|  |  |
| --- | --- |
| 产品名称 | 密级 |
|  | 机密 |
| 产品版本 | 共15页 |
|  |

DirectKafkaInputDStream及其数据流分析

|  |  |  |  |
| --- | --- | --- | --- |
| 拟制 | 中软基础平台spark小组 | 日期 | 2016-06-20 |
| 审核 |  | 日期 |  |
| 批准 |  | 日期 |  |



华为技术有限公司

版权所有 侵权必究

修订记录

|  |  |  |  |
| --- | --- | --- | --- |
| 日期 | 修订版本 | 修改描述 | 作者 |
| 2016-06-20 | 1.0 | 初稿 | x00299002 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

目 录

[1 DirectKafkaInputDStream介绍 4](#_Toc456876811)

[1.1 什么是DirectKafkaInputDStream 4](#_Toc456876812)

[1.2 Kafka介绍 6](#_Toc456876813)

[2 数据流分析 10](#_Toc456876814)

[2.1 DirectKafkaInputDStream构造 10](#_Toc456876815)

[2.2 generateJob流程 10](#_Toc456876816)

[2.3 Job执行 12](#_Toc456876817)

[2.4 CheckPoint说明 13](#_Toc456876818)

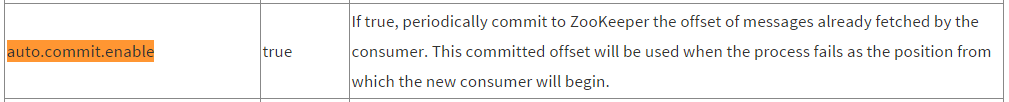
# DirectKafkaInputDStream介绍

## 什么是DirectKafkaInputDStream

SparkStreaming对接kafka数据有两种方式，一种是调用KafkaUtils. createStream接口创建的有receiver的DStream，还有一种就是KafkaUtils. createDirectStream接口创建的无receiver的Dstream。

第一种对应的DStream就是KafkaInputDStream，它的receiver有两种类型，分别为KafkaReceiver和ReliableKafkaReceiver，在创建KafkaInputDStream的时候判断配置选项“spark.streaming.receiver.writeAheadLog.enable”是否为true，是ture即为ReliableKafkaReceiver；

两者区别是KafkaReceiver通过选项“auto.commit.enable”配置是否自动提交offset到ZooKeeper，而ReliableKafkaReceiver会将选项“auto.commit.enable”强制设成false，并且在接收到block数据保存成write-ahead log之后，需要主动提交offset到ZooKeeper。



第二种就是DirectKafkaInputDStream，接口相对而言是low-level，直接获取leader的最新offset并读取topic的各个partition数据进行处理。

强烈建议用户使用DirectKafkaInputDStream的方式，其优点相对其他非常明显，具体如下：

1、内存占用：KafkaInputDStream需要有receiver在不停接收数据并且将数据保存在内存（或者内存不足时写入磁盘），而DirectKafkaInputDStream则在job产生过程中只会去获取offset，只有真正执行job的时候才会调用kafka底层接口去读取数据

2、executor占用：由于KafkaInputDStream需要receiver在不停接收数据，每个receiver都会以job的形式占用一个executor，占用处理资源

3、稳定性：从前两条就可以看出来，DirectKafkaInputDStream明显稳定性好多了；KafkaInputDStream在receiver所在网络环境影响或者内存占用过多、数据量较大情况下，结果会比较糟糕

4、流控：DirectKafkaInputDStream流控更为简单有效。对KafkaInputDStream而言，流控参数只有“spark.streaming.receiver.maxRate”会起作用，即用于receiver接收数据每秒最大条数；DirectKafkaInputDStream则是“spark.streaming.receiver.maxRate”和“spark.streaming.kafka.maxRatePerPartition”两个参数共同计算出message的最大处理条数，其中“spark.streaming.kafka.maxRatePerPartition”可以直接控制每秒所处理的partition的message最大值。

