Spark Streaming 原理介绍

目 录

Spark Streaming 原理介绍

1 Spark Streaming 简介

1.1 概述

Spark Streaming 是 Spark 核心 API 的一个扩展，可以实现高吞吐量的、具备容错机制的实时流数据的处理。支持从多种数据源获取数据，包括 Kafk、Flume、Twitter、ZeroMQ、Kinesis 以及 TCP sockets，从数据源获取数据之后，可以使用诸如 map、reduce、join 和 window 等高级函数进行复杂算法的处理。最后还可以将处理结果存储到文件系统，数据库和现场仪表盘。在“One Stack rule them all”的基础上，还可以使用 Spark 的其他子框架，如集群学习、图计算等，对流数据进行处理。

Spark Streaming 处理的数据流图：

Spark 的各个子框架，都是基于核心 Spark 的，Spark Streaming 在内部的处理机制是，接收实时流的数据，并根据一定的时间间隔拆分成一批批的数据，然后通过 Spark Engine 处理这些批数据，最终得到处理后的一批批结果数据。

对应的批数据，在 Spark 内核对应一个 RDD 实例，因此，对应流数据的 DStream 可以看成是一组 RDDs，即 RDD 的一个序列。通俗点理解的话，在流数据分成一批一批后，通过一个先进先出的队列，然后 Spark Engine 从该队列中依次取出一个个批数据，把批数据封装成一个 RDD，然后进行处理，这是一个典型的生产者消费者模型，对应的就有生产者消费者模型的问题，即如何协调生产速率和消费速率。

1.2 术语定义

- **离散流 (discretized stream) 或 DStream**：这是 Spark Streaming 对内部持续的实时数据流的抽象描述，即我们处理的一个实时数据流，在 Spark Streaming 中对应于一个

DStream 实例。

- **批数据（batch data）**：这是化整为零的第一步，将实时流数据以时间片为单位进行分批，将流处理转化为时间片数据的批处理。随着持续时间的推移，这些处理结果就形成了对应的结果数据流了。
- **时间片或批处理时间间隔（ batch interval）**：这是人为地对流数据进行定量的标准，以时间片作为我们拆分流数据的依据。一个时间片的数据对应一个 RDD 实例。
- **窗口长度（window length）**：一个窗口覆盖的流数据的时间长度。必须是批处理时间间隔的倍数，
- **滑动时间间隔**：前一个窗口到后一个窗口所经过的时间长度。必须是批处理时间间隔的倍数
- **Input DStream**：一个 input DStream 是一个特殊的 DStream，将 Spark Streaming 连接到一个外部数据源来读取数据。

1.3 Storm 与 Spark Streming 比较

- **处理模型以及延迟**

虽然两框架都提供了可扩展性(scalability)和可容错性(fault tolerance)，但是它们的处理模型从根本上说是不一样的。Storm 可以实现亚秒级时延的处理，而每次只处理一条 event，而 Spark Streaming 可以在一个短暂的时间窗口里面处理多条(batches)Event。所以说 Storm 可以实现亚秒级时延的处理，而 Spark Streaming 则有一定的时延。

- **容错和数据保证**

然而两者的代价都是容错时候的数据保证，Spark Streaming 的容错为有状态的计算提供了更好的支持。在 Storm 中，每条记录在系统的移动过程中都需要被标记跟踪，所以 Storm 只能保证每条记录最少被处理一次，但是允许从错误状态恢复时被处理多次。这就意味着可变更的状态可能被更新两次从而导致结果不正确。

任一方面，Spark Streaming 仅仅需要在批处理级别对记录进行追踪，所以他能保证每个批处理记录仅仅被处理一次，即使是 node 节点挂掉。虽然说 Storm 的 Trident library 可以保证一条记录被处理一次，但是它依赖于事务更新状态，而这个过程是很慢的，并且需要由用户去实现。

- **实现和编程 API**

Storm 主要是由 Clojure 语言实现，Spark Streaming 是由 Scala 实现。如果你想看看这两个框架是如何实现的或者你想自定义一些东西你就得记住这一点。Storm 是由 BackType 和 Twitter 开发，而 Spark Streaming 是在 UC Berkeley 开发的。

Storm 提供了 Java API ,同时也支持其他语言的 API。Spark Streaming 支持 Scala 和 Java 语言(其实也支持 Python)。

- **批处理框架集成**

Spark Streaming 的一个很棒的特性就是它是在 Spark 框架上运行的。这样你就可以想使用其他批处理代码一样来写 Spark Streaming 程序，或者是在 Spark 中交互查询。这就减少了单独编写流批量处理程序和历史数据处理程序。

- **生产支持**

Storm 已经出现好多年了，而且自从 2011 年开始就在 Twitter 内部生产环境中使用，还有其他一些公司。而 Spark Streaming 是一个新的项目，并且在 2013 年仅仅被 Sharethrough 使用(据作者了解)。

Storm 是 Hortonworks Hadoop 数据平台中流处理的解决方案，而 Spark Streaming 出现在 MapR 的分布式平台和 Cloudera 的企业数据平台中。除此之外，Databricks 是为 Spark 提供技术支持的公司，包括了 Spark Streaming。

虽然说两者都可以在各自的集群框架中运行，但是 Storm 可以在 Mesos 上运行，而 Spark Streaming 可以在 YARN 和 Mesos 上运行。

2 运行原理

2.1 Streaming 架构

SparkStreaming 是一个对实时数据流进行高通量、容错处理的流式处理系统，可以对多种数据源（如 Kdfka、Flume、Twitter、Zero 和 TCP 套接字）进行类似 Map、Reduce 和 Join 等复杂操作，并将结果保存到外部文件系统、数据库或应用到实时仪表盘。

