Spark
亚太研究院

Spark 官方文档翻译

实时流处理编程指南
(v1.2.0)

翻译者　李军

Spark 官方文档翻译团成员

前 言

伴随着大数据相关技术和产业的逐步成熟，继 Hadoop 之后，Spark 技术以集大成的无可比拟的优势，发展迅速，将成为替代 Hadoop 的下一代云计算、大数据核心技术。

Spark 是当今大数据领域最活跃最热门的高效大数据通用计算平台，基于 RDD，Spark 成功的构建起了一体化、多元化的大数据处理体系，在“One Stack to rule them all”思想的引领下，Spark 成功的使用 Spark SQL、Spark Streaming、MLLib、GraphX 近乎完美的解决了大数据中 Batch Processing、Streaming Processing、Ad-hoc Query 等三大核心问题，更为美妙的是在 Spark 中 Spark SQL、Spark Streaming、MLLib、GraphX 四大子框架和库之间可以无缝的共享数据和操作，这是当今任何大数据平台都无可匹敌的优势。

在实际的生产环境中，世界上已经出现很多一千个以上节点的 Spark 集群，以 eBay 为例，eBay 的 Spark 集群节点已经超过 2000 个，Yahoo！等公司也在大规模的使用 Spark，国内的淘宝、腾讯、百度、网易、京东、华为、大众点评、优酷土豆等也在生产环境下深度使用 Spark。2014 Spark Summit 上的信息，Spark 已经获得世界 20 家顶级公司的支持，这些公司中包括 Intel、IBM 等，同时更重要的是包括了最大的四个 Hadoop 发行商，都提供了对 Spark 非常强有力的支持。

与 Spark 火爆程度形成鲜明对比的是 Spark 人才的严重稀缺，这一情况在中国尤其严重，这种人才的稀缺，一方面是由于 Spark 技术在 2013、2014 年才在国内的一些大型企业里面被逐步应用，另一方面是由于匮乏 Spark 相关的中文资料和系统化的培训。为此，Spark 亚太研究院和 51CTO 联合推出了“Spark 亚太研究院决胜大数据时代 100 期公益大讲堂”，来推动 Spark 技术在国内的普及及落地。

具体视频信息请参考 http://edu.51cto.com/course/course_id-1659.html

与此同时，为了向 Spark 学习者提供更为丰富的学习资料，Spark 亚太研究院去年 8 月发起并号召，结合网络社区的力量构建了 Spark 中文文档专家翻译团队，翻译了 Spark 中文文档 V1.1.0 版本。2014 年 12 月，Spark 官方团队发布了 Spark 1.2.0 版本，为了让学习者了解到最新的内容，Spark 中文文档专家翻译团队又对 Spark 1.2.0 版本进行了部分更新，在此，我谨代表 Spark 亚太研究院及广大 Spark 学习爱好者向专家翻译团队所有成员热情而专业的工作致以深刻的敬意！

当然，作为相对系统的 Spark 中文文档，不足之处在所难免，大家有任何建议或者意见都可以发邮件到 marketing@sparkinchina.com；同时如果您想加入 Spark 中文文档翻译团队，也请发邮件到 marketing@sparkinchina.com 进行申请；Spark 中文

文档的翻译是一个持续更新的、不断版本迭代的过程，我们会尽全力给大家提供更高质量的 Spark 中文文档翻译。

最后，也是最重要的，请允许我荣幸的介绍一下我们的 Spark 中文文档 1.2.0 版本翻译的专家团队成员，他们分别是（排名不分先后）：

- 傅智勇，《快速开始(v1.2.0)》
- 王宇舟,《Spark 机器学习库 (v1.2.0)》
- 武扬，《在 Yarn 上运行 Spark (v1.2.0)》《Spark 调优(v1.2.0)》
- 徐骄，《Spark 配置(v1.2.0)》《Spark 作业调度(v1.2.0)》
- 蔡立宇，《Bagel 编程指南(v1.2.0)》
- harli，《Spark 编程指南 (v1.2.0)》
- 韩保礼，《Spark SQL 编程指南(v1.2.0)》
- 李丹丹，《文档首页(v1.2.0)》
- 李军，《Spark 实时流处理编程指南(v1.2.0)》
- 俞杭军，《使用 Maven 编译 Spark(v1.2.0)》
- 王之，《给 Spark 提交代码(v1.2.0)》
- Ernest ,《集群模式概览(v1.2.0)》《监控与相关工具(v1.2.0)》《提交应用程序(v1.2.0)》

Life is short, You need Spark!

Spark 亚太研究院院长 王家林
2015 年 2 月

Spark 亚太研究院决胜大数据时代 100 期公益大讲堂

简 介

作为下一代云计算的核心技术 ,Spark 性能超 Hadoop 百倍 ,算法实现仅有其 1/10 或 1/100,是可以革命 Hadoop 的目前唯一替代者，能够做 Hadoop 做的一切事情，同时速度比 Hadoop 快了 100 倍以上。目前 Spark 已经构建了自己的整个大数据处理生态系统，国外一些大型互联网公司已经部署了 Spark。甚至连 Hadoop 的早期主要贡献者 Yahoo 现在也在多个项目中部署使用 Spark ;国内的淘宝、优酷土豆、网易、Baidu、腾讯、皮皮网等已经使用 Spark 技术用于自己的商业生产系统中，国内外的应用开始越来越广泛。Spark 正在逐渐走向成熟，并在这个领域扮演更加重要的角色, 刚刚结束的 2014 Spark Summit 上的信息，Spark 已经获得世界 20 家顶级公司的支持，这些公司中包括 Intel、IBM 等，同时更重要的是包括了最大的四个 Hadoop 发行商都提供了对非常强有力的支持 Spark 的支持.

鉴于 Spark 的巨大价值和潜力，同时由于国内极度缺乏 Spark 人才，Spark 亚太研究院在完成了对 Spark 源码的彻底研究的同时，不断在实际环境中使用 Spark 的各种特性的基础之上，推出了 Spark 亚太研究院决胜大数据时代 100 期公益大讲堂，希望能够帮助大家了解 Spark 的技术。同时，对 Spark 人才培养有近一步需求的企业和个人，我们将以公开课和企业内训的方式，来帮助大家进行 Spark 技能的提升。同样，我们也为企业提供一体化的顾问式服务及 Spark 一站式项目解决方案和实施方案。

Spark 亚太研究院决胜大数据时代 100 期公益大讲堂是国内第一个 Spark 课程免费线上讲座，每周一期，从 7 月份起，每周四晚 20:00-21:30，与大家不见不散！老师将就 Spark 内核剖析、源码解读、性能优化及商业实战案例等精彩内容与大家分享，干货不容错过！

时间：从 7 月份起，每周一期，每周四晚 20:00-21:30
形式：腾讯课堂在线直播
学习条件：对云计算大数据感兴趣的技术人员
课程学习地址：http://edu.51cto.com/course/course_id-1659.html

实时流处理编程指南(v1.2.0)

（翻译者：李军）

Spark Streaming Programming Guide，原文档链接：

http://spark.apache.org/docs/latest/streaming-programming-guide.html

目录

第1章 Spark Streaming 编程指南

1.1 概述

Spark Streaming 是 Spark 核心 API 的一种扩展 ,它实现了对实时流数据的高吞吐量，低容错率的流处理。数据可以有许多来源，如 Kafka, Flume, Twitter, ZeroMQ, Kinesis 或 TCP 套接字，可以使用复杂算法对其处理实现高层次的功能，如 map,reduce,join 和 window。最后，经处理的数据可被输出到文件系统，数据库，和实时仪表盘。事实上，你可以在数据流上使用 Spark 公司的机器学习和图形处理算法。

它的内部工作原理如下: Spark Streaming 接收实时输入数据流并将数据划分为批次，其然后由 Spark Enigne 分批处理用来生成结果的最终流。

Spark Streaming 提供了一个称为*离散流*或 *DStream* 的高层次的抽象，它代表一个持续的流数据。DStreams 被创建，可以是来自诸如 Kafka、 Flume 和 Kinesis 的输入数据流，也可以通过在其他 DStreams 上应用高级操作中创建。在内部，DSTREAM 代表 RDDS 中的一个序列。

