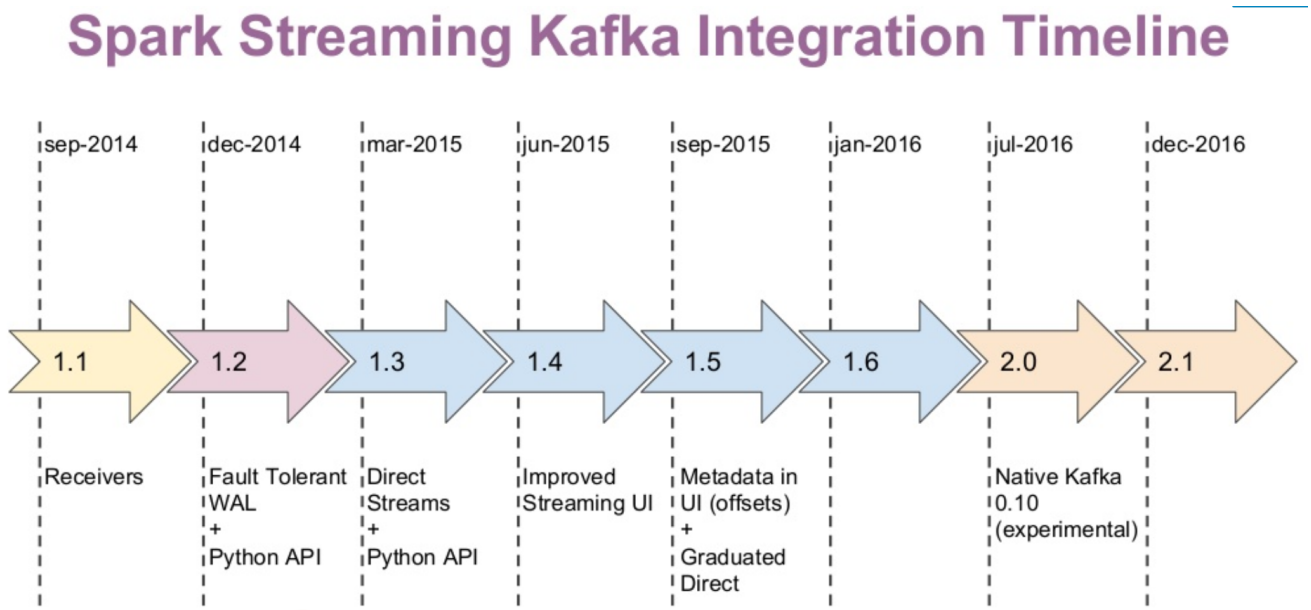
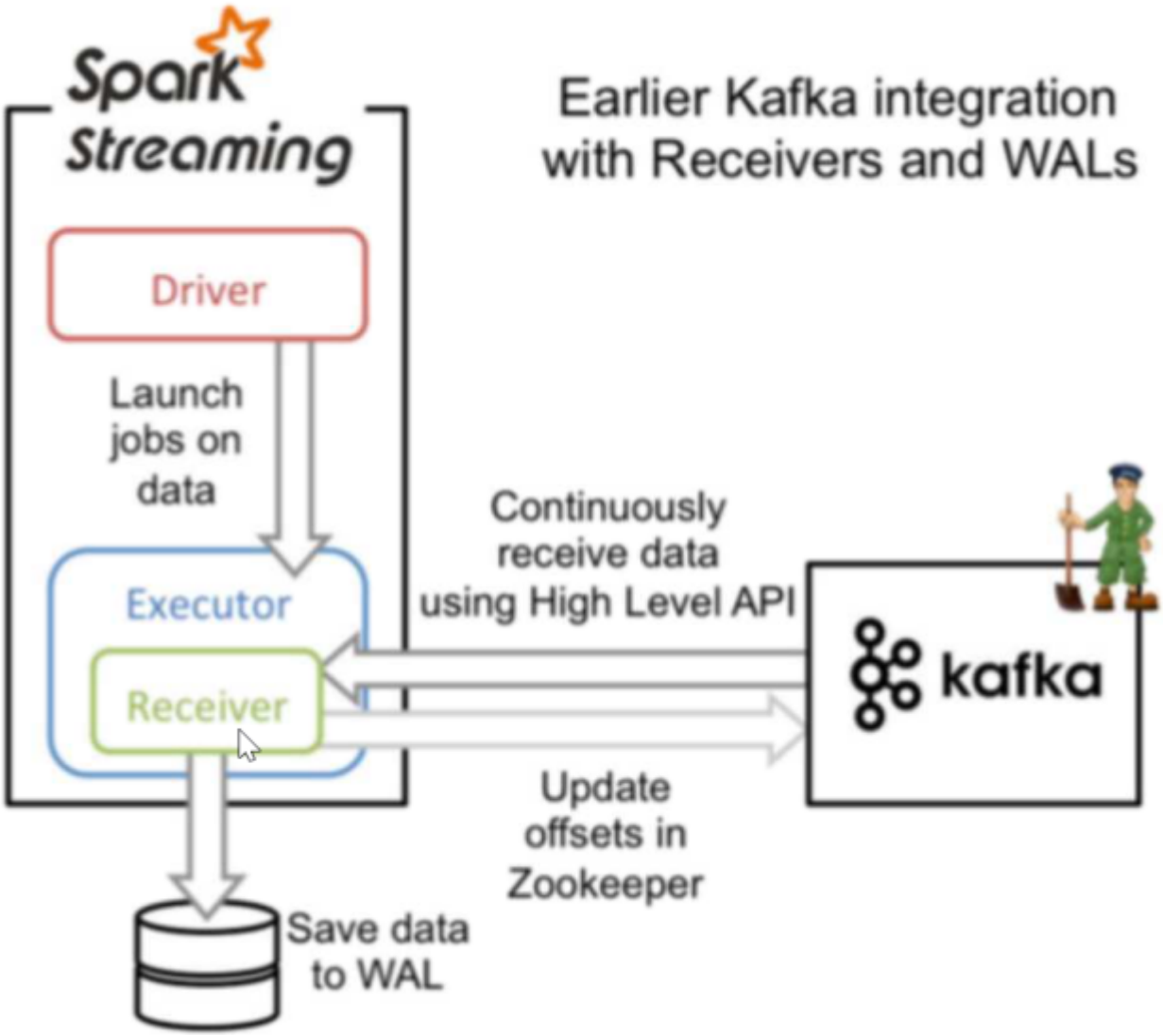
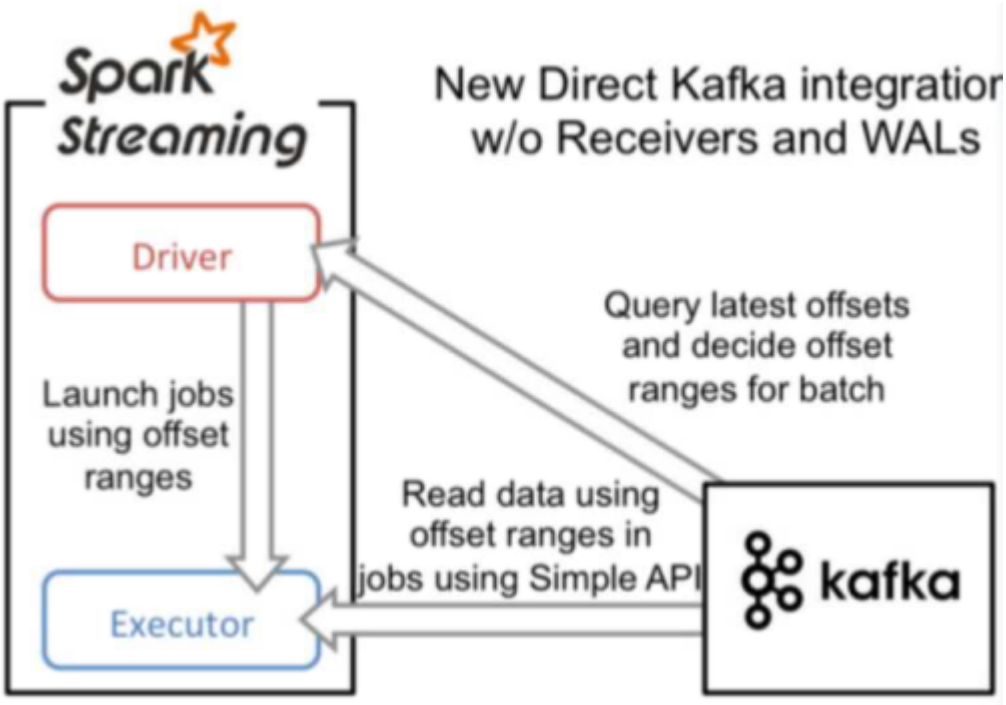
SparkStreaming和kafka的整合








```scala
package sparkStreaming02

import kafka.common.TopicAndPartition
import kafka.message.MessageAndMetadata
import kafka.serializer.StringDecoder
import kafka.utils.{ZKGroupTopicDirs, ZkUtils}
import org.I0Itec.zkclient.ZkClient
import org.apache.spark.SparkConf
import org.apache.spark.streaming.dstream.{DStream, InputDStream}
import org.apache.spark.streaming.kafka.{HasOffsetRanges, KafkaUtils, OffsetRange}
import org.apache.spark.streaming.{Duration, StreamingContext}


object KafkaDirectDemo1 {

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

    //指定组名
    val group = "g001"
    //创建SparkConf
    val conf = new SparkConf().setAppName("KafkaDirectWC").setMaster("local[2]")
    //创建SparkStreaming，并设置间隔时间
    val ssc = new StreamingContext(conf, Duration(5000))
    //指定消费的 topic 名字
    val topic = "topic"
    //指定kafka的broker地址(sparkStream的Task直连到kafka的分区上，用更加底层的API消费，效率更高)
    val brokerList = "node1:9092,node2:9092,node3:9092"


    //指定zk的地址，后期更新消费的偏移量时使用(以后可以使用Redis、MySQL来记录偏移量)
```

```
val zkQuorum = "node1:2181,node2:2181,node3:2181"
//创建 stream 时使用的 topic 名字集合，SparkStreaming可同时消费多个topic
val topics: Set[String] = Set(topic)

//创建一个 ZKGroupTopicDirs 对象,其实是指定往zk中写入数据的目录，用于保存偏移量
//每一个消费者组可以消费不同topic下的数据,一个topic下 的数据可以被多个消费者组消费
//每一个消费者组对每一个topic的消费进度是不一样,
val topicDirs = new ZKGroupTopicDirs(group, topic)
//获取 zookeeper 中的路径 "/g001/offsets/topic/"
val zkTopicPath = s"${topicDirs.consumerOffsetDir}"
println("zkTopicPath:"+zkTopicPath)

//准备kafka的参数
val kafkaParams = Map(
