# sparkstreaming的使用场景

实时数据生产源 =>消息中间键(kafka) =>处理

# kafka的作用

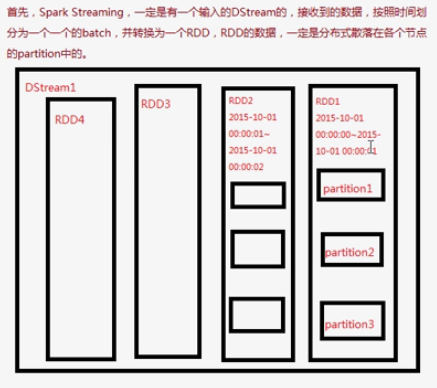
Kafka是作为实时数据处理时的一种缓冲

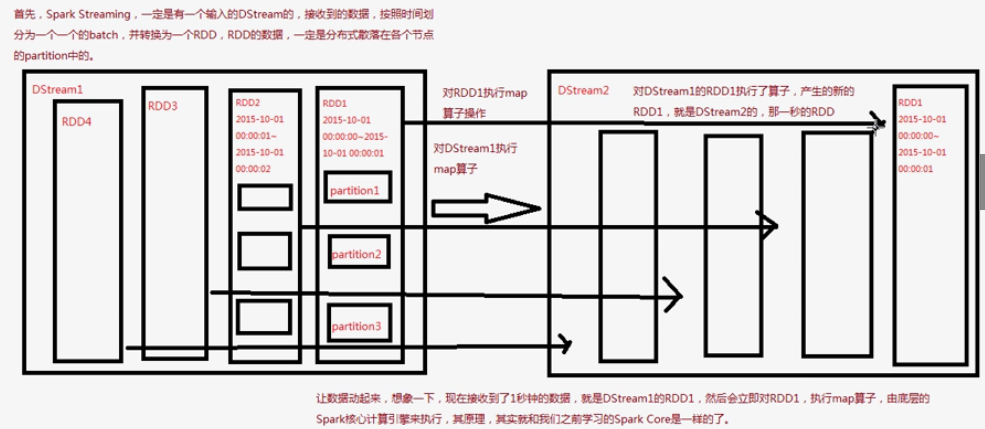
# TCPsocket flume Kafka

流处理的常用操作：map、reduce、join、window

处理后的结果可以写道database dashboard hdfshbase

核心的处理对象Dstream，是discretized stream的简称，是batch of RDD构成的：





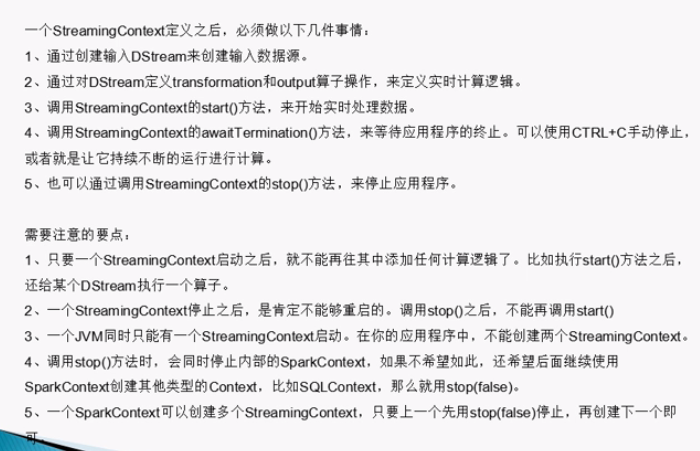
# Spark和storm的对比



Storm适用于精准、延迟低的，计算资源相对不足，有些时段高峰明显，需要调节并行度的。仅仅是流处理，不会涉及到SQL交互查询、transformation

Streaming 高吞吐量，并且可能会和一些交互查询、机器学习等结合。

# Spark streaming context



代码实操

Scm:stream的context

Scm.socketTextStream(“地址”, 端口) // 能够获取socket数据

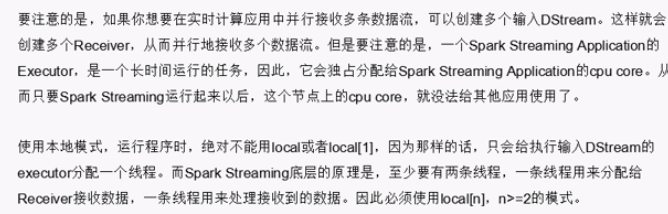
# Spark streaming的DStream和Receiver以及县城之间的关系

