다운 되면서 팔로워가 새로운 리더로 승격된 후의 상태를 나타냅니다.  
  
리더에포크를 적용하지 않는 경우에는 팔로워가 복구 과정에서 자신의 하이워터마크보다 높은 message2 를 삭제하였지만 리더에포크를 활용하는 경우에는 삭제 동작을 하기 전에 리더에포크 요청과 응답 과정을 통해서 팔로워의 하이워터마크를 올릴 수 있게 되며, 메세지 손실이 발생하지 않게 됩니다.

#### 리더에포크 변화 과정 확인

위에서 리더에포크를 사용하지 않았을 경우 와 사용할 경우에 대해서 각각의 차이점 등을 확인 해보았습니다.  
  
이번에는 실습을 통해서 리더에포크의 변화 과정을 살펴보도록 하겠습니다.  
테스트를 위한 토픽은 포스팅에서 생성하여 사용중인 jade-test01 을 이용하도록 하겠습니다.  
  
먼저 다시 describe 를 통해서 토픽의 현재 상태를 확인 해보도록 하겠습니다.

$ kafka-topics.sh --bootstrap-server kafka2:9092 --topic jade-test01 --describe

Topic: jade-test01 PartitionCount: 1

ReplicationFactor: 3 Configs: segment.bytes=1073741824

Topic: jade-test01 Partition: 0

Leader: 1 Replicas: 1,2,3 Isr: 1,2,3

\* 가로 길이에 따라서 개행 되어 있습니다.  
  
현재 파티션의 리더의 1 번 브로커임을 알 수 있습니다.  
  
그 다음에는 cat 명령을 이용해서 리더에포크 상태를 확인 하도록 하겠습니다. 먼저 1번 브로커 서버로 새로 접속 후에 log 디렉토리로 이동합니다.  
그리고 리더에포크 파일을 cat 으로 조회합니다.

$ cd /usr/local/kafka/logs/jade-test01-0

$ cat leader-epoch-checkpoint

0

1

0 0

위의 조회 결과의 의미는 다음과 같습니다.  
  
두번째 라인 1 : 현재의 리더에포크 번호  
세번째 라인 0 0 : 첫 번재 0은 리더에포크 번호, 두 번째 0은 최종 커밋 후 새로운 메세지를 전송받게 될 오프셋 번호  
  
  
리더에포크는 새로운 리더 선출이 발생하면 변경된 정보가 업데이트 됩니다.  
  
강제로 새로운 리더가 선출되도록 리더가 위치한 브로커 1을 종료 합니다.

[root]# systemctl stop kafka

\*  카프카 서비스명은 등록한 이름에 따라 다를 수 있습니다.  
  
다시 describe 로 토픽을 조회해보도록 하겠습니다.

$ kafka-topics.sh --bootstrap-server kafka2:9092 --topic jade-test01 --describe

Topic: jade-test01 PartitionCount: 1

ReplicationFactor: 3 Configs: segment.bytes=1073741824

Topic: jade-test01 Partition: 0

Leader: 2 Replicas: 1,2,3 Isr: 2,3

\* 가로 길이에 따라서 개행 되어 있습니다.  
  
리더가 변경 된 것을 확인할 수 있으며, 지금 리더는 2번 브로커 입니다.  
2번 브로커에 접속하여 다시 리더에포크 정보를 조회해봅니다.

$ cat leader-epoch-checkpoint

0

2

0 0

1 2

처음 확인했던 리더에포크에서 변화가 되었습니다.  
출력 결과의 의미는 다음과 같습니다.  
  
두번째 라인 2 : 현재 리더에포크 번호. 새로운 리더가 선출이 있었고 리더에포크는 1에서 2로 변경 되었습니다. 리더에포크 번호는 리더가 변경될 때마다 하나씩 숫자가 증가 합니다.  
세번째 라인 0 0 : 첫 번째 0은 리더에포크 번호, 두번째 0은 최종 커밋 후 새로운 메세지를 전송 받게 될 오프셋 번호  
세번째 라인 1 2 : 첫 번째 1은 리더에포크 번호, 두번째 2는 최종 커밋 후 새로운 메세지를 전송 받게 될 오프셋 번호  
  
위에서 jade-test01 토픽에 2개의 message 를 입력 하였습니다. 그래서 0,1 오프셋이었고 최종 커밋 후 새로운 메세지를 전송 받게 될 오프셋 번호인 2가 확인 되고 있습니다.  
  
팔로워는 자신의 하이워터마크보다 높은 오프셋의 메세지를 무조건 삭제 하지 않고 먼저 리더에게 리더에포크 요청을 보내 응답을 받아서 최종 커밋된 오프셋 위치를 확인 합니다.  
이와 같이 리더에포크 정보를 이용하여 장애 후 복구시 데이터의 손실 또는 데이터 불일치를 막게 됩니다.

컨트롤러

**컨트롤러**는 카프카에서 리더 선출 하는 역할을 맡고 있습니다.  
  
카프카 클러스터 중 하나의 브로커가 컨트롤러 역할을 하게 되며, 파티션의 ISR 리스트 중에서 리더를 선출 합니다.  
  
ISR 에 대한 정보는 이전 포스팅에서 확인 하시면 됩니다.