本指南将告诉您如何开始用 DStreams 编写 Spark Streaming 程序。您可以使用 Scala、Java、Python (参考 Spark 1.2 中介绍)，它们均会在本指南中给出。你会发现选项卡贯穿本指南，从而可以让你选择不通的语言。

注意: 在 Spark 1.2 中，Spark Streaming 会有 Python API 的介绍。它包含所有 DStream 转换和 Scala 中的几乎所有可用的输出操作和 Java 接口。然而，它只支持基本的

Spark 亚太研究院
www.sparkinchina.com
TEL: 4006-998-758

源，像文本文件和文本数据套接字。额外源的 API 在未来也是可用的，像 Kafka 和 Flume。进一步的信息，可以关注整个文档中提到的 Python API 特性，寻找标记 Python API。

1.2 一个简单的例子

在我们进入如何编写自己的 Spark Streaming 程序的细节前，让我们快速浏览一下一个简单的 Spark Streaming 程序是什么样子。比方说，我们要计算从数据服务器监听 TCP 套接字接收的文本数据字数。你只需要做如下操作:

首先，我们导入 Spark Streaming 类名，以及添加需要（如 DStream）其他有用的方法以及一些从 StreamingContext 到我们环境的隐式转换。StreamingContext 是所有 Spark Streaming 功能的主入口点。我们将执行两个线程，创建一个本地 StreamingContext 和一个间隔 1 秒的批处理。

```
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 secon
d.
// 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))
```

通过这个 context 对象，我们可以创建一个新的 DStream 对象从一个 TCP 源，如主机名（例：localhost）和端口（例：9999）。

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

lines 对象表示将被从数据服务器接收的数据的流。在这个 DStream 中的每个记录就是一行文字。接下来，我们要根据空格把每一行分割成单词。

```
// Split each line into words
val words = lines.flatMap(_.split(" "))
```

flatMap 是一对多的 DStream 操作。它从源 DStream 中每条记录中生成多个新记录到新创建的 DStream 中。在这种情况下，每一行将被分割为多个单词并且 words DStream 表示单词流。接下来，我们将要计算这些单词的个数。

```
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()
```

words DStream 进一步映射（一对一的转换）到一个 DStream（word，1）键值对，然后将其 reduce 得到在每批次的数据单词的频率。最后，wordCounts.print（）将打印数每秒产生的计数。

需要注意的是，当这些行被执行的时候，Spark Streaming 只设置了计算启动时它才会执行，而没开始有真正的处理。所有的转换完成后才开始处理，我们最终调用：

```
ssc.start()             // Start the computation

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

上面完整的代码可以在 Spark Streaming 示例中 NetworkWordCount 找到 。

如果您已经下载并编译 Spark，就可以运行这个如下示例。您首先需要使用如下代码运行 Netcat（在大多数类 Unix 系统中一个小工具）作为数据服务器

```
$ nc -lk 9999
```

接下来,你可以在不同的终端通过运行如下代码启动实例

```
$ ./bin/run-example streaming.NetworkWordCount localhost 9999
```

最后，在终端 netcat 服务器每秒钟会计算并打印出所有的行。如下所示：

```
# TERMINAL 1:
# Running Netcat

$ nc -lk 9999
```

```
# TERMINAL 2: RUNNING NetworkWordCount


$ ./bin/run-example streaming.NetworkWordCount localhost 9999
...
```

```
hello world
...
```

```
-------------------------------------------
Time: 1357008430000 ms
-------------------------------------------
(hello,1)
(world,1)
...
```

1.3 基础知识

接下来，我们跳过简单的例子，详细阐述如何根据 Spark Streaming 编写 streaming 应用的基础知识。

1.3.1 链接

类似与 Spark ,Spark Streaming 需要 Maven 编译才可使用。在开始编写自己的 Spark Streaming 程序之前，您需要将以下依赖添加到您的 SBT 或 Maven 项目中。

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-streaming_2.10</artifactId>
    <version>1.2.0</version>
</dependency>
```

对于来源于 Spark Streaming 的核心 API 中不存在的数据,如 Kafka、Flume 和 Kinesis，则必须到相应的 artifact 添加到 spark-streaming-xyz_2.10 依赖。一些常见的示例是如下：

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

最新列表，请参阅 Apache 的版本库 的支持来源的完整列表。

13.2 初始化 StreamingContext

使用 Scala 初始化 Spark Streaming 程序，需要创建 StreamingContext 对象,它是所有 Spark Streaming 功能的主要切入点。

一个 StreamingContext 对象可以从 SparkConf 中创建：

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

val conf = new SparkConf().setAppName(appName).setMaster(master)
val ssc = new StreamingContext(conf, Seconds(1))
```

appName 参数是应用程序奖在集群 UI 上显示的名称。master 是 Spark、Mesos 或 YARN 集群 url 地址,或者一个特殊的“local[*]”字符串可以在本地模式下运行。事实上,在一个集群上,你不想在程序里写死 master，可以通过 spark-submit 启动应用程序，在程序里接受 master。然而，对于本地测试和单元测试，您可以通过“local[*]” 运行 Spark Streaming 进程(检测到本地系统的核心)。注意，在内部创建一个 SparkContext(启动 Spark 功能)，可以作为 ssc.sparkContext 访问。

批间隔设置必须基于应用程序的延迟需求和可用的集群资源，更详细请查看性能调优部分。

一个 StreamingContext 对象也可以从现有 SparkContext 对象中创建。

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

val sc = ...                // existing SparkContext
val ssc = new StreamingContext(sc, Seconds(1))
```

当一个 context 对象被定义，你可以进行以下操作：

1.通过创建输入 DStreams 定义输入源。

2.通过在 DStreams 上应用转换和输出操作可以定义流计算。

3.使用 streamingContext.start()可以开始接收数据和处理数据。

4.使用 streamingContext.awaitTermination()可以停止正在等待中的处理。(手动或由于任何错误)

5.使用 streamingContext.stop()处理可以手动停止。

以下几点需记住：

1.一旦 context 已经启动，没有新的流计算可以设置或添加到它。

2.一旦 context 已经停止，它就不能被重新启动。

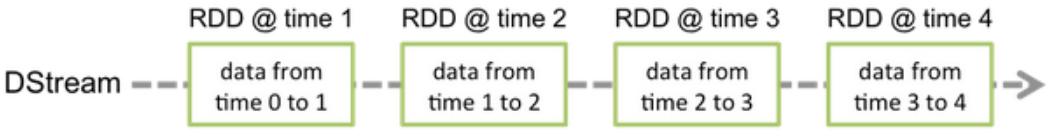
3. 在同一时间一个 JVM 只能激活一个 StreamingContext。

4.stop()在 SparkContext StreamingContext 也停止。停止只 StreamingContext,停止 ()calledstopSparkContext 假的可选参数。

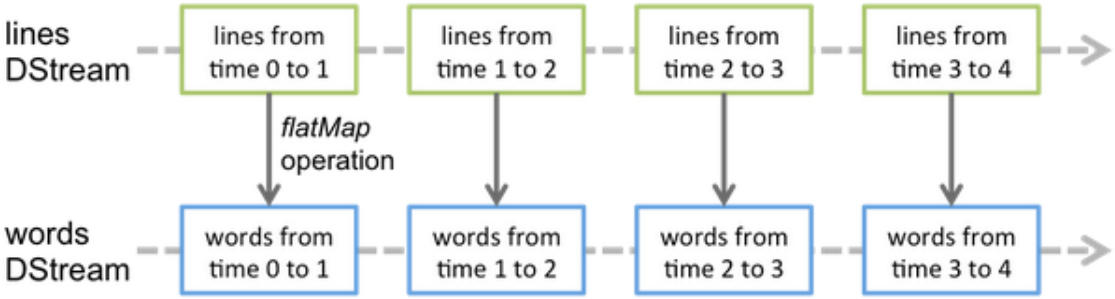
5. SparkContext 可以重复使用，创建多个 StreamingContexts，只要前一个 StreamingContext 停止 (不需要停止 SparkContext)，后一个 StreamingContext 就可以创建。