- **计算流程**：Spark Streaming 是将流式计算分解成一系列短小的批处理作业。这里的批处理引擎是 Spark Core，也就是把 Spark Streaming 的输入数据按照 batch size（如 1 秒）分成一段一段的数据（Discretized Stream），每一段数据都转换成 Spark 中的 RDD（Resilient Distributed Dataset），然后将 Spark Streaming 中对 DStream 的 Transformation 操作变为针对 Spark 中对 RDD 的 Transformation 操作，将 RDD 经过操作变成中间结果保存在内存中。整个流式计算根据业务的需求可以对中间的结果进行叠加或者存储到外部设备。下图显示了 Spark Streaming 的整个流程。

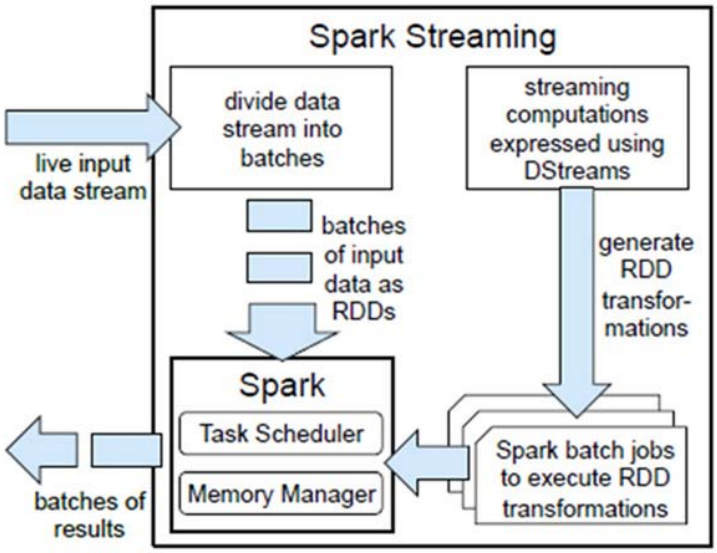
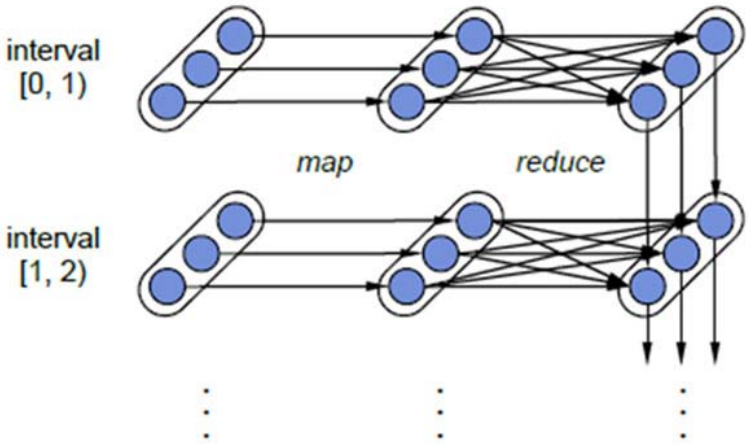


图 Spark Streaming 构架

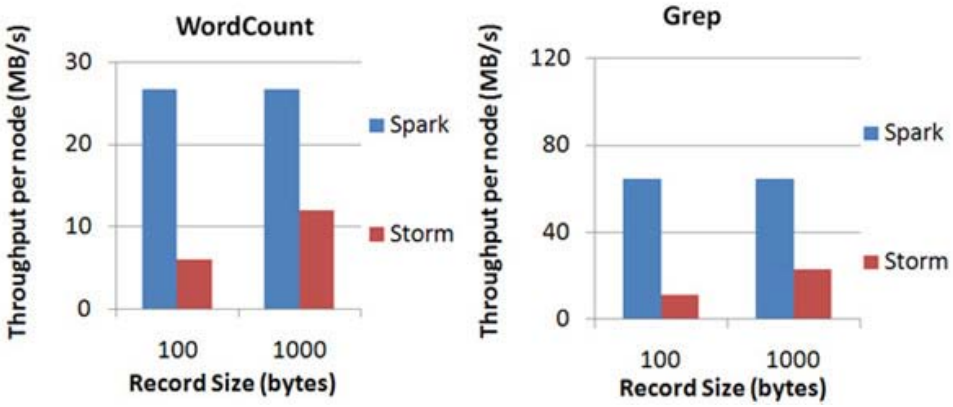
- **容错性**：对于流式计算来说，容错性至关重要。首先我们要明确一下 Spark 中 RDD 的容错机制。每一个 RDD 都是一个不可变的分布式可重算的数据集，其记录着确定性的操作继承关系（lineage），所以只要输入数据是可容错的，那么任意一个 RDD 的分区（Partition）出错或不可用，都是可以利用原始输入数据通过转换操作而重新算出的。

对于 Spark Streaming 来说，其 RDD 的传承关系如下图所示，图中的每一个椭圆形表示一个 RDD，椭圆形中的每个圆形代表一个 RDD 中的一个 Partition，图中的每一列的多个 RDD 表示一个 DStream（图中有三个 DStream），而每一行最后一个 RDD 则表示每一个 Batch Size 所产生的中间结果 RDD。我们可以看到图中的每一个 RDD 都是通过 lineage 相连接的，由于 Spark Streaming 输入数据可以来自于磁盘，例如 HDFS（多份拷贝）或是来自于网络的数据流（Spark Streaming 会将网络输入数据的每一个数据流拷贝两份到其他的机器）都能保证容错性，所以 RDD 中任意的 Partition 出错，都可以并行地在其他机器上将缺失的 Partition 计算出来。这个容错恢复方式比连续计算模型（如 Storm）的效率更高。