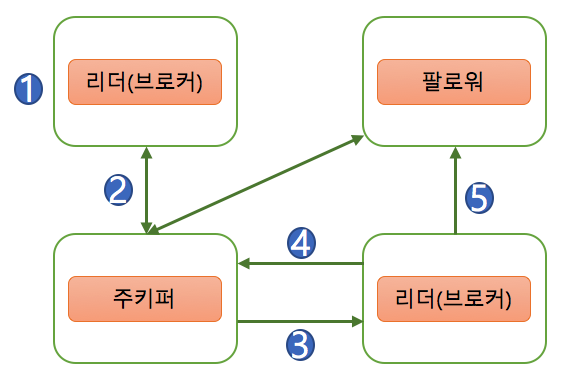
**카프카(Kafka) (4) - 내부 동작 원리와 구현 1**

안녕하세요. 이번 포스팅은 카프카의 내부 동작 원리와 구현에 대한 글로 실전 카프카 개발부터 운영까지 책을 정리한 글 입니다.

리더를 선출하기 위해 ISR 리스트 정보는 안전한 저장소에 보관되어 있어야 하는데, 가용성 보장을 위해서 주키퍼에 저장되어 있습니다.  
  
컨트롤러는 브로커가 실패하는 것을 예의 주시하고 있으며, 만약 브로커의 실패가 감지가 되면 즉시 ISR 리스트 중 하나를 새로운 파티션 리더로 선출 하며, 그리고 새로운 리더 정보를 주키퍼에 기록하고, 변경된 정보를 모든 브로커에게 전달 합니다.

### 예기치 않은 장애로 인한 리더 선출

리더 브로커가 비정상 종료 시 새로운 파티션의 리더가 선출되는 과정에 대해서 확인 해 보도록 하겠습니다.

[](https://hoing.io/storage/2022/05/kafka_controller_1.png)

**[그림 1]**

위의 **그림 1** 은 예기치 않은 장애로 인한 리더 선출 과정을 이미지화 한 것 으로 각 단계별 내용은 아래와 같습니다.  
  
1) 파티션 0 의 리더인 1번 브로커가 예기치 않게 다운이 됩니다.  
2) 주키퍼는 1번 브로커와의 연결이 끊어진 후, 0번 파티션의 ISR에서 변화가 생겼음을 감지합니다.  
3) 컨트롤러는 주키퍼 워치를 통해 0번 파티션에 변화가 생긴것을 감지하고 해당 파티션의 ISR 중에서 새로운 리더를 선출 합니다.  
4) 컨트롤러는 0번 파티션의 새로운 리더가 어떤 번호의 브로커가 되었다는 정보를 주키퍼에 기록 합니다.  
5) 이렇게 갱신된 정보는 현재 활성화 상태인 모든 브로커에게 전파 됩니다.  
  
  
리더 선출 과정은 컨트롤러에 의해 이루어집니다. 파티션이 하나인 경우 컨트롤러가 새로운 리더를 선출하고 리더 정보를 주키퍼에 기록며 다른 브로커에게 업데이트 정보를 전파하는 데는 그리고 오랜 시간이 걸리지 않습니다.  
  
예를 들어 하나의 파티션에서 리더 선출 과정에서 0.2초만에 완료가 될 경우 토픽에 1개 파티션만 있다면 0.2초만에 리더선출 과정이 완료 될 것 입니다.  
하지만 1만개의 파티션에 대해서 리더 선출이 이루어져야 한다면 전체 작업 소요시간은 약 2,000초가 걸리게 되며 분으로 환산하면 30춘이 조금 넘는 시간 입니다.  
  
1대의 브로커의 장애에 의해 리더 선출 과정에서 30여 분간 통신이 끊어지는 상황이 발생된다면, 카프카 사용에 매우 어려운 상황을 직면 하게 될 것 입니다.  
  
이런 상황을 개선하고자 2018년 11월 릴리즈된 카프카 버전 1.1.0 버전 부터는 리더 선출 작업 속도가 빨라지게 개선 되었습니다.



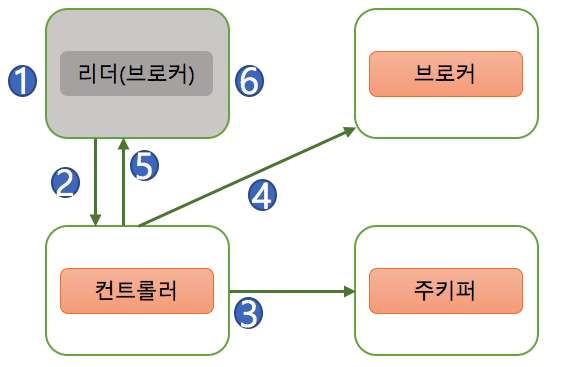
**Apache Kafka Supports 200K Partitions Per Cluster | Confluent | KR**

Since the Apache Kafka® 1.1.0 release, there has been a significant increase in the number of partitions that a single Kafka cluster can support from the deployment and the availability perspective.

카프카 1.0.0 버전에서 6분 30초가 소요 되었던 리더 선출 과정에서 불필요한 로깅을 없애고 주키퍼 비동기 API가 반영된 카프카 1.1.0 버전에서는 약 3초만에 완료 되었습니다.

### 제어된 종료에 의한 리더 선출

이전 단계 내용은 예기치 않은 장애로 인한 리더 선출 과정이었으며, 이번에는 제어된 종료 과정에서 리더 선출 작업에 대해서 확인 해보도록 하겠습니다.  
  
제어된 브로커 종료는 관리자에 의해 이루어진 자연스러운(graceful) 종료 또는 안전한 종료를 의미 합니다.

[](https://hoing.io/storage/2022/05/kafka_controller_2.png)

**[그림 2]**

위의 **그림 2** 은 제어된 종료 과정을 이미지화 한 것 으로 각 단계별 내용은 아래와 같습니다.  
  