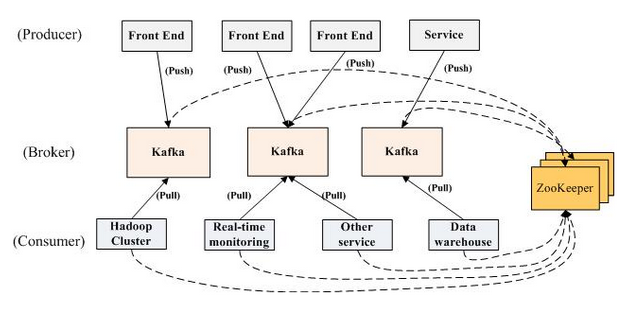
区别：

|  |  |  |  |
| --- | --- | --- | --- |
|  | KafkaInputDStream（KafkaReceiver） | KafkaInputDStream（ReliableKafkaReceiver） | DirectKafkaInputDStream |
| 有无receiver | 有 | 有 | 无 |
| offset记录 | 通过auto.commit.enable选项配置 | 是，接收到block数据保存成write-ahead log之后，更新offset到ZooKeeper | 是，不过是SparkStreaming自己记录，在generateJob之后更新，也就是每个batchTime更新 |
| 读取数据kafka类 | Consumer | Consumer | SimpleConsumer |
| 基类 | ReceiverInputDStream，ReceiverInputDStream继承自InputDStream | ReceiverInputDStream，ReceiverInputDStream继承自InputDStream | InputDStream |
| 每个batchtime生成的rdd | BlockRDD | BlockRDD | KafkaRDD，自己实现的一种rdd，compute方法重写 |
| 效率 | 低，一个receiver对应一个job，需要启动receiver进行接收数据 | 低，一个receiver对应一个job，需要启动receiver进行接收数据 | 高，直接读取topic对应的每个partition的offset，产生job之后直接读取数据进行处理 |
| 内存 | 内存占用较多，需要接收数据 | 内存占用较多，需要接收数据 | job执行才会去读取真正数据 |
| 流控参数 | “spark.streaming.receiver.maxRate”，每秒接收message最大值 | “spark.streaming.receiver.maxRate”， 每秒接收message最大值 | “spark.streaming.receiver.maxRate”和“spark.streaming.kafka.maxRatePerPartition”，分别为每秒所有partition处理message最大值及每秒每个partition的处理message最大值 |
| 流控稳定性 | 默认200ms产生一个block，然后block会放进blockingqueue中，会有堵塞；然后控制每秒的最大接收消息数 | 默认200ms产生一个block，然后block会放进blockingqueue中，会有堵塞；然后控制每秒的最大接收消息数 | 没有block堵塞，在产生job时通过两个流控参数计算每个partition的数据最大处理条数，然后对job产生时partition的offset控制即可，稳定性也更好 |

## Kafka介绍

Kafka是分布式发布-订阅消息系统。

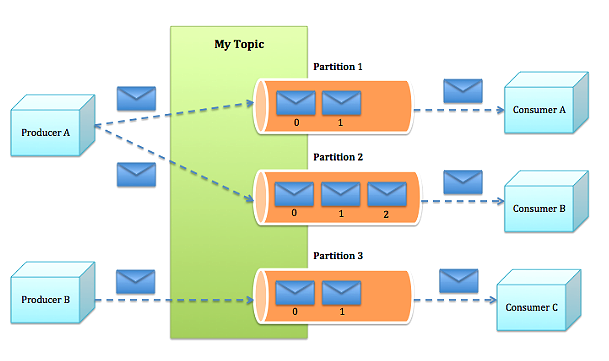
Kafka架构：



消息发送流程：

Producer根据指定的partition方法（round-robin、hash等），将消息发布到指定topic的partition里面

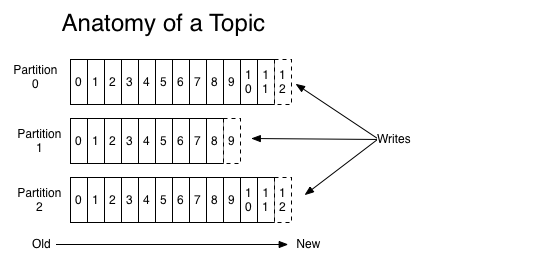
kafka集群接收到Producer发过来的消息后，将其持久化到硬盘，并保留消息指定时长（可配置），而不关注消息是否被消费



