Spark 网络资源学习笔记

harli

2015/3/5

# Spark Streaming

## [整合Kafka到Spark Streaming——代码示例和挑战](http://blog.csdn.net/stark_summer/article/details/44038247)

<http://blog.csdn.net/stark_summer/article/details/44038247>

作者Michael G. Noll是瑞士的一位工程师和研究员，效力于Verisign，是Verisign实验室的大规模数据分析基础设施（基础Hadoop）的技术主管。本文，Michael详细的演示了如何将Kafka整合到Spark Streaming中。 期间， Michael还提到了将Kafka整合到 Spark Streaming中的一些现状，非常值得阅读，虽然有一些信息在Spark 1.2版本中已发生了一些变化，比如HA策略： [通过Spark Contributor、Spark布道者陈超我们了解到](http://www.csdn.net/article/2014-12-22/2823239) ，在Spark 1.2版本中，Spark Streaming开始支持fully HA模式（选择使用），通过添加一层WAL（Write Ahead Log），每次收到数据后都会存在HDFS上，从而避免了以前版本中的数据丢失情况，但是不可避免的造成了一定的开销，需要开发者自行衡量。

以下为译文

作为一个实时大数据处理工具， [Spark Sreaming](https://spark.apache.org/streaming/) 近日一直被广泛关注，与 [Apache Storm](http://storm.apache.org/) 的对比也经常出现。但是依我说，缺少与Kafka整合，任何实时大数据处理工具都是不完整的，因此我将一个示例Spark Streaming应用程序添加到 [kafka-storm-starter](https://github.com/miguno/kafka-storm-starter) ，并且示范如何从Kafka读取，以及如何写入到Kafka。在这个过程中，我还使用Avro作为数据格式，以及Twitter Bijection进行数据序列化。

在本篇文章，我将详细地讲解这个Spark Streaming示例；同时，我还会穿插当下Spark Streaming与Kafka整合的一些焦点话题。免责声明：这是我首次试验Spark Streaming，仅作为参考。

当下，这个Spark Streaming示例被上传到GitHub，下载访问： [kafka-storm-starter](https://github.com/miguno/kafka-storm-starter)。项目的名称或许会让你产生某些误解，不过，不要在意这些细节：）

### **什么是Spark Streaming**

[Spark Streaming](http://spark.apache.org/streaming/) 是Apache Spark的一个子项目。Spark是个类似于Apache Hadoop的开源批处理平台，而Spark Streaming则是个实时处理工具，运行在Spark引擎之上。

### **Spark Streaming vs. Apache Storm**

Spark Streaming与Apache Storm有一些相似之处，后者是当下最流行的大数据处理平台。前不久，雅虎的Bobby Evans 和Tom Graves曾发表过一个“ [Spark and Storm at Yahoo!](http://yahoohadoop.tumblr.com/post/98213421641/storm-and-spark-at-yahoo-why-chose-one-over-the#_=_) ”的演讲，在这个演讲中，他们对比了两个大平台，并提供了一些选择参考。类似的，Hortonworks的P. Taylor Goetz也分享过名为 [Apache Storm and Spark Streaming Compared](http://www.slideshare.net/ptgoetz/apache-storm-vs-spark-streaming) 的讲义。

这里，我也提供了一个非常简短的对比：对比Spark Streaming，Storm的产业采用更高，生产环境应用也更稳定。但是从另一方面来说，对比Storm，Spark拥有更清晰、等级更高的API，因此Spark使用起来也更加愉快，最起码是在使用Scala编写Spark应用程序的情况（毫无疑问，我更喜欢Spark中的API）。但是，请别这么直接的相信我的话，多看看上面的演讲和讲义。

不管是Spark还是Storm，它们都是Apache的顶级项目，当下许多大数据平台提供商也已经开始整合这两个框架（或者其中一个）到其商业产品中，比如Hortonworks就同时整合了Spark和Storm，而Cloudera也整合了Spark。

### **附录：Spark中的Machines、cores、executors、tasks和receivers**

本文的后续部分将讲述许多Spark和Kafka中的parallelism问题，因此，你需要掌握一些Spark中的术语以弄懂这些环节。

* 一个Spark集群必然包含了1个以上的工者作节点，又称为从主机（为了简化架构，这里我们先抛弃开集群管理者不谈）。
* 一个工作者节点可以运行一个以上的executor
* Executor是一个用于应用程序或者工作者节点的进程，它们负责处理tasks，并将数据保存到内存或者磁盘中。每个应用程序都有属于自己的executors，一个executor则包含了一定数量的cores（也被称为slots）来运行分配给它的任务。
* Task是一个工作单元，它将被传送给executor。也就是说，task将是你应用程序的计算内容（或者是一部分）。SparkContext将把这些tasks发送到executors进行执行。每个task都会占用父executor中的一个core（slot）。
* Receiver（ [API](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.streaming.receiver.Receiver) ， [文档](http://spark.apache.org/docs/latest/streaming-custom-receivers.html) ）将作为一个长期运行的task跑在一个executor上。每个receiver都会负责一个所谓的input DStream（比如从Kafka中读取的一个输入流），同时每个receiver（ input DStream）占用一个core/slot。
* input DStream：input DStream是DStream的一个类型，它负责将Spark Streaming连接到外部的数据源，用于读取数据。对于每个外部数据源（比如Kafka）你都需要配置一个input DStream。一个Spark Streaming会通过一个input DStream与一个外部数据源进行连接，任何后续的DStream都会建立标准的DStreams。

