大数据技术之 SparkCore

Spark Core spark 核心

spark Sql

Spark Streaming 流式计算Spark MLlib 算法库Sspark 图计算。

版本：V1.1

第 **1** 章 **RDD** 概念 弹性分布式数据集

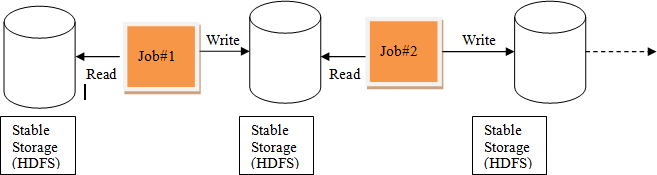
* 1. **RDD 为什么会产生**

RDD 是Spark 的基石，是实现 Spark 数据处理的核心抽象。那么 RDD 为什么会产生呢？

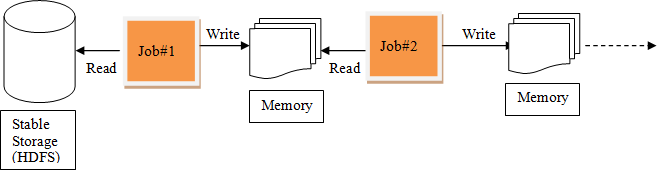
Hadoop 的 MapReduce 是一种基于数据集的工作模式，面向数据，这种工作模式一般是从存储上加载数据集，然后操作数据集，最后写入物理存储设备。数据更多面临的是一次性处理。

MR 的这种方式对数据领域两种常见的操作不是很高效。第一种是迭代式的算法。比如机器学习中 ALS、凸优化梯度下降等。这些都需要基于数据集或者数据集的衍生数据反复查询反复操作。MR 这种模式不太合适，即使多 MR 串行处理，性能和时间也是一个问题。数据的共享依赖于磁盘。另外一种是交互式数据挖掘，MR 显然不擅长。

MR 中的迭代：



Spark 中的迭代：



我们需要一个效率非常快，且能够支持迭代计算和有效数据共享的模型，Spark 应运而生。RDD 是基于工作集的工作模式，更多的是面向工作流。

但是无论是 MR 还是 RDD 都应该具有类似位置感知、容错和负载均衡等特性。

* 1. **RDD 概述**
     1. 什么是 **RDD**

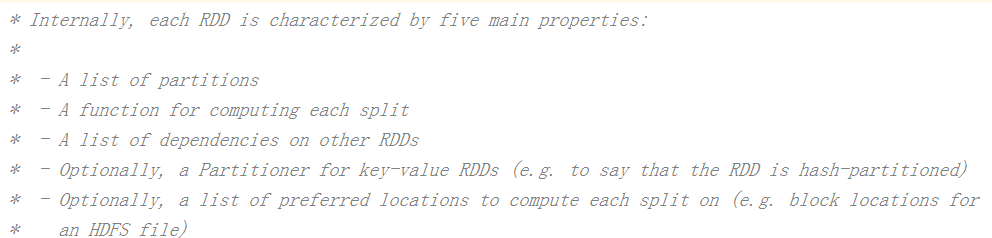
RDD（Resilient Distributed Dataset）叫做分布式数据集，是 Spark 中最基本的数据抽象， 它代表一个不可变、可分区、里面的元素可并行计算的集合。在 Spark 中，对数据的所有操作不外乎创建 RDD、转化（算子）已有 RDD 以及调用 RDD 操作进行求值（执行）。每 个 RDD 都被分为多个分区，这些分区运行在集群中的不同节点上。RDD 可以包含 Python、Java、Scala 中任意类型的对象， 甚至可以包含用户自定义的对象。RDD 具有数据流模型的特点：自动容错、位置感知性调度和可伸缩性。RDD 允许用户在执行多个查询时显式地将工作集缓存（cache）在内存中，后续的查询能够重用工作集，这极大地提升了查询速度。

RDD 支持两种操作:转化(TransFormation(抽象懒加载))操作和行动（action）操作。RDD 的转化操作是返回一个新的 RDD 的操作，比如 map()和 filter()，而行动操作则是向驱动器

程序返回结果或把结果写入外部系统的操作。比如 count() 和 first()。

Spark 采用惰性计算模式，RDD 只有第一次在一个行动操作中用到时，才会真正计算。Spark 可以优化整个计算过程。默认情况下，Spark 的 RDD 会在你每次对它们进行行动操作时重新计算。如果想在多个行动操作中重用同一个 RDD，可以使用 RDD.persist() 让Spark 把这个 RDD 缓存下来。