Kafka中的Message是以topic为基本单位组织的。

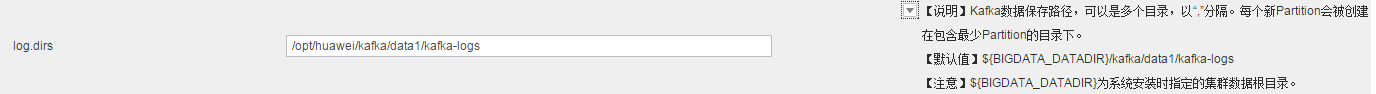
Topic- Partition- LogSegment- Message

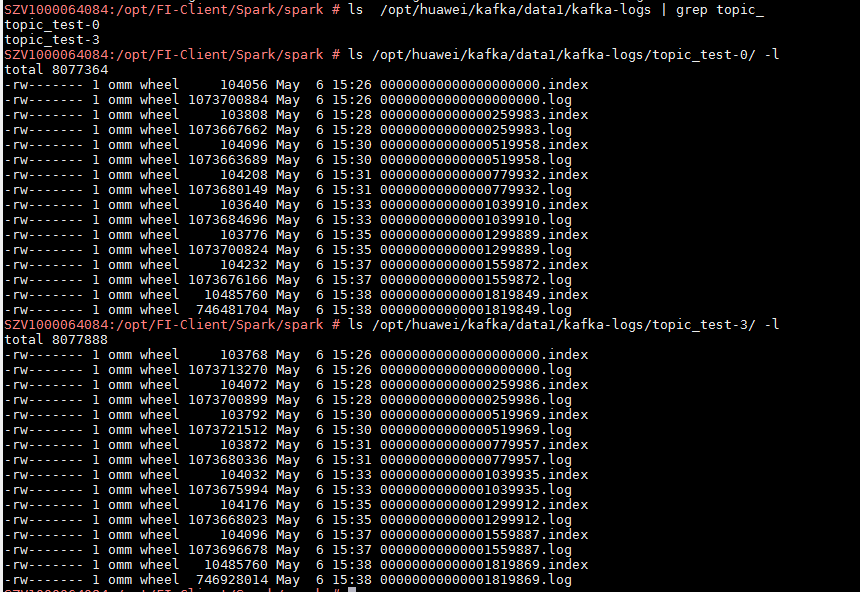
Kafka的Message存储采取了分区(partition)，分段(LogSegment)和稀疏索引这几个手段来到达了高效性。



那么message究竟如何存储的呢？

在kafka的LogSegment的文件路径配置为下图：





* segment file组成：由2大部分组成，分别为index file和data file，此2个文件一一对应，成对出现，后缀".index"和“.log”分别表示为segment索引文件、数据文件.
* segment文件命名规则：partion全局的第一个segment从0开始，后续每个segment文件名为上一个segment文件最后一条消息的offset值。数值最大为64位long大小，19位数字字符长度，没有数字用0填充。
* segment index file采取稀疏索引存储方式，它减少索引文件大小，通过mmap可以直接内存操作