在Spark的执行模型，每个应用程序都会获得自己的executors，它们会支撑应用程序的整个流程，并以多线程的方式运行1个以上的tasks，这种隔离途径非常类似Storm的执行模型。一旦引入类似YARN或者Mesos这样的集群管理器，整个架构将会变得异常复杂，因此这里将不会引入。你可以通过Spark文档中的 [Cluster Overview](http://spark.apache.org/docs/latest/cluster-overview.html) 了解更多细节。

### 整合Kafka到Spark Streaming

概述

简而言之，Spark是支持Kafka的，但是这里存在许多不完善的地方。

Spark代码库中的 [KafkaWordCount](https://github.com/apache/spark/blob/master/examples/src/main/scala/org/apache/spark/examples/streaming/KafkaWordCount.scala) 对于我们来说是个非常好的起点，但是这里仍然存在一些开放式问题。

特别是我想了解如何去做：

* 从kafaka中并行读入。在Kafka，一个话题（topic）可以有N个分区。理想的情况下，我们希望在多个分区上并行读取。这也是 [Kafka spout in Storm](https://github.com/apache/incubator-storm/tree/master/external/storm-kafka) 的工作。
* 从一个Spark Streaming应用程序向Kafka写入，同样，我们需要并行执行。

在完成这些操作时，我同样碰到了Spark Streaming和/或Kafka中一些已知的问题，这些问题大部分都已经在Spark mailing list中列出。在下面，我将详细总结Kafka集成到Spark的现状以及一些常见问题。

Kafka中的话题、分区（partitions）和parallelism

详情可以查看我之前的博文： [Apache Kafka 0.8 Training Deck and Tutorial](http://www.michael-noll.com/blog/2014/08/18/apache-kafka-training-deck-and-tutorial/) 和[Running a Multi-Broker Apache Kafka 0.8 Cluster on a Single Node](http://www.michael-noll.com/blog/2013/03/13/running-a-multi-broker-apache-kafka-cluster-on-a-single-node/) 。

Kafka将数据存储在话题中，每个话题都包含了一些可配置数量的分区。话题的分区数量对于性能来说非常重要，而这个值一般是消费者parallelism的最大数量：如果一个话题拥有N个分区，那么你的应用程序最大程度上只能进行N个线程的并行，最起码在使用Kafka内置Scala/Java消费者API时是这样的。

与其说应用程序，不如说Kafka术语中的消费者群（consumer group）。消费者群，通过你选择的字符串识别，它是逻辑消费者应用程序集群范围的识别符。同一个消费者群中的所有消费者将分担从一个指定Kafka话题中的读取任务，同时，同一个消费组中所有消费者从话题中读取的线程数最大值即是N（等同于分区的数量），多余的线程将会闲置。

多个不同的Kafka消费者群可以并行的运行：毫无疑问，对同一个Kafka话题，你可以运行多个独立的逻辑消费者应用程序。这里，每个逻辑应用程序都会运行自己的消费者线程，使用一个唯一的消费者群id。而每个应用程序通常可以使用不同的read parallelisms（见下文）。当在下文我描述不同的方式配置read parallelisms时，我指的是如何完成这些逻辑消费者应用程序中的一个设置。

这里有一些简单的例子

* 你的应用程序使用“terran”消费者群id对一个名为“zerg.hydra”的kafka话题进行读取，这个话题拥有10个分区。如果你的消费者应用程序只配置一个线程对这个话题进行读取，那么这个线程将从10个分区中进行读取。
* 同上，但是这次你会配置5个线程，那么每个线程都会从2个分区中进行读取。
* 同上，这次你会配置10个线程，那么每个线程都会负责1个分区的读取。
* 同上，但是这次你会配置多达14个线程。那么这14个线程中的10个将平分10个分区的读取工作，剩下的4个将会被闲置。

这里我们不妨看一下现实应用中的复杂性——Kafka中的再平衡事件。在Kafka中，再平衡是个生命周期事件（lifecycle event），在消费者加入或者离开消费者群时都会触发再平衡事件。这里我们不会进行详述，更多再平衡详情可参见我的 [Kafka training deck](http://www.michael-noll.com/blog/2014/08/18/apache-kafka-training-deck-and-tutorial/) 一文。

你的应用程序使用消费者群id“terran”，并且从1个线程开始，这个线程将从10个分区中进行读取。在运行时，你逐渐将线程从1个提升到14个。也就是说，在同一个消费者群中，parallelism突然发生了变化。毫无疑问，这将造成Kafka中的再平衡。一旦在平衡结束，你的14个线程中将有10个线程平分10个分区的读取工作，剩余的4个将会被闲置。因此如你想象的一样，初始线程以后只会读取一个分区中的内容，将不会再读取其他分区中的数据。

现在，我们终于对话题、分区有了一定的理解，而分区的数量将作为从Kafka读取时parallelism的上限。但是对于一个应用程序来说，这种机制会产生一个什么样的影响，比如一个Spark Streaming job或者 Storm topology从Kafka中读取数据作为输入。

**1. Read parallelism：** 通常情况下，你期望使用N个线程并行读取Kafka话题中的N个分区。同时，鉴于数据的体积，你期望这些线程跨不同的NIC，也就是跨不同的主机。在Storm中，这可以通过TopologyBuilder#setSpout()设置Kafka spout的parallelism为N来实现。在Spark中，你则需要做更多的事情，在下文我将详述如何实现这一点。

