# Spark Streaming Programming Guide

* [Overview](http://spark.apache.org/docs/latest/streaming-programming-guide.html#overview)
* [A Quick Example](http://spark.apache.org/docs/latest/streaming-programming-guide.html#a-quick-example)
* [Basic Concepts](http://spark.apache.org/docs/latest/streaming-programming-guide.html#basic-concepts)
  + [Linking](http://spark.apache.org/docs/latest/streaming-programming-guide.html#linking)
  + [Initializing StreamingContext](http://spark.apache.org/docs/latest/streaming-programming-guide.html#initializing-streamingcontext)
  + [Discretized Streams (DStreams)](http://spark.apache.org/docs/latest/streaming-programming-guide.html#discretized-streams-dstreams)
  + [Input DStreams and Receivers](http://spark.apache.org/docs/latest/streaming-programming-guide.html#input-dstreams-and-receivers)
  + [Transformations on DStreams](http://spark.apache.org/docs/latest/streaming-programming-guide.html#transformations-on-dstreams)
  + [Output Operations on DStreams](http://spark.apache.org/docs/latest/streaming-programming-guide.html#output-operations-on-dstreams)
  + [DataFrame and SQL Operations](http://spark.apache.org/docs/latest/streaming-programming-guide.html#dataframe-and-sql-operations)
  + [MLlib Operations](http://spark.apache.org/docs/latest/streaming-programming-guide.html#mllib-operations)
  + [Caching / Persistence](http://spark.apache.org/docs/latest/streaming-programming-guide.html#caching--persistence)
  + [Checkpointing](http://spark.apache.org/docs/latest/streaming-programming-guide.html#checkpointing)
  + [Accumulators, Broadcast Variables, and Checkpoints](http://spark.apache.org/docs/latest/streaming-programming-guide.html#accumulators-broadcast-variables-and-checkpoints)
  + [Deploying Applications](http://spark.apache.org/docs/latest/streaming-programming-guide.html#deploying-applications)
  + [Monitoring Applications](http://spark.apache.org/docs/latest/streaming-programming-guide.html#monitoring-applications)
* [Performance Tuning](http://spark.apache.org/docs/latest/streaming-programming-guide.html#performance-tuning)
  + [Reducing the Batch Processing Times](http://spark.apache.org/docs/latest/streaming-programming-guide.html#reducing-the-batch-processing-times)
  + [Setting the Right Batch Interval](http://spark.apache.org/docs/latest/streaming-programming-guide.html#setting-the-right-batch-interval)
  + [Memory Tuning](http://spark.apache.org/docs/latest/streaming-programming-guide.html#memory-tuning)
* [Fault-tolerance Semantics](http://spark.apache.org/docs/latest/streaming-programming-guide.html#fault-tolerance-semantics)
* [Where to Go from Here](http://spark.apache.org/docs/latest/streaming-programming-guide.html#where-to-go-from-here)

# Overview

**Spark Streaming是核心Spark API的扩展，可实现实时数据流的可扩展，高吞吐量，容错流处理**。 数据可以从诸如Kafka，Flume，Kinesis或TCP套接字的许多来源中获取，并且可以使用以map，reducer，join和Windows等高级功能表示的复杂算法进行处理。 最后，处理后的数据可以推送到文件系统，数据库和实时仪表板。 实际上，您可以将Spark的机器学习和图形处理算法应用于数据流。



在内部，它的工作原理如下。 Spark Streaming接收实时输入数据流并将数据分成批，然后由Spark引擎进行处理，以批量生成最终的结果流。



Spark Streaming提供称为离散流或DStream的高级抽象，它代表连续的数据流。 可以从诸如**Kafka，Flume和Kinesis等来源的输入数据流**中**创建DStream**，或者**通过对其他DStream应用高级操作**来创建。 **在内部，DStream表示为RDD序列**。

本指南介绍如何开始**使用DStreams编写Spark Streaming程序**。 您可以在Scala，Java或Python（在Spark 1.2中引入）中编写Spark Streaming程序，所有这些都在本指南中介绍。 您可以在本指南中找到标签，让您可以选择不同语言的代码段。

注意：有一些API在Python中是不同的或不可用的。 在本指南中，您将发现Python API的标签突出显示这些差异。

## 一个快速的例子

在我们详细介绍如何编写自己的Spark Streaming程序之前，我们来看一下简单的Spark Streaming程序。 假设我们想计算从在TCP套接字上侦听的数据服务器接收到的文本数据中的字数。 所有你需要做的是如下。

* [**Python**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_python_0)

**首先**，我们**导入StreamingContext**，它是所有流功能的主要入口点。 我们创建一个带有两个执行线程的本地StreamingContext，**批处理间隔为1秒**。

***from pyspark import SparkContext***

***from pyspark.streaming import StreamingContext***

***# Create a local StreamingContext with two working thread and batch interval of 1 second***

***sc = SparkContext("local[2]", "NetworkWordCount")***

***ssc = StreamingContext(sc, 1)***

**使用这种上下文，我们可以创建一个DStream，它表示从TCP源（称为主机名）（例如localhost）和端口（例如9999））的流数据。**

# Create a DStream that will connect to hostname:port, like localhost:9999

lines = ssc.socketTextStream("localhost", 9999)

这行DStream表示将从数据服务器接收的数据流。 **此DStream中的每条记录都是一行文本。 接下来，我们要将空格的行分割成单词。**

# Split each line into words

words = lines.flatMap(lambda line: line.split(" "))

**flatMap是一个一对多的DStream操作，通过从源DStream中的每个记录生成多个新记录来创建新的DStream。** 在这种情况下，每一行将被分割成多个单词，并将单词流表示为单词DStream。 接下来，我们想计算这些话。

# Count each word in each batch

pairs = words.map(lambda word: (word, 1))

wordCounts = pairs.**reduceByKey**(lambda x, y: x + y)

# Print the first ten elements of each RDD generated in this DStream to the console

wordCounts.pprint()

**词DStream进一步映射（一对一变换）到（字，1）对的DStream**，**然后减少以获得每批数据中的单词的频率。** 最后，wordCounts.pprint（）将打印每秒产生的几个计数。

请注意，当执行这些行时，S**park Streaming仅设置启动时执行的计算，并且尚未开始实际处理。** 在所有转换设置完成后开始处理，我们终于打电话

ssc.start() # Start the computation

ssc.awaitTermination() # Wait for the computation to terminate

完整的代码可以在Spark Streaming示例NetworkWordCount中找到。

如果您已经下载并构建了Spark，则可以运行以下示例。 您将首先需要运行Netcat（大多数类Unix系统中的一个小型实用程序）作为数据服务器

$ nc -lk 9999

Then, in a different terminal, you can start the example by using

* [**Scala**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_scala_1)
* [**Java**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_java_1)
* [**Python**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_python_1)

$ ./bin/spark-submit examples/src/main/python/streaming/network\_wordcount.py localhost 9999

然后，在运行netcat服务器的终端中输入的任何行将每秒计数并打印在屏幕上。 它会看起来像下面这样。

|  |  |  |
| --- | --- | --- |
| # TERMINAL 1:  # Running Netcat  $ nc -lk 9999  hello world  ... |  | * [**Scala**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_scala_2) * [**Java**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_java_2) * [**Python**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_python_2)   # TERMINAL 2: RUNNING network\_wordcount.py  $ ./bin/spark-submit examples/src/main/python/streaming/network\_wordcount.py localhost 9999  ...  -------------------------------------------  Time: 2014-10-14 15:25:21  -------------------------------------------  (hello,1)  (world,1)  ... |

# 基本概念

接下来，我们超越了简单的例子，并阐述了Spark Streaming的基础知识。

链接

与Spark类似，Spark Streaming可通过Maven Central获得。 要编写自己的Spark Streaming程序，您必须将以下依赖项添加到SBT或Maven项目中。

跟别人最新的概述的

* [**Maven**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_Maven_3)
* [**SBT**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_SBT_3)

<dependency>

<groupId>org.apache.spark</groupId>

<artifactId>spark-streaming\_2.11</artifactId>

<version>2.1.1</version>

</dependency>

要从Spark Streaming核心API中不存在的来源（如Kafka，Flume和Kinesis）中获取数据，您必须将相应的工件spark-streaming-xyz\_2.11添加到依赖关系中。 例如，一些常见的如下。

|  |  |
| --- | --- |
| Source | Artifact |
| Kafka | spark-streaming-kafka-0-8\_2.11 |
| Flume | spark-streaming-flume\_2.11 |
| Kinesis | spark-streaming-kinesis-asl\_2.11 [Amazon Software License] |
|  |  |

有关最新的列表，请参阅Maven存储库以获取支持的源和工件的完整列表。

## 初始化StreamingContext

要初始化Spark Streaming程序，必须创建一个StreamingContext对象，它是所有Spark Streaming功能的主要入口点。

* [**Scala**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_scala_4)
* [**Java**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_java_4)
* [**Python**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_python_4)

