[**Spark Streaming编程指南**](http://www.cnblogs.com/cenyuhai/p/3577204.html)

**Overview**

Spark Streaming属于Spark的核心api，它支持高吞吐量、支持容错的实时流数据处理。

它可以接受来自Kafka, Flume, Twitter, ZeroMQ和TCP Socket的数据源，使用简单的api函数比如 map, reduce, join, window等操作，还可以直接使用内置的机器学习算法、图算法包来处理数据。



它的工作流程像下面的图所示一样，接受到实时数据后，给数据分批次，然后传给Spark Engine处理最后生成该批次的结果。



它支持的数据流叫Dstream，直接支持Kafka、Flume的数据源。Dstream是一种连续的RDDs，下面是一个例子帮助大家理解Dstream。

**A Quick Example**

[复制代码](javascript:void(0);)

// 创建StreamingContext，1秒一个批次  
val ssc = new StreamingContext(sparkConf, Seconds(1));

// 获得一个DStream负责连接 监听端口:地址

val lines = ssc.socketTextStream(serverIP, serverPort);

// 对每一行数据执行Split操作

val words = lines.flatMap(\_.split(" "));

// 统计word的数量

val pairs = words.map(word => (word, 1));

val wordCounts = pairs.reduceByKey(\_ + \_);

// 输出结果

wordCounts.print();

ssc.start(); // 开始

ssc.awaitTermination(); // 计算完毕退出

[复制代码](javascript:void(0);)

具体的代码可以访问这个页面：

https://github.com/apache/incubator-spark/blob/master/examples/src/main/scala/org/apache/spark/streaming/examples/NetworkWordCount.scala

如果已经装好Spark的朋友，我们可以通过下面的例子试试。

首先，启动Netcat，这个工具在Unix-like的系统都存在，是个简易的数据服务器。

使用下面这句命令来启动Netcat：

$ nc -lk 9999

接着启动example

$ ./bin/run-example org.apache.spark.streaming.examples.NetworkWordCount local[2] localhost 9999

在Netcat这端输入hello world，看Spark这边的

[复制代码](javascript:void(0);)

# TERMINAL 1:

# Running Netcat

$ nc -lk 9999

hello world

...

# TERMINAL 2: RUNNING NetworkWordCount or JavaNetworkWordCount

$ ./bin/run-example org.apache.spark.streaming.examples.NetworkWordCount local[2] localhost 9999

...

-------------------------------------------

Time: 1357008430000 ms

-------------------------------------------

(hello,1)

(world,1)

...

[复制代码](javascript:void(0);)

**Basics**

下面这块是如何编写代码的啦，哇咔咔！

首先我们要在SBT或者Maven工程添加以下信息：

groupId = org.apache.spark

artifactId = spark-streaming\_2.10

version = 0.9.0-incubating

[复制代码](javascript:void(0);)

//需要使用一下数据源的，还要添加相应的依赖  
Source Artifact

Kafka spark-streaming-kafka\_2.10

Flume spark-streaming-flume\_2.10

Twitter spark-streaming-twitter\_2.10

ZeroMQ spark-streaming-zeromq\_2.10

MQTT spark-streaming-mqtt\_2.10

[复制代码](javascript:void(0);)

接着就是实例化

new StreamingContext(master, appName, batchDuration, [sparkHome], [jars])

这是之前的例子对DStream的操作。



**Input Sources**

除了sockets之外，我们还可以这样创建Dstream

streamingContext.fileStream(dataDirectory)

这里有3个要点：

（1）dataDirectory下的文件格式都是一样

（2）在这个目录下创建文件都是通过移动或者重命名的方式创建的

（3）一旦文件进去之后就不能再改变

假设我们要创建一个Kafka的Dstream。

import org.apache.spark.streaming.kafka.\_

KafkaUtils.createStream(streamingContext, kafkaParams, ...)