Spark Streaming 中 RDD 的 lineage 关系图

- **实时性**：对于实时性的讨论，会牵涉到流式处理框架的应用场景。Spark Streaming 将流式计算分解成多个 Spark Job，对于每一段数据的处理都会经过 Spark DAG 图分解以及 Spark 的任务集的调度过程。对于目前版本的 Spark Streaming 而言，其最小的 Batch Size 的选取在 0.5~2 秒钟之间（Storm 目前最小的延迟是 100ms 左右），所以 Spark Streaming 能够满足除对实时性要求非常高（如高频实时交易）之外的所有流式准实时计算场景。
- **扩展性与吞吐量**：Spark 目前在 EC2 上已能够线性扩展到 100 个节点（每个节点 4Core），可以以数秒的延迟处理 6GB/s 的数据量（60M records/s），其吞吐量也比流行的 Storm 高 2～5 倍，图 4 是 Berkeley 利用 WordCount 和 Grep 两个用例所做的测试，在 Grep 这个测试中，Spark Streaming 中的每个节点的吞吐量是 670k records/s，而 Storm 是 115k records/s。



Spark Streaming 与 Storm 吞吐量比较图

2.2 编程模型

DStream（Discretized Stream）作为 Spark Streaming 的基础抽象，它代表持续性的数据流。这些数据流既可以通过外部输入源赖获取，也可以通过现有的 Dstream 的 transformation 操作来获得。在内部实现上，DStream 由一组时间序列上连续的 RDD 来表示。每个 RDD 都包含了自己特定时间间隔内的数据流。如图 7-3 所示。

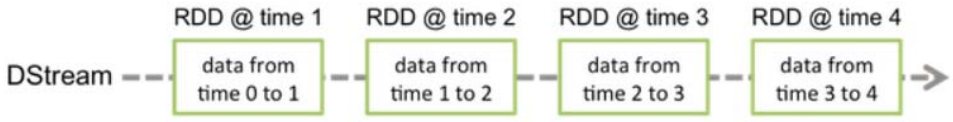
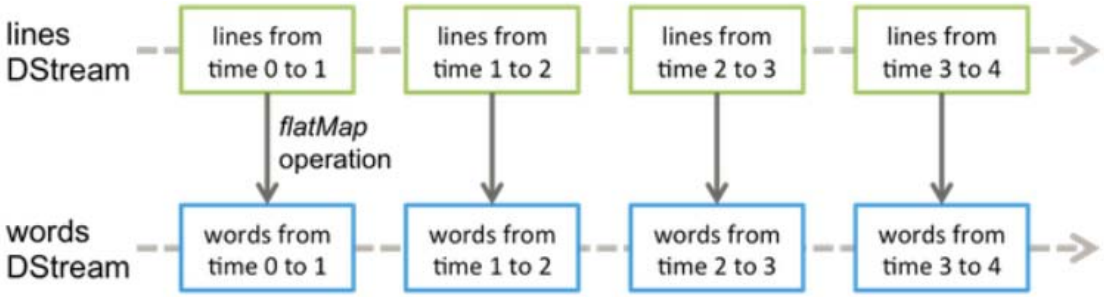


图 7-3　DStream 中在时间轴下生成离散的 RDD 序列



对 DStream 中数据的各种操作也是映射到内部的 RDD 上来进行的，如图 7-4 所示，对 Dtream 的操作可以通过 RDD 的 transformation 生成新的 DStream。这里的执行引擎是 Spark。

2.2.1 如何使用 Spark Streaming

作为构建于 Spark 之上的应用框架，Spark Streaming 承袭了 Spark 的编程风格，对于已经了解 Spark 的用户来说能够快速地上手。接下来以 Spark Streaming 官方提供的 WordCount 代码为例来介绍 Spark Streaming 的使用方式。

```
import org.apache.spark._
import org.apache.spark.streaming._
import org.apache.spark.streaming.StreamingContext._

// Create a local StreamingContext with two working thread and batch interval of 1 second.
// The master requires 2 cores to prevent from a starvation scenario.
val conf = new SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
val ssc = new StreamingContext(conf, Seconds(1))

// Create a DStream that will connect to hostname:port, like localhost:9999
val lines = ssc.socketTextStream("localhost", 9999)

// Split each line into words
val words = lines.flatMap(_.split(" "))
import org.apache.spark.streaming.StreamingContext._
// Count each word in each batch
val pairs = words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)

// Print the first ten elements of each RDD generated in this DStream to the console
wordCounts.print()
```

```
ssc.start()             // Start the computation
ssc.awaitTermination()  // Wait for the computation to terminate
```

1. **创建 StreamingContext 对象** 同 Spark 初始化需要创建 SparkContext 对象一样，使用 Spark Streaming 就需要创建 StreamingContext 对象。创建 StreamingContext 对象所需的参数与 SparkContext 基本一致，包括指明 Master，设定名称（如 NetworkWordCount)。需要注意的是参数 Seconds(1)，Spark Streaming 需要指定处理数据的时间间隔，如上例所示的 1s，那么 Spark Streaming 会以 1s 为时间窗口进行数据处理。此参数需要根据用户的需求和集群的处理能力进行适当的设置；