A [StreamingContext](http://spark.apache.org/docs/latest/api/python/pyspark.streaming.html#pyspark.streaming.StreamingContext) object can be created from a [SparkContext](http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.SparkContext) object.

from pyspark import SparkContext

from pyspark.streaming import StreamingContext

sc = SparkContext(master, appName)

ssc = StreamingContext(sc, 1)

appName参数是应用程序在集群UI上显示的名称。 master是Spark，Mesos或YARN集群URL，或者以本地模式运行的特殊“local [\*]”字符串。 实际上，当在集群上运行时，您不需要在程序中硬编码主机，而是使用spark-submit启动应用程序并在其中接收。 但是，对于本地测试和单元测试，您可以通过“本地[\*]”来运行Spark Streaming（检测本地系统中的核心数）。

**批处理间隔必须根据应用程序的延迟要求和可用的集群资源进行设置。** 有关详细信息，请参阅“性能调优”部分。

定义上下文后，您必须执行以下操作。

1.通过创建输入DStreams来定义输入源。

2.通过对DStreams进行转换和输出操作来定义流式计算。

3.开始接收数据并使用streamingContext.start（）进行处理。

4.使用streamingContext.awaitTermination（）等待处理停止（手动或由于任何错误）。

可以使用streamingContext.stop（）手动停止处理。

要记住：

一旦上下文已经开始，就不能建立或添加新的流计算。

一旦上下文停止，就无法重新启动。

只有一个StreamingContext可以同时在JVM中处于活动状态。

**StreamingContext上的stop（）也会停止SparkContext。** 要**仅停止StreamingContext，请将stop（）的可选参数stopSparkContext设置为false。**

只要先前的StreamingContext在创建下一个StreamingContext之前停止（不停止SparkContext），SparkContext就可以重新创建多个StreamingContexts。

## 离散流（DStreams）

离散流或DStream是Spark Streaming提供的基本抽象。 它**表示连续的数据流**，**无论是从源接收的输入数据流，还是通过转换输入流生成的经处理的数据流。** 在内部，DStream由连续的一系列RDD表示，这是Spark对不可变的分布式数据集的抽象（详见Spark编程指南）。 DStream中的每个RDD都包含一定间隔的数据，如下图所示。



对DStream应用的任何操作都将转换为底层RDD上的操作。 例如，在将流的流转换为单词的早期示例中，将flatMap操作应用于行DStream中的每个RDD上，以生成单词DStream的RDD。 如下图所示。



这些基础的RDD转换由Spark引擎计算。 DStream操作可以隐藏大部分这些细节，为开发人员提供更高级的API以方便使用。 这些操作将在后面的章节中详细讨论。

## 输入DStreams和接收器

输入DStreams是表示从流媒体源接收的输入数据流的DStream。在快速示例中，线是一个输入DStream，因为它表示从netcat服务器接收的数据流。每个输入DStream（文件流除本节稍后讨论）与Receiver（Scala doc，Java doc）对象相关联，该对象从源接收数据并将其存储在Spark的内存中进行处理。

### Spark Streaming提供两类内置流式传输源。

基本资料来源：StreamingContext API中直接提供的资源。示例：文件系统和套接字连接。

高级资料来源：Kafka，Flume，Kinesis等资源可通过额外的实用工具类获得。这些需要链接到链接部分中讨论的额外依赖关系。

我们将在本节稍后讨论每个类别中的一些来源。

请注意，如果要在流式应用程序中并行接收多个数据流，则可以创建多个输入DStream（在“性能调优”部分进一步讨论）。这将创建多个接收器，同时接收多个数据流。但是请注意，Spark工作人员/执行者是一个长期运行的任务，因此它占据分配给Spark Streaming应用程序的核心之一。因此，重要的是要记住，Spark Streaming应用程序需要分配足够的内核（或线程，如果在本地运行）来处理接收到的数据，以及运行接收器。

要记住的

当本地运行Spark Streaming程序时，不要使用“local”或“local [1]”作为主URL。 这两者之一意味着只有一个线程将用于在本地运行任务。 如果您正在使用基于接收器（例如插座，卡夫卡，水槽等）的输入DStream，那么单线程将用于运行接收器，不会留出线程来处理接收到的数据。 因此，当在本地运行时，始终使用“local [n]”作为主URL，其中n>要运行的接收器数量（有关如何设置主机的信息，请参阅Spark属性）。

将逻辑扩展到在集群上运行，分配给Spark Streaming应用程序的内核数量必须大于接收器数量。 否则系统将收到数据，但无法处理。

### 基本资料

我们已经看过ssc.socketTextStream（...）在快速示例中，它通过TCP套接字连接接收的文本数据创建了一个DStream。除了套接字之外，StreamingContext API提供了从文件创建DStreams作为输入源的方法。

文件流：从与HDFS API（即HDFS，S3，NFS等）兼容的任何文件系统上的文件读取数据，DStream可以创建为：

Scala

Java的

Python

**streamingContext.textFileStream（DataDirectory目录）**

Spark Streaming将监视目录dataDirectory并处理在该目录中创建的任何文件（不支持嵌套目录中写入的文件）。注意

文件必须具有相同的数据格式。

必须通过将数据原子移动或重命名到数据目录中，在dataDirectory中创建文件。

移动后，不得更改文件。因此，如果连续追加文件，则不会读取新数据。

对于简单的文本文件，有一个更容易的方法streamingContext.textFileStream（dataDirectory）。而文件流不需要运行接收器，因此不需要分配内核。

Python API fileStream在Python API中不可用，只有textFileStream可用。

基于自定义接收器的流：可以通过自定义接收器接收的数据流创建DStream。有关详细信息，请参阅自定义接收器指南。

将RDD队列作为流：对于使用测试数据测试Spark Streaming应用程序，还可以使用streamingContext.queueStream（queueOfRDDs）创建基于RDD队列的DStream。推送到队列中的每个RDD将被视为DStream中的一批数据，并像流一样处理。

有关从套接字和文件的流的更多详细信息，请参阅StreamingContext for Scala，JavaStreamingContext for Java和Python的StreamingContext中相关函数的API文档。

### 高级来源

Python API从Spark 2.1.1开始，在这些源代码中，Kafka，Kinesis和Flume都可以在Python API中使用。

这种类别的源需要与外部非Spark库进行连接，其中一些具有复杂的依赖关系（例如Kafka和Flume）。因此，为了最小化与依赖关系的版本冲突相关的问题，从这些源创建DStream的功能已被移动到可以在必要时显式链接到单独的库。

请注意，这些高级源在Spark shell中不可用，因此基于这些高级源的应用程序无法在shell中测试。如果您真的想在Spark shell中使用它们，则必须下载相应的Maven工件的JAR及其依赖项，并将其添加到类路径中。

其中一些先进的资料如下。

Kafka：Spark Streaming 2.1.1与Kafka代理版本0.8.2.1或更高版本兼容。有关详细信息，请参阅“卡夫卡集成指南”。

Flume：Spark Streaming 2.1.1与Flume 1.6.0兼容。有关详细信息，请参阅“Flume Integration Guide”。

Kinesis：Spark Streaming 2.1.1与Kinesis Client Library 1.2.1兼容。有关详细信息，请参阅“Kinesis集成指南”。

### 自定义来源

Python API Python中尚不支持。

也可以通过自定义数据源创建输入DStream。所有你需要做的是实现一个用户定义的接收器（见下一部分来了解这是什么），可以从自定义源接收数据并将其推入到Spark中。有关详细信息，请参阅自定义收件人指南

接收机可靠性

基于可靠性可以有两种数据源。来源（如Kafka和Flume）允许传输的数据被确认。如果从这些可靠来源接收数据的系统正确地确认接收到的数据，则可以确保由于任何种类的故障而不会丢失任何数据。这导致两种接收器：

