```
package lab.streaming.exercise.ch6



import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.{SparkConf, SparkContext}
import kafka.serializer.StringDecoder
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.streaming.{Milliseconds, Seconds,
StreamingContext}
import org.apache.spark.sql.SparkSession


case class Person(name: String, age: Int)

/**
 * 1. Kafka 다운로드....
 *   1.1 # cd ~/download
 *   1.2 # wget http://mirror.apache-kr.org/kafka/0.10.2.1/
kafka_2.11-0.10.2.1.tgz
 *   1.3 # tar xvfz kafka_2.11-0.10.2.1.tgz
 *
 * 2. Zookeeper 설정....(Kafka 내부 포함 Zookeeper 사용....)
 *   2.1 # cd ~/apps/kafka/config
 *   2.2 # vi zookeeper.properties
 *          "i" 키 입력 => 입력모드 전환.... 키보드 화살표로 이동하여 필요 정보 입
력 및 수정....
 *          dataDir=~/apps/kafka/zookeeper         //--기존 정보 수
정....dataDir=/tmp/zookeeper
 *          server.1=localhost:2889:3889
                                                              //--신규
추가....
 *          "Esc" 키 입력 => 입력모드에서 나오기....
 *          ":wq" 키 입력 => 저장 후 vi 종료....
 *   2.3 # cat zookeeper.properties       //--수정 내용 확인....
 *   2.4 # cd ~/apps/kafka
 *   2.5 # mkdir zookeeper
 *   2.6 # cd zookeeper
 *   2.7 # echo '1' > myid
 *   2.8 # cat myid               //--myid에 '1' 입력 되어 있음 확인....
 *
 * 3. Kafka 설정....
 *   3.1 # cd ~/apps/kafka/config
 *   3.2 # vi server.properties
 *          "i" 키 입력 => 입력모드 전환.... 키보드 화살표로 이동하여 필요 정보 입
력 및 수정....
 *          log.dirs=~/apps/kafka/logs             //--기존 정보 수
정....log.dirs=/tmp/kafka-logs
 *          delete.topic.enable=true               //--토픽 삭제 가능하
도록 설정....               //--기존 정보 주석 제거....   //--기존 정보 없을 경
우 신규 추가....(0.8.2.2 같은 OLD 버전)....
 *          "Esc" 키 입력 => 입력모드에서 나오기....
 *          ":wq" 키 입력 => 저장 후 vi 종료....
 *   3.3 # cat server.properties //--수정 내용 확인....
```

```
 *   3.4 # cd ~/apps/kafka
 *   3.5 # mkdir logs
 *
 * 4. Zookeeper 실행....
 *   4.1 # cd ~/apps/kafka/bin
 *   4.2 # nohup ./zookeeper-server-start.sh ../config/
zookeeper.properties > nohup_zookeeper.out &
 *   4.3 # jps           //--QuorumPeerMain 프로세스 확인....
 *
 * 5. Kafka 실행....
 *   5.1 # cd /kafka_2.11-0.10.2.1/bin
 *   5.2 # nohup ./kafka-server-start.sh ../config/server.properties
> nohup_kafka.out &
 *   5.2 # nohup env JMX_PORT=9995 ./kafka-server-start.sh ../
config/server.properties > nohup_kafka.out &               //--JMX
포트 지정하여 실행할 경우....
 *   5.3 # jps           //--Kafka 프로세스 확인....
 *
 * 6. Kafka 토픽 생성....
 *   6.1 # cd /kafka_2.11-0.10.2.1/bin
 *   6.2 # ./kafka-topics.sh --create --topic log --partitions 4 --
replication-factor 1 --zookeeper CentOS7-14:2181   //--"log" 토픽 생
성....
 *   6.3 # ./kafka-topics.sh --list --zookeeper
CentOS7-14:2181  //--토픽 리스트 확인....
 *   6.4 # ./kafka-topics.sh --describe --topic log --zookeeper
CentOS7-14:2181  //--"log" 토픽 상세 정보 보기....
 *   6.5 # ./kafka-topics.sh --delete --topic log --zookeeper
CentOS7-14:2181  //--"log" 토픽 삭제.... 필요시 실행....
 *   6.6 # ./kafka-topics.sh --alter --topic log --partitions 8 --
zookeeper CentOS7-14:2181                  //--partitions 개수 변
경.... //--Option "[replication-factor]" can't be used with
option"[alter]"
 *
 * 7. Kafka Consumer 실행.... (별도 SSH 세션에서 실행....)