**2. Downstream processing parallelism：** 一旦使用Kafka，你希望对数据进行并行处理。鉴于你的用例，这种等级的parallelism必然与read parallelism有所区别。如果你的用例是计算密集型的，举个例子，对比读取线程，你期望拥有更多的处理线程；这可以通过从多个读取线程shuffling或者网路“fanning out”数据到处理线程实现。因此，你通过增长网络通信、序列化开销等将访问交付给更多的cores。在Storm中，你通过[shuffle grouping](https://storm.apache.org/documentation/Concepts.html) 将Kafka spout shuffling到下游的bolt中。在Spark中，你需要通过DStreams上的 [repartition](https://spark.apache.org/docs/1.1.0/streaming-programming-guide.html#transformations-on-dstreams) 转换来实现。

通常情况下，大家都渴望去耦从Kafka的parallelisms读取，并立即处理读取来的数据。在下一节，我将详述使用 Spark Streaming从Kafka中的读取和写入。

### 从Kafka中读取

Spark Streaming中的Read parallelism

类似Kafka，Read parallelism中也有分区的概念。了解Kafka的per-topic话题与[RDDs in Spark](http://spark.apache.org/docs/1.1.0/programming-guide.html) 中的分区没有关联非常重要。

Spark Streaming中的 [KafkaInputDStream](https://github.com/apache/spark/blob/master/external/kafka/src/main/scala/org/apache/spark/streaming/kafka/KafkaInputDStream.scala) （又称为Kafka连接器）使用了Kafka的[高等级消费者API](http://kafka.apache.org/documentation.html#highlevelconsumerapi) ，这意味着在Spark中为Kafka设置 read parallelism将拥有两个控制按钮。

**1. Input DStreams的数量。** 因为Spark在每个Input DStreams都会运行一个receiver（=task），这就意味着使用多个input DStreams将跨多个节点并行进行读取操作，因此，这里寄希望于多主机和NICs。

**2. Input DStreams上的消费者线程数量。** 这里，相同的receiver（=task）将运行多个读取线程。这也就是说，读取操作在每个core/machine/NIC上将并行的进行。

在实际情况中，第一个选择显然更是大家期望的。

为什么会这样？首先以及最重要的，从Kafka中读取通常情况下会受到网络/NIC限制，也就是说，在同一个主机上你运行多个线程不会增加读的吞吐量。另一方面来讲，虽然不经常，但是有时候从Kafka中读取也会遭遇CPU瓶颈。其次，如果你选择第二个选项，多个读取线程在将数据推送到blocks时会出现锁竞争（在block生产者实例上，BlockGenerator的“+=”方法真正使用的是“synchronized”方式）。

input DStreams建立的RDDs分区数量：KafkaInputDStream将储存从Kafka中读取的每个信息到Blocks。从我的理解上，一个新的Block由 spark.streaming.blockInterval在毫秒级别建立，而每个block都会转换成RDD的一个分区，最终由DStream建立。如果我的这种假设成立，那么由KafkaInputDStream建立的RDDs分区数量由batchInterval / spark.streaming.blockInterval决定，而batchInterval则是数据流拆分成batches的时间间隔，它可以通过StreamingContext的一个构造函数参数设置。举个例子，如果你的批时间价格是2秒（默认情况下），而block的时间间隔是200毫秒（默认情况），那么你的RDD将包含10个分区。如果有错误的话，可以提醒我。

选项1：控制input DStreams的数量

下面这个例子可以从 [Spark Streaming Programming Guide](https://spark.apache.org/docs/1.1.0/streaming-programming-guide.html#reducing-the-processing-time-of-each-batch) 中获得：

val ssc: StreamingContext = ??? // ignore for now

val kafkaParams: Map[String, String] = Map("group.id" -> "terran", /\* ignore rest \*/)

val numInputDStreams = 5

val kafkaDStreams = (1 to numInputDStreams).map { \_ => KafkaUtils.createStream(...) }

在这个例子中，我们建立了5个input DStreams，因此从Kafka中读取的工作将分担到5个核心上，寄希望于5个主机/NICs（之所以说是寄希望于，因为我也不确定Spark Streaming task布局策略是否会将receivers投放到多个主机上）。所有Input Streams都是“terran”消费者群的一部分，而Kafka将保证topic的所有数据可以同时对这5个input DSreams可用。换句话说，这种“collaborating”input DStreams设置可以工作是基于消费者群的行为是由Kafka API提供，通过KafkaInputDStream完成。

在这个例子中，我没有提到每个input DSream会建立多少个线程。在这里，线程的数量可以通过KafkaUtils.createStream方法的参数设置（同时，input topic的数量也可以通过这个方法的参数指定）。在下一节中，我们将通过实际操作展示。

但是在开始之前，在这个步骤我先解释几个Spark Streaming中常见的几个问题，其中有些因为当下Spark中存在的一些限制引起，另一方面则是由于当下Kafka input DSreams的一些设置造成：

当你使用我上文介绍的多输入流途径，而这些消费者都是属于同一个消费者群，它们会给消费者指定负责的分区。这样一来则可能导致syncpartitionrebalance的失败，系统中真正工作的消费者可能只会有几个。为了解决这个问题，你可以把再均衡尝试设置的非常高，从而获得它的帮助。然后，你将会碰到另一个坑——如果你的receiver宕机（OOM，亦或是硬件故障），你将停止从Kafka接收消息。