1.可靠的接收器 - 当数据已被接收并以复制形式存储在Spark中时，可靠的接收器正确地将确认发送到可靠的源。

2.不可靠的接收器 - 不可靠的接收器不向源发送确认。这可以用于不支持确认的源，甚至可以用于不希望或需要进行确认复杂度的可靠源。

“定制接收器指南”中讨论了如何编写可靠接收器的细节。

# DStreams转换

与RDD类似，转换允许修改来自输入DStream的数据。 DStreams支持常规Spark RDD上可用的许多转换。 一些常见的如下。

|  |  |
| --- | --- |
| Transformation | Meaning |
| **map**(*func*) | 通过传递源DStream的每个元素通过函数func返回一个新的DStream。 |
| **flatMap**(*func*) | 类似于map，但每个输入项可以映射到0个或更多的输出项。 |
| **filter**(*func*) | 通过仅选择func返回true的源DStream的记录来返回新的DStream。 |
| **repartition**(*numPartitions*) | 通过创建更多或更少的分区来更改此DStream中的并行级别。 |
| **union**(*otherStream*) | 返回一个新的DStream，它包含源DStream和otherDStream中元素的并集。 |
| **count**() | 通过计算源DStream的每个RDD中的元素数量，返回单元素RDD的新DStream。 |
| **reduce**(*func*) | 通过使用函数func（其接受两个参数并返回一个）聚合源DStream的每个RDD中的元素来返回单个元素RDD的新DStream。 该函数应该是关联的和可交换的，以便可以并行计算。 |
| **countByValue**() | 当调用类型为K的元素的DStream时，返回（K，Long）对的新DStream，其中每个键的值是其源DStream的每个RDD中的频率。 |
| **reduceByKey**(*func*, [*numTasks*]) | 当（K，V）对的DStream被调用时，返回一个新的DStream（K，V）对，其中使用给定的reduce函数聚合每个键的值。 注意：默认情况下，它使用Spark的默认并行任务数（2为本地模式，群集模式中的数字由config属性spark.default.parallelism确定）进行分组。 您可以传递可选的numTasks参数来设置不同数量的任务。 |
| **join**(*otherStream*, [*numTasks*]) | 当（K，V）和（K，W）对的两个DStream被调用时，返回一个新的（K，（V，W））对的DStream与每个键的所有元素对。 |
| **cogroup**(*otherStream*, [*numTasks*]) | 当调用（K，V）和（K，W）对的DStream时，返回一个新的DStream（K，Seq [V]，Seq [W]）元组。 |
| **transform**(*func*) | 通过对源DStream的每个RDD应用RDD到RDD函数来返回一个新的DStream。 这可以用于对DStream进行任意RDD操作。 |
| **updateStateByKey**(*func*) | 返回一个新的“状态”DStream，其中每个密钥的状态通过在密钥的先前状态应用给定的功能和密钥的新值来更新。 这可以用于维护每个密钥的任意状态数据。 |
|  |  |

其中一些转换值得深入讨论。

## UpdateStateByKey操作

updateStateByKey操作允许您在使用新的信息持续更新时保持任意状态。 要使用这个，你将不得不做两个步骤。

1.定义状态 - 状态可以是任意数据类型。

2.定义状态更新功能 - 使用函数指定如何使用上一个状态更新状态，并从输入流中指定新值。

在每个批处理中，Spark将对所有现有密钥应用状态更新功能，无论它们是否在批次中都有新数据。 如果update函数返回None，则键值对将被消除。

我们来举例说明一下。 假设你想保持在文本数据流中看到的每个单词的运行计数。 这里，运行计数是状态，它是一个整数。 我们将更新功能定义为：

* [**Scala**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_scala_6)
* [**Java**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_java_6)
* [**Python**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_python_6)

def updateFunction(newValues, runningCount):

if runningCount is None:

runningCount = 0

return sum(newValues, runningCount) # add the new values with the previous running count to get the new count

这应用于包含单词的DStream（例如，在前面的示例中包含（包括（1，对）对的对）DStream）。

runningCounts = pairs.updateStateByKey（updateFunction）

将为每个单词调用更新函数，其中newValues具有1的序列（来自（单词，1）对）和runningCount具有先前的计数。有关完整的Python代码，请查看stateful\_network\_wordcount.py的示例。

请注意，使用updateStateByKey需要配置检查点目录，这在检查点部分将详细讨论。

**变换操作**

变换操作（以及其变体如transformWith）允许将任意RDD到RDD函数应用于DStream。它可用于应用任何未在DStream API中公开的RDD操作。例如，将数据流中的每个批处理与其他数据集相结合的功能不会直接暴露在DStream API中。但是，您可以轻松地使用转换来做到这一点。这使得非常强大的可能性。例如，可以通过将输入数据流与预先计算的垃圾邮件信息（也可以使用Spark一起生成）进行实时数据清理，然后基于它进行过滤。

* [**Scala**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_scala_7)
* [**Java**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_java_7)
* [**Python**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_python_7)

spamInfoRDD = sc.pickleFile(...) # RDD containing spam information

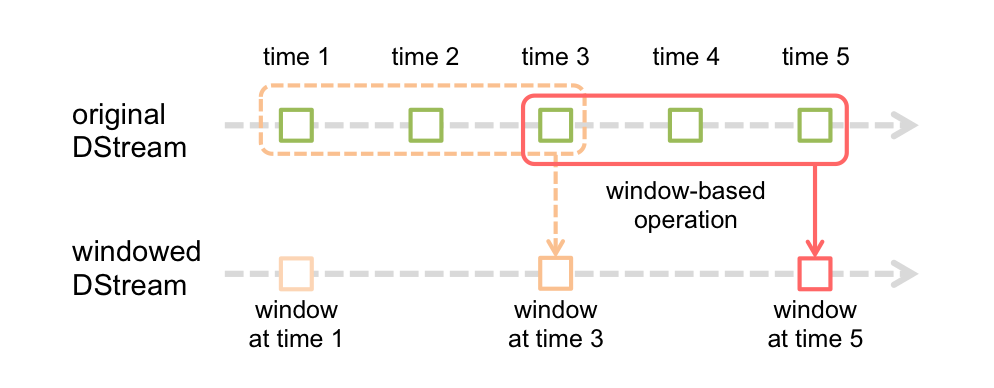
# join data stream with spam information to do data cleaning

cleanedDStream = wordCounts.transform(lambda rdd: rdd.join(spamInfoRDD).filter(...))

请注意，提供的函数在每个批次间隔中被调用。 这允许您进行时变RDD操作，即RDD操作，分区数，广播变量等可以在批次之间更改。

# 窗口操作

Spark Streaming还提供了窗口计算，允许您在数据的滑动窗口上应用转换。 下图说明了这个滑动窗口。



如图所示，每当窗口滑过源DStream时，落在窗口内的源RDD被组合起来并进行操作以产生窗口DStream的RDD。在这种具体情况下，操作应用于最近3个时间单位的数据，并以2个时间单位滑动。这表明任何窗口操作都需要指定两个参数。

窗口长度 - 窗口的持续时间（图中的3）。

滑动间隔 - 执行窗口操作的间隔（图中的2）。

这两个参数必须是源DStream的批间隔的倍数（图中的1）。

我们以一个例子来说明窗口操作。假设您希望通过在过去30秒的数据中每10秒产生一个字数来扩展上述示例。为此，我们必须在最近30秒的数据中对（word，1）对的对DStream应用reduceByKey操作。这是使用reduceByKeyAndWindow操作完成的。

* [**Scala**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_scala_8)
* [**Java**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_java_8)
* [**Python**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_python_8)

# Reduce last 30 seconds of data, every 10 seconds

windowedWordCounts = pairs.reduceByKeyAndWindow(lambda x, y: x + y, lambda x, y: x - y, 30, 10)

一些常见的窗口操作如下。 所有这些操作都采用上述两个参数 - windowLength和slideInterval。

|  |  |
| --- | --- |
| Transformation | Meaning |
| **window**(*windowLength*, *slideInterval*) | 返回基于源DStream的窗口批次计算的新DStream。 |
| **countByWindow**(*windowLength*, *slideInterval*) | 返回流中元素的滑动窗口数。 |
| **reduceByWindow**(*func*, *windowLength*, *slideInterval*) | 返回一个新的单元素流，通过使用func在滑动间隔内通过流中的元素聚合创建。 该函数应该是关联的和可交换的，以便它可以并行计算。 |
| **reduceByKeyAndWindow**(*func*, *windowLength*, *slideInterval*, [*numTasks*]) | 当调用（K，V）对的DStream时，返回一个新的DStream（K，V）对，其中每个键的值在滑动窗口中使用给定的减少函数功能进行聚合。 注意：默认情况下，它使用Spark的默认并行任务数（2为本地模式，群集模式中的数字由config属性spark.default.parallelism确定）进行分组。 您可以传递可选的numTasks参数来设置不同数量的任务。 |
| **reduceByKeyAndWindow**(*func*, *invFunc*, *windowLength*, *slideInterval*, [*numTasks*]) | 上述reduceByKeyAndWindow（）的更有效的版本，其中使用前一个窗口的reduce值递增地计算每个窗口的reduce值。 这是通过减少进入滑动窗口的新数据，以及“逆减”离开窗口的旧数据来完成的。 一个例子是当窗口滑动时，“添加”和“减”键的数量。 然而，它仅适用于“可逆减函数”，即具有相应“反减”功能的减函数（作为参数invFunc）。 像reduceByKeyAndWindow一样，reduce任务的数量可以通过可选参数进行配置。 请注意，使用此操作必须启用检查点。 |
| **countByValueAndWindow**(*windowLength*, *slideInterval*, [*numTasks*]) | 当（K，V）对的DStream被调用时，返回（K，Long）对的新DStream，其中每个键的值是其滑动窗口内的频率。 像reduceByKeyAndWindow一样，reduce任务的数量可以通过可选参数进行配置。 |
|  |  |