 *   7.1 # cd /kafka_2.11-0.10.2.1/bin
 *   7.2 # ./kafka-console-consumer.sh --topic log --zookeeper
CentOS7-14:2181  //--zookeeper 리스트 사용....(0.8.2.2 같은 OLD 버전)....
 *   7.2 # ./kafka-console-consumer.sh --topic log --bootstrap-
server CentOS7-14:9092   //--bootstrap-server(broker) 리스트 사용....
 *   7.3 # 토픽에 데이터가 들어오면 Consumer가 토픽에서 데이터를 가져와 콘솔에 출력
함....
 *
 * 8. Kafka Producer 실행.... (별도 SSH 세션에서 실행....)
 *   8.1 # cd /kafka_2.11-0.10.2.1/bin
 *   8.2 # ./kafka-console-producer.sh --sync --topic log --broker-
list CentOS7-14:9092
 *   8.3 # 콘솔에 데이터 입력.... => 토픽에 전송됨....
 *
 * 9. KafkaWordCountProducer 실행....(로컬 실행....)
 *   9.1 run : Run As > Scala Application
 *
 **/
```

```
object Exercise1 {

  def main(args: Array[String]): Unit = {

    /* 6.2.1절
    //application을 submit해서 netcat으로 실행 확인 해야 한다.
    //netcat을 먼저 실행하고 submit하면 확인 할 수 있다.
    /** 2. Socket Server(Netcat) 실행....
                                    2.1 CentOS7-14 SSH 연결(MobaXterm)
                                    2.2 # cd ~/apps/spark
                                    2.3 # yum install nc -
y               //--Netcat(nc) 설치....(필요시....)
                                    2.4 # nc -lk 9999
                //--Netcat 실행(port 9999)....
          **/2.5 # Netcat 콘솔에 데이터 입력....        //--Netcat 데이터
송신....

    val conf = new
SparkConf().setMaster("local[3]").setAppName("SocketSample")
    val ssc = new StreamingContext(conf, Seconds(3))


    val ds = ssc.socketTextStream("localhost", 9000)


    ds.print()
    */


    /* 6.2.3절
    import scala.collection.mutable
    val rdd1 = ssc.sparkContext.parallelize(List("a", "b", "c"))
    val rdd2 = ssc.sparkContext.parallelize(List("c", "d", "e"))
    val queue = mutable.Queue(rdd1, rdd2)

    val ds = ssc.queueStream(queue) // RDD로 구성된 Queue를 이용해
DStream을 만드는 방법
                                    // DStream : 스파크 스트리밍에서 사용
하는 새로운 데이터 모델, 고정되지 않고 끊임없이 생성되는 연속된 데이터를 나타내기 위한 추
상 모델

    ds.print()
    */


    /*
    import org.apache.spark.streaming.kafka._

    val conf = new
SparkConf().setMaster("local[3]").setAppName("KafkaSample")
    val sc = new SparkContext(conf)
    val ssc = new StreamingContext(sc, Seconds(3))
    val zkQuorum = "localhost:2181" //주키퍼 쿼럼 정보
    val groupId = "test-consumer-group1" //컨슈머 그룹명
```

```
    val topics = Map("test" -> 3) //토픽 이름, 스레드 수

    val ds1 = KafkaUtils.createStream(ssc, zkQuorum, groupId,
topics) // stream data가 없으면 error 발생
    val ds2 = KafkaUtils.createDirectStream[String, String,
StringDecoder, StringDecoder](ssc,
        Map("metadata.broker.list" -> "localhost:9092"),
        Set("test")) // kafka를 실행시켜야 된다.

    ds1.print
    ds2.print
    */

    /* 6.3절
    import scala.collection.mutable

    val conf = new
SparkConf().setMaster("local[3]").setAppName("StreamingOps")
    val ssc = new StreamingContext(conf, Seconds(3))

    val rdd1 = ssc.sparkContext.parallelize(List("a", "b", "c", "c",
"c"))
    val rdd2 = ssc.sparkContext.parallelize(List("1,2,3,4,5"))
    val rdd3 = ssc.sparkContext.parallelize(List(("k1", "r1"),
("k2", "r2"), ("k3", "r3")))
    val rdd4 = ssc.sparkContext.parallelize(List(("k1", "s1"),
("k2", "s2")))
    val rdd5 = ssc.sparkContext.range(1, 6)

    val q1 = mutable.Queue(rdd1)
    val q2 = mutable.Queue(rdd2)
    val q3 = mutable.Queue(rdd3)
    val q4 = mutable.Queue(rdd4)
    val q5 = mutable.Queue(rdd5)

    val ds1 = ssc.queueStream(q1, false)
    val ds2 = ssc.queueStream(q2, false)
    val ds3 = ssc.queueStream(q3, false)
    val ds4 = ssc.queueStream(q4, false)
    val ds5 = ssc.queueStream(q5, false)

    //ds1.print    // 기본적으로 각 RDD의 맨 앞쪽 10개의 요소를 출력,
print(20)과 같이 출력할 요소의 개수를  직접 지정해서 변경 할 수 있다.