2. **创建 InputDStream** 如同 Storm 的 Spout，Spark Streaming 需要指明数据源。如上例所示的 socketTextStream，Spark Streaming 以 socket 连接作为数据源读取数据。当然 Spark Streaming 支持多种不同的数据源，包括 Kafka、 Flume、HDFS/S3、Kinesis 和 Twitter 等数据源；

3. **操作 DStream** 对于从数据源得到的 DStream，用户可以在其基础上进行各种操作，如上例所示的操作就是一个典型的 WordCount 执行流程 :对于当前时间窗口内从数据源得到的数据首先进行分割，然后利用 Map 和 ReduceByKey 方法进行计算，当然最后还有使用 print()方法输出结果；

4. **启动 Spark Streaming** 之前所作的所有步骤只是创建了执行流程，程序没有真正连接上数据源，也没有对数据进行任何操作，只是设定好了所有的执行计划，当 ssc.start()启动后程序才真正进行所有预期的操作。

至此对于 Spark Streaming 的如何使用有了一个大概的印象，在后面的章节我们会通过源代码深入探究一下 Spark Streaming 的执行流程。

2.2.2 DStream 的输入源

在 Spark Streaming 中所有的操作都是基于流的，而输入源是这一系列操作的起点。输入 DStreams 和 DStreams 接收的流都代表输入数据流的来源，在 Spark Streaming 提供两种内置数据流来源：

- **基础来源** 在 StreamingContext API 中直接可用的来源。例如：文件系统、Socket（套接字）连接和 Akka actors；
- **高级来源** 如 Kafka、Flume、Kinesis、Twitter 等，可以通过额外的实用工具类创建。

2.2.2.1 基础来源

在前面分析怎样使用 Spark Streaming 的例子中我们已看到 ssc.socketTextStream()方法，可以通过 TCP 套接字连接，从从文本数据中创建了一个 DStream。除了套接字，

StreamingContext 的 API 还提供了方法从文件和 Akka actors 中创建 DStreams 作为输入源。

Spark Streaming 提供了 streamingContext.fileStream(dataDirectory)方法可以从任何文件系统(如：HDFS、S3、NFS 等）的文件中读取数据，然后创建一个 DStream。Spark Streaming 监控 dataDirectory 目录和在该目录下任何文件被创建处理(不支持在嵌套目录下写文件)。需要注意的是：读取的必须是具有相同的数据格式的文件；创建的文件必须在 dataDirectory 目录下，并通过自动移动或重命名成数据目录；文件一旦移动就不能被改变，如果文件被不断追加,新的数据将不会被阅读。对于简单的文本文，可以使用一个简单的方法 streamingContext.textFileStream(dataDirectory)来读取数据。

Spark Streaming 也可以基于自定义 Actors 的流创建 DStream ，通过 Akka actors 接受数据流，使用方法 streamingContext.actorStream(actorProps, actor-name)。

Spark Streaming 使用 streamingContext.queueStream(queueOfRDDs)方法可以创建基于 RDD 队列的 DStream ,每个 RDD 队列将被视为 DStream 中一块数据流进行加工处理。

2.2.2.2 高级来源

这一类的来源需要外部 non-Spark 库的接口，其中一些有复杂的依赖关系(如 Kafka、Flume)。因此通过这些来源创建 DStreams 需要明确其依赖。例如，如果想创建一个使用 Twitter tweets 的数据的 DStream 流，必须按以下步骤来做：

1）在 SBT 或 Maven 工程里添加 spark-streaming-twitter_2.10 依赖。

2)开发 :导入 TwitterUtils 包 ,通过 TwitterUtils.createStream 方法创建一个 DStream。

3）部署：添加所有依赖的 jar 包(包括依赖的 spark-streaming-twitter_2.10 及其依赖)，然后部署应用程序。

需要注意的是，这些高级的来源一般在 Spark Shell 中不可用，因此基于这些高级来源的应用不能在 Spark Shell 中进行测试。如果你必须在 Spark shell 中使用它们，你需要下载相应的 Maven 工程的 Jar 依赖并添加到类路径中。

其中一些高级来源如下：

- **Twitter** Spark Streaming 的 TwitterUtils 工具类使用 Twitter4j ,Twitter4J 库支持通过任何方法提供身份验证信息，你可以得到公众的流，或得到基于关键词过滤流。
- **Flume** Spark Streaming 可以从 Flume 中接受数据。
- **Kafka** Spark Streaming 可以从 Kafka 中接受数据。
- **Kinesis** Spark Streaming 可以从 Kinesis 中接受数据。

需要重申的一点是在开始编写自己的 SparkStreaming 程序之前，一定要将高级来源依赖的 Jar 添加到 SBT 或 Maven 项目相应的 artifact 中。常见的输入源和其对应的 Jar 包如下图所示。

Source	Artifact
Kafka	spark-streaming-kafka_2.10
Flume	spark-streaming-flume_2.10
Kinesis	spark-streaming-kinesis-asl_2.10 [Amazon Software License]
Twitter	spark-streaming-twitter_2.10
ZeroMQ	spark-streaming-zeromq_2.10
MQTT	spark-streaming-mqtt_2.10