连接操作

最后，它值得强调的是，您可以轻松地在Spark Streaming中执行不同类型的连接。

# Stream-stream joins

流可以非常容易地与其他流连接。

* [**Scala**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_scala_9)
* [**Java**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_java_9)
* [**Python**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_python_9)

stream1 = ...

stream2 = ...

joinedStream = stream1.join(stream2)

这里，在每个批间隔中，由stream1生成的RDD将与stream2生成的RDD相连。 你也可以做leftOuterJoin，rightOuterJoin，fullOuterJoin。 此外，在流的窗口上进行联接通常是非常有用的。 这也很简单。

* [**Scala**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_scala_10)
* [**Java**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_java_10)
* [**Python**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_python_10)

windowedStream1 = stream1.window(20)

windowedStream2 = stream2.window(60)

joinedStream = windowedStream1.join(windowedStream2)

# Stream-dataset joins

这在前面已经说明了DStream.transform操作。 这是另一个加入窗口流与数据集的例子。

* [**Scala**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_scala_11)
* [**Java**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_java_11)
* [**Python**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_python_11)

dataset = ... # some RDD

windowedStream = stream.window(20)

joinedStream = windowedStream.transform(lambda rdd: rdd.join(dataset))

实际上，您也可以动态地更改要加入的数据集。 每个批次间隔评估提供给变换的功能，因此将使用数据集参考指向的当前数据集。

DStream转换的完整列表可在API文档中找到。 对于Scala API，请参阅DStream和PairDStreamFunction。 对于Java API，请参阅JavaDStream和JavaPairDStream。 对于Python API，请参阅DStream。

# DStreams的输出操作

输出操作允许将DStream的数据推送到外部系统，如数据库或文件系统。 由于输出操作实际上允许外部系统使用变换后的数据，所以它们触发所有DStream变换的实际执行（类似于RDD的动作）。 目前，定义了以下输出操作：

|  |  |
| --- | --- |
| Output Operation | Meaning |
| **print**() | 在运行流应用程序的驱动程序节点上的DStream中打印每批数据的前十个元素。 这对开发和调试很有用。  Python API这在Python API中称为pprint（）。 |
| **saveAsTextFiles**(*prefix*, [*suffix*]) | 将此DStream的内容另存为文本文件。 每个批处理间隔的文件名是根据前缀和后缀“prefix-TIME\_IN\_MS [.suffix]”生成的。 |
| **saveAsObjectFiles**(*prefix*, [*suffix*]) | 将此DStream的内容保存为序列化Java对象的SequenceFiles。 每个批处理间隔的文件名是根据前缀和后缀“prefix-TIME\_IN\_MS [.suffix]”生成的。  Python API这在Python API中是不可用的。 |
| **saveAsHadoopFiles**(*prefix*, [*suffix*]) | 将此DStream的内容另存为Hadoop文件。 每个批处理间隔的文件名是根据前缀和后缀“prefix-TIME\_IN\_MS [.suffix]”生成的。  Python API这在Python API中是不可用的。 |
| **foreachRDD**(*func*) | 对于从流生成的每个RDD应用函数func的最通用的输出运算符。 该功能应将每个RDD中的数据推送到外部系统，例如将RDD保存到文件，或将其通过网络写入数据库。 请注意，函数func在运行流应用程序的驱动程序进程中执行，通常会在其中具有将强制流式传输RDD的计算的RDD操作。 |
|  |  |

# 使用foreachRDD的设计模式

dstream.foreachRDD是一个强大的原语，允许将数据发送到外部系统。 然而，重要的是要正确有效地了解如何使用这个原语。 避免一些常见的错误如下。

通常向外部系统写入数据需要创建一个连接对象（例如与远程服务器的TCP连接），并使用它将数据发送到远程系统。 为此，开发人员可能无意中尝试在Spark驱动程序创建连接对象，然后尝试在Spark工作人员中使用它来将记录保存在RDD中。 例如（在Scala中）

* [**Scala**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_scala_12)
* [**Java**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_java_12)
* [**Python**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_python_12)

def sendRecord(rdd):

connection = createNewConnection() # executed at the driver

rdd.foreach(lambda record: connection.send(record))

connection.close()

dstream.foreachRDD(sendRecord)

这是不正确的，因为这需要将连接对象序列化并从驱动程序发送给工作人员。 这种连接对象很少在机器之间传输。 此错误可能会显示为序列化错误（连接对象不可序列化），初始化错误（连接对象需要在工作人员初始化）等。正确的解决方案是在工作人员创建连接对象。

但是，这可能会导致另一个常见的错误 - 为每个记录创建一个新的连接。 例如，

* [**Scala**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_scala_13)
* [**Java**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_java_13)
* [**Python**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_python_13)

def sendRecord(record):

connection = createNewConnection()

connection.send(record)

connection.close()

dstream.foreachRDD(lambda rdd: rdd.foreach(sendRecord))

通常，创建连接对象具有时间和资源开销。 因此，创建和销毁每个记录的连接对象可能会引起不必要的高开销，并可显着降低系统的总体吞吐量。 一个更好的解决方案是使用rdd.foreachPartition - 创建一个连接对象，并使用该连接在RDD分区中发送所有记录。

* [**Scala**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_scala_14)
* [**Java**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_java_14)
* [**Python**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_python_14)

def sendPartition(iter):

connection = createNewConnection()

for record in iter:

connection.send(record)

connection.close()

dstream.foreachRDD(lambda rdd: rdd.foreachPartition(sendPartition))

这样可以在多个记录上摊销连接创建开销。

最后，可以通过在多个RDD /批次之间重复使用连接对象来进一步优化。 可以维护连接对象的静态池，而不是将多个批次的RDD推送到外部系统时重新使用，从而进一步减少开销。

* [**Scala**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_scala_15)
* [**Java**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_java_15)
* [**Python**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_python_15)

def sendPartition(iter):

# ConnectionPool is a static, lazily initialized pool of connections

connection = ConnectionPool.getConnection()

for record in iter:

connection.send(record)

# return to the pool for future reuse

ConnectionPool.returnConnection(connection)

dstream.foreachRDD(lambda rdd: rdd.foreachPartition(sendPartition))

请注意，池中的连接应根据需要懒惰创建，如果不使用一段时间，则会超时。 这实现了最有效地将数据发送到外部系统。

## 其他要记住的要点：

DStreams通过输出操作执行懒惰，就像RDD由RDD操作懒惰地执行。 具体来说，DStream输出操作中的RDD动作强制处理接收到的数据。 因此，如果您的应用程序没有任何输出操作，或者具有dstream.foreachRDD（）等输出操作，而在其中没有任何RDD操作，则不会执行任何操作。 系统将简单地接收数据并将其丢弃。

默认情况下，输出操作是一次一个执行的。 它们按照它们在应用程序中定义的顺序执行。

# DataFrame和SQL操作

您可以轻松地在流数据上使用DataFrames和SQL操作。 您必须使用StreamingContext正在使用的SparkContext创建一个SparkSession。 此外，这必须做到这一点，可以在驱动程序故障时重新启动。 这是通过创建一个简单实例化的SparkSession单例实例来实现的。 这在以下示例中显示。 它修改早期的字数示例，以使用DataFrames和SQL生成单词计数。 将每个RDD转换为DataFrame，注册为临时表，然后使用SQL进行查询。

* [**Scala**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_scala_16)
* [**Java**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_java_16)
* [**Python**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_python_16)

# Lazily instantiated global instance of SparkSession

def getSparkSessionInstance(sparkConf):

if ("sparkSessionSingletonInstance" not in globals()):

globals()["sparkSessionSingletonInstance"] = SparkSession \

.builder \

.config(conf=sparkConf) \

.getOrCreate()

return globals()["sparkSessionSingletonInstance"]

...

# DataFrame operations inside your streaming program

words = ... # DStream of strings

def process(time, rdd):

print("========= %s =========" % str(time))

try:

# Get the singleton instance of SparkSession

spark = getSparkSessionInstance(rdd.context.getConf())

# Convert RDD[String] to RDD[Row] to DataFrame

rowRdd = rdd.map(lambda w: Row(word=w))

wordsDataFrame = spark.createDataFrame(rowRdd)

# Creates a temporary view using the DataFrame

wordsDataFrame.createOrReplaceTempView("words")

# Do word count on table using SQL and print it

wordCountsDataFrame = spark.sql("select word, count(\*) as total from words group by word")

wordCountsDataFrame.show()

except:

pass

words.foreachRDD(process)

See the full [source code](https://github.com/apache/spark/blob/v2.1.1/examples/src/main/python/streaming/sql_network_wordcount.py).