    //ds1.map((_, 1)).print // RDD 각 요소를 튜플로 반환, RDD의 map()연산과
동일
    //ds2.flatMap(_.split(", ")).print
    //ds2.map(_.split(", ")).print // [Ljava.lang.String;@3880fb42
<= 왜 이렇게 나올까? map과 flatMap의 차이... flatMap은 여러개의 인자를 반환,
map 하나만 반환
    //ds1.count.print // DStream에 포함된 모든 요소의 개수를 반환, 리턴 되는
값의 타입 Long이 아닌 DStream
    //ds1.countByValue().print // key값 기준으로 count
    //ds1.reduce(_ + "," +_).print // DStream에 포함된 RDD값들을 집계해서
최종적으로 하나의 값으로 변환
```

```
    //ds1.map((_, 1L)).reduceByKey(_+_).print // 튜플 타입시에는
reduceByKey 사용
    //ds1.filter(_!="c").print//DStream 모든 요소에 func함수를 적용하고 그
결과가 true인 요소만을 포함한 새로운 DStream을 반환
    //ds1.union(ds2).print // 두개의 DStream의 요소를 모두 포함한 새로운
DStream을 생성
    //ds3.join(ds4).print // 키와 값 쌍으로 구성된 두개의 DStream을 키를 이용
해 join
    //ds3.leftOuterJoin(ds4).print
    //ds3.rightOuterJoin(ds4).print
    //ds3.fullOuterJoin(ds4).print
    */


    //6.4절
    /* 6.4..1 ~ 6.4.2
    import scala.collection.mutable

    val conf = new
SparkConf().setMaster("local[3]").setAppName("StreamingOps")
    val ssc = new StreamingContext(conf, Seconds(3))


    val rdd1 = ssc.sparkContext.parallelize(List("a", "b", "c", "c",
"c"))
    val rdd2 = ssc.sparkContext.parallelize(List("1,2,3,4,5"))
    val rdd3 = ssc.sparkContext.parallelize(List(("k1", "r1"),
("k2", "r2"), ("k3", "r3")))
    val rdd4 = ssc.sparkContext.parallelize(List(("k1", "s1"),
("k2", "s2")))
    val rdd5 = ssc.sparkContext.range(1, 6)

    val q1 = mutable.Queue(rdd1)
    val q2 = mutable.Queue(rdd2)
    val q3 = mutable.Queue(rdd3)
    val q4 = mutable.Queue(rdd4)
    val q5 = mutable.Queue(rdd5)

    val ds1 = ssc.queueStream(q1, false)
    val ds2 = ssc.queueStream(q2, false)
    val ds3 = ssc.queueStream(q3, false)
    val ds4 = ssc.queueStream(q4, false)
    val ds5 = ssc.queueStream(q5, false)

    val other = ssc.sparkContext.range(1, 3)

    //ds5.transform(_ subtract other).print //transform(func)을 사용하
면 RDD 클래스 타입에서만 제공되던 메서드도 사용 할 수 있다.

    val t1 = ssc.sparkContext.parallelize(List("a", "b", "c"))
    val t2 = ssc.sparkContext.parallelize(List("b", "c"))
    val t3 = ssc.sparkContext.parallelize(List("a", "a", "a"))
    val q6 = mutable.Queue(t1, t2, t3)
```

```
    val ds6 = ssc.queueStream(q6, true) //queueStream(, true):배치가 수행될 때마다 한 개씩의 RDD를 읽어 들이도록 설정
                                                //t1을 읽고, 3초 뒤에 t2를 읽고, 3초 뒤에 t3를 읽는다.
    ssc.checkpoint(".") //checkpoint는 현재의 작업 상태를 HDFS와 같은 영속성을 가진 저장소에 저장해 놓는 메소드
                              //queueStream은 checkpoint를 지원하지 않음

    val updateFunc = (newValues : Seq[Long], currentValue: Option[Long]) // 상태 업데이트를 위한 함수 정의
                         => Option(currentValue.getOrElse(0L) + newValues.sum)

    ds6.map((_, 1L)).updateStateByKey(updateFunc).print // PairDStreamFuunctions 변환 후 updateStatebyKey()메서드 호출
    ds6.saveAsTextFiles("src/main/output/test", "txt")
    */


    /*6.4.3
    import scala.collection.mutable
    val conf = new SparkConf().setMaster("local[*]").setAppName("WindowSample")
    val sc = new SparkContext(conf);
    val ssc = new StreamingContext(sc, Seconds(1))

    ssc.checkpoint(".")