另外，输入 DStream 也可以创建自定义的数据源，需要做的就是实现一个用户定义的接收器。

2.2.3 DStream 的操作

与 RDD 类似，DStream 也提供了自己的一系列操作方法，这些操作可以分成三类：普通的转换操作、窗口转换操作和输出操作。

2.2.3.1 普通的转换操作

普通的转换操作如下表所示：

转换	描述
map(*func*)	源 DStream 的每个元素通过函数 func 返回一个新的 DStream。
flatMap(*func*)	类似与 map 操作，不同的是每个输入元素可以被映射出0或者更多的输出元素。
filter(*func*)	在源 DSTREAM 上选择 Func 函数返回仅为 true 的元素,最终返回一个新的 DSTREAM 。
repartition(*numPartitions*)	通过输入的参数 numPartitions 的值来改变 DStream 的分区大小。
union(*otherStream*)	返回一个包含源 DStream 与其他 DStream 的元素合并后的新 DSTREAM。
count()	对源 DStream 内部的所含有的 RDD 的元素数量进行计数，返回一个

	内部的 RDD 只包含一个元素的 DStreaam。
reduce(*func*)	使用函数 func（有两个参数并返回一个结果）将源 DStream 中每个 RDD 的元素进行聚 合操作,返回一个内部所包含的 RDD 只有一个元素的新 DStream。
countByValue()	计算 DStream 中每个 RDD 内的元素出现的频次并返回新的 DStream[(K,Long)]，其中 K 是 RDD 中元素的类型，Long 是元素出现的频次。
reduceByKey(*func*, [*numTasks*])	当一个类型为（K，V）键值对的 DStream 被调用的时候,返回类型为类型为（K，V）键值对的新 DStream,其中每个键的值 V 都是使用聚合函数 func 汇总。注意：默认情况下，使用 Spark 的默认并行度提交任务（本地模式下并行度为2，集群模式下位8），可以通过配置 numTasks 设置不同的并行任务数。
join(*otherStream*, [*numTasks*])	当被调用类型分别为（K，V）和（K，W）键值对的2个 DStream 时，返回类型为（K，(V，W)）键值对的一个新 DSTREAM。
cogroup(*otherStream*, [*numTasks*])	当被调用的两个 DStream 分别含有(K, V) 和(K, W)键值对时,返回一个(K, Seq[V], Seq[W])类型的新的 DStream。
transform(*func*)	通过对源 DStream 的每 RDD 应用 RDD-to-RDD 函数返回一个新的 DStream，这可以用来在 DStream 做任意 RDD 操作。
updateStateByKey(*func*)	返回一个新状态的 DStream,其中每个键的状态是根据键的前一个状态和键的新值应用给定函数 func 后的更新。这个方法可以被用来维持每个键的任何状态数据。

在上面列出的这些操作中，transform()方法和 updateStateByKey()方法值得我们深入的探讨一下：

- **transform(func)操作**

该 transform 操作（转换操作）连同其其类似的 transformWith 操作允许 DStream 上应用任意 RDD-to-RDD 函数。它可以被应用于未在 DStream API 中暴露任何的 RDD 操作。例如，在每批次的数据流与另一数据集的连接功能不直接暴露在 DStream API 中，但可以轻松地使用 transform 操作来做到这一点，这使得 DStream 的功能非常强大。例如，你可以通过连接预先计算的垃圾邮件信息的输入数据流（可能也有 Spark 生成的），然后基于此做实时数据清理的筛选，如下面官方提供的伪代码所示。事实上，也可以在 transform 方法中使用机器学习和图形计算的算法。

- **updateStateByKey 操作**

该 updateStateByKey 操作可以让你保持任意状态，同时不断有新的信息进行更新。要使用此功能，必须进行两个步骤 ：

（1） 定义状态 - 状态可以是任意的数据类型。

（2） 定义状态更新函数 - 用一个函数指定如何使用先前的状态和从输入流中获取的新值更新状态。

让我们用一个例子来说明，假设你要进行文本数据流中单词计数 。在这里，正在运行的计数是状态而且它是一个整数。我们定义了更新功能如下：

```
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)
}
```

此函数应用于含有键值对的 DStream 中（如前面的示例中，在 DStream 中含有（word，1）键值对）。它会针对里面的每个元素（如 wordCount 中的 word）调用一下更新函数，newValues 是最新的值，runningCount 是之前的值。

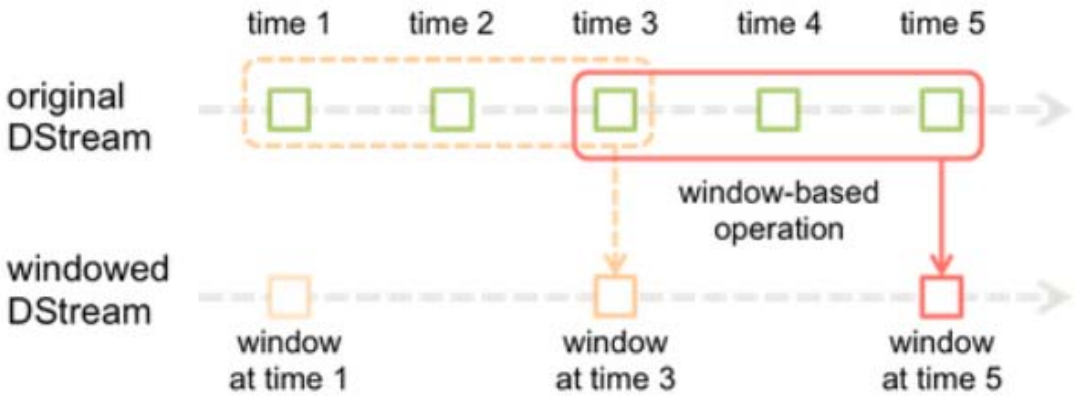
```
val runningCounts = pairs.updateStateByKey[Int](updateFunction _)
```