您还可以对来自不同线程的流数据（即与正在运行的StreamingContext异步）上定义的表运行SQL查询。 只需确保您将StreamingContext设置为记住足够数量的流数据，以便查询可以运行。 否则，不知道任何异步SQL查询的StreamingContext将在查询完成之前删除旧的流数据。 例如，如果要查询最后一个批次，但您的查询可能需要5分钟才能运行，然后调用streamingContext.remember（Minutes（5））（以Scala或其他语言的等价物）。

有关DataFrames的更多信息，请参阅DataFrames和SQL指南。

# MLlib操作

您还可以轻松使用MLlib提供的机器学习算法。 首先，有流式传输机器学习算法（例如，流式线性回归，流式传输KMeans等），其可以同时从流数据学习并将该模型应用于流数据。 除此之外，对于更大类的机器学习算法，您可以离线学习一个学习模型（即使用历史数据），然后将该模型在线应用于流数据。 有关详细信息，请参阅MLlib指南。

# 缓存/持久性

与RDD类似，DStreams还允许开发人员将流的数据保留在内存中。也就是说，在DStream上使用persist（）方法会自动将该DStream的每个RDD保留在内存中。如果DStream中的数据将被多次计算（例如，相同数据上的多个操作），这是有用的。对于基于窗口的操作，如reduceByWindow和reduceByKeyAndWindow以及基于状态的操作，如updateStateByKey，这是隐含的。因此，基于窗口的操作生成的DStream会自动保存在内存中，而开发人员不会调用persist（）。

对于通过网络接收数据（例如Kafka，Flume，套接字等）的输入流，默认持久性级别被设置为将数据复制到两个节点进行容错。

请注意，与RDD不同，DStream的默认持久性级别将数据序列化在内存中。这在“性能调优”部分进一步讨论。有关不同持久性级别的更多信息，请参见“Spark编程指南”。

# 检查点

流式应用程序必须全天候运行，因此必须能够适应与应用程序逻辑无关的故障（例如，系统故障，JVM崩溃等）。为了实现这一点，Spark Streaming需要检查足够的信息到容错存储系统，以便可以从故障中恢复。检查点有两种类型的数据。

元数据检查点 - 将定义流计算的信息保存到容错存储（如HDFS）中。这用于从运行流应用程序驱动程序的节点的故障中恢复（稍后详细讨论）。元数据包括：

配置 - 用于创建流应用程序的配置。

DStream操作 - 定义流应用程序的DStream操作集。

不完整的批次 - 工作排队但尚未完成的批次。

数据检查点 - 将生成的RDD保存到可靠的存储。在一些将多个批次的数据进行组合的状态变换中，这是必要的。在这种转换中，生成的RDD依赖于以前批次的RDD，这导致依赖链的长度随时间而增加。为了避免恢复时间的这种无限增加（与依赖关系链成比例），有状态转换的中间RDD被定期检查到可靠的存储（例如HDFS）以切断依赖关系链。

总而言之，元数据检查点主要用于从驱动程序故障中恢复，而数据或RDD检查点是必要的，即使是使用状态变换的基本功能。

何时启用检查点

对于具有以下任一要求的应用程序，必须启用检查点：

使用有状态转换 - 如果在应用程序中使用updateStateByKey或reduceByKeyAndWindow（具有反向功能），则必须提供检查点目录以允许定期的RDD检查点。

从运行应用程序的驱动程序的故障中恢复 - 元数据检查点用于使用进度信息进行恢复。

请注意，无需上述有状态转换的简单流媒体应用程序就可以运行，而不启用检查点。在这种情况下，驱动程序故障的恢复也将是部分的（一些接收但未处理的数据可能会丢失）。这通常是可以接受的，许多运行Spark Streaming应用程序。未来对非Hadoop环境的支持预计会有所改善。

如何配置检查点

可以通过在保存检查点信息的容错，可靠的文件系统（例如，HDFS，S3等）中设置目录来启用检查点。这是通过使用streamingContext.checkpoint（checkpointDirectory）完成的。这将允许您使用上述有状态转换。此外，如果要使应用程序从驱动程序故障中恢复，您应该重写流式应用程序以具有以下行为。

当程序第一次启动时，它将创建一个新的StreamingContext，设置所有流，然后调用start（）。

当程序在失败后重新启动时，它将从checkpoint目录中的检查点数据重新创建一个StreamingContext。

* [**Scala**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_scala_17)
* [**Java**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_java_17)
* [**Python**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_python_17)

This behavior is made simple by using StreamingContext.getOrCreate. This is used as follows.

# Function to create and setup a new StreamingContext

def functionToCreateContext():

sc = SparkContext(...) # new context

ssc = StreamingContext(...)

lines = ssc.socketTextStream(...) # create DStreams

...

ssc.checkpoint(checkpointDirectory) # set checkpoint directory

return ssc

# Get StreamingContext from checkpoint data or create a new one

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()

如果checkpointDirectory存在，那么将从检查点数据重建上下文。 如果目录不存在（即第一次运行），则将调用函数functionToCreateContext来创建新的上下文并设置DStream。 请参阅Python示例recoverable\_network\_wordcount.py。 此示例将网络数据的字数添加到文件中。

您还可以从检查点数据显式创建一个StreamingContext，并使用StreamingContext.getOrCreate（checkpointDirectory，None）开始计算。

除了使用getOrCreate之外，还需要确保在失败时自动重新启动驱动程序进程。这只能由用于运行应用程序的部署基础架构完成。这在“部署”部分进一步讨论。

请注意，RDD的检查点会导致保存到可靠存储的成本。这可能会导致RDD得到检查点的批次的处理时间增加。因此，需要仔细设置检查点的间隔。在小批量（例如1秒）的情况下，每个批次的检查点可能会显着降低操作吞吐量。相反，检查点太少会导致谱系和任务大小增长，这可能会产生不利影响。对于需要RDD检查点的状态转换，默认间隔是至少10秒的批间隔的倍数。它可以通过使用dstream.checkpoint（checkpointInterval）进行设置。通常，DStream的5到10个滑动间隔的检查点间隔是一个很好的设置。

# 累加器，广播变量和检查点

在Spark Streaming中，无法从检查点恢复累加器和广播变量。 如果启用检查点并使用累加器或广播变量，则必须为累加器和广播变量创建延迟实例化的单例实例，以便在驱动程序重新启动失败后重新实例化。 这在以下示例中显示。

* [**Scala**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_scala_18)
* [**Java**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_java_18)
* [**Python**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_python_18)

def getWordBlacklist(sparkContext):

if ("wordBlacklist" not in globals()):

globals()["wordBlacklist"] = sparkContext.broadcast(["a", "b", "c"])

return globals()["wordBlacklist"]

def getDroppedWordsCounter(sparkContext):

if ("droppedWordsCounter" not in globals()):

globals()["droppedWordsCounter"] = sparkContext.accumulator(0)

return globals()["droppedWordsCounter"]

def echo(time, rdd):

# Get or register the blacklist Broadcast

blacklist = getWordBlacklist(rdd.context)

# Get or register the droppedWordsCounter Accumulator

droppedWordsCounter = getDroppedWordsCounter(rdd.context)

# Use blacklist to drop words and use droppedWordsCounter to count them

def filterFunc(wordCount):

if wordCount[0] in blacklist.value:

droppedWordsCounter.add(wordCount[1])

False

else:

True

counts = "Counts at time %s %s" % (time, rdd.filter(filterFunc).collect())

wordCounts.foreachRDD(echo)

See the full [source code](https://github.com/apache/spark/blob/v2.1.1/examples/src/main/python/streaming/recoverable_network_wordcount.py).

