[](http://spark.apache.org/docs/latest/index.html)**1.6.1**

* [Overview](http://spark.apache.org/docs/latest/index.html)
* [Programming Guides](http://spark.apache.org/docs/latest/streaming-kafka-integration.html)
* [API Docs](http://spark.apache.org/docs/latest/streaming-kafka-integration.html)
* [Deploying](http://spark.apache.org/docs/latest/streaming-kafka-integration.html)
* [More](http://spark.apache.org/docs/latest/api.html)

**Spark Streaming + Kafka Integration Guide**

[Apache Kafka](http://kafka.apache.org/) is publish-subscribe messaging rethought（重新考虑） as a distributed, partitioned, replicated commit log service. Here we explain how to configure Spark Streaming to receive data from Kafka. There are two approaches to this - the old approach using Receivers and Kafka’s high-level API, and a new experimental approach (introduced in Spark 1.3) without using Receivers. They have different programming models, performance characteristics, and semantics guarantees, so read on for more details.

**Approach 1: Receiver-based Approach**

This approach uses a Receiver to receive the data. The Receiver is implemented using the Kafka high-level consumer API. As with all receivers, the data received from Kafka through a Receiver is stored in Spark executors, and then jobs launched by Spark Streaming processes the data.

However, under default configuration, this approach can lose data under failures (see [receiver reliability](http://spark.apache.org/docs/latest/streaming-programming-guide.html#receiver-reliability). To ensure zero-data loss, you have to additionally enable Write Ahead Logs in Spark Streaming (introduced in Spark 1.2). This synchronously saves all the received Kafka data into write ahead logs on a distributed file system (e.g HDFS), so that all the data can be recovered on failure. See [Deploying section](http://spark.apache.org/docs/latest/streaming-programming-guide.html#deploying-applications) in the streaming programming guide for more details on Write Ahead Logs.

Next, we discuss how to use this approach in your streaming application.

1. **Linking:** For Scala/Java applications using SBT/Maven project definitions, link your streaming application with the following artifact (see [Linking section](http://spark.apache.org/docs/latest/streaming-programming-guide.html#linking) in the main programming guide for further information).
2. groupId = org.apache.spark
3. artifactId = spark-streaming-kafka\_2.10
4. version = 1.6.1

For Python applications, you will have to add this above library and its dependencies when deploying your application. See the *Deploying*subsection below.

1. **Programming:** In the streaming application code, import KafkaUtils and create an input DStream as follows.
   * [**Scala**](http://spark.apache.org/docs/latest/streaming-kafka-integration.html#tab_scala_0)
   * [**Java**](http://spark.apache.org/docs/latest/streaming-kafka-integration.html#tab_java_0)
   * [**Python**](http://spark.apache.org/docs/latest/streaming-kafka-integration.html#tab_python_0)
2. import org.apache.spark.streaming.kafka.\_
3. val kafkaStream = KafkaUtils.createStream(streamingContext,
4. [ZK quorum], [consumer group id], [per-topic number of Kafka partitions to consume])

You can also specify the key and value classes and their corresponding decoder classes using variations of createStream. See the [API docs](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.streaming.kafka.KafkaUtils$)and the [example](https://github.com/apache/spark/blob/master/examples/src/main/scala/org/apache/spark/examples/streaming/KafkaWordCount.scala).

**Points to remember:**

* + Topic partitions in Kafka does not correlate to partitions of RDDs generated in Spark Streaming. So increasing the number of topic-specific partitions in the KafkaUtils.createStream() only increases the number of threads using which topics that are consumed within a single receiver. It does not increase the parallelism of Spark in processing the data. Refer to the main document for more information on that.
  + Multiple Kafka input DStreams can be created with different groups and topics for parallel receiving of data using multiple receivers.
  + If you have enabled Write Ahead Logs with a replicated file system like HDFS, the received data is already being replicated in the log. Hence, the storage level in storage level for the input stream to StorageLevel.MEMORY\_AND\_DISK\_SER (that is, useKafkaUtils.createStream(..., StorageLevel.MEMORY\_AND\_DISK\_SER)).

1. **Deploying:** As with any Spark applications, spark-submit is used to launch your application. However, the details are slightly different for Scala/Java applications and Python applications.

For Scala and Java applications, if you are using SBT or Maven for project management, then package spark-streaming-kafka\_2.10 and its dependencies into the application JAR. Make sure spark-core\_2.10 and spark-streaming\_2.10 are marked as provided dependencies as those are already present in a Spark installation. Then use spark-submit to launch your application (see [Deploying section](http://spark.apache.org/docs/latest/streaming-programming-guide.html#deploying-applications) in the main programming guide).