Spark用户讨论 [markmail.org/message/…](http://markmail.org/message/257a5l3oqyftsjxj)

这里，我们需要对“停止从Kafka中接收”问题 [做一些解释](http://apache-spark-user-list.1001560.n3.nabble.com/spark-streaming-and-the-spark-shell-tp3347.html) 。当下，当你通过ssc.start()开启你的streams应用程序后，处理会开始并一直进行，即使是输入数据源（比如Kafka）变得不可用。也就是说，流不能检测出是否与上游数据源失去链接，因此也不会对丢失做出任何反应，举个例子来说也就是重连或者结束执行。类似的，如果你丢失这个数据源的一个receiver，那么 [你的流应用程序可能就会生成一些空的RDDs](http://apache-spark-user-list.1001560.n3.nabble.com/spark-streaming-and-the-spark-shell-td3347.html#a3375) 。

这是一个非常糟糕的情况。最简单也是最粗糙的方法就是，在与上游数据源断开连接或者一个receiver失败时，重启你的流应用程序。但是，这种解决方案可能并不会产生实际效果，即使你的应用程序需要将Kafka配置选项auto.offset.reset设置到最小——因为Spark Streaming中一些已知的bug，可能导致你的流应用程序发生一些你意想不到的问题，在下文Spark Streaming中常见问题一节我们将详细的进行介绍。

选择2：控制每个input DStream上小发着线程的数量

在这个例子中，我们将建立一个单一的input DStream，它将运行3个消费者线程——在同一个receiver/task，因此是在同一个core/machine/NIC上对Kafka topic “zerg.hydra”进行读取。

val ssc: StreamingContext = ??? // ignore for now

val kafkaParams: Map[String, String] = Map("group.id" -> "terran", ...)

val consumerThreadsPerInputDstream = 3

val topics = Map("zerg.hydra" -> consumerThreadsPerInputDstream)

val stream = KafkaUtils.createStream(ssc, kafkaParams, topics, ...)

KafkaUtils.createStream方法被重载，因此这里有一些不同方法的特征。在这里，我们会选择Scala派生以获得最佳的控制。

结合选项1和选项2

下面是一个更完整的示例，结合了上述两种技术：

val ssc: StreamingContext = ???

val kafkaParams: Map[String, String] = Map("group.id" -> "terran", ...)

val numDStreams = 5

val topics = Map("zerg.hydra" -> 1)

val kafkaDStreams = (1 to numDStreams).map { \_ =>

KafkaUtils.createStream(ssc, kafkaParams, topics, ...)

}

我们建立了5个input DStreams，它们每个都会运行一个消费者线程。如果“zerg.hydra”topic拥有5个分区（或者更少），那么这将是进行并行读取的最佳途径，如果你在意系统最大吞吐量的话。

Spark Streaming中的并行Downstream处理

在之前的章节中，我们覆盖了从Kafka的并行化读取，那么我们就可以在Spark中进行并行化处理。那么这里，你必须弄清楚Spark本身是如何进行并行化处理的。类似Kafka，Spark将parallelism设置的与（RDD）分区数量有关， [通过在每个RDD分区上运行task进行](http://spark.apache.org/docs/1.1.0/programming-guide.html#resilient-distributed-datasets-rdds) 。在有些文档中，分区仍然被称为“slices”。

在任何Spark应用程序中，一旦某个Spark Streaming应用程序接收到输入数据，其他处理都与非streaming应用程序相同。也就是说，与普通的Spark数据流应用程序一样，在Spark Streaming应用程序中，你将使用相同的工具和模式。更多详情可见[Level of Parallelism in Data Processing](https://spark.apache.org/docs/1.1.0/streaming-programming-guide.html#level-of-parallelism-in-data-processing) 文档。

因此，我们同样将获得两个控制手段：

**1. input DStreams的数量** ，也就是说，我们在之前章节中read parallelism的数量作为结果。这是我们的立足点，这样一来，我们在下一个步骤中既可以保持原样，也可以进行修改。

**2. DStream转化的重分配** 。这里将获得一个全新的DStream，其parallelism等级可能增加、减少，或者保持原样。在DStream中每个返回的RDD都有指定的N个分区。DStream由一系列的RDD组成，DStream.repartition则是通过RDD.repartition实现。接下来将对RDD中的所有数据做随机的reshuffles，然后建立或多或少的分区，并进行平衡。同时，数据会在所有网络中进行shuffles。换句话说，DStream.repartition非常类似Storm中的shuffle grouping。

因此，repartition是从processing parallelism解耦read parallelism的主要途径。在这里，我们可以设置processing tasks的数量，也就是说设置处理过程中所有core的数量。间接上，我们同样设置了投入machines/NICs的数量。

一个DStream转换相关是 [union](https://spark.apache.org/docs/1.1.0/streaming-programming-guide.html#transformations-on-dstreams) 。这个方法同样在StreamingContext中，它将从多个DStream中返回一个统一的DStream，它将拥有相同的类型和滑动时间。通常情况下，你更愿意用StreamingContext的派生。一个union将返回一个由Union RDD支撑的UnionDStream。Union RDD由RDDs统一后的所有分区组成，也就是说，如果10个分区都联合了3个RDDs，那么你的联合RDD实例将包含30个分区。换句话说，union会将多个 DStreams压缩到一个 DStreams或者RDD中，但是需要注意的是，这里的parallelism并不会发生改变。你是否使用union依赖于你的用例是否需要从所有Kafka分区进行“in one place”信息获取决定，因此这里大部分都是基于语义需求决定。举个例子，当你需要执行一个不用元素上的（全局）计数。