如果我们需要自定义流的receiver，可以查看https://spark.incubator.apache.org/docs/latest/streaming-custom-receivers.html

**Operations**

对于Dstream，我们可以进行两种操作，*transformations* 和 *output*

**Transformations**

[复制代码](javascript:void(0);)

Transformation Meaning

map(func) 对每一个元素执行func方法

flatMap(func) 类似map函数，但是可以map到0+个输出

filter(func) 过滤

repartition(numPartitions) 增加分区，提高并行度

union(otherStream) 合并两个流

count() 　　　　　　　　 统计元素的个数

reduce(func) 对RDDs里面的元素进行聚合操作，2个输入参数，1个输出参数

countByValue() 针对类型统计，当一个Dstream的元素的类型是K的时候，调用它会返回一个新的Dstream，包含<K,Long>键值对，Long是每个K出现的频率。

reduceByKey(func, [numTasks]) 对于一个(K, V)类型的Dstream，为每个key，执行func函数，默认是local是2个线程，cluster是8个线程，也可以指定numTasks

join(otherStream, [numTasks]) 把(K, V)和(K, W)的Dstream连接成一个(K, (V, W))的新Dstream

cogroup(otherStream, [numTasks]) 把(K, V)和(K, W)的Dstream连接成一个(K, Seq[V], Seq[W])的新Dstream

transform(func) 转换操作，把原来的RDD通过func转换成一个新的RDD  
updateStateByKey(func) 针对key使用func来更新状态和值，可以将state该为任何值

[复制代码](javascript:void(0);)

**UpdateStateByKey Operation**

使用这个操作，我们是希望保存它状态的信息，然后持续的更新它，使用它有两个步骤：

（1）定义状态，这个状态可以是任意的数据类型

（2）定义状态更新函数，从前一个状态更改新的状态

下面展示一个例子：

def updateFunction(newValues: Seq[Int], runningCount: Option[Int]): Option[Int] = {

val newCount = ... // add the new values with the previous running count to get the new count

Some(newCount)

}

它可以用在包含(word, 1) 的Dstream当中，比如前面展示的example

val runningCounts = pairs.updateStateByKey[Int](updateFunction \_)

它会针对里面的每个word调用一下更新函数，newValues是最新的值，runningCount是之前的值。

**Transform Operation**

和transformWith一样，可以对一个Dstream进行RDD->RDD操作，比如我们要对Dstream流里的RDD和另外一个数据集进行join操作，但是Dstream的API没有直接暴露出来，我们就可以使用transform方法来进行这个操作，下面是例子：

val spamInfoRDD = sparkContext.hadoopFile(...) // RDD containing spam information

val cleanedDStream = inputDStream.transform(rdd => {

rdd.join(spamInfoRDD).filter(...) // join data stream with spam information to do data cleaning

...

})

另外，我们也可以在里面使用机器学习算法和图算法。

**Window Operations**

、

先举个例子吧，比如前面的word count的例子，我们想要每隔10秒计算一下最近30秒的单词总数。

我们可以使用以下语句：

// Reduce last 30 seconds of data, every 10 seconds

val windowedWordCounts = pairs.reduceByKeyAndWindow(\_ + \_, Seconds(30), Seconds(10))

这里面提到了windows的两个参数：

（1）window length：window的长度是30秒，最近30秒的数据

（2）slice interval：计算的时间间隔

通过这个例子，我们大概能够明白窗口的意思了，定期计算滑动的数据。

下面是window的一些操作函数，还是有点儿理解不了window的概念，Meaning就不翻译了，直接删掉

[复制代码](javascript:void(0);)

Transformation

window(windowLength, slideInterval)

countByWindow(windowLength, slideInterval)

reduceByWindow(func, windowLength, slideInterval)

reduceByKeyAndWindow(func, windowLength, slideInterval, [numTasks])

reduceByKeyAndWindow(func, invFunc, windowLength, slideInterval, [numTasks])

countByValueAndWindow(windowLength, slideInterval, [numTasks])

[复制代码](javascript:void(0);)

**Output Operations**

Output Operation Meaning

print() 打印到控制台

foreachRDD(func) 对Dstream里面的每个RDD执行func，保存到外部系统

saveAsObjectFiles(prefix, [suffix]) 保存流的内容为SequenceFile, 文件名 : "prefix-TIME\_IN\_MS[.suffix]".

saveAsTextFiles(prefix, [suffix]) 保存流的内容为文本文件, 文件名 : "prefix-TIME\_IN\_MS[.suffix]".

saveAsHadoopFiles(prefix, [suffix]) 保存流的内容为hadoop文件, 文件名 : "prefix-TIME\_IN\_MS[.suffix]".