# 部署应用程序

本节讨论部署Spark Streaming应用程序的步骤。

## 要求

要运行Spark Streaming应用程序，您需要具备以下功能。

集群管理器集群 - 这是任何Spark应用程序的一般要求，并在部署指南中详细讨论。

打包应用程序JAR - 您必须将流式应用程序编译为JAR。如果您使用spark-submit启动应用程序，则不需要在JAR中提供Spark和Spark Streaming。但是，如果您的应用程序使用高级源（例如Kafka，Flume），那么您将必须将他们链接的额外工件及其依赖项打包在用于部署应用程序的JAR中。例如，使用KafkaUtils的应用程序必须在应用程序JAR中包含spark-streaming-kafka-0-8\_2.11及其所有传递依赖项。

为执行程序配置足够的内存 - 由于接收到的数据必须存储在内存中，所以执行程序必须配置足够的内存来保存接收到的数据。请注意，如果您正在进行10分钟的窗口操作，系统必须在内存中保留至少10分钟的数据。因此，应用程序的内存要求取决于其中使用的操作。

配置检查点 - 如果流应用程序需要它，则Hadoop API兼容的容错存储（例如HDFS，S3等）中的目录必须配置为检查点目录，并且流程应用程序以检查点信息的方式写入用于故障恢复。有关详细信息，请参阅检查点部分。

配置应用程序驱动程序的自动重新启动 - 要自动从驱动程序故障中恢复，用于运行流应用程序的部署基础架构必须监视驱动程序进程，如果失败，则重新启动驱动程序。不同的集群管理器有不同的工具来实现这一点。

Spark独立 - 可以提交Spark应用程序驱动程序以在Spark Standalone集群中运行（请参阅集群部署模式），即应用程序驱动程序本身在其中一个工作节点上运行。此外，可以指示独立的群集管理器来监督驱动程序，如果驱动程序由于非零退出代码而失败，或由于运行驱动程序的节点发生故障，则重新启动它。有关详细信息，请参阅“Spark Standalone”指南中的群集模式和监督。

YARN - 纱线支持类似的机制来自动重新启动应用程序。有关详细信息，请参阅YARN文档。

Mesos - 马拉松已被用来实现这一点与Mesos。

配置写入日志 - 自Spark 1.2以来，我们引入了写入前端日志来实现强大的容错保证。如果启用，则从接收器接收的所有数据都将写入配置检查点目录中的写入日志。这可以防止驱动程序恢复时的数据丢失，从而确保零数据丢失（在容错语义部分中详细讨论）。可以通过将配置参数spark.streaming.receiver.writeAheadLog.enable设置为true来实现。然而，这些更强的语义可能以单个接收机的接收吞吐量为代价。这可以通过并行运行更多的接收器来更正，以增加总吞吐量。另外，建议在启用写入日志时，在日志已经存储在复制的存储系统中时，禁用在Spark中接收到的数据的复制。这可以通过将输入流的存储级别设置为StorageLevel.MEMORY\_AND\_DISK\_SER来完成。使用S3（或任何不支持刷新的文件系统）写入日志时，请记住启用spark.streaming.driver.writeAheadLog.closeFileAfterWrite和spark.streaming.receiver.writeAheadLog.closeFileAfterWrite。有关详细信息，请参阅Spark Streaming配置。请注意，启用I / O加密时，Spark不会将写入写入日志的数据加密。如果需要加密写入日志数据，则应将其存储在本地支持加密的文件系统中。

设置最大接收速率 - 如果集群资源不够大，流媒体应用程序能够像接收到的那样快速处理数据，则可以通过设置记录/秒的最大速率限制来对接收方进行速率限制。请参阅接收器的spark.streaming.receiver.maxRate和用于Direct Kafka方法的spark.streaming.kafka.maxRatePerPartition的配置参数。在Spark 1.5中，我们引入了一个称为背压的功能，无需设置此速率限制，因为Spark Streaming会自动计算速率限制，并在处理条件发生变化时动态调整速率限制。可以通过将配置参数spark.streaming.backpressure.enabled设置为true来启用此背压。

## 升级应用程序代码

如果一个正在运行的Spark Streaming应用程序需要使用新的应用程序代码进行升级，然后有两种可能的机制。升级的Spark Streaming应用程序与现有应用程序并行启动并运行。一旦新的（接收与旧的相同的数据）已经被加热，并且准备好黄金时段，旧的可以被关闭。请注意，这可以用于支持将数据发送到两个目的地（即早期和升级的应用程序）的数据源。现有应用程序正常关闭（请参阅StreamingContext.stop（...）或JavaStreamingContext.stop（..） 。）用于正常关机选项），确保在关闭之前已经接收的数据被完全处理。然后可以启动升级的应用程序，这将从较早的应用程序停止的同一点开始处理。请注意，只有在支持源端缓存的输入源（如Kafka和Flume）时才可以进行此操作，因为数据需要在先前的应用程序关闭和升级的应用程序尚未启动时进行缓冲。从升级前代码的早期检查点信息重新启动不能完成。检查点信息基本上包含序列化的Scala / Java / Python对象，并尝试使用新的修改的类反序列化对象可能会导致错误。在这种情况下，可以使用不同的检查点目录启动升级的应用程序，也可以删除以前的检查点目录。监视ApplicationsBeyond Spark的监视功能，还有Spark Streaming特有的附加功能。当使用StreamingContext时，Spark Web UI会显示一个附加的Streaming选项卡，显示有关运行的接收器（接收器是否活动，接收到的记录数量，接收器错误等）和完成的批次（批处理时间，排队延迟等）的统计信息）。这可以用于监视流应用的进度。Web UI中的以下两个指标特别重要：处理时间 - 处理每批数据的时间。计划延迟 - 批处理在队列中等待处理先前批次的时间。如果批量处理时间始终超过批次间隔和/或排队延迟不断增加，则表示系统无法以批量生产的速度处理批次并落后。在这种情况下，请考虑减少批处理时间。还可以使用StreamingListener接口监视Spark Streaming程序的进度，从而允许您获取接收器状态和处理时间。请注意，这是一个开发人员API，将来可能会有所改进（即更多的信息报告）。

# 性能调优

在群集上的Spark Streaming应用程序中获得最佳性能需要进行一些调整。本节介绍了可以调整以提高应用程序性能的许多参数和配置。在高层次上，你需要考虑两件事情：

1.通过有效利用集群资源，减少每批数据的处理时间。

2.设置正确的批量大小，使得批量的数据可以像接收到的那样快速处理（即数据处理与数据摄取保持一致）。

减少批处理时间

Spark中可以进行一些优化，以最小化每个批处理的处理时间。这些已在“调音指南”中详细讨论过。本节重点介绍一些最重要的。

数据接收并行度水平

通过网络接收数据（如Kafka，Flume，插座等）需要将数据反序列化并存储在Spark中。如果数据接收成为系统的瓶颈，则考虑并行化数据接收。请注意，每个输入DStream都会创建一个接收单个数据流的单个接收器（在工作机上运行）。因此，可以通过创建多个输入DStream并配置它们以从源接收数据流的不同分区来实现接收多个数据流。例如，接收两个数据主题的单个Kafka输入DStream可以分为两个Kafka输入流，每个只接收一个主题。这将运行两个接收器，允许并行接收数据，从而增加总体吞吐量。这些多个DStream可以联合起来创建一个单独的DStream。然后，应用于单个输入DStream的转换可以应用于统一流。这样做完成如下。

* [**Scala**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_scala_19)
* [**Java**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_java_19)
* [**Python**](http://spark.apache.org/docs/latest/streaming-programming-guide.html#tab_python_19)

numStreams = 5

kafkaStreams = [KafkaUtils.createStream(...) for \_ in range (numStreams)]

unifiedStream = streamingContext.union(\*kafkaStreams)

unifiedStream.pprint()

应考虑的另一个参数是接收器的块间隔，由块配置参数spark.streaming.blockInterval决定。对于大多数接收机，接收到的数据在存储在Spark的内存中之前将它们合并成一组数据。每个批次中的块数确定了将用于以类似地图的转换处理接收到的数据的任务数。每个接收器每批次的任务数量将是大约（批间隔/块间隔）。例如，200 ms的块间隔将每2秒批次创建10个任务。如果任务数量太少（即每台计算机的核心数量少），则所有可用内核都不会被用于处理数据，所以效率会降低。要增加给定批间隔的任务数量，请减少块间隔。但是，建议的块间隔的最小值约为50 ms，低于此值的任务启动开销可能是一个问题。

使用多个输入流/接收器接收数据的替代方法是明确地重新分配输入数据流（使用inputStream.repartition（<数量的分区>））。在进一步处理之前，会将接收到的批次数据分布在集群中指定数量的计算机上。

# 数据处理中的并行度水平

如果在任何计算阶段中使用的并行任务数量不够高，则集群资源可能未得到充分利用。例如，对于诸如reduceByKey和reduceByKeyAndWindow之类的分布式还原操作，默认并行任务数由spark.default.parallelism配置属性控制。您可以将并行级别作为参数传递（请参阅PairDStreamFunctions文档），或者将spark.default.parallelism配置属性设置为更改默认值。

## 数据序列化

通过调整序列化格式可以减少数据串行化的开销。在流式传输的情况下，有两种类型的数据被序列化。

输入数据：默认情况下，通过接收器接收的输入数据通过StorageLevel.MEMORY\_AND\_DISK\_SER\_2存储在执行程序的存储器中。也就是说，将数据序列化为字节以减少GC开销，并复制以容忍执行器故障。此外，数据首先保存在内存中，并且只有在内存不足以容纳流计算所需的所有输入数据时，才会溢出到磁盘。这个序列化显然有开销 - 接收器必须反序列化接收的数据，并使用Spark的序列化格式重新序列化它。