**注意：** RDDs是无序的。因此，当你union RDDs时，那么结果RDD同样不会拥有一个很好的序列。如果你需要在RDD中进行sort。

你的用例将决定需要使用的方法，以及你需要使用哪个。如果你的用例是CPU密集型的，你希望对zerg.hydra topic进行5 read parallelism读取。也就是说，每个消费者进程使用5个receiver，但是却可以将processing parallelism提升到20。

val ssc: StreamingContext = ???

val kafkaParams: Map[String, String] = Map("group.id" -> "terran", ...)

val readParallelism = 5

val topics = Map("zerg.hydra" -> 1)

val kafkaDStreams = (1 to readParallelism).map { \_ =>

KafkaUtils.createStream(ssc, kafkaParams, topics, ...)

}

//> collection of five \*input\* DStreams = handled by five receivers/tasks

val unionDStream = ssc.union(kafkaDStreams) // often unnecessary, just showcasing how to do it

//> single DStream

val processingParallelism = 20

val processingDStream = unionDStream(processingParallelism)

//> single DStream but now with 20 partitions

在下一节中，我将把所有部分结合到一起，并且联合实际数据处理进行讲解。

### 写入到Kafka

写入到Kafka需要从foreachRDD输出操作进行：

通用的输出操作者都包含了一个功能（函数），让每个RDD都由Stream生成。这个函数需要将每个RDD中的数据推送到一个外部系统，比如将RDD保存到文件，或者通过网络将它写入到一个数据库。需要注意的是，这里的功能函数将在驱动中执行，同时其中通常会伴随RDD行为，它将会促使流RDDs的计算。

**注意：** 重提“功能函数是在驱动中执行”，也就是Kafka生产者将从驱动中进行，也就是说“功能函数是在驱动中进行评估”。当你使用foreachRDD从驱动中读取Design Patterns时，实际过程将变得更加清晰。

在这里，建议大家去阅读Spark文档中的 [Design Patterns for using foreachRDD](http://spark.apache.org/docs/1.1.0/streaming-programming-guide.html#output-operations-on-dstreams)一节，它将详细讲解使用foreachRDD读外部系统中的一些常用推荐模式，以及经常出现的一些陷阱。

在我们这个例子里，我们将按照推荐来重用Kafka生产者实例，通过生产者池跨多个RDDs/batches。 我通过 [Apache Commons Pool](http://commons.apache.org/proper/commons-pool/) 实现了这样一个工具，已经上传到[GitHub](https://github.com/miguno/kafka-storm-starter/blob/develop/src/main/scala/com/miguno/kafkastorm/kafka/PooledKafkaProducerAppFactory.scala) 。这个生产者池本身通过 [broadcast variable](http://spark.apache.org/docs/latest/programming-guide.html#broadcast-variables) 提供给tasks。

最终结果看起来如下：

val producerPool = {

// See the full code on GitHub for details on how the pool is created

val pool = createKafkaProducerPool(kafkaZkCluster.kafka.brokerList, outputTopic.name)

ssc.sparkContext.broadcast(pool)

}

stream.map { ... }.foreachRDD(rdd => {

rdd.foreachPartition(partitionOfRecords => {

// Get a producer from the shared pool

val p = producerPool.value.borrowObject()

partitionOfRecords.foreach { case tweet: Tweet =>

// Convert pojo back into Avro binary format

val bytes = converter.value.apply(tweet)

// Send the bytes to Kafka

p.send(bytes)

}

// Returning the producer to the pool also shuts it down

producerPool.value.returnObject(p)

})

})

需要注意的是， Spark Streaming每分钟都会建立多个RDDs，每个都会包含多个分区，因此你无需为Kafka生产者实例建立新的Kafka生产者，更不用说每个Kafka消息。上面的步骤将最小化Kafka生产者实例的建立数量，同时也会最小化TCP连接的数量（通常由Kafka集群确定）。你可以使用这个池设置来精确地控制对流应用程序可用的Kafka生产者实例数量。如果存在疑惑，尽量用更少的。

### 完整示例

下面的代码是示例Spark Streaming应用程序的要旨（所有代码参见 [这里](https://github.com/miguno/kafka-storm-starter/blob/develop/src/test/scala/com/miguno/kafkastorm/spark/KafkaSparkStreamingSpec.scala) ）。这里，我做一些解释：

* 并行地从Kafka topic中读取Avro-encoded数据。我们使用了一个最佳的read parallelism，每个Kafka分区都配置了一个单线程 input DStream。
* 并行化Avro-encoded数据到pojos中，然后将他们并行写到binary，序列化可以通过[Twitter Bijection](https://github.com/twitter/bijection) 执行。
* 通过Kafka生产者池将结果写回一个不同的Kafka topic。