**Persistence**

Dstream中的RDD也可以调用persist()方法保存在内存当中，但是基于window和state的操作，

reduceByWindow,reduceByKeyAndWindow,updateStateByKey它们就是隐式的保存了，系统已经帮它自动保存了。（需要设置checkpoint目录）

从网络接收的数据(such as, Kafka, Flume, sockets, etc.)，默认是保存在两个节点来实现容错性，以序列化的方式保存在内存当中。

**RDD Checkpointing**

状态的操作是基于多个批次的数据的。它包括基于window的操作和updateStateByKey。因为状态的操作要依赖于上一个批次的数据，所以它要根据时间，不断累积元数据。为了清空数据，它支持周期性的检查点，通过把中间结果保存到hdfs上。因为检查操作会导致保存到hdfs上的开销，所以设置这个时间间隔，要很慎重。对于小批次的数据，比如一秒的，检查操作会大大降低吞吐量。但是检查的间隔太长，会导致任务变大。通常来说，5-10秒的检查间隔时间是比较合适的。

ssc.checkpoint(hdfsPath) //设置检查点的保存位置

dstream.checkpoint(checkpointInterval) //设置检查点间隔

对于必须设置检查点的Dstream，比如通过updateStateByKey和reduceByKeyAndWindow创建的Dstream，默认设置是至少10秒。

**Performance Tuning**

对于调优，可以从两个方面考虑：

（1）利用集群资源，减少处理每个批次的数据的时间

（2）给每个批次的数据量的设定一个合适的大小

**Level of Parallelism**

像一些分布式的操作，比如reduceByKey和reduceByKeyAndWindow，默认的8个并发线程，可以通过对应的函数提高它的值，或者通过修改参数spark.default.parallelism来提高这个默认值。

**Task Launching Overheads**

通过进行的任务太多也不好，比如每秒50个，发送任务的负载就会变得很重要，很难实现亚秒级的时延了，当然可以通过压缩来降低批次的大小。

**Setting the Right Batch Size**

要使流程序能在集群上稳定的运行，要使处理数据的速度跟上数据流入的速度。最好的方式计算这个批量的大小，我们首先设置batch size为5-10秒和一个很低的数据输入速度。确实系统能跟上数据的速度的时候，我们可以根据经验设置它的大小，通过查看日志看看Total delay的多长时间。如果delay的小于batch的，那么系统可以稳定，如果delay一直增加，说明系统的处理速度跟不上数据的输入速度。

**24/7 Operation**

Spark默认不会忘记元数据，比如生成的RDD，处理的stages，但是Spark Streaming是一个24/7的程序，它需要周期性的清理元数据，通过spark.cleaner.ttl来设置。比如我设置它为600，当超过10分钟的时候，Spark就会清除所有元数据，然后持久化RDDs。但是这个属性要在SparkContext 创建之前设置。

但是这个值是和任何的window操作绑定。Spark会要求输入数据在过期之后必须持久化到内存当中，所以必须设置delay的值至少和最大的window操作一致，如果设置小了，就会报错。

**Monitoring**

除了Spark内置的监控能力，还可以StreamingListener这个接口来获取批处理的时间, 查询时延, 全部的端到端的试验。

**Memory Tuning**

Spark Stream默认的序列化方式是[StorageLevel.MEMORY\_ONLY\_SER](https://spark.incubator.apache.org/docs/latest/api/core/index.html#org.apache.spark.storage.StorageLevel$)，而不是RDD的[StorageLevel.MEMORY\_ONLY](https://spark.incubator.apache.org/docs/latest/api/core/index.html#org.apache.spark.storage.StorageLevel$)。

默认的，所有持久化的RDD都会通过被Spark的LRU算法剔除出内存，如果设置了spark.cleaner.ttl，就会周期性的清理，但是这个参数设置要很谨慎。一个更好的方法是设置spark.streaming.unpersist为true，这就让Spark来计算哪些RDD需要持久化，这样有利于提高GC的表现。

推荐使用concurrent mark-and-sweep GC，虽然这样会降低系统的吞吐量，但是这样有助于更稳定的进行批处理。

**Fault-tolerance Properties**

**Failure of a Worker Node**

下面有两种失效的方式：

1.使用hdfs上的文件，因为hdfs是可靠的文件系统，所以不会有任何的数据失效。

2.如果数据来源是网络，比如Kafka和Flume，为了防止失效，默认是数据会保存到2个节点上，但是有一种可能性是接受数据的节点挂了，那么数据可能会丢失，因为它还没来得及把数据复制到另外一个节点。