2.2.3.2 窗口转换操作

Spark Streaming 还提供了窗口的计算，它允许你通过滑动窗口对数据进行转换，窗口转换操作如下：

转换	描述
window(*windowLength*, *slideInterval*)	返回一个基于源 DStream 的窗口批次计算后得到新的 DStream。
countByWindow(*windowLength*,*slideInterval*)	返回基于滑动窗口的 DStream 中的元素的数量。
reduceByWindow(*func*, *windowLength*,*slideInterval*)	基于滑动窗口对源 DStream 中的元素进行聚合操作，得到一个新的 DStream。
reduceByKeyAndWindow(*func*,*windowLength*, *slideInterval*, [*numTasks*])	基于滑动窗口对（K，V）键值对类型的 DStream 中的值按 K 使用聚合函数 func 进行聚合操作，得到一个新的 DStream。
reduceByKeyAndWindow(*func*, *invFunc*,*windowLength*, *slideInterval*, [*numTasks*])	一个更高效的 reduceByKkeyAndWindow()的实现版本，先对滑动窗口中新的时间间隔内数据增量聚合并移去最早的与新增数据量的时间间隔内的数据统计量。例如，计算 t+4秒这个时刻过去5秒窗口的 WordCount ,那么我们可以将 t+3时刻过去5

	秒的统计量加上[t+3，t+4]的统计量，在减去[t-2，t-1]的统计量，这种方法可以复用中间三秒的统计量，提高统计的效率。
countByValueAndWindow(*windowLength*,*slideInterval*, [*numTasks*])	基于滑动窗口计算源 DStream 中每个 RDD 内每个元素出现的频次并返回 DStream[(K,Long)] ,其中 K 是 RDD 中元素的类型，Long 是元素频次。与 countByValue 一样，reduce 任务的数量可以通过一个可选参数进行配置。



批处理间隔示意图

在 Spark Streaming 中，数据处理是按批进行的，而数据采集是逐条进行的，因此在 Spark Streaming 中会先设置好批处理间隔（batch duration），当超过批处理间隔的时候就会把采集到的数据汇总起来成为一批数据交给系统去处理。

对于窗口操作而言，在其窗口内部会有 N 个批处理数据，批处理数据的大小由窗口间隔（window duration）决定，而窗口间隔指的就是窗口的持续时间，在窗口操作中，只有窗口的长度满足了才会触发批数据的处理。除了窗口的长度，窗口操作还有另一个重要的参数就是滑动间隔（slide duration），它指的是经过多长时间窗口滑动一次形成新的窗口，滑动窗口默认情况下和批次间隔的相同，而窗口间隔一般设置的要比它们两个大。在这里必须注意的一点是滑动间隔和窗口间隔的大小一定得设置为批处理间隔的整数倍。

如批处理间隔示意图所示，批处理间隔是 1 个时间单位，窗口间隔是 3 个时间单位，滑动间隔是 2 个时间单位。对于初始的窗口 time 1-time 3，只有窗口间隔满足了才触发数据的处理。这里需要注意的一点是，初始的窗口有可能流入的数据没有撑满，但是随着时间的推进，窗口最终会被撑满。当每个 2 个时间单位，窗口滑动一次后，会有新的数据流入窗口，这时窗口会移去最早的两个时间单位的数据，而与最新的两个时间单位的数据进行汇总形成新的窗口（time3-time5）。

对于窗口操作，批处理间隔、窗口间隔和滑动间隔是非常重要的三个时间概念，是理解窗口操作的关键所在。

2.2.3.3 输出操作

Spark Streaming 允许 DStream 的数据被输出到外部系统，如数据库或文件系统。由于输出操作实际上使 transformation 操作后的数据可以通过外部系统被使用，同时输出操作触发所有 DStream 的 transformation 操作的实际执行（类似于 RDD 操作）。以下表列出了目前主要的输出操作：

转换	描述
print()	在 Driver 中打印出 DStream 中数据的前10个元素。
saveAsTextFiles(*prefix*, [*suffix*])	将 DStream 中的内容以文本的形式保存为文本文件，其中每次批处理间隔内产生的文件以 prefix-TIME_IN_MS[.suffix]的方式命名。
saveAsObjectFiles(*prefix*, [*suffix*])	将 DStream 中的内容按对象序列化并且以 SequenceFile 的格式保存。其中每次批处理间隔内产生的文件以 prefix-TIME_IN_MS[.suffix]的方式命名。
saveAsHadoopFiles(*prefix*, [*suffix*])	将 DStream 中的内容以文本的形式保存为 Hadoop 文件，其中每次批处理间隔内产生的文件以 prefix-TIME_IN_MS[.suffix]的方式命名。
foreachRDD(*func*)	最基本的输出操作，将 func 函数应用于 DStream 中的 RDD 上，这个操作会输出数据到外部系统，比如保存 RDD 到文件或者网络数据库等。需要注意的是 func 函数是在运行该 streaming 应用的 Driver 进程里执行的。

dstream.foreachRDD 是一个非常强大的输出操作，它允许将数据输出到外部系统。但是，如何正确高效地使用这个操作是很重要的，下面展示了如何去避免一些常见的错误。

通常将数据写入到外部系统需要创建一个连接对象（如 TCP 连接到远程服务器），并用它来发送数据到远程系统。出于这个目的，开发者可能在不经意间在 Spark driver 端创建了连接对象，并尝试使用它保存 RDD 中的记录到 Spark worker 上，如下面代码：

```
dstream.foreachRDD { rdd =>
  val connection = createNewConnection()  // executed at the driver
  rdd.foreach { record =>
    connection.send(record) // executed at the worker
  }
}
```