// Set up the input DStream to read from Kafka (in parallel)

val kafkaStream = {

val sparkStreamingConsumerGroup = "spark-streaming-consumer-group"

val kafkaParams = Map(

"zookeeper.connect" -> "zookeeper1:2181",

"group.id" -> "spark-streaming-test",

"zookeeper.connection.timeout.ms" -> "1000")

val inputTopic = "input-topic"

val numPartitionsOfInputTopic = 5

val streams = (1 to numPartitionsOfInputTopic) map { \_ =>

KafkaUtils.createStream(ssc, kafkaParams, Map(inputTopic -> 1), StorageLevel.MEMORY\_ONLY\_SER).map(\_.\_2)

}

val unifiedStream = ssc.union(streams)

val sparkProcessingParallelism = 1 // You'd probably pick a higher value than 1 in production.

unifiedStream.repartition(sparkProcessingParallelism)

}

// We use accumulators to track global "counters" across the tasks of our streaming app

val numInputMessages = ssc.sparkContext.accumulator(0L, "Kafka messages consumed")

val numOutputMessages = ssc.sparkContext.accumulator(0L, "Kafka messages produced")

// We use a broadcast variable to share a pool of Kafka producers, which we use to write data from Spark to Kafka.

val producerPool = {

val pool = createKafkaProducerPool(kafkaZkCluster.kafka.brokerList, outputTopic.name)

ssc.sparkContext.broadcast(pool)

}

// We also use a broadcast variable for our Avro Injection (Twitter Bijection)

val converter = ssc.sparkContext.broadcast(SpecificAvroCodecs.toBinary[Tweet])

// Define the actual data flow of the streaming job

kafkaStream.map { case bytes =>

numInputMessages += 1

// Convert Avro binary data to pojo

converter.value.invert(bytes) match {

case Success(tweet) => tweet

case Failure(e) => // ignore if the conversion failed

}

}.foreachRDD(rdd => {

rdd.foreachPartition(partitionOfRecords => {

val p = producerPool.value.borrowObject()

partitionOfRecords.foreach { case tweet: Tweet =>

// Convert pojo back into Avro binary format

val bytes = converter.value.apply(tweet)

// Send the bytes to Kafka

p.send(bytes)

numOutputMessages += 1

}

producerPool.value.returnObject(p)

})

})

// Run the streaming job

ssc.start()

ssc.awaitTermination()

[更多的细节和解释可以在这里看所有源代码。](https://github.com/miguno/kafka-storm-starter/blob/develop/src/test/scala/com/miguno/kafkastorm/spark/KafkaSparkStreamingSpec.scala)

就我自己而言，我非常喜欢 Spark Streaming代码的简洁和表述。在Bobby Evans和 Tom Graves讲话中没有提到的是，Storm中这个功能的等价代码是非常繁琐和低等级的： [kafka-storm-starter](https://github.com/miguno/kafka-storm-starter) 中的 [KafkaStormSpec](https://github.com/miguno/kafka-storm-starter/blob/develop/src/test/scala/com/miguno/kafkastorm/integration/KafkaStormSpec.scala) 会运行一个Stormtopology来执行相同的计算。同时，规范文件本身只有非常少的代码，当然是除下说明语言，它们能更好的帮助理解；同时，需要注意的是，在Storm的Java API中，你不能使用上文Spark Streaming 示例中所使用的匿名函数，比如map和foreach步骤。取而代之的是，你必须编写完整的类来获得相同的功能，你可以查看 [AvroDecoderBolt](https://github.com/miguno/kafka-storm-starter/blob/develop/src/main/scala/com/miguno/kafkastorm/storm/bolts/AvroDecoderBolt.scala) 。这感觉是将Spark的API转换到Java，在这里使用匿名函数是非常痛苦的。

最后，我同样也非常喜欢 [Spark的说明文档](http://spark.apache.org/documentation.html) ，它非常适合初学者查看，甚至还包含了一些 [进阶使用](http://spark.apache.org/docs/1.1.0/tuning.html) 。关于Kafka整合到Spark，上文已经基本介绍完成，但是我们仍然需要浏览mailing list和深挖源代码。这里，我不得不说，维护帮助文档的同学做的实在是太棒了。

### 知晓Spark Streaming中的一些已知问题

你可能已经发现在Spark中仍然有一些尚未解决的问题，下面我描述一些我的发现：

一方面，在对Kafka进行读写上仍然存在一些含糊不清的问题，你可以在类似[Multiple Kafka Receivers and Union](http://apache-spark-user-list.1001560.n3.nabble.com/Multiple-Kafka-Receivers-and-Union-td14901.html) 和 [How to scale more consumer to Kafka stream](http://apache-spark-user-list.1001560.n3.nabble.com/How-to-scale-more-consumer-to-Kafka-stream-td13883.html) mailing list的讨论中发现。

另一方面，Spark Streaming中一些问题是因为Spark本身的固有问题导致，特别是故障发生时的数据丢失问题。换句话说，这些问题让你不想在生产环境中使用Spark。