    val input = for (i <- mutable.Queue(1 to 100: _*)) yield sc.parallelize(i :: Nil) //yield vector 타입으로 반환, :: => New List를 생성한다. 왜 Nil이 필요?
    val ds = ssc.queueStream(input)

    //ds.window(Seconds(3), Seconds(2)).print //매 2초마다 최근 3초간의 데이터를 읽어와서 새로운 DStream을 생성
    //ds.countByWindow(Seconds(3), Seconds(2)).print //윈도우에 포함된 요소의 개수를 포함한 DStream을 생성
    //ds.reduceByWindow((a, b) => Math.max(a, b), Seconds(3), Seconds(2)).print //윈도우에 포함된 요소에 reduce()함수를 적용한 결과로 구성된 DStream을 생성
    //ds.map(v => (v % 2, 1)).reduceByKeyAndWindow((a: Int, b: Int) => a + b, Seconds(4), Seconds(2)).print
    //주어진 값을 홀수/짝수로 나누고 reduce()함수를 수행

    val invFnc = (v1 : Int, v2 : Int) => {
      v1 -v2 //v1:직전 총합, v2 : 제외된 값의 총합
    }

    val reduceFnc = (v1 : Int, v2 : Int) => {
      v1 + v2
    }

    //ds.map(v => ("sum", v)).reduceByKeyAndWindow(reduceFnc, invFnc, Seconds(3), Seconds(2)).print
```

```
    //효율적인 reduce()연산을 수행하기 위함, reduceFnc는 리듀스 연산을 수행할 리듀
스 함수, invFnc는 역리듀스 함수

    //ds.countByValueAndWindow(Seconds(3), Seconds(2)).print //윈도우
내에 포함된 요소들을 값을 기준으로 각 값에 해당하는 요소의 개수를 포함하는 새로운
DStream을 생성
    //ds.saveAsTextFiles("src/main/output/test", "txt")
    */



    /*6.5
    import scala.collection.mutable
    val conf = new
SparkConf().setMaster("local[*]").setAppName("DataFrameSample")
    val ssc = new StreamingContext(conf, Seconds(1))
    val sc = ssc.sparkContext

    val rdd1 = sc.parallelize(Person("P1", 20) :: Nil)
    val rdd2 = sc.parallelize(Person("P2", 10) :: Nil)
    val queue = mutable.Queue(rdd1, rdd2)
    val ds = ssc.queueStream(queue)

    ds.foreachRDD(rdd => {
      val spark =
SparkSession.builder.config(sc.getConf).getOrCreate()
      import spark.implicits._
      val df = spark.createDataFrame(rdd)
      df.select("name", "age").show
    })
    */



    //ssc.start()
    //ssc.awaitTermination()
  }
}
```

==========

6장_2번째

```
package lab.streaming.exercise.ch6

import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.{SparkConf, SparkContext}
import kafka.serializer.StringDecoder
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.streaming.{Milliseconds, Seconds,
StreamingContext}
import org.apache.spark.sql.SparkSession
import scala.collection.mutable
```

```
//netcat 실행시켜서(nc -lk 9999) spark application submit 시켜서
checkpoint 저장되는거 확인하면 된다.

object Exercise2 {

    def updateFnc(newValues: Seq[Int], currentValue: Option[Int]):
Option[Int] = {
        Option(currentValue.getOrElse(0) + newValues.sum)
      }

    def createSSC(checkpointDir: String) = {
        //ssc 생성
        val conf = new
SparkConf().setMaster("local[3]").setAppName("CheckPointSample")
        val sc = new SparkContext(conf);
        val ssc = new StreamingContext(sc, Milliseconds(3000))
        //checkpoint
        ssc.checkpoint(checkpointDir)
        //DStream 생성
        val ids1 = ssc.socketTextStream("localhost", 9999)
        val ids2 = ids1.flatMap(_.split(" ")).map((_, 1))
        //updateStateByKey
        ids2.updateStateByKey(updateFnc).print
        //return
        ssc
      }

      def main(args: Array[String]) {
        val checkpointDir = "/home/jazzim/apps/spark/checkpoint"
        val ssc = StreamingContext.getOrCreate(checkpointDir, () =>
createSSC(checkpointDir))
        ssc.start()
        ssc.awaitTermination()
      }

    }
```