1) 관리자가 브로커 종료 명령어를 실행하고, SIG\_TERM 신호가 브로커에게 전달 됩니다.  
2) SIG\_TERM 신호를 받은 브로커는 컨트롤러에게 알립니다.  
3) 컨트롤러는 리더 선출 작업을 진행하고, 해당 정보를 주피커에 기록 합니다.  
4) 컨트롤러는 새로운 리더 정보를 다른 브로커들에게 전송하게 됩니다.  
5) 컨트롤러는 종료 요청을 보낸 브로커에게 정상 종료한다는 응답을 보내게 됩니다.  
6) 응답을 받은 브로커는 캐시에 있는 내용을 디스크에 저장하고 종료 하게 됩니다.

### 종료 방식의 차이

제어된 종료 와 예기치 않은 종료(급작스러운 종료)와의 큰 차이는 다운타임 입니다.  
  
제어된 종료를 사용하면 카프카 내부적으로 파티션들이 다운타임을 최소화 할 수 있습니다. 이유는 브로커가 종료 되기 전, 컨트롤러는 해당 브로커가 리더도 할당된 전체 파티션에 대해서 리더 재 선출 작업을 진행하기 때문 입니다.  
  
물론 제어된 종료라도 리더 선출 작업 시간 동안 일시적인 다운타임은 발생할 수 있습니다. 다만 리더 선출 작업 대상 파티션들의 리더들이 활성화된 상태에서 컨트롤러는 순차적으로 하나의 파티션 마다 리더를 선출하게 되므로 결과적으로 각 파티션별 다운타임을 최소화 할 수 있습니다.  
  
  
예기치 않은 종료(장애에 의한)에 의해 리더 선출 과정은 이미 대상 파티션들의 리더가 종료 된 상태가 되고, 파티션들의 다운타임은 새로운 리더 선출 작업이 완료 될 때 까지 지속되게 됩니다. 컨트롤러는 순차적으로 하나의 파티션마디 리더를 선출하게 되며, 첫번째 대상 파티션의 다운타임은 길지 않을수 있지만 마지막 리더 선출 대상의 파티션은 다운타임이 오랜시간이 걸려서 수행될 것 입니다.  
  
또한 제어된 종료의 경우 (리더)브로커는 자신의 모든 로그를 디스크에 동기화한 후 종료됨에 따라 이후 다시 브로커가 재시작할 때 로그 복구 시간이 짧아지게 됩니다.  
다양한 장점이 있는 제어된 종료를 사용하려면 control.shutdown.enable = true 설정이 브로커 설정 파일인 server.properties 파일에 설정이 되어 있어야 합니다.  
  
  
브로커의 설정파일에 옵션이 명시되어 있지 않다고 해서 옵션이 반영되지 않은 것은 아닙니다. 기본값은 따로 명시하지 않아도 적용되므로 현재 브로커의 설정을 확인해볼 필요는 있습니다.  
  
카프카에서 제공하는 **kafka-configs.sh** 명령어를 이용해서 --broker 옵션과 확인하고자 하는 브로커 아이디를 통해서 현재 브로커의 설정 상태를 확인할 수 있습니다.

$ kafka-configs.sh --bootstrap-server kafka1:9092 \

--broker 1 --describe --all

All configs for broker 1 are:

log.cleaner.min.compaction.lag.ms=0 sensitive=false synonyms={DEFAULT\_CONFIG:log.cleaner.min.compaction.lag.ms=0}

offsets.topic.num.partitions=50 sensitive=false synonyms={DEFAULT\_CONFIG:offsets.topic.num.partitions=50}

log.flush.interval.messages=9223372036854775807 sensitive=false synonyms={DEFAULT\_CONFIG:log.flush.interval.messages=9223372036854775807}

controller.socket.timeout.ms=30000 sensitive=false synonyms={DEFAULT\_CONFIG:controller.socket.timeout.ms=30000}

principal.builder.class=null sensitive=false synonyms={}

log.flush.interval.ms=null sensitive=false synonyms={}

< 내용 중략 >

현재 사용하는 브로커의 제어된 종료 설정 상태를 확인 하여 설정이 되어 있지 않다면 설정하는 것을 권장됩니다.

## 로그(로그 세그먼트)

카프카의 토픽으로 들어오는 메세지(레코드)는 **세그먼트(segement, 로그 세그먼트 라고도 함)** 라는 파일에 저장되게 됩니다.  
  
메세지는 정해진 형식에 맞추어 순차적으로 로그 세그먼트 파일에 기록 됩니다. 로그 세그먼트에는 메세지의 내용만 저장되는 것이 아니라 메세지의 키, 밸류, 오프셋, 메세지 크기 같은 정보도 함께 저장 되며, 로그 세그먼트 파일들은 브로커의 로컬 디스크에 보관되어 있습니다.   
  
하나의 로그 세그먼트 크기가 너무 커져버리면 파일을 관리하기 어렵기 때문에 로그 세그먼트의 최대 크기는 1GB 가 기본값 입니다.  
로그 세그먼트가 1GB보다 커지는 경우에는 기본적으로 롤링(rolling) 전략을 적용합니다.  
  
로그 세그먼트의 크기가 1GB에 도달하면 해당 세그먼트 파일을 클로즈(close) 하고, 새로운 로그 세그먼트를 생성하는 방식으로 진행 합니다.  
  
1GB 크기에 설정에 의해서 파일이 롤링이 되지만, 파일이 무한정 늘어날 경우를 대비하여 관리자는 로그 세그먼트에 대한 관리 계획을 수립해야 합니다.  
  