1.3.3 离散流(DStreams)

*离散流*或 *DStream* 是 Spark Streaming 提供的基本抽象。它代表了一个连续的数据流，或者从源端接收到的，或通过将输入流中产生处理后的数据流。在内部，它是由 RDDS 的连续序列表示，这是 Spark 的一个不可改变的，分布式的数据集抽象。在 DStream 中每个 RDD 包含数据从某一个时间间隔，如图所示。



任何在 DStream 上使用的操作都会转化为底层 RDDS 操作。例如，在前面的示例从行 stream 中转换为单词，在 flatmap 操作是在每个 RDD 施加在 lines DStream 生成的该 RDDS 上的 words DStream。如下图所示。



这些底层的 RDD 转换是由 Spark engine 计算的到。为使用方便,该 DStream 操作隐藏了大部分的细节，并提供给开发者更高层的 API。这些操作细节在后面的章节中讨论。

1.3.4 输入离散流

输入 DStreams 和 DStreams 接收的流都代表输入数据流的来源。在下面简单的例子中，lines 是一个输入流，它接收到 Netcat 服务器的数据流。每个输入流(除文件流，在这一节后部分讨论)与接收器(Scala 文档,Java 文档)对象从源接收数据并将其存储在 Spark 的内存进行处理。

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、套接字连接和 Akka actors。
- *高级来源*：来源如 Kafka、Flume、Kinesis、Twitter 等，可以通过额外的实用工具类创建。这些需要额外的连接可以参考讨论这个链接的讨论部分。

我们将在本节后面讨论目前每个类别里的一些来源。

注意，如果你想在你的 Streaming 应用中接收多个并行数据流，您可以创建多个输入 DStreams(进一步的讨论参考性能调优部分)，同时这将创建多个接收器接收多个数据流。但是请注意，Spark worker 或 executor 作为一个长时间运行的任务，它将占 Spark Streaming 应用中分配的一个内核。因此，重要的是要记住，Spark Streaming 应用需要分配足够的核心(或线程，如果在本地运行)来处理接收的数据以及运行的接收器。

记住要点：

- 当一个 Spark Streaming 程序在本地运行时，不要使用“local”或“local[1]”作为 master URL。这意味着只有一个线程将被用于在本地运行的任务。如果您使用的是基于一个接收器(例如套接字、Kafka、Flume 等)的输入 DStream，则将使用单线程运行接收器，不留其它线程处理接收到的数据。因此，在本地运行时，总是使用“local[n]”作为 master URL，其中 n >运行接收器的数量(参见 Spark 有关如何设置 master 的信息)。

- 在集群中合理运行，分配给 Spark Streaming 应用的核心数量必须超过接收器的数量，否则系统将能接收数据，但无法处理。

基础来源

我们已在查看 ssc.socketTextStream(...)的简单例子，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 API 提供了方法从文件和 Akka actors 中创建 DStreams 作为输入源。

文件流: 在任何文件系统，通过 HDFS API(如：HDFS、S3、NFS 等等)从文件中读取数据，可以创建一个 DStream。

```
streamingContext.fileStream[KeyClass, ValueClass, InputFormatClass](dataDirectory)
```

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)。请注意：

- 必须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。所以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)，文件流不需要运行一个接收器，因此不需要配置内核。

Python API 在 Spark 1.2 Python API 中， 文件系统还不可以使用，仅能使用 textFileStream 读取本地文件。

基于自定义 Actors 的流: DStreams 可以被创建，通过 Akka actors 接受数据流，使用方法 streamingContext.actorStream(actorProps, actor-name)。更多详细信息，请参见自定义接收器指南。

Python API 目前基于 actor 的流在 Python API 中不能使用，仅能在包含 Java 或 Scala 包中使用。

基于 RDD 队列的流: 通过测试数据测试一个 Spark Streaming 应用，可以创建一个 DStream 基于 RDD 队列,使用 streamingContext.queueStream(queueOfRDDs)。每个 RDD 队列将被视为 DStream 中一块数据，流一样加工处理。

更多关于套接字流、文件流、actor 流的详细资料，请查看相关 API 文档：Scala 的 StreamingContext，Java 的 JavaStreamingContext 和 Python 的 StreamingContext。

高级来源

Python API 在 Spark 1.2 Python API 中，这些来源不可用。

这一类的来源需要外部 non-Spark 库的接口，其中一些与复杂的依赖关系(如 Kafka and Flume)。因此，以最小化适应版本冲突问题，这些来源的功能创建 DStreams 已经搬到独立的库，必要时可以明确。例如，如果您想创建一个 DStream 使用 Twitter tweets 的数据流，您必须按以下步骤来做。

1. ***前提***：在 SBT 或 Maven 工程里添加 spark-streaming-twitter_2.10 依赖。
2. ***开发***：导入 TwitterUtils 包，如下所示通过 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 spark-streaming-twitter_2.10 及其依赖)，然后部署应用程序。进一步解释参考部署部分。

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

TwitterUtils.createStream(ssc)
```

注意，这些高级的来源在 Spark shell 中不可用，因此基于这些高级来源的应用不能在 shell 进行测试。如果你真得想在 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 3.0.3 版本的 Twitter4j，通过使用 Twitter 的流 API 获取公众的 tweets 的流。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。查看 API 文档(Scala, Java)和(TwitterPopularTags 和 TwitterAlgebirdCMS)例子。
- Flume: Spark Streaming 1.2.0 可以从 Flume 1.4.0 中接受数据。更多详细请参考 Flume 集成指南。
- Kafka: Spark Streaming 1.2.0 可以从 Kafka 0.8.0 中接受数据。更多详细请参考 Kafka 集成指南。
- Kinesis: 更多详细请参考 Kinesis 集成指南。

自定义来源

Python API 在 Spark 1.2 Python API 中，这些来源不可用。

输入 DStreams 也可以创建自定义的数据源，所以你需要做的就是实现一个用户定义的接收器(详细了解参考下一节)可以接收来自自定义数据源的数据，把它推到 Spark。有关详细信息，请参考自定义接收器指南。

接收器的可靠性

这里有两种可靠的数据来源。来源(如 Kafka 和 Flume)允许传输数据，如果系统接收数据从这些可靠的来源承认正确的数据，它可以确保没有数据丢失，因为任何一种失败，将导致两种接收器接受失败。

1．可靠的接收器 - 一个可靠的接收器承认可靠来源的数据，能接受并存储在 Spark 上。

2．不可靠的接收器 - 这些是接收器的来源不支持承认。即使是可靠的来源，一个也许实现了一个不可靠的接收器，不去承认正确的复杂性。

如何编写可靠的接收器请参考自定义接收器指南。

1.3.5 离散流转换

类似 RDD 转换允许输入的流可以被修改。DStreams 支持多种变换的基本 Spark RDD 的使用。通用的操作如下：

Transformation	Meaning
map(*func*)	源 DSTREAM 的每个元素通过函数 func 返回一个新的 DSTREAM。

Spark 亚太研究院
www.sparkinchina.com
TEL: 4006-998-758

flatMap(*func*)	类似的图，但每个输入项可以被映射到 0 或者更多的输出项。
filter(*func*)	在源 DSTREAM 上选择 FUNC 函数返回仅为 true 的记录,最终返回一个新的 DSTREAM 。
repartition(*numPartitions*)	通过创建更多或更少的分区改变平行于这个 DSTREAM 的层次。
union(*otherStream*)	返回一个包含源 DStream 与其他 DStream 元素联合后新的 DSTREAM。
count()	在源 DSTREAM 的每 RDD 的数目计数并返回包含单元素 RDDS 新的 DSTREAM。
reduce(*func*)	使用函数 func （有两个参数并返回一个结果）将源 DStream 中的每个 RDD 进行元素聚合,返回一个单元素 RDDs 新的 DStream.该函数应该关联，使得它可以并行地进行计算。
countByValue()	当请求 DStream 类型为 K 元素的 DStream,返回类型为（K，Long）的键值对的新 DSTREAM,其中每个键的值就

	是它的源 DSTREAM 每个 RDD 频率。
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,其中每个键的值都是使用给定的 reduce 函数汇总。注意：默认情况下，使用 Spark 的并行任务默认号码（2 为本地模式，并且在集群模式的数目是由配置属性确定 spark.default.parallelism）进行分组。你可以通过一个可选 numTasks 参数设置不同数量的任务。
join(*otherStream*, [*numTasks*])	当被调用类型分别为的（K，V）和（K，W）键值对的 2 个 DStream 时，返回包含所有键值对每个键的元素,类型为（K，(V，W)）键值对的一个新 DSTREAM。
cogroup(*otherStream*, [*numTasks*])	当被调用的 DStream 含有(K, V) 和(K, W)键值对时,返回(K, Seq[V], Seq[W])元组的新 DStream.
transform(*func*)	通过源 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。

Spark 亚太研究院
www.sparkinchina.com
TEL: 4006-998-758

updateStateByKey(*func*)	返回一个新 “状态”的 DStream,所在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后的更新。这可以被用来维持每个键的任何状态数据。

最后两个转换是值得再次高亮显示。

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，你就必须做两个步：

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，从输入流中的新值更新状态。

让我们用一个例子来说明。假设你要维护可见的文本数据流中每个字的运行计数。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：