For Python applications which lack SBT/Maven project management, spark-streaming-kafka\_2.10 and its dependencies can be directly added to spark-submit using --packages (see [Application Submission Guide](http://spark.apache.org/docs/latest/submitting-applications.html)). That is,

./bin/spark-submit --packages org.apache.spark:spark-streaming-kafka\_2.10:1.6.1 ...

Alternatively, you can also download the JAR of the Maven artifact spark-streaming-kafka-assembly from the [Maven repository](http://search.maven.org/#search|ga|1|a%3A%22spark-streaming-kafka-assembly_2.10%22%20AND%20v%3A%221.6.1%22) and add it tospark-submit with --jars.

**Approach 2: Direct Approach (No Receivers)**

This new receiver-less “direct” approach has been introduced in Spark 1.3 to ensure stronger end-to-end guarantees. Instead of using receivers to receive data, this approach periodically queries Kafka for the latest offsets in each topic+partition, and accordingly defines the offset ranges to process in each batch. When the jobs to process the data are launched, Kafka’s simple consumer API is used to read the defined ranges of offsets from Kafka (similar to read files from a file system). Note that this is an experimental feature introduced in Spark 1.3 for the Scala and Java API, in Spark 1.4 for the Python API.

This approach has the following advantages over the receiver-based approach (i.e. Approach 1).

* *Simplified Parallelism:* No need to create multiple input Kafka streams and union them. With directStream, Spark Streaming will create as many RDD partitions as there are Kafka partitions to consume, which will all read data from Kafka in parallel. So there is a one-to-one mapping between Kafka and RDD partitions, which is easier to understand and tune.

简化并行：不再需要创建多个kafka的输入流，然后再将其合并。用directStream，spark Streaming讲根据kafka要消费的分区来创建对应的RDD，所有的数据都是并行的葱kafka中读取。简单来讲就是RDD和kafka的分区做了一一对应。

* *Efficiency:* Achieving zero-data loss in the first approach required the data to be stored in a Write Ahead Log, which further replicated the data. This is actually inefficient as the data effectively gets replicated twice - once by Kafka, and a second time by the Write Ahead Log. This second approach eliminates the problem as there is no receiver, and hence no need for Write Ahead Logs. As long as you have sufficient Kafka retention, messages can be recovered from Kafka.

 高效：实现零数据丢失，在spark1.2中，需要讲数据存储在一个预写日志（Write Ahead Log），实际上是数据做了二次复制 - 一次通过kafka复制到spark，第二被预写到hdfs存储中，效率是很低下的。spark1.3提供的方法消除了该问题，因为没有接收器，因此没有必要预写日志。

* *Exactly-once semantics:* The first approach uses Kafka’s high level API to store consumed offsets in Zookeeper. This is traditionally the way to consume data from Kafka. While this approach (in combination with write ahead logs) can ensure zero data loss (i.e. at-least once semantics), there is a small chance some records may get consumed twice under some failures. This occurs because of inconsistencies between data reliably received by Spark Streaming and offsets tracked by Zookeeper. Hence, in this second approach, we use simple Kafka API that does not use Zookeeper. Offsets are tracked by Spark Streaming within its checkpoints. This eliminates inconsistencies between Spark Streaming and Zookeeper/Kafka, and so each record is received by Spark Streaming effectively exactly once despite failures. In order to achieve exactly-once semantics for output of your results, your output operation that saves the data to an external data store must be either idempotent, or an atomic transaction that saves results and offsets (see [Semantics of output operations](http://spark.apache.org/docs/latest/streaming-programming-guide.html#semantics-of-output-operations) in the main programming guide for further information).

消费一次原则（语义）:在spark1.2之前的版本中，spark steaming通过kafka的高级消费API来读取数据，并且将偏移量保存在zookeeper中。 这是传统上从kafka消费数据的方式。虽然这种方法（结合Write Ahead Log日志）可以保证零数据丢失（即至少一次语义），但还是在发生故障的情况下有很少的几率会消费两次。这是由于zookeeper保存偏移量和spark steaming接收数据之间的不一致造成的。因此，在spark1.3中，我们使用简单的Kafak API,而不是使用zookeeper（高级api），偏移量由spark streaming通过checkpoints来保存，这消除了spark streaming 和zookeeper/kafka之间的不一致，所以即使发生失败，收到的每条记录由spark streaming 有且只能消费一次。

Note that one disadvantage of this approach is that it does not update offsets in Zookeeper, hence Zookeeper-based Kafka monitoring tools will not show progress. However, you can access the offsets processed by this approach in each batch and update Zookeeper yourself (see below).

Next, we discuss how to use this approach in your streaming application.