관리 계획으로는 크게 로그 세그먼트의 삭제 또는 컴팩션(compaction) 으로 구분할 수 있습니다.

### 로그 세그먼트 삭제

로그 세그먼트 삭제 옵션은 브로커의 설정 파일인 server.properties 에서 log.cleanup.policy 가 delete 로 명시되어야 합니다.  
  
해당 값은 기본값으로 적용됨으로 별도로 설정하지 않아도 로그 세그먼트는 삭제 정책이 적용되게 됩니다.  
  
  
세그먼트 삭제 테스트를 진행해보도록 하겠습니다. 계속 사용하였던 jade-test01 토픽을 이용하도록 하겠습니다.  
토픽 생성에 관한 내용은 이전 포스팅 아래 링크에서 확인 하시면 됩니다.



**카프카(Kafka) (4) - 내부 동작 원리와 구현 1**

토픽 생성하기

먼저 콘솔 컨슈머를 통해서 jade-test01 토픽의 메세지를 확인 해보도록 하겠습니다.  
--from-beginning 옵션을 통해서 처음 부터 메세지를 가져 올 수 있습니다.

$ kafka-console-consumer.sh --bootstrap-server kafka1:9092 \

--topic jade-test01 --from-beginning

출력)

test message1

test message2

토픽을 생성하고 입력한 2개의 메세지를 잘 가져온 것 같습니다.  
  
이제 jade-test01 토픽의 retention.ms 옵션을 조정해서 메세지를 삭제하도록 하겠으며 옵션 조정은 kafka-configs.sh 를 이용합니다.

$ kafka-configs.sh --bootstrap-server kafka1:9092 \

--topic jade-test01 --add-config retention.ms=0 --alter

리턴 메세지)

Completed updating config for topic jade-test01.

위의 명령어는 retention.ms=0 이라는 설정을 추가하는 명령어 입니다. 여기서 retention.ms=0 이란 로그 세그먼트 보관 시간이 해당 숫자 보다 크다면 세그먼트를 삭제한다는 의미 입니다. 보관 시간이 0 보다 클 수 밖에 없으므로, 현재 저장된 모든 메세지는 삭제가 될 것 입니다.  
  
정상적으로 옵션이 반영 되었는지를 확인 해보기 위해서 먼저 describe 로 확인 해보겠습니다.

$ kafka-topics.sh --bootstrap-server kafka1:9092 \

--topic jade-test01 --describe

결과)

Topic: jade-test01 PartitionCount: 1

ReplicationFactor: 3 Configs: segment.bytes=1073741824,retention.ms=0

Topic: jade-test01 Partition: 0

Leader: 2 Replicas: 1,2,3 Isr: 2,3,1

\* 가로 길이에 따른 개행이 되어 있습니다.  
  
출력 내용을 확인 해보면 Configs 라인에 retention.ms=0 이 추가 되어 있는 것을 확인할 수 있습니다.  
  
로그 세그먼트 삭제는 기본적으로 5분 단위로 진행 됩니다. 따라서 명령어를 수행하고 바로 삭제 되는 것이 아니라 최대 5분 뒤에 수행이 되게 됩니다.  
  
디렉토리의 로그 파일을 확인 해본 결과 아래 와 같이 용량이 0 인 새로운 파일명 2.log 파일이 새로 생성이 된 것을 확인 할 수 있습니다.

$ ls -arlt

total 40

-rw-r--r--. 1 root root 10 May 22 03:27 00000000000000000002.snapshot

-rw-r--r--. 1 root root 8 May 22 18:13 leader-epoch-checkpoint

-rw-r--r--. 1 root root 0 May 22 18:15 00000000000000000002.log

당연히 콘솔 컨슈머로 조회를 해봐도 아무런 결과가 나오지는 않습니다.

$ kafka-console-consumer.sh --bootstrap-server kafka1:9092 \

--topic jade-test01 --from-beginning

< 결과 없음, 즉 메세지가 없음 >

설정 하였던 옵션을 다시 제거해보도록 하겠습니다.

$ kafka-configs.sh --bootstrap-server kafka1:9092 \

--topic jade-test01 --delete-config retention.ms --alter

리턴 메세지)

Completed updating config for topic jade-test01.

다시 describe 를 통해서 옵션이 변경 되었는지 살펴보도록 하겠습니다.

$ kafka-topics.sh --bootstrap-server kafka1:9092 \

--topic jade-test01 --describe

결과)

Topic: jade-test01 PartitionCount: 1

ReplicationFactor: 3 Configs: segment.bytes=1073741824

Topic: jade-test01 Partition: 0

Leader: 2 Replicas: 1,2,3 Isr: 2,3,1

\* 가로 길이에 따른 개행이 되어 있습니다.  
  
출력 내용에서 설정하였던 retention.ms 옵션이 삭제 된 것을 확인 할 수 있습니다.  
  
카프카에서는 토픽마다 보관주기를 조정하여(설정하여) 얼마만큼의 기간동안 카프카에 로그를 저장할지를 결정하고 관리할 수 있습니다.  
  
관리자가 토픽에 별도의 retention.ms 옵션을 설정하지 않으면 카프카의 server.properties에 적용된 옵션값이 적용 됩니다.  
카프카의 기본값은 7일이며, 토픽 단위로 별도로 설정하지 않았다면 모든 세그먼트는 7일이 지난 후 삭제가 되게 됩니다.  
  
세그먼트 관리에 대한 설정중에는 기간이 아닌 용량으로도 설정할 수 있으며 retention.bytes 라는 옵션을 지정하여 사용 합니다.