* 在Spark 1.1版本的驱动中，Spark并不会考虑那些已经接收却因为种种原因没有进行处理的元数据（ [点击这里查看更多细节](https://www.mail-archive.com/user@spark.apache.org/msg10572.html) ）。因此，在某些情况下，你的Spark可能会丢失数据。Tathagata Das指出驱动恢复问题会在Spark的1.2版本中解决，当下已经释放。
* 1.1版本中的Kafka连接器是基于Kafka的高等级消费者API。这样就会造成一个问题，Spark Streaming不可以依赖其自身的KafkaInputDStream将数据从Kafka中重新发送，从而无法解决下游数据丢失问题（比如Spark服务器发生故障）。
* 有些人甚至认为这个版本中的Kafka连接器不应该投入生产环境使用，因为它是基于Kafka的高等级消费者API。取而代之，Spark应该使用简单的消费者API（就像Storm中的Kafka spout），它将允许你控制便宜和分区分配确定性。
* 但是当下Spark社区已经在致力这些方面的改善，比如Dibyendu Bhattacharya的Kafka连接器。后者是Apache Storm Kafka spout的一个端口，它基于Kafka所谓的简单消费者API，它包含了故障发生情景下一个更好的重放机制。
* 即使拥有如此多志愿者的努力，Spark团队更愿意非特殊情况下的Kafka故障恢复策略，他们的目标是“在所有转换中提供强保证，通用的策略”，这一点非常难以理解。从另一个角度来说，这是浪费Kafka本身的故障恢复策略。这里确实难以抉择。
* 这种情况同样也出现在写入情况中，很可能会造成数据丢失。
* Spark的Kafka消费者参数auto.offset.reset的使用同样与Kafka的策略不同。在Kafka中，将auto.offset.reset设置为最小是消费者将自动的将offset设置为最小offset，这通常会发生在两个情况：第一，在ZooKeeper中不存在已有offsets；第二，已存在offset，但是不在范围内。而在Spark中，它会始终删除所有的offsets，并从头开始。这样就代表着，当你使用auto.offset.reset = “smallest”重启你的应用程序时，你的应用程序将完全重新处理你的Kafka应用程序。更多详情可以在下面的两个讨论中发现： [1](http://apache-spark-user-list.1001560.n3.nabble.com/spark-streaming-and-the-spark-shell-td3347.html#a3387) 和 [2](http://markmail.org/message/257a5l3oqyftsjxj) 。
* Spark-1341：用于控制Spark Streaming中的数据传输速度。这个能力可以用于很多情况，当你已经受Kafka引起问题所烦恼时（比如auto.offset.reset所造成的），然后可能让你的应用程序重新处理一些旧数据。但是鉴于这里并没有内置的传输速率控制，这个功能可能会导致你的应用程序过载或者内存不足。

在这些故障处理策略和Kafka聚焦的问题之外之外，扩展性和稳定性上的关注同样不可忽视。再一次，仔细观看 [Bobby和Tom的视频](http://yahoohadoop.tumblr.com/post/98213421641/storm-and-spark-at-yahoo-why-chose-one-over-the-other) 以获得更多细节。在Spark使用经验上，他们都永远比我更丰富。

当然，我也有我的 [评论](https://www.mail-archive.com/user@spark.apache.org/msg11505.html) ，在 G1 garbage（在Java 1.7.0u4+中） 上可能也会存在问题。但是，我从来都没碰到这个问题。

### Spark使用技巧和敲门

在我实现这个示例的代码时，我做了一些重要的笔记。虽然这不是一个全面的指南，但是在你开始Kafka整合时可能发挥一定的作用。它包含了 [Spark Streaming programming guide](http://spark.apache.org/docs/1.1.0/streaming-programming-guide.html) 中的一些信息，也有一些是来自Spark用户的mailing list。

#### 通用

* 当你建立你的Spark环境时，对Spark使用的cores数量配置需要特别投入精力。你必须为Spark配置receiver足够使用的cores（见下文），当然实际数据处理所需要的cores的数量也要进行配置。在Spark中，每个receiver都负责一个input DStream。同时，每个receiver（以及每个input DStream） occies一个core，这样做是服务于每个文件流中的读取（详见文档）。举个例子，你的作业需要从两个input streams中读取数据，但是只访问两个cores，这样一来，所有数据都只会被读取而不会被处理。
* 注意，在一个流应用程序中，你可以建立多个input DStreams来并行接收多个数据流。在上文从Kafka并行读取一节中，我曾演示过这个示例作业。
* 你可以使用 broadcast variables在不同主机上共享标准、只读参数，相关细节见下文的优化指导。在示例作业中，我使用了broadcast variables共享了两个参数：第一，Kafka生产者池（作业通过它将输出写入Kafka）；第二，encoding/decoding Avro数据的注入（从Twitter Bijection中）。 [Passing functions to Spark](http://spark.apache.org/docs/1.1.0/programming-guide.html#passing-functions-to-spark) 。
* 你可以使用累加器参数来跟踪流作业上的所有全局“计数器”，这里可以对照Hadoop作业计数器。在示例作业中，我使用累加器分别计数所有消费的Kafka消息，以及所有对Kafka的写入。如果你对累加器进行命名，它们同样可以在Spark UI上展示。
* 不要忘记import Spark和Spark Streaming环境：

// Required to gain access to RDD transformations via implicits.

import org.apache.spark.SparkContext.\_

// Required when working on `PairDStreams` to gain access to e.g. `DStream.reduceByKey`

// (versus `DStream.transform(rddBatch => rddBatch.reduceByKey()`) via implicits.