```
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)
}
```

此函数用于含有字的 DStream(也就是说前面的示例中 ,在 DSTREAM 含(word ,1) 键值对)。

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

每一个字都将调用更新函数，newValues 具有序列 1（从（word，1）键值对）且 runningCount 表示以前的计数。有关完整的 Scala 代码，看看这个例子 stateful_network_wordcount.py。

Note that using updateStateByKey requires the checkpoint directory to be configured, which is discussed in detail in the checkpointing section.

转换操作

该转换操作（连同其变化如 transformWith ）允许 DStream 上应用任意 RDD 到 RDD

函数。它可以被应用于未在 DStream API 中暴露任何 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。但是，您可以轻松地使用转换来做到这一点。这使得功能非常强大。例如，如果你想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。

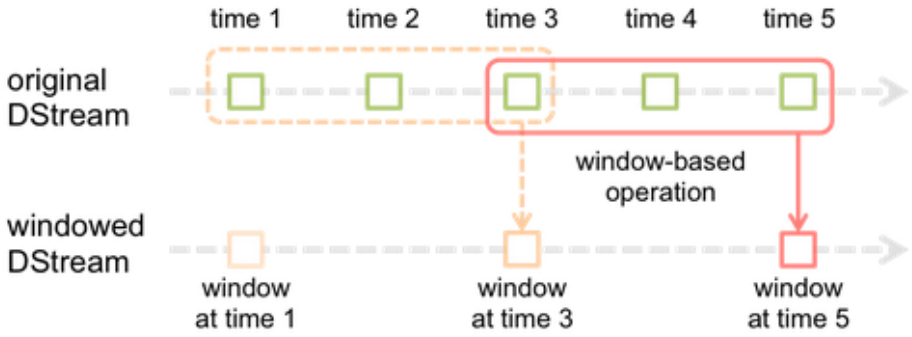
```
val spamInfoRDD = ssc.sparkContext.newAPIHadoopRDD(...) // RDD containing spam information

val cleanedDStream = wordCounts.transform(rdd => {
  rdd.join(spamInfoRDD).filter(...) // join data stream with spam information to do data cleaning
  ...
})
```

事实上，你也可以在转换方法中使用机器学习和 图形计算的算法。

窗口操作

最后，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。下图阐述了这种滑动窗口。



如该图所示，每次当窗口*滑动*过源 DStream，落入窗口内源 RDDS 被组合和操作时，产生的加窗 DStream 的 RDDS。在此特定情况下，该操作是由 2 个滑动时间单元施加在数据的最后 3 个时间单位。这表明任何窗口操作需要指定两个参数。

- *窗口长度* -该窗口（图中 3）的持续时间
- *滑动间隔* - （图 2），在其上执行的窗口操作的时间间隔。

这两个参数必须是源 DSTREAM（1 中所示）的批次间隔的倍数。

让我们来说明窗口的操作用一个例子。比方说，你想扩展 前面的例子所产生过的数据，每 10 秒的最后 30 秒字数技术。要做到这一点，我们在数据的最后 30 秒的 DStream 的键值对(word ,1)上应用 reduceByKey 操作 。这是通过使用操作 reduceByKeyAndWindow。

```
// Reduce last 30 seconds of data, every 10 seconds
val windowedWordCounts = pairs.reduceByKeyAndWindow((a:Int,b:Int) => (a + b), Seconds(30), Seconds(10))
```

一些常见的窗口的操作如下所述。所有这些操作用到所述两个参数- windowLength 和 slideInterval。

Transformation	Meaning
window(*windowLength*, *slideInterval*)	返回一个基于源 DStream 的窗口批次计算得到新的 DStream。
countByWindow(*windowLength*,*slideInterval*)	返回一个流中的元素的滑动窗口计数。
reduceByWindow(*func*, *windowLength*,*slideInterval*)	返回一个新的单件流 ,通过 FUNC 及在使用滑动间隔对数据流中的元素聚合。该函数应该联合以便它可以正确地在并行计算。
reduceByKeyAndWindow(*func*,*windowLength*, *slideInterval*, [*numTasks*])	当调用 DStream(K ,V)对时 ,返回新(K ,V)对的 DSTREAM ，其中每个键的值是根据给定的 reduce 函数 FUNC 和滑动窗口批次汇总。注意：默认情况下，这里使用 Spark 的并行任务默

	认号码（2 本地模式，并在集群模式下的数量由配置属性决定 spark.default.parallelism）做分组。你可以通过一个可选 numTasks 参数设置不同数量的任务。
reduceByKeyAndWindow(*func*, *invFunc*,*windowLength*, *slideInterval*, [*numTasks*])	一种比之前更有效的版本是 reduceByKeyAndWindow（），其中使用减少先前窗口的值的增量来计算每个窗口的减少值。这是通过减少输入的滑动窗口中的新数据完成的，而“逆减少”即离开窗口的旧数据。一个例子是“添加”和“减”键作为滑动窗口的数量。然而，它仅适用于“可逆的 reduce 函数”，即，那些 reduce 函数具有一个对应的“逆 reduce”函数（作为参数函数 invFunc 比如在 reduceByKeyAndWindow，减少任务的数量是通过一个可选的参数进行配置。
countByValueAndWindow(*windowLength*,*slideInterval*, [*numTasks*])	当被调用的（K，V）对的 DStream，返回新的（K，Long）对的 DSTREAM，对其中每个键的值是其在一个滑动窗口频率。和 reduceByKeyAndWindow 一样，减少任务的数量是通过一个可选的参数进行配置。

API 文档中提供 DSTREAM 转换的完整列表。对于 Scala 的 API，参考 DSTREAM 和 PairDStreamFunctions。对于 Java 的 API，参考 JavaDStream 和 JavaPairDStream。想要 Python 的 API，请查看 DStream。

1.3.6 在 DStreams 输出操作

输出操作允许 DSTREAM 的数据被推送出外部系统，如数据库或文件系统。由于输出操作实际上使变换后的数据通过外部系统被使用，它们触发所有 DSTREAM 转换的实际执行（类似于 RDDS 操作）。目前，以下输出操作被定义为：

Output Operation	Meaning
print()	首先在 Dirver 上打印每一批 DStream 数据中的 10 个元素。这对于开发和调试。
saveAsObjectFiles(*prefix*, [*suffix*])	将 DSTREAM 的内容为序列化对象并保存为 SequenceFile 文件。根据产生在每个批次间隔的文件名前缀和后缀：“前缀 TIME_IN_MS [.后缀]” 。
saveAsTextFiles(*prefix*, [*suffix*])	保存此 DSTREAM 的内容作为文本文件。根据产生在每个批次间隔的文件名前缀和后缀：“前缀 TIME_IN_MS [.后缀]” 。
saveAsHadoopFiles(*prefix*, [*suffix*])	保存此 DSTREAM 的内容为 Hadoop 的文件。根据产生在每个批次间隔的文件名前缀和后缀：“前缀 TIME_IN_MS [.后缀]” 。

foreachRDD(*func*)	Note that the function *func* is executed at the driver, and will usually have RDD actions in it that will force the computation of the streaming RDDs. 最通用的输出操作，是使用函数 FUNC，从流中生成的每个 RDD。这个函数可以将每个 RDD 的数据推送到外部系统，如保存 RDD 为文件，或通过网络写到数据库。注意，函数 FUNC 是在 driver 执行 ,并且通常将具有在它的 RDD 行为，将强制执行 RDDS 的计算。

设计模式中使用 foreachRDD

dstream.foreachRDD 是一个强大的原语，它允许数据发送到外部系统。但是，要了解如何正确，有效地使用这种原语是很重要的。为避免一些常见的错误现报告如下:

- 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尝试在 Spark driver 创建连接对象 ,并尝试使用它保存 RDDS 记录到 Spark worker。例如(在 Scala)