### 로그 세그먼트 컴팩션

**컴팩션(compaction)** 은 카프카에서 제공하는 로그 세그먼트 관리 정책 중 하나로, 로그를 삭제하지 않고 컴팩션하여 보관할 수 있는 방법 입니다.   
  
로그 컴팩션은 기본적으로 로컬 디스크에 저장되어 있는 세그먼트를 대상으로 실행되는데, 현재 활성화된 세그먼트는 제외하고 나머지 세그먼트들을 대상으로 컴팩션이 실행 됩니다.  
  
컴팩션하더라도 카프카의 로컬 디스크에 로그를 무기한 보관한다면, 로그의 용량은 감당할 수 없이 커질 것 입니다.  
따라서 카프카에서는 단순하게 메세지를 컴팩션만 해서 보관하기 보다는 좀더 효율적인 방법으로 컴팩션을 합니다.  
  
카프카에서 로그 세그먼트를 컴팩션하면 메세지(레코드)의 키값을 기준으로 마지막의 데이터만 보관하게 됩니다.  
  
메세지의 키값을 기준으로 컴팩션하는 방법이 다소 생소 할 수 있으며, 로그 컴팩션 기능을 이용하는 대표적인 예제는 바로 카프카의 \_\_consumer\_offset 토픽 입니다.  
  
\_\_consumer\_offset 토픽은 카프카의 내부 토픽으로, 컨슈머 그룹의 정보를 저장하는 토픽 입니다. 각 컨슈머 그룹의 중요한 정보는 해당 컨슈머 그룹이 어디까지 읽었는지를 나타내는 오프셋 커밋 정보이며, \_\_consumer\_offset 에 키(컨슈머 그룹명, 토픽명)와 밸류(오프셋 커밋 정보) 형태로 메세지가 저장 됩니다.  
  
  
예를 들어 CG01 컨슈머 그룹이 T01 토픽을 컨슘하고, 첫 번째 메세지를 읽고 커밋했다고 가정한다면 이 정보는 키 와 밸류 형태의 메세지로 \_\_consumer\_offset 토픽에 저장 됩니다.  
  
따라서 키는 CG01(컨슈머 그룹), T01(토픽명) 이 되고 밸류는 1(오프셋) 인 메세지가 \_\_consumer\_offset 토픽에 저장되게 됩니다.  
  
그리고 1시간 뒤에 컨슘하면서 두번째 메세지를 읽고 커밋하면 밸류는 2인 메세지가 \_\_consumer\_offset 에 저장됩니다.  
  
그리고 다시 또 1시간 뒤에 세번째 메세지를 읽고 커밋을 하면 밸류는 3인 메세지가 \_consumer\_offset 에 저장됩니다.  
  
그래서 현재 \_\_consume\_offset 토픽에 저장된 메세지는 총 3개로 밸류가 1,2,3 인 메세지 입니다.  
이후 로그 컴팩션이 동작하면 CG01,T01 키값의 밸류 중에서 마지막 메세지인 3만 남게 됩니다. 컨슈머 그룹은 항상 마지막으로 커밋된 오프셋 정보가 중요하므로, 과거에 커밋된 정보들은 삭제 되어도 무방 합니다.  
  
이렇게 로그 컴팩션은 메세지의 키값을 기준으로 과거 정보는 중요하지 않고 가장 마지막 값이 필요한 경우에만 사용 합니다.  
  
  
또 다른 사용 예시로는 현재 구매 현황 상태를 보여주는 시스템에서도 로그 컴팩션을 이용할 수 있습니다.  
이때는 고유한 사용자 아이디가 메세지의 키값이고 현재의 구매 상태 정보가 메세지의 밸류값이 되게 됩니다.  
  
구매 상태 정보는 주문 완료 -> 배송 준비 -> 배송 중 -> 배송 완료 의 총 4단계로 나타냅니다.  
  
구매한 사용자 아이디(메세지의 키) 를 기준으로 최종 상태(메세지의 밸류)만 사용자에게 노출하면 되므로, 카프카의 로그 컴팩션 기능을 활용할 수 있습니다.  
  
프로듀서가 카프카로 메세지를 전송할 때, 메세지에는 메세지(레코드)의 키 와 밸류를 같이 전송하게 됩니다. 키는 필수이며, 밸류는 필수는 아닙니다. 따라서 로그 컴팩션 기능을 사용하고자 한다면 키프카로 메세지를 전송할때도 키를 필수로 전송해야 합니다.

[텍스트, 체척계이(가) 표시된 사진

자동 생성된 설명](https://hoing.io/storage/2022/05/log_-compaction_1.png)