通过流操作生成的持久RDD：通过流式计算生成的RDD可能会持久存储在内存中。例如，窗口操作会将数据保留在内存中，因为它们将被多次处理。但是，与StorageLevel.MEMORY\_ONLY的Spark Core默认情况不同，默认情况下，通过流式计算生成的持久化RDD将通过StorageLevel.MEMORY\_ONLY\_SER（即序列化）进行持久化，以最大程度降低GC开销。

在这两种情况下，使用Kryo序列化可以减少CPU和内存开销。有关详细信息，请参阅“Spark调整指南”。对于Kryo，请考虑注册自定义类，并禁用对象引用跟踪（请参阅“配置指南”中的Kryo相关配置）。

在流媒体应用程序需要保留的数据量不大的特定情况下，可以将数据（两种类型）作为反序列化对象持久化，而不会导致过多的GC开销。例如，如果您使用几秒钟的批处理间隔并且没有窗口操作，那么可以通过明确地相应地设置存储级别来尝试禁用持久化数据中的序列化。这将减少由于序列化引起的CPU开销，潜在地提高性能，而不需要太多的GC开销。

## 任务启动开销

如果每秒启动的任务数量很高（例如，每秒50个或更多），那么将任务发送到从站的开销可能很大，并且将难以实现亚秒级延迟。可以通过以下更改减少开销：

执行模式：在独立模式或粗粒度Mesos模式下运行Spark可以导致比细粒度的Mesos模式更好的任务启动时间。有关详细信息，请参阅“在Mesos上运行”指南。

这些更改可能会将批处理时间缩短100秒，从而允许次秒批次大小可行。

## 设置正确的批次间隔

对于运行在群集上的Spark Streaming应用程序，该系统应该能够像收到的那样快速处理数据。换句话说，批处理的数据应该像生成一样快。通过监视流媒体用户界面中的处理时间可以找到应用程序是否正确，批处理时间应小于批处理间隔。

根据流式传输计算的性质，所使用的批次间隔可能会对应用程序在一组固定的集群资源上持续的数据速率产生重大影响。例如，让我们考虑早期的WordCountNetwork示例。对于特定数据速率，系统可能能够每2秒跟踪报告字数（即，2秒的批次间隔），但不能每500毫秒。因此，需要设置批次间隔，以便可以维持生产中的预期数据速率。

为您的应用程序找出正确的批量大小的好方法是以保守的批次间隔（例如5-10秒）和低数据速率进行测试。要验证系统是否能够跟上数据速率，您可以检查每个处理批次遇到的端到端延迟的值（要查找Spark驱动程序log4j日志中的“总延迟”，或者使用StreamingListener接口）。如果延迟保持与批量大小相当，那么系统是稳定的。否则，如果延迟持续增加，则意味着系统不能跟上，因此不稳定。一旦你有一个稳定的配置的想法，你可以尝试增加数据速率和/或减少批量大小。注意，只要延迟降低到低值（即，小于批量大小），由于临时数据速率增加导致的延迟的瞬间增加可能是正常的。

## 内存调优

调整Spark应用程序的内存使用情况和GC行为已在“调优指南”中进行了详细的讨论。强烈建议您阅读。在本节中，我们将在Spark Streaming应用程序的上下文中专门讨论一些调整参数。

Spark Streaming应用程序所需的集群内存量在很大程度上取决于所使用的转换类型。例如，如果要在最近10分钟的数据中使用窗口操作，那么您的集群应该有足够的内存来容纳内存中10分钟的数据。或者如果要使用带有大量密钥的updateStateByKey，则必需的内存将会很高。相反，如果你想做一个简单的map-filter-store操作，那么所需的内存将会很低。

一般来说，由于通过接收器接收到的数据与StorageLevel.MEMORY\_AND\_DISK\_SER\_2一起存储，所以不适合内存的数据将会溢出到磁盘上。这可能会降低流媒体应用程序的性能，因此建议您提供足够的流量应用程序所需的内存。最好尝试一下小尺寸的内存使用量，并相应地进行估算。

内存调优的另一个方面是垃圾收集。对于需要低延迟的流式应用程序，由JVM垃圾收集引起的大量暂停是不希望的。

有几个参数可以帮助您调整内存使用量和GC开销：

DStreams的持续级别：如前面在Data Serialization部分所述，输入数据和RDD默认保持为序列化字节。与反序列化持久性相比，这减少了内存使用量和GC开销。启用Kryo序列化进一步减少了序列化大小和内存使用。可以通过压缩（参见Spark配置spark.rdd.compress）实现内存使用的进一步减少，而CPU时间是代价的。

清除旧数据：默认情况下，DStream转换生成的所有输入数据和持久的RDD都将被自动清除。 Spark Streaming决定何时根据所使用的转换来清除数据。例如，如果您使用10分钟的窗口操作，则Spark Streaming将保留最近10分钟的数据，并主动丢弃旧数据。通过设置streamingContext.remember，数据可以保持更长的持续时间（例如交互式地查询较旧的数据）。

CMS垃圾收集器：强烈推荐使用并行标记和扫描GC，以保持GC相关的暂停始终如一。即使并行GC已知降低系统的整体处理吞吐量，仍然建议使用它来实现更一致的批处理时间。确保在驱动程序（使用--driver-java-options-spark-submit）和执行程序（使用Spark配置spark.executor.extraJavaOptions）中设置CMS GC。

其他提示：为了进一步降低GC开销，以下是一些更多的提示。

使用OFF\_HEAP存储级别保持RDD。在Spark编程指南中查看更多详细信息。

使用更小的堆大小的执行器。这将降低每个JVM堆内的GC压力。

## 要记住的要点：

DStream与单个接收器相关联。为了获得读取并行性，需要创建多个接收机，即多个DStream。接收器在执行器中运行。它占据一个核心。确保在接收机插槽预订后有足够的内核进行处理，即spark.cores.max应考虑接收机插槽。接收器以循环方式分配给执行者。

当从流源接收数据时，接收器创建数据块。每个blockInterval毫秒生成一个新的数据块。在N = batchInterval / blockInterval的batchInterval期间创建N个数据块。这些块由当前执行器的BlockManager分发给其他执行程序的块管理器。之后，在驱动程序上运行的网络输入跟踪器通知块位置进行进一步处理。

在驱动程序中为在batchInterval期间创建的块创建一个RDD。在batchInterval期间生成的块是RDD的分区。每个分区都是火花中的任务。 blockInterval == batchinterval意味着创建单个分区，并且可能是在本地进行处理。

除非非本地计划启动，否则块上的映射任务将在执行程序（接收块，另一个块被复制）中进行处理，而不管块间隔如何，具有更大的块间隔意味着更大的块。 spark.locality.wait的高值增加了处理本地节点上的块的机会。在这两个参数之间需要找到平衡，以确保更大的块在本地被处理。

而不是依赖于batchInterval和blockInterval，你可以定义通过调用inputDstream.repartition（n）来计算分区数。这样会随机地重新组织RDD中的数据，以创建n个分区。是的，为了更大的并行性。虽然以洗牌为代价。 RDD的处理由司机的调度员作为一项工作安排。在给定的时间点，只有一个工作是活跃的。因此，如果一个作业正在执行其他作业排队。如果您有两个dstream，将会形成两个RDD，并且将创建两个将被安排在另一个之后的作业。为了避免这种情况，你可以联合两个dstream。这将确保为dstream的两个RDD形成单个unionRDD。这个unionRDD被认为是一个单一的工作。然而，RDD的分区不受影响。如果批处理时间超过批量间隔，那么显然，接收器的内存将开始填充，并将最终导致抛出异常（最可能的是BlockNotFoundException）。目前没有办法暂停接收机。使用SparkConf配置spark.streaming.receiver.maxRate，接收机的速率可以受到限制。

# 容错语义

在本节中，我们将讨论Spark Streaming应用程序在发生故障时的行为。

背景

要了解Spark Streaming提供的语义，让我们记住Spark的RDD的基本容错语义。

RDD是一个不可变的，确定性可重新计算的分布式数据集。每个RDD记住在容错输入数据集上使用的确定性操作的谱系来创建它。

2.如果RDD的任何分区由于工作节点故障而丢失，则可以使用业务流程从原始容错数据集重新计算该分区。

假设所有RDD转换都是确定性的，则最终变换的RDD中的数据将始终是相同的，而不管Spark集群中的故障如何。

Spark运行在容错文件系统（如HDFS或S3）中的数据上。因此，从容错数据生成的所有RDD也是容错的。但是，Spark Streaming并不是这样，因为在大多数情况下，数据通过网络接收（除非使用fileStream）。为了为所有生成的RDD实现相同的容错属性，接收到的数据将在集群中的工作节点中的多个Spark执行器之间复制（默认复制因子为2）。这导致系统中需要在发生故障时恢复的两种数据：