  "metadata.broker.list" -> brokerList,
  "group.id" -> group,
  //从头开始读取数据,从第一条数据开始消费
  //LargestTimeString,从消费者启动后产生的最新数据开始消费
  "auto.offset.reset" -> kafka.api.OffsetRequest.SmallestTimeString
)

//zookeeper 的host 和 ip，创建一个 client,用于跟新偏移量量的
//是zookeeper的客户端，可以从zk中读取偏移量数据，并更新偏移量
val zkClient = new ZkClient(zkQuorum)

//查询该路径下是否字节点（默认有字节点为我们自己保存不同 partition 时生成的）
// /g001/offsets/topic/0/10001"
// /g001/offsets/topic/1/30001"
// /g001/offsets/topic/2/10001"
//zkTopicPath  -> /g001/offsets/topic/
//从返回结果我们可以判断之前是否已经维护过偏移量
val children = zkClient.countChildren(zkTopicPath)

var kafkaStream: InputDStream[(String, String)] = null

//如果 zookeeper 中有保存 offset，我们会利用这个 offset 作为 kafkaStream 的起始位置
var fromOffsets: Map[TopicAndPartition, Long] = Map()

//如果保存过 offset
if (children > 0) {
  for (i <- 0 until children) {
    // /g001/offsets/wordcount/0/10001

    // /g001/offsets/wordcount/0
    val partitionOffset = zkClient.readData[String](s"$zkTopicPath/${i}")
    // wordcount/0
    val tp = TopicAndPartition(topic, i)
    //将不同 partition 对应的 offset 增加到 fromOffsets 中
    // wordcount/0 -> 10001
    //fromOffsets记录了每一个topic下每一个分区,对应的偏移量,记录的是上次消费的位置
    fromOffsets += (tp -> partitionOffset.toLong)
  }

  //Key: kafka的key   values: "hello tom hello jerry"
```

```
      //这个会将 kafka 的消息进行 transform，最终 kafak 的数据都会变成 (kafka的key, message) 这样的tuple
      val messageHandler = (mmd: MessageAndMetadata[String, String]) => (mmd.key(), mmd.message())

      //通过KafkaUtils创建直连的DStream（fromOffsets参数的作用是:按照前面计算好了的偏移量继续消费数据）
      //[String, String, StringDecoder, StringDecoder,     (String, String)]
      //  key    value    key的解码方式   value的解码方式