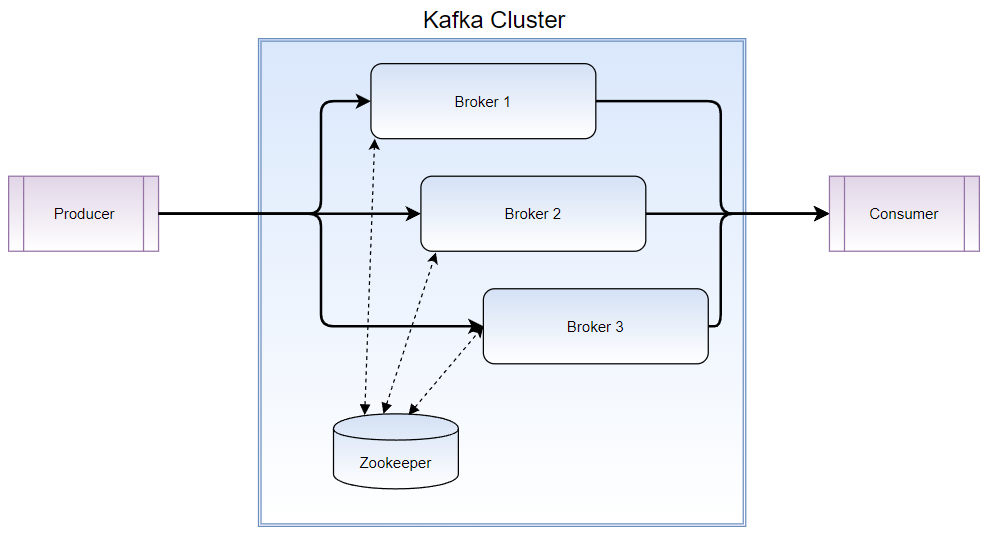
[[**Introduction to Topic Log Compaction in Apache Kafka**](https://medium.com/swlh/introduction-to-topic-log-compaction-in-apache-kafka-3e4d4afd2262)]

로그 컴팩션의 장점은 **빠른 장애 복구** 입니다. 장애 복구 시 전체 로그를 복구하지 않고, 메세지의 키를 기준으로 최신의 상태만 복구를 합니다. 따라서 전체 로그를 복구할 때 보다 복구 시간을 줄일 수 있다는 장점이 있습니다.  
  
  
컨슈머가 처리한 메세지들에서 오류가 발견되어 재처리가 필요한 상황이라고 가정한다면 로그 컴팩션 기능이 사용하지 않는다면 재처리를 위해서 모든 로그를 다시 적용을 해야 겠지만, 로그 컴팩션을 이용하여 마지막 메세지만 빠르게 처리할 수 있습니다.  
  
하지만 빠른 재처리라는 장점이 있다고 하여 모든 토픽에서 로그 컴팩트 적용은 고민이 필요 한 부분 입니다.  
키 값 기준으로 최종값만 필요한 워크로드에 적용하는 것이 바람직합니다.  
  
또한 카프카에서 로그 컴팩트 작업이 실행되는 동안 브로커의 과도한 입출력(I/O) 부하가 발생할 수 있으므로 이점도 주의 해야 합니다.

**시스템 아키텍처**

클러스터를 실행하기 위해 아래와 같은 프로세스들을 시작해야 합니다.

* **Zookeeper** : 클러스터 노드 간의 상태를 유지하기 위해 Kafka에서 사용
* **Kafka brokers** : 데이터를 저장하고 내보내는 파이프라인의 "파이프"
* **Producers** : 클러스터에 데이터를 입력
* **Consumers** : 클러스터로부터 데이터를 조회



*이 다이어그램의 각 블록은 네트워크의 다른 시스템에 구성할 수도 있습니다.*

# [Kafka] Option 및 Config 정리

Zookeeper Config (zookeeper.properties)

|  |  |
| --- | --- |
| 옵션 | 설명 |
| dataDir=/data/zookeeper/data | 주키퍼의 상태, 스냅션, 트랜잭션 로그들을 저장하고 업데이트하는 디렉토리의 위치 |
| dataLogDir=/data/zookeeper/logs | 트랜잭션 로그를 저장하는 디렉터리. 특별한 설정을 하지 않으면 dataDir에 저장.  성능상 다른 디스크의 디렉터리에 분리하는것이 좋다. |
| clientPort=2181 | 클라이언트로 요청을 받기 위한 포트 |
| initLimit=5 | 처음 주키퍼의 follower가 leader에 접속하거나 데이터를 동기화 시키기 위해 사용되는  최대 시간을 제한하기 위해 사용하는 timeout 단위.  초기에 팔로워가 리더에 접속하거나 데이터를 동기화 시키기 위한 시간으로 단위는 tickTime 이며  initLimit \* tickTime으로 계산된다. |
| syncLimit=2 | follower가 leader로부터 얼마나 sync에 뒤쳐질 수 있는지를 제한하는 timeout 단위. 이 시간동안 sync가 안되면 해당 팔로워는 클러스터에서 제외된다. |
| maxClientCnxns=0 | 클라이언트로 부터 동시에 접속할 수 있는 연결 수를 지정. 연결수는 클라이언트 IP당 개수이며, 기본값은 10이며 0은 무제한이다. |
| admin.enableServer=false | 주키퍼는 default로 admin server를 기동하는데에 8080 port를 사용하는데, 주키퍼 기동 시  port 충돌을 방지하기 위해 false 로 사용 |
| server.1=xxx.xx.xx.xxx:2888:3888 server.2=xxx.xx.xx.xxx:2888:3888 server.3=xxx.xx.xx.xxx:2888:3888 | 주키퍼 클러스터(앙상블) 구성을 위한 서버 설정이며, server.myid(서버를 식별하는 ID) 형식으로 사용.  IP는 각 서버의 IP를 입력하고, 2888 포트는 follower가 leader에 접속하기 위한 포트이며,  3888 포트는 leader를 선출하는데 사용되는 포트 |
| tickTime=2000 | 주키퍼가 사용하는 기본 시간 단위(milliseconds)로, heartbeats를 보내는데에 사용된다.  default는 2000. |

initLimit, syncLimit 두 가지 timeout들은 모두 앞서 설정한 tickTime의 시간 단위를 사용한다. 즉, 위 설정에서 initLimit의 timeout은 5 ticks이며 tickTime이 2000 milliseconds이기 때문에, '10초'로 설정된 것이다.