**Failure of the Driver Node**

为了支持24/7不间断的处理，Spark支持驱动节点失效后，重新恢复计算。Spark Streaming会周期性的写数据到hdfs系统，就是前面的检查点的那个目录。驱动节点失效之后，StreamingContext可以被恢复的。

为了让一个Spark Streaming程序能够被恢复，它需要做以下操作：

（1）第一次启动的时候，创建 StreamingContext，创建所有的streams，然后调用start()方法。

（2）恢复后重启的，必须通过检查点的数据重新创建StreamingContext。

下面是一个实际的例子：

通过StreamingContext.getOrCreate来构造StreamingContext，可以实现上面所说的。

[复制代码](javascript:void(0);)

// Function to create and setup a new StreamingContext

def functionToCreateContext(): StreamingContext = {

val ssc = new StreamingContext(...) // new context

val lines = ssc.socketTextStream(...) // create DStreams

...

ssc.checkpoint(checkpointDirectory) // set checkpoint directory

ssc

}

// Get StreaminContext from checkpoint data or create a new one

val context = StreamingContext.getOrCreate(checkpointDirectory, functionToCreateContext \_)

// Do additional setup on context that needs to be done,

// irrespective of whether it is being started or restarted

context. ...

// Start the context

context.start()

context.awaitTermination()

[复制代码](javascript:void(0);)

在stand-alone的部署模式下面，驱动节点失效了，也可以自动恢复，让别的驱动节点替代它。这个可以在本地进行测试，在提交的时候采用supervise模式，当提交了程序之后，使用jps查看进程，看到类似DriverWrapper就杀死它，如果是使用YARN模式的话就得使用其它方式来重新启动了。

这里顺便提一下向客户端提交程序吧，之前总结的时候把这块给落下了。

[复制代码](javascript:void(0);)

./bin/spark-class org.apache.spark.deploy.Client launch

[client-options] \

<cluster-url> <application-jar-url> <main-class> \

[application-options]

cluster-url: master的地址.

application-jar-url: jar包的地址，最好是hdfs上的,带上hdfs：//...否则要所有的节点的目录下都有这个jar的   
main-class: 要发布的程序的main函数所在类.   
Client Options:   
--memory <count> (驱动程序的内存，单位是MB)   
--cores <count> (为你的驱动程序分配多少个核心)   
--supervise (节点失效的时候，是否重新启动应用)   
--verbose (打印增量的日志输出)

[复制代码](javascript:void(0);)

在未来的版本，会支持所有的数据源的可恢复性。

为了更好的理解基于HDFS的驱动节点失效恢复，下面用一个简单的例子来说明：

[复制代码](javascript:void(0);)

Time Number of lines in input file Output without driver failure Output with driver failure

1 　　10 　　　　　　　　　　　　　　　　　10 　　　　　　　　　　　　　　　　10

2 　　20 　　　　　　　　　　　　　　　　　20 　　　　　　　　　　　　　　　　20

3 　　30 　　　　　　　　　　　　　　　　　30 　　　　　　　　　　　　　　　　30

4 　　40 　　　　　　　　　　　　　　　　　40 　　　　　　　　　　　　　　　　[DRIVER FAILS] no output

5 　　50 　　　　　　　　　　　　　　　　　50 　　　　　　　　　　　　　　　　no output

6 　　60 　　　　　　　　　　　　　　　　　60 　　　　　　　　　　　　　　　　no output

7 　　70 　　　　　　　　　　　　　　　　　70 　　　　　　　　　　　　　　　　[DRIVER RECOVERS] 40, 50, 60, 70

8 　　80 　　　　　　　　　　　　　　　　　80 　　　　　　　　　　　　　　　　80

9 　　90 　　　　　　　　　　　　　　　　　90 　　　　　　　　　　　　　　　　90

10 　100 　　　　　　　　　　　　　　　　 100 　　　　　　　　　　　　　　　100

[复制代码](javascript:void(0);)

在4的时候出现了错误，40,50,60都没有输出，到70的时候恢复了，恢复之后把之前没输出的一下子全部输出。