      kafkaStream = KafkaUtils.createDirectStream[String, String, StringDecoder, StringDecoder, (String, String)](ssc, kafkaParams, fromOffsets, messageHandler)
    } else {
      //如果未保存，根据 kafkaParam 的配置使用最新(largest)或者最旧的（smallest） offset
      kafkaStream = KafkaUtils.createDirectStream[String, String, StringDecoder, StringDecoder](ssc, kafkaParams, topics)
    }

    //偏移量的范围
    var offsetRanges = Array[OffsetRange]()

    //从kafka读取的消息，DStream的Transform方法可以将当前批次的RDD获取出来
    //该transform方法计算获取到当前批次RDD,然后将RDD的偏移量取出来，然后在将RDD返回到DStream
    val transform: DStream[(String, String)] = kafkaStream.transform { rdd =>
      //得到该 rdd 对应 kafka 的消息的 offset
      //该RDD是一个KafkaRDD，可以获得偏移量的范围
      offsetRanges = rdd.asInstanceOf[HasOffsetRanges].offsetRanges
      rdd
    }

    val messages: DStream[String] = transform.map(_._2)

    //依次迭代DStream中的RDD
    messages.foreachRDD { rdd =>
      //对RDD进行操作，触发Action
      rdd.foreachPartition(partition =>
        partition.foreach(x => {
          println(x)
        })
      )

      for (o <- offsetRanges) {
        //  /g001/offsets/topic/0
        val zkPath = s"${topicDirs.consumerOffsetDir}/${o.partition}"
        //将该 partition 的 offset 保存到 zookeeper
        //  /g001/offsets/topic/0/20000
        ZkUtils.updatePersistentPath(zkClient, zkPath, o.untilOffset.toString)
      }
    }

    ssc.start()
    ssc.awaitTermination()


  }
```

```
}
```

```
package sparkday12

import java.lang
import org.apache.kafka.clients.consumer.ConsumerRecord
import org.apache.kafka.common.TopicPartition
import org.apache.kafka.common.serialization.StringDeserializer
import org.apache.log4j.{Level, Logger}