Dstream是从其他服务（比如nc，socket）等接受数据，除了文件数据源外，所有的数据源都绑定一个receiver。

支持的数据源：

1. 基本数据源：文本、socket、akka
2. 高级数据源：
3. 自定义数据源：

一个应用会独占executor，一个ssc应用至少分配给他receiver数目 + 1个core



# KafkaUtil提供了对kafka发送的数据进行接受的API。

package cn.spark.study.streaming;

import java.util.Arrays;

import java.util.HashMap;

import java.util.Map;

import org.apache.spark.SparkConf;

import org.apache.spark.api.java.function.FlatMapFunction;

import org.apache.spark.api.java.function.Function2;

import org.apache.spark.api.java.function.PairFunction;

import org.apache.spark.streaming.Durations;

import org.apache.spark.streaming.api.java.JavaDStream;

import org.apache.spark.streaming.api.java.JavaPairDStream;

import org.apache.spark.streaming.api.java.JavaPairReceiverInputDStream;

import org.apache.spark.streaming.api.java.JavaStreamingContext;

import org.apache.spark.streaming.kafka.KafkaUtils;

import scala.Tuple2;

/\*\*

\* 基于Kafka receiver方式的实时wordcount程序

\* @author Administrator

\*

\*/

public class KafkaReceiverWordCount {

public static void main(String[] args) {

SparkConf conf = new SparkConf()

.setMaster("local[2]")

.setAppName("KafkaWordCount");

JavaStreamingContext jssc = new JavaStreamingContext(conf, Durations.seconds(5));

// 使用KafkaUtils.createStream()方法，创建针对Kafka的输入数据流

Map<String, Integer> topicThreadMap = new HashMap<String, Integer>();

topicThreadMap.put("WordCount", 1);

JavaPairReceiverInputDStream<String, String> lines = KafkaUtils.createStream(

jssc,

"192.168.1.107:2181,192.168.1.108:2181,192.168.1.109:2181",

"DefaultConsumerGroup",

topicThreadMap);

// 然后开发wordcount逻辑

JavaDStream<String> words = lines.flatMap(

new FlatMapFunction<Tuple2<String,String>, String>() {

private static final long serialVersionUID = 1L;

@Override

public Iterable<String> call(Tuple2<String, String> tuple)

throws Exception {

return Arrays.asList(tuple.\_2.split(" "));

}

});

JavaPairDStream<String, Integer> pairs = words.mapToPair(

new PairFunction<String, String, Integer>() {

private static final long serialVersionUID = 1L;

@Override

public Tuple2<String, Integer> call(String word)

throws Exception {

return new Tuple2<String, Integer>(word, 1);

}

});

JavaPairDStream<String, Integer> wordCounts = pairs.reduceByKey(

new Function2<Integer, Integer, Integer>() {

private static final long serialVersionUID = 1L;

@Override

public Integer call(Integer v1, Integer v2) throws Exception {

return v1 + v2;

}

});

wordCounts.print();

jssc.start();

jssc.awaitTermination();

jssc.close();

}

}

upStateByKey



# transform操作

val finalDStream = searchWordCountsDSteram.transform(searchWordCountsRDD => {

val countSearchWordsRDD = searchWordCountsRDD.map(tuple => (tuple.\_2, tuple.\_1))

val sortedCountSearchWordsRDD = countSearchWordsRDD.sortByKey(false)

val sortedSearchWordCountsRDD = sortedCountSearchWordsRDD.map(tuple => (tuple.\_1, tuple.\_2))

val top3SearchWordCounts = sortedSearchWordCountsRDD.take(3)

for(tuple <- top3SearchWordCounts) {

println(tuple)

}

searchWordCountsRDD

})

finalDStream.print()

# window操作

val searchWordCountsDSteram = searchWordPairsDStream.reduceByKeyAndWindow(

(v1: Int, v2: Int) => v1 + v2,

Seconds(60),

Seconds(10))

# foreachRDD操作

## 通过静态线程池和DStream的foreachRDD来想数据库写入数据



这段代码仍然在ssc.start()前面。

## 通过foreachRDD和SQL输出

categoryProductCountsDStream.foreachRDD(categoryProductCountsRDD => {

val categoryProductCountRowRDD = categoryProductCountsRDD.map(tuple => {

val category = tuple.\_1.split("\_")(0)

val product = tuple.\_1.split("\_")(1)

val count = tuple.\_2

Row(category, product, count)

})

val structType = StructType(Array(

StructField("category", StringType, true),

StructField("product", StringType, true),

StructField("click\_count", IntegerType, true)))

val hiveContext = new HiveContext(categoryProductCountsRDD.context)

val categoryProductCountDF = hiveContext.createDataFrame(categoryProductCountRowRDD, structType)

categoryProductCountDF.registerTempTable("product\_click\_log")

val top3ProductDF = hiveContext.sql(

"SELECT category,product,click\_count "

+ "FROM ("

+ "SELECT "

+ "category,"

+ "product,"

+ "click\_count,"

+ "row\_number() OVER (PARTITION BY category ORDER BY click\_count DESC) rank "

+ "FROM product\_click\_log"

+ ") tmp "

+ "WHERE rank<=3")

top3ProductDF.show()

})

## 试一下

foreachRDD能否放在ssc.start()之后