Kafka Config (server.properties)

|  |  |
| --- | --- |
| 옵션 | 설명 |
| broker.id.generation.enable  =true | 해당 서버에서 사용할 브로커의 ID이며, 각 브로커의 고유 정수로 설정해야 함.  **broker.id.generation.enable** 옵션을 사용하여 서버에서 브로커 id 자동 생성을 활성화. |
| listeners=  PLAINTEXT:  //xxx.xx.xx.xxx:9092,EXTERNAL:  //xxx.xx.xx.xxx:9093 | Broker가 사용하는 호스트와 포트를 지정.  내부 브로커들 간의 통신을 위한 엔드포인트와 외부 클라이언트를 위한 엔드포인트를 구분하려면  **advertised.listeners** 옵션을 사용해야 함. |
| listener.security.protocol.map=  PLAINTEXT:PLAINTEXT,  EXTERNAL:PLAINTEXT | 위 listeners 옵션에서 사용된 PLAINTEXT 는 보안 프로토콜이 아니기 때문에 해당 옵션을 설정해야함 |
| num.network.threads=3 | 서버에서 네트워크로부터 요청을 받고 네트워크로 응답을 전송할 때 사용할 스레드 수.   (네트워크 요청을 처리하는 Thread의 개수) |
| num.io.threads=8 | 서버에서 request를 처리할 때 사용하는 스레드 수.(I/O가 생길때 마다 생성되는 Thread의 개수) |
| socket.send.buffer.bytes=  102400 | 네트워크 요청을 처리할 소켓 서버가 사용하는 수신 메모리 사이즈 |
| socket.receive.buffer.bytes=  102400 | 네트워크 요청을 처리할 소켓 서버가 사용하는 송신 메모리 사이즈 |
| socket.request.max.bytes=  104857600 | 서버가 받을 수 있는 최대 요청 사이즈이며, JAVA의 Heap 보다 작게 설정해야 함. |
| log.dirs=/data/kafka/logs | 로그 파일을 저장할 쉼표로 구분된 디렉터리 목록.  - 로그 파일 종류 1) Log Segment File(.log) : 전송되는 topic 메시지와 같이 실제 메타데이터가 저장되는 파일  2) Index File(.index) : offset 번호를 가지고 해당 데이터를 빨리 찾기위한 인덱스 파일  3) Time-based Index File(.timeindex) : timestamp를 기반으로 데이터를 빨리 찾기위한 인덱스 파일  4) leader-epoch-checkpoint : 리더가 변경이 되었을 때 offset 정보를 저장하는 파일 |
| num.partitions=1 | 토픽당 파티션의 수를 의미하며, 입력한 수만큼 병렬처리를 할 수 있지만 데이터 파일도 그만큼 늘어남. |
| num.recovery.threads.per.data.  dir=1 | 기동 시 로그 복구와, 셧 다운 시 플러시에 사용할 데이터 디렉토리 당 스레드 수. 장애에 빠르게 대응하고자 한다면 높은 값을 설정해야함. |
| offsets.topic.replication.factor  =3 | 오프셋 토픽(\_\_consumer\_offsets)의 replication factor를 지정. 클러스터 크기가 이 값보다 작으면  internal topic 생성에 실패한다.  \_\_consumer\_offsets : 각 topic에 대한 커밋된 오프셋에 대한 정보를 저장하는 데 사용된다. |
| default.replication.factor=3 | 자동으로 생성된 토픽에 대한 기본 복제 인수(factor)다. 기본값은 1. |
| transaction.state.log.  replication.factor=3 | 트랜잭션 토픽(\_\_transaction\_state)의 replication factor를 지정. 클러스터 크기가 이 값보다 작으면  internal topic 생성에 실패한다.  카프카 트랜잭션은 프로듀서와 컨슈머단에서 옵션을 설정해야 함. \_\_transaction\_state : 이 topic은 프로듀서가 보내는 데이터의 트랜젝션을 기록하는데 사용되며,  기록된 데이터는 트랜젝션이 활성화된 컨슈머가 데이터를 가져가는데 사용한다. |
| transaction.state.log.min.isr=1 | 트랜잭션 토픽에 대한 min.insync.replicas 설정을 재정의함. |
| min.insync.replicas=2 | 프로듀서가 acks=all(-1)로 설정하여 메시지를 보낼 때, write를 성공하기 위한 최소 복제본의 수.  ex) N개의 Replica가 있고 해당 옵션이 2일 경우, N-2개의 장애를 허용할 수 있음.  datahub 서버의 acks는 all이고 해당 값은 1로 설정돼있는데, 데이터 유실을 없게하려면 1보다  큰 값으로 설정해야 할 것으로 생각됨. (보통 replica가 3일때, 2로 많이 사용한다함) |
| acks=all | 프로듀서가 kafka에 데이터를 전달하고 데이터를 잘 받았는지 확인하는 옵션.  1) acks=0 : 프로듀서가 데이터를 전달한 후 확인하지 않는 것. 리더 파티션은 데이터가 저장된 이후  몇번째 오프셋에 저장되는지 리턴하게 되는데, 0으로 설정하면 이를 받지 않고 전달만 하게 됨.  이렇게 지정하는 경우 retries 옵션도 무의미하게 되며 데이터 유실이 발생하는 경우가 발생할 수 있지만  속도는 훨씬 빨라짐. 데이터보다 속도에 중점을 두는 경우 사용할 수 있는 옵션.  2) acks=1 : 데이터를 전달하고 리더 파티션에 정상적으로 적재되었는지를 확인하는 옵션. 리더 파티션에  적재가 되었다는 응답을 받지 못하면 정상적으로 전달되었다고 판단하지 않으며, 데이터를 재전송하게 됨.  그렇다고 데이터 유실이 발생하지 않는 것은 아닌데, 리더 파티션으로 데이터를 전달한 직후 팔로워  파티션과 데이터가 동기화되기 전에 리더 파티션에 문제가 발생하면 데이터의 유실이 발생할 수 있음.  3) acks=all(-1) : 리더 파티션과 팔로워 파티션에 데이터가 모두 적재되었는지를 확인하는 옵션으로 당연히  0, 1 옵션을 사용하는 경우보다 속도가 느리지만 그만큼의 안정성을 보장하게 됨. |
| delete.topic.enable=true | --delete --topic 명령을 사용하기 위해 해당 옵션을 true로 설정해야 함. |
| log.retention.hours=12 | 로그 파일(메시지)의 수명. 기본값은 168(7일).  hours 단위로 지정하는 것이 아닌 분단위 minutes, 밀리초단위 ms 설정도 지원.  일반적으로 3일 정도로 설정하여 사용한다고 하는데, 일배치로 마이그레이션을 하기 위해  1일 미만으로 설정 예정. |
| log.segment.bytes=1073741824 | 로그 세그먼트 파일의 최대 크기. 이 크기에 도달하면 새 로그 세그먼트가 생성됨. |
| log.retention.check.interval.ms  =300000 | 로그를 삭제하기 위한 check 간격으로 ms단위임. |
| zookeeper.connect=zookeeper-01:2181,zookeeper-02:2181,  zookeeper-03:2181 | 연결된 zookeeper 서버 정보 |
| zookeeper.connection.  timeout.ms  =18000 | 주키퍼 연결 최대 대기 시간 |
| group.initial.rebalance.delay.ms  =0 | 그룹 코디네이터가 새 그룹에서 처음 리밸런스(리더 역할을 할 클라이언트를 선출하며 파티션을  분배하는 과정)를 수행하기 전에, 더 많은 컨슈머가 그룹에 들어올 수 있도록 기다리는 시간.   더 오래 기다리면 앞으로 리밸런스를 덜 할 순 있지만, 리밸런스를 시작하기까지 시간이 더 오래 걸린다.  개발 및 테스트 중에는 테스트 실행 시간을 지연시키지 않기 위해 이것을 0으로 설정하는 것이  바람직할 수 있다. |
| auto.create.topics.enable  =false | 브로커 서버의 토픽 자동 생성을 활성화 함.  Connect의 **topic.creation.enable**옵션과 독립적이며, Connect의 옵션이 우선 적용됨. |
| confluent.topic.replication.facto  r=1 | Confluent Control Center 모니터링을 위한 옵션. |