import org.apache.spark.SparkConf
import org.apache.spark.streaming.dstream.InputDStream
import org.apache.spark.streaming.kafka010._
import org.apache.spark.streaming.{Seconds, StreamingContext}
import redis.clients.jedis.Jedis

object KafkaDirectDemo3 {
  // 过滤日志
  Logger.getLogger("org").setLevel(Level.WARN)
  def main(args: Array[String]): Unit = {
    val conf = new SparkConf().setMaster("local[2]").setAppName("xx")
      //每秒钟每个分区kafka拉取消息的速率
      .set("spark.streaming.kafka.maxRatePerPartition", "100")
      // 序列化
      .set("spark.serilizer", "org.apache.spark.serializer.KryoSerializer")
    val ssc = new StreamingContext(conf, Seconds(3))
    //启动一参数设置
    val groupId = "ts001"
    // kafka配置参数
    val kafkaParams = Map[String, Object](
      "bootstrap.servers" -> "node1:9092,node2:9092,node3:9092",
      // kafka的key和value的解码方式
      "key.deserializer" -> classOf[StringDeserializer],
      "value.deserializer" -> classOf[StringDeserializer],
      "group.id" -> groupId,
      // 从头开始消费
      "auto.offset.reset" -> "earliest",
      "enable.auto.commit" -> (false: lang.Boolean)
    )
    val topics = Array("ts")
    //启动二参数设置  （获取Redis中的kafka偏移量）
    var formdbOffset: Map[TopicPartition, Long] = JedisOffset(groupId)
    //拉取kafka数据
    val stream: InputDStream[ConsumerRecord[String, String]] =
    // 首先判断一下 我们要消费的kafka数据是否是第一次消费，之前有没有消费过
      if (formdbOffset.size == 0) {
        KafkaUtils.createDirectStream[String, String](
          ssc,
          /**
            * 本地策略

            * 一般使用LocationStrategies的PreferConsistent方法。
```

```
        它会将分区数据尽可能均匀地分配给所有可用的Executor。
        题外话：本地化策略看到这里就行了，下面讲的是一些特殊情况。
        情况一