```
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 发送到 worker。跨机器间很少转换这种连接对象。此错误可能表现为序列错误（连接对不可序列化），初始化错误（连接对象在 worker 中需要初始化），等等。正确的解决办法是在 worker 创建的连接对象。

Spark 亚太研究院
www.sparkinchina.com
TEL: 4006-998-758

但是，这可能会导致另一个常见的错误 - 为每一个记录的一个新的连接。例如，

```
dstream.foreachRDD(rdd => {
    rdd.foreach(record => {
        val connection = createNewConnection()
        connection.send(record)
        connection.close()
    })
})
```

通常情况下，创建一个连接对象有时间和资源开销。因此，创建和销毁的每个记录中的连接对象可以招致不必要的高开销，并可以显著降低系统的整体吞吐量。一个更好的解决方案是使用 rdd.foreachPartition -创建一个单独的连接对象，并使用该连接发送的所有记录在 RDD 分区。

```
dstream.foreachRDD(rdd => {
    rdd.foreachPartition(partitionOfRecords => {
        val connection = createNewConnection()
        partitionOfRecords.foreach(record => connection.send(record))
        connection.close()
    })
})
```

这分摊了接创建多条记录连接的开销。

最后，这可以进一步通过重用在多个 RDDS / batches 的连接对象进行了优化。一种能够保持连接对象的静态池，可以重用作多个 batches 的 RDDS 推到外部系统，从而进一步降低了开销。

```
dstream.foreachRDD(rdd => {
    rdd.foreachPartition(partitionOfRecords => {
        // ConnectionPool is a static, lazily initialized pool of connections
        val connection = ConnectionPool.getConnection()
        partitionOfRecords.foreach(record => connection.send(record))
        ConnectionPool.returnConnection(connection)  // return to the pool for future reuse
    })
})
```

需要注意的是，在池中的连接应该按需延迟创建和超时(如果一段时间内不使用)。这实现了最有效的数据发送到外部系统。

另一点要记住：

- DStreams 由输出操作延迟方式执行的，就像 RDDS 由 RDD actions 延迟方式执行。具体来讲，DSTREAM 输出操作内部的 RDD actions 迫使所接收的数据的处理。因此，如果你的应用程序没有任何输出操作，或有在内部没有任何 RDD action 输出操作，如 dstream.foreachRDD（ ），那么什么都不会得到执行。系统将简单地接收的数据，并丢弃它。
- 默认情况下，输出操作一个在一次一执行。并且它们在它们的应用中定义的顺序执行。

1.3.7 缓存/持久化

类似 RDDS，DStreams 还允许开发者把流的数据持久化到内存中。也就是说，在 DStream 用 persist() 方法将自动持久化 DSTREAM 中每一个 RDD 到内存中。如果在 DSTREAM 的数据将被计算多次（例如，相同的数据上做多个操作），这是很有用的。对于像基于窗口的操作 reduceByWindow 和 reduceByKeyAndWindow 和基于状态的操作例如 updateStateByKey，这是隐含为真的。因此，通过基于窗口的操作产生 DStreams 会自动持久化到内存中，无需开发人员调用 persist()。

对于通过网络接收数据（例如，Kafka，Flume，安全套接字等）的输入数据流，默认持久化等级被设定为将数据复制到用于容错的两个节点。

需要注意的是，不像 RDDS，DStreams 的默认持久化等级将序列化数据到内存中。这是在进一步讨论性能调优部分。在不同的持久化等级的详细信息，可以查找 Spark 编程指南。

1.3.8 检查点

一个流应用必须运行 24/7，因此必须与任务失败不相关(如系统故障、JVM 崩溃等等)。为了这种可能，Spark Streaming 需要检查足够的错误信息、容错存储系统，从而可以从任务失败中恢复过来。有两种类型的数据可以设置检查点。

- 元数据检查点——保存定义流计算的信息，像 HDFS 容错存储。这可以用来恢复从节点运行流应用的任务失败(稍后详细讨论)。元数据包括：
  - *配置——用于创建流的配置应用程序。*
  - *流操作——定义一组流操作的流应用程序。*
  - *不完整的批处理-批处理任务进行队列但尚未完成。*
- 检查点——保存生成 RDD 的可靠存储。在某些跨多个批处理合并数据转换上，这是必要的。这种转换，生成的 RDD 取决于 RDD 之前的批处理，这会导致依赖链的

长度随时间不断增加。为了避免这种无限增加的时间恢复(附属关系链)，定期转换中间 RDD 的状态，设置检查点，切断依赖链，可以增加可靠存储(例如 HDFS)。

总结：元数据检查点主要是恢复失败的驱动程序，而数据或 RDD 检查点是必要的，为了基本功能使用。

当启用检查点

为应用程序启用检查点有下列要求：

- *使用有状态的转换——如果 updateStateByKey 或 reduceByKeyAndWindow(逆函数)在应用程序中使用，则必须提供检查点目录允许定期 RDD 检查。*
- *从故障中恢复驱动运行应用——元数据使用检查点恢复程序信息。*

请注意，简单的流应用没有设置检查点，就不会有上述状态转换运行。从故障中恢复驱动应用程序也将部分在这种情况下(有些收到但未加工的数据可能会丢失)。这通常是可接受的，许多以这种方式运行流应用程序。支持 non-Hadoop 环境在未来有望改善。

如何配置检查点

检查点可以通过设置一个目录启用容错，可靠的文件系统(如：HDFS,S3 等)的检查点信息将被保存。这是通过使用 streamingContext.checkpoint(checkpointDirectory)，这将允许您使用上述状态转换。此外，如果你想要使应用程序从故障中恢复，你应该重写你的流应用程序有以下操作：

- 当程序被首次开始，它将创建一个新的 StreamingContext，设置所有的流，然后调用 start()。
- 当程序因为故障而正在重启，它将重新创建一个 StreamingContext，并检查在检查点目录中的数据。

这种操作可以通过使用 StreamingContext.getOrCreate 简单创建，使用如下：

```
// 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 StreamingContext 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()
```

如果检查点目录存在，通过检查点数据，context 对象可以重新创建。如果该目录不存在(即首次运行)，那么该函数 functionToCreateContext 将创建一个新的 context 对象，同时设置 DStreams。参考 Scala 中 RecoverableNetworkWordCount 示例，这个例子将网络数据中单词汇总，并插入一个文件中。

除了使用 getOrCreate 也需要确保驱动程序出现故障会自动重新启动。这只能通过部署基础设施，用于运行应用程序。进一步讨论请参考部署章节。

注意检查点抽样导致储蓄可靠存储的成本。这可能导致这些批次的处理时间的增加,抽样得到设置检查点。因此,需要设置检查点间隔的小心。在小批量的大小(1 秒)说,检查点每批

Spark 亚太研究院
www.sparkinchina.com
TEL: 4006-998-758

可能显著降低操作的吞吐量。相反,检查点太少导致的血统和任务大小增长可能有不利影响。对于需要抽样检查点的状态转换,默认间隔的多个批处理间隔至少 10 秒。它可以设置通过使用 dstream.checkpoint(checkpointInterval)。通常,一个检查点间隔的 5 - 10 倍的滑动时间间隔 DStream 设置尝试是好事。