其中每条message的数据结构：



# 数据流分析

## DirectKafkaInputDStream构造

调用接口KafkaUtils. createDirectStream接口创建Dstream，有两种scala接口：

def createDirectStream[K: ClassTag, V: ClassTag, KD <: Decoder[K]: ClassTag, VD <: Decoder[V]: ClassTag] (ssc: StreamingContext, kafkaParams: Map[String, String], topics: Set[String])

def createDirectStream[K: ClassTag, V: ClassTag, KD <: Decoder[K]: ClassTag, VD <: Decoder[V]: ClassTag, R: ClassTag] (ssc: StreamingContext, kafkaParams: Map[String, String], fromOffsets: Map[TopicAndPartition, Long], messageHandler: MessageAndMetadata[K, V] => R)

createDirectStream可以接收多个topic的数据，topics: Set[String]就是各个topic的名字

fromOffsets就是初始化从partition的那个offset开始进行数据处理

第一种接口会根据"auto.offset.reset"配置参数去更新offset，假如是“smallest”就调用kc.getEarliestLeaderOffsets(topicPartitions)去获取最初的offset开始产生Job，默认不配置的话就是调用kc.getLatestLeaderOffsets(topicPartitions)获取目前所在的offset开始处理

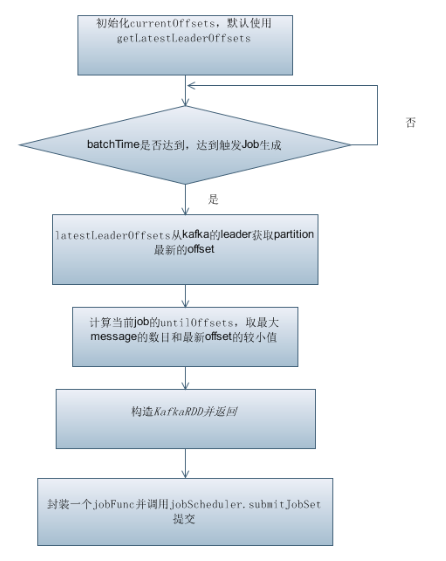
## generateJob流程

首先，调用DStream的action操作（print或者foreachRDD方法，saveAsTextFiles等方法实际调用的就是foreachRDD）将该DStream注册到DStreamGraph的outputStreams内，在JobGenerator每经过一个batchTime的时候就去触发遍历outputStreams 的generateJob方法，当层层调用到DirectKafkaInputDStream的compute方法时，就会产生一个kafkaRDD，最后把这些RDD封装成一个Job即可。

**val** jobFunc = () => {  
 **val** emptyFunc = { (iterator: Iterator[T]) => {} }  
 context.sparkContext.runJob(rdd, emptyFunc)  
}  
*Some*(**new** Job(time, jobFunc))

DirectKafkaInputDStream的compute流程：

1. 获取每个partition的最新的offset，一个partition对应一个kafka的leader
2. 根据“spark.streaming.receiver.maxRate / numPartitions”和“spark.streaming.kafka.maxRatePerPartition”参数计算每秒的最大处理message数，然后再换算成batchTime所处理的最大message数，与最新的latest\_offset比较，选择Math.min(currentOffsets(tp) + mmp, latest\_offset))作为该batchTime所对应的untilOffsets
3. 构造kafkaRDD，处理数据为currentOffsets到untilOffsets
4. 更新currentOffsets为untilOffsets，为下次Job生成做准备，也就是这时offset就已经更新完成了，并不等待Job运行成功
5. 返回kafkaRDD

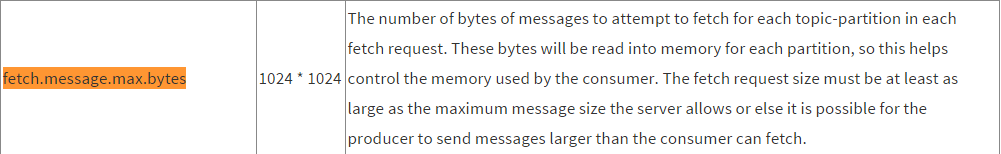


## Job执行

Job产生之后调用JobScheduler的submitJobSet接口，并调用jobExecutor.execute(new JobHandler(job)))进行执行。