        如果你的Executor和kafka broker在同一台机器上，可以用PreferBrokers，
        这将优先将分区调度到kafka分区leader所在的主机上。
        题外话：废话，Executor是随机分布的，我怎么知道是不是在同一台服务器上？
        除非是单机版的are you明白？
        情况二
        分区之间的负荷有明显的倾斜，可以用PreferFixed。
        这个允许你指定一个明确的分区到主机的映射（没有指定的分区将会使用连续的地址）。
        题外话：就是出现了数据倾斜了呗
            */
          LocationStrategies.PreferConsistent,
          /**
            * 消费者策略
            * ConsumerStrategies.Subscribe，能够订阅一个固定的topics的集合。
            * SubscribePattern 能够
            * 根据你感兴趣的topics进行匹配。需要注意的是，不同于 0.8的集成，
            * 使用subscribe or SubscribePattern 可以支持在运行的streaming中增加分区。
            * 而Assign不可以动态的改变消费的分区模式，那么一般都会在开始读取固定的数据时候才能使用
            */
          ConsumerStrategies.Subscribe[String, String](topics, kafkaParams)
        )
      } else {
        // 第一次消费数据，没有任何的消费信息数据
        KafkaUtils.createDirectStream(
          ssc,
          LocationStrategies.PreferConsistent,
          ConsumerStrategies.Assign[String, String](
            formdbOffset.keys, kafkaParams, formdbOffset)
        )
      }
    //数据偏移量处理。
    stream.foreachRDD({
      rdd =>
        // 获得偏移量对象数组
        val offsetRange: Array[OffsetRange] =
          rdd.asInstanceOf[HasOffsetRanges].offsetRanges
        //逻辑处理
        rdd.map(_.value())
          .map((_, 1))
          .reduceByKey(_ + _).foreach(println)
        // 偏移量存入redis
        val jedis: Jedis = JedisConnectionPool.getConnection()
        for (or <- offsetRange) {
          jedis.hset(groupId, or.topic + "-" + or.partition, or.untilOffset.toString)
        }
        jedis.close()
    })
    // 启动Streaming程序
    ssc.start()
    ssc.awaitTermination()

}
```

```
}
```