1. **Linking:** This approach is supported only in Scala/Java application. Link your SBT/Maven project with the following artifact (see [Linking section](http://spark.apache.org/docs/latest/streaming-programming-guide.html#linking)in the main programming guide for further information).
2. groupId = org.apache.spark
3. artifactId = spark-streaming-kafka\_2.10
4. version = 1.6.1
5. **Programming:** In the streaming application code, import KafkaUtils and create an input DStream as follows.
   * [**Scala**](http://spark.apache.org/docs/latest/streaming-kafka-integration.html#tab_scala_1)
   * [**Java**](http://spark.apache.org/docs/latest/streaming-kafka-integration.html#tab_java_1)
   * [**Python**](http://spark.apache.org/docs/latest/streaming-kafka-integration.html#tab_python_1)
6. import org.apache.spark.streaming.kafka.\_
7. val directKafkaStream = KafkaUtils.createDirectStream[
8. [key class], [value class], [key decoder class], [value decoder class] ](
9. streamingContext, [map of Kafka parameters], [set of topics to consume])

You can also pass a messageHandler to createDirectStream to access MessageAndMetadata that contains metadata about the current message and transform it to any desired type. See the [API docs](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.streaming.kafka.KafkaUtils$) and the [example](https://github.com/apache/spark/blob/master/examples/src/main/scala/org/apache/spark/examples/streaming/DirectKafkaWordCount.scala).

In the Kafka parameters, you must specify either metadata.broker.list or bootstrap.servers. By default, it will start consuming from the latest offset of each Kafka partition. If you set configuration auto.offset.reset in Kafka parameters to smallest, then it will start consuming from the smallest offset.

You can also start consuming from any arbitrary offset using other variations of KafkaUtils.createDirectStream. Furthermore, if you want to access the Kafka offsets consumed in each batch, you can do the following.

* + [**Scala**](http://spark.apache.org/docs/latest/streaming-kafka-integration.html#tab_scala_2)
  + [**Java**](http://spark.apache.org/docs/latest/streaming-kafka-integration.html#tab_java_2)
  + [**Python**](http://spark.apache.org/docs/latest/streaming-kafka-integration.html#tab_python_2)

// Hold a reference to the current offset ranges, so it can be used downstream

var offsetRanges = Array[OffsetRange]()

directKafkaStream.transform { rdd =>

offsetRanges = rdd.asInstanceOf[HasOffsetRanges].offsetRanges

rdd

}.map {

...

}.foreachRDD { rdd =>

for (o <- offsetRanges) {

println(s"${o.topic} ${o.partition} ${o.fromOffset} ${o.untilOffset}")

}

...

}

You can use this to update Zookeeper yourself if you want Zookeeper-based Kafka monitoring tools to show progress of the streaming application.

Note that the typecast to HasOffsetRanges will only succeed if it is done in the first method called on the directKafkaStream, not later down a chain of methods. You can use transform() instead of foreachRDD() as your first method call in order to access offsets, then call further Spark methods. However, be aware that the one-to-one mapping between RDD partition and Kafka partition does not remain after any methods that shuffle or repartition, e.g. reduceByKey() or window().

Another thing to note is that since this approach does not use Receivers, the standard receiver-related (that is, [configurations](http://spark.apache.org/docs/latest/configuration.html) of the formspark.streaming.receiver.\* ) will not apply to the input DStreams created by this approach (will apply to other input DStreams though). Instead, use the [configurations](http://spark.apache.org/docs/latest/configuration.html) spark.streaming.kafka.\*. An important one is spark.streaming.kafka.maxRatePerPartition which is the maximum rate (in messages per second) at which each Kafka partition will be read by this direct API.

1. **Deploying:** This is same as the first approach.

# End-to-end semantics

This stream ensures that every records is effectively received and

transformed exactly once, but gives no guarantees on whether the transformed data are