1.数据接收和复制 - 这个数据在单个工作节点的故障中幸存下来，因为它的副本存在于其他节点之上。

2.数据收到但缓冲进行复制 - 由于不复制，恢复此数据的唯一方法是从源中重新获取。

此外，我们应该关注的有两种失败：

1.工作节点失败 - 运行执行程序的任何工作节点都可能会失败，并且这些节点上的所有内存中的数据都将丢失。如果任何接收器在故障节点上运行，则其缓冲的数据将丢失。

2.驱动程序节点的失败 - 如果运行Spark Streaming应用程序的驱动程序节点出现故障，则显然SparkContext将丢失，并且所有具有内存中数据的执行程序都将丢失。

有了这个基础知识，让我们了解Spark Streaming的容错语义。

## 定义

流系统的语义往往是通过系统可以处理每个记录的次数来捕获的。系统可以在所有可能的操作条件下提供三种类型的保证（尽管有故障等）

1.最多一次：每个记录将被处理一次或根本不处理。

2.至少一次：每个记录将被处理一次或多次。这比一次更强大，因为它确保不会丢失任何数据。但可能有重复。

3.完全一次：每个记录将被精确处理一次 - 不会丢失任何数据，也不会多次处理数据。这显然是三者的最强保证。

## 基本语义

在任何流处理系统中，广义上说，处理数据有三个步骤。

1.接收数据：使用接收器或其他方式从数据源接收数据。

2.转换数据：使用DStream和RDD转换来转换接收到的数据。

3.推出数据：将最终转换的数据推送到外部系统，如文件系统，数据库，仪表板等。

如果流式应用程序必须实现端到端的一次性保证，则每个步骤都必须提供一次完整的保证。也就是说，每个记录必须被精确地接收一次，转换完成一次，并被推送到下游系统一次。让我们在Spark Streaming的上下文中了解这些步骤的语义。

1.接收数据：不同的输入源提供不同的保证。这将在下一小节中详细讨论。

2.转换数据：由于RDD提供的保证，所有已收到的数据将被处理完全一次。即使有故障，只要接收到的输入数据可访问，最终变换的RDD将始终具有相同的内容。

3.推出数据：默认情况下，输出操作至少保留一次语义，因为它取决于输出操作的类型（幂等于或不符合）和下游系统的语义（支持交易）。但用户可以实现自己的事务机制来实现一次性语义。这将在本节后面的更详细的讨论。

## 接收数据语义

不同的输入源提供不同的保证，从至少一次到一次。阅读更多详情。

使用文件

如果所有输入数据已经存在于像HDFS这样的容错文件系统中，Spark Streaming总是可以从任何故障中恢复并处理所有数据。这给出了一次一次的语义，意味着无论什么失败，所有的数据将被精确地处理一次。

使用基于接收器的数据源

对于基于接收器的输入源，容错语义取决于故障场景和接收器的类型。正如我们之前讨论的，有两种类型的接收器：

1.可靠的接收机 - 这些接收机只有在确保接收到的数据已被复制后才能确认可靠的源。如果这样的接收机失败，则源将不会接收缓冲（未复制）数据的确认。因此，如果接收机重新启动，源将重新发送数据，并且不会由于故障而丢失数据。

2.不可靠的接收器 - 这样的接收器不发送确认，因此在由于工作人员或驱动器故障而失败时可能会丢失数据。

取决于使用什么类型的接收器，我们实现以下语义。如果工作节点出现故障，则可靠的接收器没有数据丢失。使用不可靠的接收器，收到但未复制的数据可能会丢失。如果驱动程序节点出现故障，那么除了这些损失之外，在内存中接收和复制的所有过去的数据都将丢失。这将影响有状态转换的结果。

为了避免过去收到的数据丢失，Spark 1.2引入了写入提前日志，将接收到的数据保存到容错存储中。通过启用写入日志和可靠的接收器，数据丢失为零。在语义方面，它至少提供一次保证。

下表总结了失败的语义：

|  |  |  |
| --- | --- | --- |
| Deployment Scenario | Worker Failure | Driver Failure |
| *Spark 1.1 or earlier,* OR  *Spark 1.2 or later without write ahead logs* | 缓冲数据丢失与不可靠的接收器  零数据丢失与可靠的接收器  至少一次语义 | 缓冲数据丢失与不可靠的接收器  所有接收器丢失过去的数据  未定义的语义 |
| *Spark 1.2 or later with write ahead logs* | 零数据丢失与可靠的接收器  至少一次语义 | 零数据丢失与可靠的接收器和文件  至少一次语义 |
|  |  |  |

使用Kafka Direct API

在Spark 1.3中，我们引入了一个新的Kafka Direct API，可以确保所有的Kafka数据都被Spark Streaming一次收到。 与此同时，如果您实现一次性输出操作，您可以实现端到端的一次性保证。 在“卡夫卡整合指南”中进一步讨论了这种方法。

## 输出操作语义

输出操作（如foreachRDD）至少具有一次语义，即在发生工作失败时，转换后的数据可能会多次写入外部实体。虽然使用saveAs \*\*\*文件操作（因为文件将被相同的数据简单地覆盖）保存到文件系统是可以接受的，但是为了实现一次一次的语义，可能需要额外的努力。有两种方法。

幂等更新：多次尝试总是写入相同的数据。例如，saveAs \*\*\*文件总是将相同的数据写入生成的文件。

事务更新：所有更新都是事务性的，以便更新完全按原子进行。执行此操作的一种方法如下。

使用批处理时间（在foreachRDD中可用）和RDD的分区索引来创建标识符。该标识符唯一地标识流应用中的Blob数据。

使用标识符事务地更新外部系统（即，一次，原子地）。也就是说，如果标识符尚未提交，则以原子方式提交分区数据和标识符。否则，如果已经提交，请跳过更新。

dstream.foreachRDD { (rdd, time) =>

rdd.foreachPartition { partitionIterator =>

val partitionId = TaskContext.get.partitionId()

val uniqueId = generateUniqueId(time.milliseconds, partitionId)

// use this uniqueId to transactionally commit the data in partitionIterator

}

}

# Where to Go from Here

* Additional guides
  + [Kafka Integration Guide](http://spark.apache.org/docs/latest/streaming-kafka-integration.html)
  + [Kinesis Integration Guide](http://spark.apache.org/docs/latest/streaming-kinesis-integration.html)
  + [Custom Receiver Guide](http://spark.apache.org/docs/latest/streaming-custom-receivers.html)
* Third-party DStream data sources can be found in [Third Party Projects](http://spark.apache.org/third-party-projects.html)
* API documentation
  + Scala docs
    - [StreamingContext](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.streaming.StreamingContext) and [DStream](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.streaming.dstream.DStream)
    - [KafkaUtils](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.streaming.kafka.KafkaUtils$), [FlumeUtils](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.streaming.flume.FlumeUtils$), [KinesisUtils](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.streaming.kinesis.KinesisUtils$),
  + Java docs
    - [JavaStreamingContext](http://spark.apache.org/docs/latest/api/java/index.html?org/apache/spark/streaming/api/java/JavaStreamingContext.html), [JavaDStream](http://spark.apache.org/docs/latest/api/java/index.html?org/apache/spark/streaming/api/java/JavaDStream.html) and [JavaPairDStream](http://spark.apache.org/docs/latest/api/java/index.html?org/apache/spark/streaming/api/java/JavaPairDStream.html)
    - [KafkaUtils](http://spark.apache.org/docs/latest/api/java/index.html?org/apache/spark/streaming/kafka/KafkaUtils.html), [FlumeUtils](http://spark.apache.org/docs/latest/api/java/index.html?org/apache/spark/streaming/flume/FlumeUtils.html), [KinesisUtils](http://spark.apache.org/docs/latest/api/java/index.html?org/apache/spark/streaming/kinesis/KinesisUtils.html)
  + Python docs
    - [StreamingContext](http://spark.apache.org/docs/latest/api/python/pyspark.streaming.html#pyspark.streaming.StreamingContext) and [DStream](http://spark.apache.org/docs/latest/api/python/pyspark.streaming.html#pyspark.streaming.DStream)
    - [KafkaUtils](http://spark.apache.org/docs/latest/api/python/pyspark.streaming.html#pyspark.streaming.kafka.KafkaUtils)
* More examples in [Scala](https://github.com/apache/spark/tree/master/examples/src/main/scala/org/apache/spark/examples/streaming) and [Java](https://github.com/apache/spark/tree/master/examples/src/main/java/org/apache/spark/examples/streaming) and [Python](https://github.com/apache/spark/tree/master/examples/src/main/python/streaming)
* [Paper](http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-259.pdf) and [video](http://youtu.be/g171ndOHgJ0) describing Spark Streaming.