1.3.9 部署应用程序

这一章节主要讨论部署一个 Spark Streaming 应用的步骤。

必备

运行一个 Spark Streaming 应用，你必须有以下步骤：

- *集群与集群管理器——这是任何 Spark 应用程序的一般要求,详细讨论参考部署指南。*
- *应用程序 JAR 打包——你必须将流应用程序编译成一个 JAR。如果你使用的是 spark-submit 启动应用程序，那么你将不需要提供 JAR 的 Spark 和 Spark Streaming。然而 ,部署 JAR 应用程序 ,如果您的应用程序使用高级来源(如 ,Kafka、Flume、 Twitter)，那么您将不得不提供额外的链接，包括他们的依赖包。例如，一个应用程序使用 TwitterUtils 必须包括 spark-streaming-twitter_2.10 及其应用程序 JAR 依赖包。*
- *配置足够的内存为 worker——因为接收到的数据必须存储在内存中,执行者必须配置足够的内存来保存接收到的数据。请注意,如果你在做 10 分钟的窗口操作,系统必须保持至少 10 分钟的数据在内存中。因此应用程序的内存需求取决于使用的操作。*
- *配置检查点——如果流应用程序需要它,然后在 Hadoop API 兼容容错存储目录(例如 HDFS,S3 等)必须配置作为检查点目录和流媒体应用程序编写,检查点信息可以用于故障恢复。详情请参见检查点部分。*

- *配置自动重启应用程序驱动的——从一个司机自动恢复失败,部署基础设施,用于运行流媒体应用程序必须监视驱动过程和重新司机如果失败了。不同的集群经理有不同的工具来实现这一目标。*
  - *Spark 独立——Spark 可以提交给应用程序驱动程序运行在独立的 Spark 集群(见集群部署模式),也就是说,应用程序驱动程序本身一个职工的节点上运行。此外,独立集群管理器可以指示 tosupervise 驱动程序,并重新启动它如果司机失败由于非零退出代码,或由于故障节点的运行驱动程序。看到 Spark 独立集群模式和监督指导的更多细节。*
  - *YARN - Yarn 支持类似的机制来自动重新启动应用程序。更多细节请参阅 YARN 文档。*
  - *Mesos——Marathon 就是用 Mesos 来实现的。*
- *(实验在 Spark1.2)配置提前写日志——在 Spark1.2 中,我们引入了一个新实验的特点实现强大的容错担保之前写日志。如果启用,所有从接收机接收到的数据被写入之前写日志配置检查点中的目录。这可以防止数据丢失在司机复苏,从而确保了零数据丢失(容错语义部分中详细讨论)。这可以通过设置启用配置 parameterspark.streaming.receiver.writeAheadLogs.enable 为 true。然而,这些强大的语义可能在个别接收器的接收吞吐量的成本。这可以纠正了并行运行多个接收者增加总吞吐量。此外,建议接收的数据的复制在 Spark 启用被禁用时提前写日志的日志已经存储在一个存储系统复制。这可以通过设置存储输入流 toStorageLevel.MEMORY_AND_DISK_SER 水平。*

升级应用程序代码

如果正在运行的 Spark Streaming 应用程序需要升级使用新的应用程序代码，那么有两种可能的机制。

亚太研究院
www.sparkinchina.com
TEL: 4006-998-758

- 升级后的 Spark Streaming 应用程序启动和并行运行现有的应用程序。一旦新的（与旧的接收到相同的数据）已被预热并就绪，旧的程序就可以停用。请注意，这种方式适合于支持数据发送到两个目的地（即早期和升级的应用程序）的数据源来完成。
- 现有的应用程序是正常关闭（见 StreamingContext.stop（...）或 JavaStreamingContext.stop（...）为正常关机的选项），确保已接收关闭之前已经完全处理数据。然后在升级应用程序可以启动，这将从之前程序停止的同一点开始处理。注意，这只能由输入信号源完成,当之前的程序停止和升级的应用尚不起来的时候，它支持输入源侧缓冲（如 Kafka 和 Flume）.

其它考虑的地方

如果由接收器接收数据的速度比我们处理的速度快，你可以通过配置 parameterspark.streaming.receiver.maxRate 来限制。

1.3.10 监控应用

超出 Spark 监视功能，对于 Spark Streaming 有额外的功能。当一个 StreamingContext 被使用，Spark web 界面将显示了一个额外的流选项卡，显示统计信息运行接收器(接收器是否活跃 ,收到的记录数量 ,接收错误等等)和完成批处理(批处理时间、排队延迟等)，这可以用于监控流媒体应用的进展。

以下两个指标在 web 界面尤为重要：

- *处理时间* -每一批数据处理时间。
- *调度延迟*-一个批处理在排队中等待的时间直到前一个批处理完成。

如果批处理时间持续超过一批间隔或排队延迟不断增加，那么就表明该系统无法处理批量程序，在这种情况下，考虑减少批处理时间。

一个 Spark Streaming 程序可以使用 StreamingListener 接口监控，它允许你获得接收器状态和处理时间。注意 ,这是一个开发人员 API ,在未来它可能会改进(更多信息报道)。

1.4 性能优化

获取一个集群上的 a Spark Streaming 应用程序的最佳性能需要一些调整的。本节介绍了一些可以调整的参数和配置，以提高你的应用程序的性能。在较高的角度上，你需要考虑两件事情：

1. 通过有效地利用群集资源减少每批数据的处理时间。
2. 设置合适的批次大小，使得数据的批次可以尽可能快地接收并被处理（即，数据处理紧跟在数据摄取）。

1.4.1 减少每个批次的处理时间

在 Spark 中可以做很多优化从而减少每个批次的处理时间。这些都被详细讨论在调优指南中。本节重点介绍一些最重要的。

并行结构的数据接收

通过网络接收（如 Kafka, Flume, socket 等）数据需要数据被反序列化并存储在 Spark 中。如果数据接收成为系统中的瓶颈，那么考虑并行数据接收。请注意，每个输入 DStream 创建单个接收器（运行在 worker 的机器上）用来接收单一的数据流。接收多个数据流可因此通过创建多个输入 DStreams 并配置它们以从源数据流中的不同分区接收数据来实现。例如，一个单一的 Kafka 输入 DStream 接收数据的两个主题可以被分成两个 Kafka 输入流，每个仅接收一个主题。这将在两个 worker 运行两个 receiver，因而允许并行接收数据，并提高整体的吞吐量。这些多 DSTREAM 可以联合在一起创建一个单一的 DSTREAM。然后这个单一输入 DSTREAM 转换为统一的流。可以做如下操作：

```
val numStreams = 5
val kafkaStreams = (1 to numStreams).map { i => KafkaUtils.createStream(...) }
val unifiedStream = streamingContext.union(kafkaStreams)
unifiedStream.print()
```

应当考虑的另一个参数是 receiver 的阻塞间隔。对于大多数的 receiver，在存储到 Spark 之前接收到的数据被合并在一起形成大块数据。在每批次的块的数量决定了将被用于处理在地图样变换的那些接收数据的任务数量。这种阻塞间隔由 的配置参数 spark.streaming.blockInterval 确定,其默认值是 200 毫秒。

另一种从多个输入流/receiver 接受数据是显示地对输入数据流重新分区（使用 inputStream.repartition（<分区数>））。在集群中进一步处理之前,它将数据分发接收批次到指定数目的机器上。

Spark 亚太研究院
www.sparkinchina.com
TEL: 4006-998-758

并行结构的数据处理

如果在计算的任何阶段使用的并行任务的数量不够高,则群集资源利用不足。例如，对于分布式 reduce 的操作如 reduceByKey 和 reduceByKeyAndWindow ,并行任务的默认数量由[配置属性]（(configuration.html#spark-properties) spark.default.parallelism ）决定。您可以通过并行的等级作为自变量（见[PairDStreamFunctions]（ api/scala/index.html#org.apache.spark.streaming.dstream.PairDStreamFunctions ）文档），或者设置配置属性 spark.default.parallelism 来更改默认值。

数据序列化

数据序列化的开销显著，特别是当要实现亚秒级的 batch 规模。有两个方面:

- Spark RDD 数据序列化 :请参阅调优指南中数据序列化的详细讨论。但是 ,请注意，不像 Spark，默认情况下 RDDS 是以序列化的字节数组持久化，以减少相关的 GC 暂停。
- 输入数据的序列化：将外部数据导入 Spark，收到的字节数据（例如，从网络）需要从字节反序列化并按照 Spark 的序列化格式重新序列化。因此，输入数据的反序列化的开销可能是一个瓶颈。

任务启动管理开销

如果每秒发起的任务数目很高（例如，每秒 50 或更多），那么给从站发送的任务开销可能显著和将难以达到亚秒级延迟的。以下更改可以减少开销：

- 任务序列化 :使用 Kryo 序列化任务可降低任务的大小 ,因此降低其发送到从站的时间。
- 执行模式：在独立模式或粗粒 Mesos 模式下运行 Spark 带来更好的任务启动时间比细粒度 Mesos 模式。更多细节请参考 Mesos 运行指南。

这些变化可能会缩短 100S 毫秒批处理时间，从而使亚秒级 batch 规模是可行的。

1.4.2 设定正确的批次大小

为了使一个集群上 Spark Streaming 应用程序稳定运行，系统处理与接收数据速度应该能够一致。换句话说，batch 数据应该被处理与被产生时那样快。可以通过在 Spark Streaming web 用户界面找到监视的处理时间,从而判断这对于应用程序是否正确设置，其中所述批量的处理时间应小于批次间隔。

根据流计算的性质 ,所用的间歇时间间隔可能对在一组固定的群集资源的应用程序上持续的数据传输速率产生显著影响。例如，让我们考虑之前的 WordCountNetwork 例子。

对于一个特定的数据速率，系统能够跟上每 2 秒（即 2 秒间歇间隔）报告字计数，,而不是每 500 毫秒。因此需要设置这样的批次间隔，使得在生产中的预期数据速率可被维持。

为你的应用程序找出正确的 batch 大小一个好的方法是用一个保守的批次时间间隔(比如 5-10 秒）和低数据速率进行测试。为了验证该系统是否能够跟上数据传输速率，可以检查每一个处理 batch 所经历的端到端延迟的值(或是查找 Spark driver log4j 的 log4j 日志中“延迟总计”的值，或使用 StreamingListener 接口）。如果延迟保持与 batch 大小一致，则系统是稳定的。否则，如果延迟不断增加，这意味着该系统是无法跟上，因此是不稳定的。一旦你有一个稳定的配置的想法，你可以尝试提高数据速率和/或减少批量大小。请注意，由于临时数据率的增加导致延迟的瞬时增大也是可接受的，只要延迟会降低到较低的值（即，低于批量大小）。

1.4.3 内存优化

Spark 应用的内存优化和 GC 的性能已经在调优指南中很详细的讨论了。我们建议您阅读。在本节中，我们强调的是几个自定义,即强烈建议在 Spark Streaming 应用尽量减少 GC 相关的暂停并实现更加一致的批处理时间。

- DStreams 的默认持久化等级：不像 RDDS，DStreams 的默认的持久化级别会序列化内存中的数据（也就是 DSTREAM 为 StorageLevel.MEMORY_ONLY_SER 而 RDDS 为 StorageLevel.MEMORY_ONLY）。即使保持数据序列化会带来更高的序列化/反序列化的开销，但它可以显著减少 GC 暂停。
- 清除持久化的 RDDS：默认情况下，按照 Spark 的内置策略（LRU）,Spark Streaming 产生的所有持续化的 RDDS 会从内存中清除。如果设置了 spark.cleaner.ttl，那么持久化的 RDDS 时间超过该值时会定期清除。正如 早些时候提到的，这需要在 Spark Streaming 程序使用的操作的基础上小心设置。然而，通过设置配置属性 spark.streaming.unpersist 为 true 来启用 更智能的非持久化 RDDS。这使得系统找出哪些 RDDS 是没有必要的维持并非持久化他们。这可能会减少 Spark 的 RDD 内存使用情况，也能潜在的提高 GC 的性能。
- 并发垃圾收集器：使用并发 mark-and-sweep GC 进一步减少 GC 暂停的可变性。即使我们知道并发 GC 会减小整个系统处理的吞吐量，但其仍然建议使用以实现更一致的批次处理时间。

1.5 容错特性

在本节中，我们将要讨论一个节点发生故障时的 Spark Streaming 应用程序的行为。要理解这一点，让我们记住 Spark 的 RDDS 基本容错特性。

1.RDD 是一个不可变的，唯一的可重新计算的，分布式的数据集。每个 RDD 记得在容错输入数据集中创建它的唯一操作的血统。

Spark 亚太研究院
www.sparkinchina.com
TEL: 4006-998-758

2.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的容错数据集使用操作的血统重新计算。
3 假设 RDD 转换都是确定的，集群出现故障会改变 RDD 中的数据。

因为在 Spark Streaming 的所有数据转换是基于 RDD 操作，只要输入数据集存在，所有的中间数据可以重新计算。牢记这些特性，我们将详细讨论失败的语义。

Spark 运行在像 HDFS 和 S3 容错文件系统的数据。因此，所有从容错数据生成的 RDDS 也都是容错。然而，这不是因为在大多数情况下，数据接收通过网络（当 fileStreamis 用于除外）为 Spark 流的情况。以实现相同的容错特性为所有所生成的 RDDS 的，所接收的数据被复制之间在集群中的工作节点的多个 Sparkworker（默认复制因子是 2）。这导致两种类型的数据中，需要回收的故障事件的系统：

1. *接收并复制数据 - 该数据生存的单一工作节点故障的副本存在于节点之一。*
2. *收到但缓冲复制数据 - 因为这是不可复制的，只有这样，才能恢复数据从源再次得到它。*

此外，有两种故障，我们应该关心：

1. *一个工作节点的故障 - 任何一个工作节点上运行 worker 可以失败的，并且所有内存中的数据在这些节点上都将丢失。如果有任何接收机中失败的节点上运行，则其缓冲的数据将丢失。*
2. *驱动程序节点失败 - 如果运行 Spark 流媒体应用的驱动器节点出现故障，那么很明显的 SparkContext 丢失，并与他们在内存中的数据全部 worker 丢失。*

有了这些基本的知识，让我们明白 Spark 流的容错语义。

容错文件作为输入源

基于接收机的输入源，则容错语义取决于故障情形和接收器的类型两者。正如我们前面所讨论的，有两种类型的接收机：

1. *可靠的接收器 - 这些接收器后，才确保接收到的数据被复制承认可靠的消息来源。如果这样的接收器发生故障时，被缓冲的（非复制的）数据不能确认到源。如果接收器重新启动后，源将重新发送数据，因此没有数据会因丢失的故障。*
2. *不可靠的接收器 - 他们失败时，由于工人或驱动程序故障这样的接收器丢失数据。*

取决于什么类型的接收器被用于我们实现以下语义。如果工人节点发生故障，则没有数据丢失与可靠的接收器。不可靠的接收器，接收到的数据，但不会复制可能会丢失。如果驾驶员节点发生故障，则除了这些损耗，所有的被接收到的和复制的内存中的过去的数据将丢失。这将影响到状态转换的结果。

为了避免这种损失近接收到的数据，Spark1.2 引入了提前写日志，其中保存接收到的数据容错存储的一个实验性的功能。随着提前写日志启用和可靠的接收器，还有零数据丢失，并恰好一次语义。

下表总结了根据故障的语义：

部署方案	Worker 故障	驱动故障
Spark1.1 及更早版本，或 Spark1.2 不写头日志	不可靠的接收器缓冲的数据丢失 零数据丢失与可靠的接收器和文件	不可靠的接收器缓冲的数据丢失 与所有的接收器丢失了过去的数据 零数据丢失的文件