//

// See also <http://spark.apache.org/docs/1.1.0/programming-guide.html#working-with-key-value-pairs>

import org.apache.spark.streaming.StreamingContext.toPairDStreamFunctions

如果你是 Twitter Algebird的爱好者，你将会喜欢使用Count-Min Sketch和Spark中的一些特性，代表性的，你会使用reduce或者reduceByWindow这样的操作（比如，[DStreams上的转换](http://spark.apache.org/docs/1.1.0/streaming-programming-guide.html#transformations-on-dstreams) ）。Spark项目包含了 [Count-Min Sketch](https://github.com/apache/spark/blob/master/examples/src/main/scala/org/apache/spark/examples/streaming/TwitterAlgebirdCMS.scala) 和 [HyperLogLog](https://github.com/apache/spark/blob/master/examples/src/main/scala/org/apache/spark/examples/streaming/TwitterAlgebirdHLL.scala) 的示例介绍。

如果你需要确定Algebird数据结构的内存介绍，比如Count-Min Sketch、HyperLogLog或者Bloom Filters，你可以使用SparkContext日志进行查看，更多细节参见 [Determining Memory Consumption](http://spark.apache.org/docs/1.1.0/tuning.html#determining-memory-consumption) 。

Kafka整合

我前文所述的一些增补：

* 你可能需要修改Spark Streaming中的一些Kafka消费者配置。举个例子，如果你需要从Kafka中读取大型消息，你必须添加fetch.message.max.bytes消费设置。你可以使用KafkaUtils.createStream(…)将这样定制的Kafka参数给Spark Streaming传送。

测试

* 首先，确定 已经 在一个finally bloc或者测试框架的teardown method中使用stop()关闭了StreamingContext 和/或 SparkContext，因为在同一个程序（或者JVM？）中Spark不支持并行运行两种环境。
* 根据我的经验，在使用sbt时，你希望在测试中将你的建立配置到分支JVM中。最起码在kafka-storm-starter中，测试必须并行运行多个线程，比如ZooKeeper、Kafka和Spark的内存实例。开始时，你可以参考 [build.sbt](https://github.com/miguno/kafka-storm-starter/blob/develop/build.sbt) 。
* 同样，如果你使用的是Mac OS X，你可能期望关闭JVM上的IPv6用以阻止DNS相关超时。这个问题与Spark无关，你可以查看 [.sbtopts](https://github.com/miguno/kafka-storm-starter/blob/develop/.sbtopts) 来获得关闭IPv6的方法。

#### 性能调优

* 确定你理解作业中的运行时影响，如果你需要与外部系统通信，比如Kafka。在使用foreachRDD时，你应该阅读中 [Spark Streaming programming guide](http://spark.apache.org/docs/1.1.0/streaming-programming-guide.html#output-operations-on-dstreams) 中的Design Patterns一节。举个例子，我的用例中使用Kafka产生者池来优化 Spark Streaming到Kafka的写入。在这里，优化意味着在多个task中共享同一个生产者，这个操作可以显著地减少由Kafka集群建立的新TCP连接数。
* 使用Kryo做序列化，取代默认的Java serialization，详情可以访问 [Tuning Spark](http://spark.apache.org/docs/1.1.0/tuning.html#serialized-rdd-storage) 。我的例子就使用了Kryo和注册器，举个例子，使用Kryo生成的Avro-generated Java类（见 [KafkaSparkStreamingRegistrator](https://github.com/miguno/kafka-storm-starter/blob/develop/src/main/scala/com/miguno/kafkastorm/spark/serialization/KafkaSparkStreamingRegistrator.scala) ）。除此之外，在Storm中类似的问题也可以使用Kryo来解决。
* 通过将spark.streaming.unpersist设置为true将Spark Streaming 作业设置到明确持续的RDDs。这可以显著地减少Spark在RDD上的内存使用，同时也可以改善GC行为。（点击访问 [来源](http://spark.apache.org/docs/1.1.0/streaming-programming-guide.html#memory-tuning) ）
* 通过MEMORY\_ONLY\_SER开始你的储存级别P&S测试（在这里，RDD被存储到序列化对象，每个分区一个字节）。取代反序列化，这样做更有空间效率，特别是使用Kryo这样的高速序列化工具时，但是会增加读取上的CPU密集操作。这个优化对 Spark Streaming作业也非常有效。对于本地测试来说，你可能并不想使用\*\_2派生（2=复制因子）。

#### 总结

完整的Spark Streaming示例代码可以在 [kafka-storm-starter](https://github.com/miguno/kafka-storm-starter/) 查看。这个应用包含了Kafka、Zookeeper、Spark，以及上文我讲述的示例。

总体来说，我对我的初次Spark Streaming体验非常满意。当然，在Spark/Spark Streaming也存在一些需要特别指明的问题，但是我肯定Spark社区终将解决这些问题。在这个过程中，得到了Spark社区积极和热情的帮助，同时我也非常期待Spark 1.2版本的新特性。

在大型生产环境中，基于Spark还需要一些TLC才能达到Storm能力，这种情况我可能将它投入生产环境中么？大部分情况下应该不会，更准确的说应该是现在不会。那么在当下，我又会使用Spark Streaming做什么样的处理？这里有两个想法，我认为肯定存在更多：

* 它可以非常快的原型数据流。如果你因为数据流太大而遭遇扩展性问题，你可以运行 Spark Streaming，在一些样本数据或者一部分数据中。
* 搭配使用Storm和Spark Streaming。举个例子，你可以使用Storm将原始、大规模输入数据处理到易管理等级，然后使用Spark Streaming来做下一步的分析，因为后者可以开箱即用大量有趣的算法、计算指令和用例。

感谢Spark社区对大数据领域所作出的贡献！

翻译/童阳

文章出处：推酷-CSDN