KafkaRDD的执行：

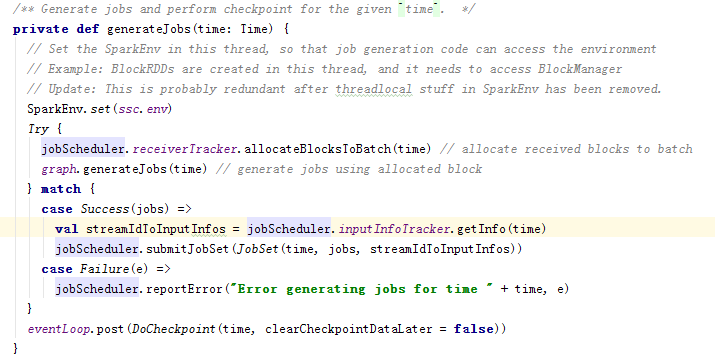
1. 构造一个new KafkaRDDIterator(part, context)
2. KafkaRDDIterator内构造一个SimpleConsumer，用于连接leader获取partition数据
3. KafkaRDD重写了getNext方法，每次调用fetchBatch得到一定大小的messages，该字节大小由“fetch.message.max.bytes”参数控制，默认为1024 \* 1024，getNext方法每次返回一条数据，处理完一个fetch的messages，就继续调用fetchBatch，直到所有offset内的数据处理完毕

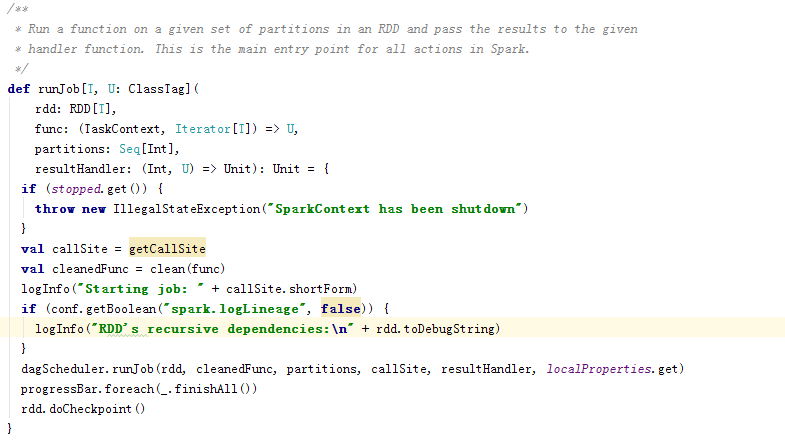


## CheckPoint说明

CheckPoint有两种形式，一种是checkpoint文件，还有一种是rdd的文件，前者在JobGenerator的generateJobs最后生成，后者是在dagScheduler.runJob之后做checkpoint。

代码如下图：





第一种的checkpoint文件最大数是由参数“spark.sql.bigdata.streaming.checkPointFileslimit”控制的，默认为10个，存储的数据是当前StreamingContext的类序列化之后的数据；

第二种是将rdd序列化写入文件，触发一个job用于写checkpoint文件，“rdd.context.runJob(rdd, ReliableCheckpointRDD.writeCheckpointFile[T](cpDir, broadcastedConf) \_)”

-------------结束------------

**附spark.apache.org对createDirectStream的api描述：**

Create an input stream that directly pulls messages from Kafka Brokers without using any receiver. This stream can guarantee that each message from Kafka is included in transformations exactly once (see points below).

Points to note: - No receivers: This stream does not use any receiver. It directly queries Kafka - Offsets: This does not use Zookeeper to store offsets. The consumed offsets are tracked by the stream itself. For interoperability with Kafka monitoring tools that depend on Zookeeper, you have to update Kafka/Zookeeper yourself from the streaming application. You can access the offsets used in each batch from the generated RDDs (see [HasOffsetRanges](https://spark.apache.org/docs/1.5.1/api/java/org/apache/spark/streaming/kafka/HasOffsetRanges.html" \o "interface in org.apache.spark.streaming.kafka)). - Failure Recovery: To recover from driver failures, you have to enable checkpointing in the StreamingContext. The information on consumed offset can be recovered from the checkpoint. See the programming guide for details (constraints, etc.). - 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.