Spark 亚太研究院
www.sparkinchina.com
TEL: 4006-998-758

Spark1.2 写头日志	零数据丢失与可靠的接收器和文件	零数据丢失与可靠的接收器和文件

容错输出流操作

因为所有数据建模转换为 RDDs 都是他们血统的确定性操作，任何重新计算总是会导致同样的结果。因此，所有 DStream 转换只保证一次容错，最后结果都是相同的，即使是有一个工人节点故障。然而，输出操作(如 foreachRDD)至少有一次容错，也就是说，当一个节点出现故障，转换后的数据可能会不止一次写入外部实体。虽然这是接受存储 HDFS 使用 saveAs***Files 操作(如文件由相同的数据只会书写过度) ,可能需要额外的事务机制实现只有一次容错输出操作。

1.6 从 0.9.1 或以下到 1.x 的迁移指南

Spark 0.9.1 和 Spark 1.0 之间，还有以确保未来 API 的稳定做了几个变化。本节详细阐述将现有代码迁移到 1.0 所需的步骤。

输入 DStreams：

创建一个输入流（例如，StreamingContext.socketStream， FlumeUtils.createStream 等）的所有操作现在在 Scala 中返回 InputDStream / ReceiverInputDStream （而不是 DSTREAM）， 在 Java 中返回 JavaInputDStream / JavaPairInputDStream /JavaReceiverInputDStream / JavaPairReceiverInputDStream (而不是 JavaDStream)。这确保了输入数据流的功能特性在未来可被添加到这些类而不破坏二进制兼容性。请注意，您现有的 Spark Streaming 应用程序不应该要求任何变化（因为这些新类是 DSTREAM / JavaDStream 的子类），但可能需要重新编译以 Spark 1.0。

定制网络接收器：

在之前的 Spark Streaming，自定义网络接收器可以在 Scala 中使用类 NetworkReceiver 定义。然而 ,在 API 中的错误处理和报告方面是有限的 ,并且不能从 Java 调用。从 Starting Spark 1.0，这个类已经被取代为 Receiver 并有以下优点。

- 方法如 stop 和 restart 已添加到更好地控制接收器的生命周期。更多细节请参考自定义的接收器指导。
- 自定义接收器可以同时使用 Scala 和 Java 实现。

从现有的早期 NetworkReceiver 迁移到新的自定义接收器，你需要做到以下几点。

- 使你的自定义接收器类继承 org.apache.spark.streaming.receiver.Receiver 而不是 org.apache.spark.streaming.dstream.NetworkReceiver。
- 此前，BlockGenerator 对象必须通过 custom receiver 创建，接收到的数据被存储在 Spark。它必须通过 OnStart( )和的 onStop( ) 方法显式启动和停止。新的 Receiver

类使得这种不必要的，因为它增加了一套名为方法 store(<data>），可以调用它来存储数据。因此，迁移您的自定义网络接收器，删除任何 BlockGenerator 对象（反正 Spark1.0 后不再存在），并使用 store(...) 方法存储接收到的数据。

基于 Actor 的接收器：

数据可能已被使用任何 Akka Actors 通过继承自 actor 类 org.apache.spark.streaming.receivers.Receiver。这已更名为 org.apache.spark.streaming.receiver.ActorHelper 和 pushBlock（...）方法来存储接收到的数据已被重新命名为 store(...)。在其他辅助类 org.apache.spark.streaming.receivers 包也被搬到了 org.apache.spark.streaming.receiver 包，并为更容易区分而更名。

1.7 从这里到哪

- API 文档
  - Scala 的文档
    - StreamingContext 和 DSTREAM
    - KafkaUtils，FlumeUtils，KinesisUtils，TwitterUtils，ZeroMQUtils 和 MQTTUtils
  - Java 的文档
    - JavaStreamingContext，JavaDStream 和 PairJavaDStream
    - KafkaUtils，FlumeUtils，KinesisUtils TwitterUtils，ZeroMQUtils 和 MQTTUtils
- 在 Scala 和 Java 的更多实例
- 文献和视频描述 Spark Streaming。

Spark 亚太研究院

Spark 亚太研究院是中国最专业的一站式大数据 Spark 解决方案供应商和高品质大数据企业级完整培训与服务供应商，以帮助企业规划、架构、部署、开发、培训和使用 Spark 为核心，同时提供 Spark 源码研究和应用技术训练。针对具体 Spark 项目，提供完整而彻底的解决方案。包括 Spark 一站式项目解决方案、Spark 一站式项目实施方案及 Spark 一体化顾问服务。

官网： www.sparkinchina.com

视频课程：

《大数据 Spark 实战高手之路》 国内第一个 Spark 视频系列课程

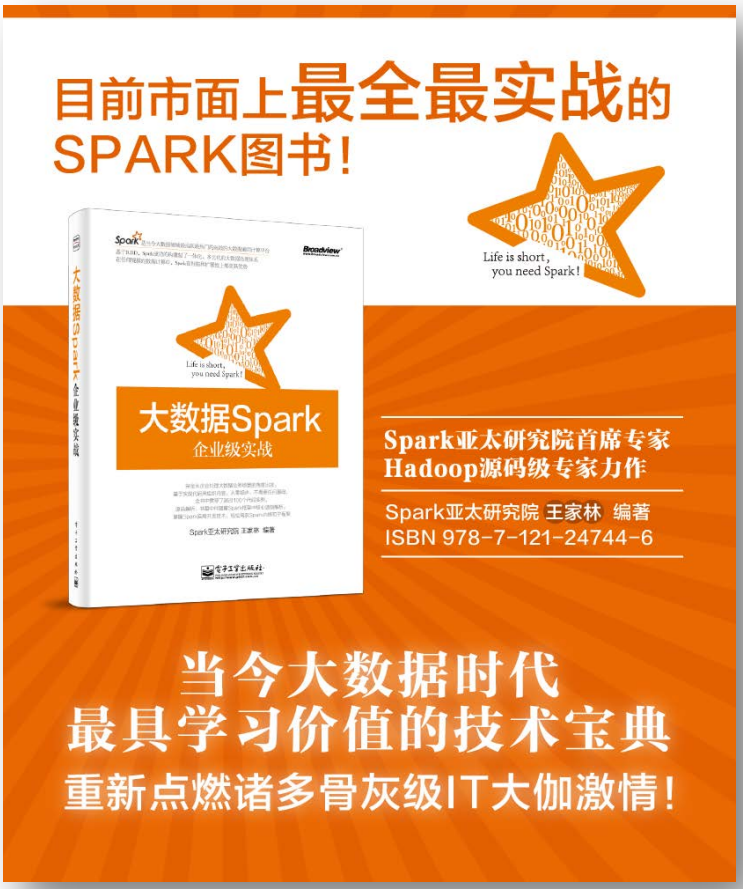
从零起步，分阶段无任何障碍逐步掌握大数据统一计算平台 Spark，从 Spark 框架编写和开发语言 Scala 开始，到 Spark 企业级开发，再到 Spark 框架源码解析、Spark 与 Hadoop 的融合、商业案例和企业面试，一次性彻底掌握 Spark，成为云计算大数据时代的幸运儿和弄潮儿，笑傲大数据职场和人生！

- 第一阶段：熟练的掌握 Scala 语言
  课程学习地址：http://edu.51cto.com/pack/view/id-124.html
- 第二阶段：精通 Spark 平台本身提供给开发者 API
  课程学习地址：http://edu.51cto.com/pack/view/id-146.html
- 第三阶段：精通 Spark 内核
  课程学习地址：http://edu.51cto.com/pack/view/id-148.html
- 第四阶段：掌握基于 Spark 上的核心框架的使用
  课程学习地址：http://edu.51cto.com/pack/view/id-149.html
- 第五阶段：商业级别大数据中心黄金组合：Hadoop+ Spark
  课程学习地址：http://edu.51cto.com/pack/view/id-150.html
- 第六阶段：Spark 源码完整解析和系统定制
  课程学习地址：http://edu.51cto.com/pack/view/id-151.html

图书：

《大数据 spark 企业级实战》

京东购买官网：http://item.jd.com/11622851.html
当当购买官网：http://product.dangdang.com/23631792.html
亚马逊购买官网：http://www.amazon.cn/dp/B00RMD8KI2/



咨询电话：4006-998-758

QQ 交流群： 1 群：317540673（已满）
2 群：297931500（已满）
3 群：317176983
4 群：324099250

微信公众号：spark-china