这是不正确的，这需要连接对象进行序列化并从 Driver 端发送到 Worker 上。连接对象很少在不同机器间进行这种操作，此错误可能表现为序列化错误（连接对不可序列化），初始化错误（连接对象在需要在 Worker 上进行需要初始化）等等，正确的解决办法是在 worker 上创建的连接对象。

通常情况下，创建一个连接对象有时间和资源开销。因此，创建和销毁的每条记录的连接对象可能招致不必要的资源开销 ，并显著降低系统整体的吞吐量 。一个更好的解决方案是使用 rdd.foreachPartition 方法创建一个单独的连接对象，然后使用该连接对象输出的所有 RDD 分区中的数据到外部系统。

这缓解了创建多条记录连接的开销。最后，还可以进一步通过在多个 RDDs/ batches 上重用连接对象进行优化。一个保持连接对象的静态池可以重用在多个批处理的 RDD 上将其输出到外部系统，从而进一步降低了开销。

需要注意的是，在静态池中的连接应该按需延迟创建，这样可以更有效地把数据发送到外部系统。另外需要要注意的是：DStreams 延迟执行的，就像 RDD 的操作是由 actions 触发一样。默认情况下，输出操作会按照它们在 Streaming 应用程序中定义的顺序一个个执行。

2.3 容错、持久化和性能调优

2.3.1 容错

DStream 基于 RDD 组成，RDD 的容错性依旧有效，我们首先回忆一下 SparkRDD 的基本特性。

- RDD 是一个不可变的、确定性的可重复计算的分布式数据集。RDD 的某些 partition 丢失了，可以通过血统（lineage）信息重新计算恢复；
- 如果 RDD 任何分区因 worker 节点故障而丢失 ,那么这个分区可以从原来依赖的容错数据集中恢复；
- 由于 Spark 中所有的数据的转换操作都是基于 RDD 的，即使集群出现故障，只要输入数据集存在，所有的中间结果都是可以被计算的。

Spark Streaming 是可以从 HDFS 和 S3 这样的文件系统读取数据的，这种情况下所有的数据都可以被重新计算，不用担心数据的丢失。但是在大多数情况下，Spark Streaming 是基于网络来接受数据的，此时为了实现相同的容错处理，在接受网络的数据时会在集群的多个 Worker 节点间进行数据的复制（默认的复制数是 2），这导致产生在出现故障时被处理的两种类型的数据：

1）Data received and replicated ：一旦一个 Worker 节点失效，系统会从另一份还存在的数据中重新计算。

2）Data received but buffered for replication ：一旦数据丢失，可以通过 RDD 之间的依赖关系，从 HDFS 这样的外部文件系统读取数据。

此外，有两种故障，我们应该关心：

（1）Worker 节点失效：通过上面的讲解我们知道，这时系统会根据出现故障的数据的类型，选择是从另一个有复制过数据的工作节点上重新计算，还是直接从从外部文件系统读取数据。

（2）Driver（驱动节点）失效 ：如果运行 Spark Streaming 应用时驱动节点出现故障，那么很明显的 StreamingContext 已经丢失，同时在内存中的数据全部丢失。对于这种情况，Spark Streaming 应用程序在计算上有一个内在的结构——在每段 micro-batch 数据周期性地执行同样的 Spark 计算。这种结构允许把应用的状态（亦称 checkpoint）周期性地保存到可靠的存储空间中，并在 driver 重新启动时恢复该状态。具体做法是在 ssc.checkpoint(<checkpoint directory>)函数中进行设置，Spark Streaming 就会定期把 DStream 的元信息写入到 HDFS 中，一旦驱动节点失效，丢失的 StreamingContext 会通过已经保存的检查点信息进行恢复。

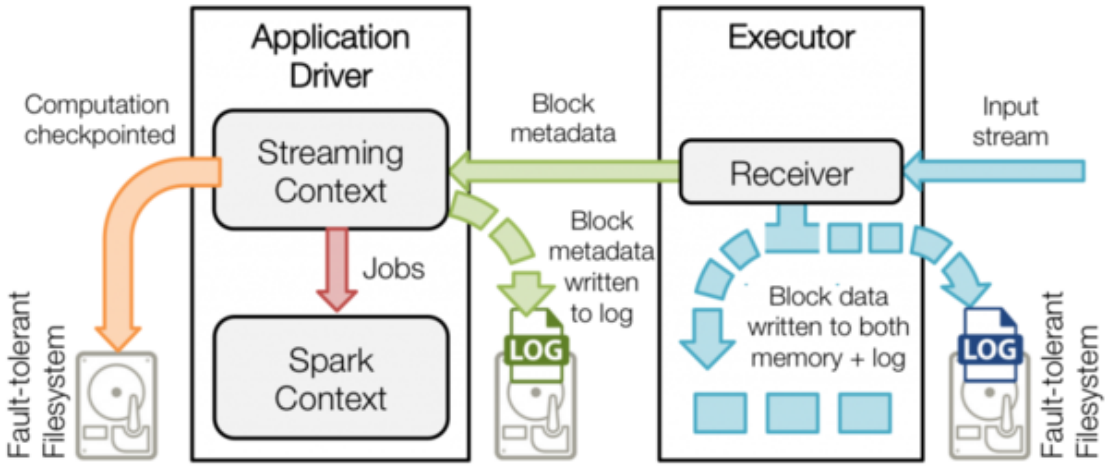
最后我们谈一下 Spark Stream 的容错在 Spark 1.2 版本的一些改进：