Connect (connect-distributed.properties)

|  |  |
| --- | --- |
| 옵션 | 설명 |
| bootstrap.servers=kafka-broker-01:9092,kafka-broker-02:9092,kafka-  broker-03:9092,kafka-broker-04:9092  ,kafka-broker-05:9092 | 카프카 클러스터에 대한 초기 커넥션을 구축하는 데 사용할 호스트/포트 쌍 리스트.  kafka broker 서버 = bootstrap 서버. |
| group.id=connect-cluster | connect를 같은 클러스터로 묶기 위한 그룹 설정. |
| key.converter=org.apache.kafka.connect.  storage.StringConverter value.converter=org.apache.kafka.  connect.json.  JsonConverter | 카프카 커넥트 포맷과 카프카에 기록한 직렬화된 포맷 간을 변환할 때 사용할 컨버터 클래스.  카프카에서 쓰거나 읽은 메세지 키 포맷은 이 클래스가 제어하며,  컨버터는 커넥터와는 독립적이기 때문에, 사용하는 커넥터와는 무관하게 어떤 직렬화 포맷으로도  작업할 수 있다. |
| key.converter.schemas.enable=false value.converter.schemas.enable=false | key/value값이 내부 schema와 data를 모두 포함하는 복합 객체로 처리되도록 하는 설정 |
| offset.storage.topic=connect-offsets | connect offset을 저장할 kafka topic명. |
| offset.storage.replication.factor=3 | offset을 저장하는 토픽을 생성할때 사용할 replication factor. default는 3 |
| config.storage.topic=connect-configs | connector 설정을 저장할 kafka topic명 |
| config.storage.replication.factor=3 | 설정을 저장하는 topic을 생성할 때 사용할 replication factor. default는 3 |
| status.storage.topic=connect-status | connector와 task 상태를 저장할 topic명. |
| status.storage.replication.factor=3 | 상태를 저장하는 topic을 생성할 때 사용할 replication factor. default는 3 |
| offset.flush.interval.ms=10000 | task에서 offset 커밋을 시도하는 간격 |
| plugin.path=/home/genius/plugins | plugin이 들어있는 path 리스트. |
| connector.client.config.override.policy=All | 커넥터가 broker의 모든 설정을 바꿀 수 있도록 All로 정의. |
| topic.creation.enable=true | source connector에서 사용하는 topic의 자동 생성을 허용할지 여부. |

운영보안: JMX 익스포터 대시보드 사용

Kafka 개발 환경

• 아파치 카프카 2.6

• 아파치 주키퍼 3.5.9

• 컨플루언트 카프카 6.1

• 자바 1.8

• 파이썬 3.7.9

• 앤서블 2.9.12

• 운영체제: 온프레미스(센트OS7), AWS(아마존 리눅스2)