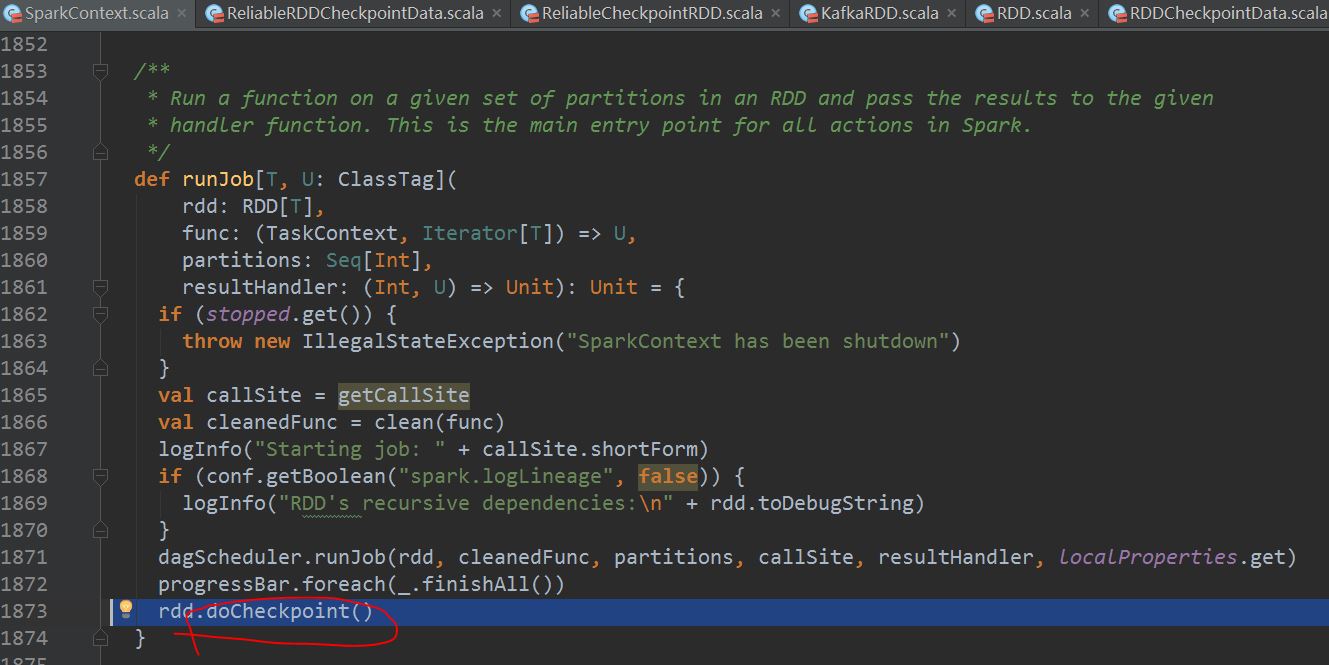
outputted exactly once. For end-to-end exactly-once semantics, you have to either ensure

that the output operation is idempotent, or use transactions to output records atomically.

See the programming guide for more details.

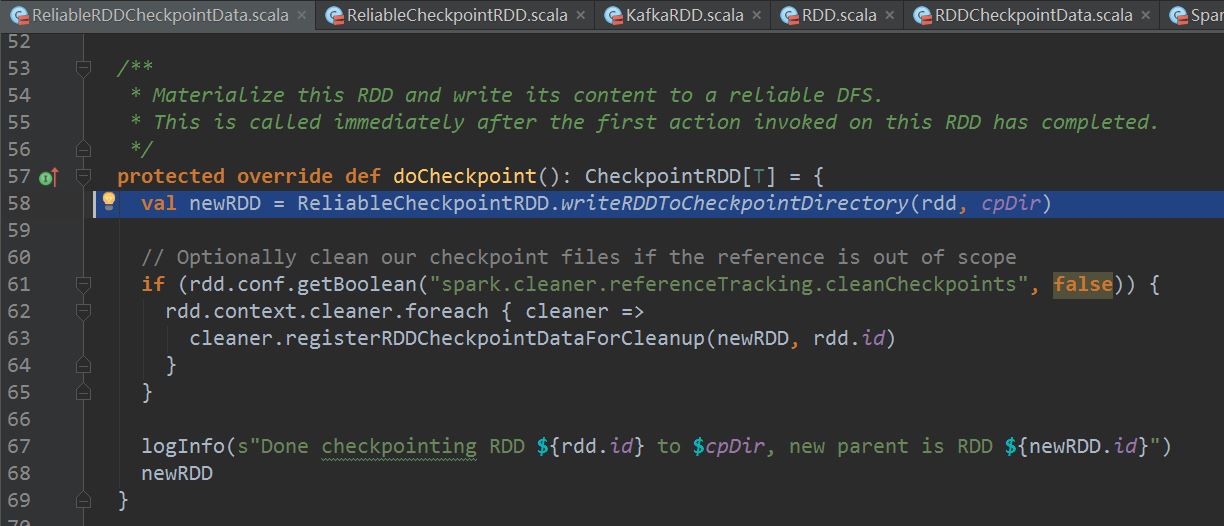
# doCheckPoint

Action执行完成以后立即执行checkpoint操作



org.apache.spark.rdd.ReliableRDDCheckpointData#doCheckpoint

执行checkpoint时要先将原先的RDD物化到可靠的存储，然后在生成CheckPointRDD



org.apache.spark.rdd.ReliableCheckpointRDD$#writeRDDToCheckpointDirectory

将原先的RDD最终结果写入到可靠的存储，然后在生成并返回ReliableCheckpointRDD