实时流处理系统必须要能在 24/7 时间内工作，因此它需要具备从各种系统故障中恢复过来的能力。最开始，SparkStreaming 就支持从 driver 和 worker 故障恢复的能力。然而有些数据源的输入可能在故障恢复以后丢失数据。在 Spark1.2 版本中，Spark 已经在 SparkStreaming 中对预写日志（也被称为 journaling）作了初步支持，改进了恢复机制，并使更多数据源的零数据丢失有了可靠。

对于文件这样的源数据，driver 恢复机制足以做到零数据丢失，因为所有的数据都保存在了像 HDFS 或 S3 这样的容错文件系统中了。但对于像 Kafka 和 Flume 等其它数据源，有些接收到的数据还只缓存在内存中，尚未被处理，它们就有可能会丢失。这是由于 Spark 应用的分布操作方式引起的。当 driver 进程失败时，所有在 standalone/yarn/mesos 集群运行的 executor，连同它们在内存中的所有数据，也同时被终止。对于 Spark Streaming 来说，从诸如 Kafka 和 Flume 的数据源接收到的所有数据，在它们处理完成之前，一直都缓存在 executor 的内存中。纵然 driver 重新启动，这些缓存的数据也不能被恢复。为了避免这种数据损失，在 Spark1.2 发布版本中引进了预写日志（WriteAheadLogs）功能。

预写日志功能的流程是：1）一个 SparkStreaming 应用开始时（也就是 driver 开始时），相关的 StreamingContext 使用 SparkContext 启动接收器成为长驻运行任务。这些接收器接收并保存流数据到 Spark 内存中以供处理。2）接收器通知 driver。3）接收块中的元数据（metadata）被发送到 driver 的 StreamingContext。这个元数据包括：（a）定位其在 executor 内存中数据的块 referenceid，（b）块数据在日志中的偏移信息（如果启用了）。

用户传送数据的生命周期如下图所示。



类似 Kafka 这样的系统可以通过复制数据保持可靠性。允许预写日志两次高效地复制同样的数据：一次由 Kafka，而另一次由 SparkStreaming。Spark 未来版本将包含 Kafka 容错机制的原生支持，从而避免第二个日志。

2.3.2 持久化

与 RDD 一样，DStream 同样也能通过 persist()方法将数据流存放在内存中，默认的持久化方式是 MEMORY_ONLY_SER，也就是在内存中存放数据同时序列化的方式，这样做的好处是遇到需要多次迭代计算的程序时，速度优势十分的明显。而对于一些基于窗口的操作，如 reduceByWindow、reduceByKeyAndWindow，以及基于状态的操作，如 updateStateBykey，其默认的持久化策略就是保存在内存中。

对于来自网络的数据源（Kafka、Flume、sockets 等），默认的持久化策略是将数据保存在两台机器上，这也是为了容错性而设计的。

另外，对于窗口和有状态的操作必须 checkpoint，通过 StreamingContext 的 checkpoint 来指定目录，通过 Dtream 的 checkpoint 指定间隔时间，间隔必须是滑动间隔(slide interval)的倍数。

2.3.3 性能调优

1. 优化运行时间

- **增加并行度** 确保使用整个集群的资源，而不是把任务集中在几个特定的节点上。对于包含 shuffle 的操作，增加其并行度以确保更为充分地使用集群资源；
- **减少数据序列化，反序列化的负担** Spark Streaming 默认将接受到的数据序列化后存储，以减少内存的使用。但是序列化和反序列话需要更多的 CPU 时间，因此更加高效的序列化方式（Kryo）和自定义的系列化接口可以更高效地使用 CPU；
- **设置合理的 batch duration（批处理时间间）** 在 Spark Streaming 中，Job 之间有可能存在依赖关系，后面的 Job 必须确保前面的作业执行结束后才能提交。若前面的 Job

执行的时间超出了批处理时间间隔，那么后面的 Job 就无法按时提交，这样就会进一步拖延接下来的 Job，造成后续 Job 的阻塞。因此设置一个合理的批处理间隔以确保作业能够在这个批处理间隔内结束时必须的；

- **减少因任务提交和分发所带来的负担** 通常情况下，Akka 框架能够高效地确保任务及时分发，但是当批处理间隔非常小（500ms）时，提交和分发任务的延迟就变得不可接受了。使用 Standalone 和 Coarse-grained Mesos 模式通常会比使用 Fine-grained Mesos 模式有更小的延迟。

2. 优化内存使用

- **控制 batch size（批处理间隔内的数据量）** Spark Streaming 会把批处理间隔内接收到的所有数据存放在 Spark 内部的可用内存区域中，因此必须确保当前节点 Spark 的可用内存中少能容纳这个批处理时间间隔内的所有数据，否则必须增加新的资源以提高集群的处理能力；
- **及时清理不再使用的数据** 前面讲到 Spark Streaming 会将接受的数据全部存储到内部可用内存区域中，因此对于处理过的不再需要的数据应及时清理，以确保 Spark Streaming 有富余的可用内存空间。通过设置合理的 spark.cleaner.ttl 时长来及时清理超时的无用数据，这个参数需要小心设置以免后续操作中所需要的数据被超时错误处理；
- **观察及适当调整 GC 策略** GC 会影响 Job 的正常运行，可能延长 Job 的执行时间，引起一系列不可预料的问题。观察 GC 的运行情况，采用不同的 GC 策略以进一步减小内存回收对 Job 运行的影响。

参考资料：

（1）《Spark Streaming》 http://blog.debugo.com/spark-streaming/