* + 1. **RDD** 的属性



1. 一组分片（Partition），即数据集的基本组成单位。对于 RDD 来说，每个分片都会被一个计算任务处理，并决定并行计算的粒度。用户可以在创建 RDD 时指定 RDD 的分片个数，如果没有指定，那么就会采用默认值。默认值就是程序所分配到的 CPU Core 的数目。
2. 一个 core 计算每个分区的函数。Spark 中 RDD 的计算是以分片为单位的，每个 RDD 都会实现 compute 函数以达到这个目的。compute 函数会对迭代器进行复合，不需要保存每次计算的结果。
3. RDD 之间的依赖关系。RDD 的每次转换都会生成一个新的 RDD，所以 RDD 之间就会形成类似于流水线一样的前后依赖关系(血统关系)。在部分分区数据丢失时，Spark 可以通过这个依赖关系重新计算丢失的分区数据，而不是对 RDD 的所有分区进行重新计算。
4. 一个 Partitioner，即 RDD 的分片函数。当前 Spark 中实现了两种类型的分片函数，一个是基于哈希的 HashPartitioner，另外一个是基于范围的 RangePartitioner。只有对于于 key- value 的 RDD，才会有 Partitioner，非 key-value 的 RDD 的 Parititioner 的值是 None。Partitioner 函数不但决定了 RDD 本身的分片数量，也决定了 parent RDD Shuffle 输出时的分片数量。
5. 一个列表（FIFO），存储存取每个 Partition 的优先位置（preferred location）。对于一个HDFS 文件来说，这个列表保存的就是每个 Partition 所在的块的位置。按照“移动数据不如移动计算”的理念，Spark 在进行任务调度的时候，会尽可能地将计算任务分配到其所要处理数据块的存储位置。

RDD 是一个应用层面的逻辑概念。一个 RDD 多个分片。RDD 就是一个元数据记录集，记录了 RDD 内存所有的关系数据。

* 1. **RDD 的弹性**

1. 自动进行内存和磁盘数据存储的切换 存储弹性。

Spark 优先把数据放到内存中，如果内存放不下，就会放到磁盘里面，程序进行自动的存储切换

1. 基于血统的高效容错机制 （过程弹性）

在 RDD 进行转换和动作的时候，会形成 RDD 的 Lineage 依赖链，当某一个 RDD 失效的时候，可以通过重新计算上游的 RDD 来重新生成丢失的 RDD 数据。

1. Task 如果失败会自动进行特定次数的重试

RDD 的计算任务如果运行失败，会自动进行任务的重新计算，默认次数是 4 次。

1. Stage 如果失败会自动进行特定次数的重试(是否执行 shuffle )

如果Job 的某个 Stage 阶段计算失败，框架也会自动进行任务的重新计算，默认次数也是 4 次。

1. Checkpoint 和 Persist 可主动或被动触发

RDD 可以通过 Persist 持久化将 RDD 缓存到内存或者磁盘，当再次用到该 RDD 时直接读取就行。也可以将 RDD 进行检查点，检查点会将数据存储在 HDFS 中，该 RDD 的所有父 RDD 依赖都会被移除。

1. 数据调度弹性

Spark 把这个 JOB 执行模型抽象为通用的有向无环图 DAG，可以将多 Stage 的任务串联或并行执行，调度引擎自动处理 Stage 的失败以及Task 的失败。

1. 数据分片的高度弹性

可以根据业务的特征，动态调整数据分片的个数，提升整体的应用执行效率。

RDD 全称叫做弹性分布式数据集(Resilient Distributed Datasets)，它是一种分布式的内存抽象，表示一个只读的记录分区的集合，它只能通过其他 RDD 转换而创建，为此， RDD 支持丰富的转换操作(如 map, join, filter, groupBy 等)，通过这种转换操作，新的 RDD 则包含了如何从其他 RDDs 衍生所必需的信息，所以说 RDDs 之间是有依赖关系的。基于RDDs 之间的依赖，RDDs 会形成一个有向无环图 DAG，该 DAG 描述了整个流式计算的流程，实际执行的时候，RDD 是通过血缘关系(Lineage)一气呵成的，即使出现数据分区丢

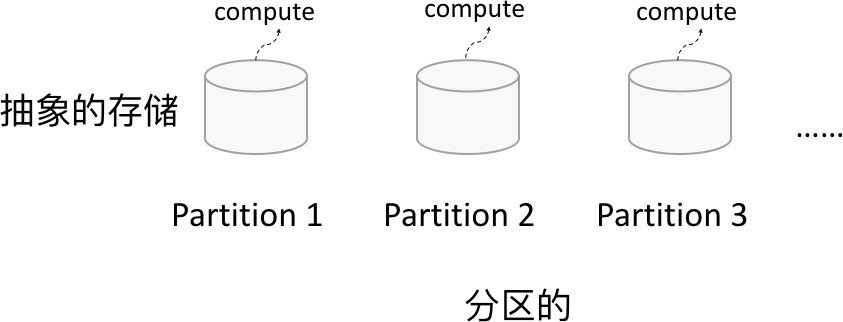
失，也可以通过血缘关系重建分区，总结起来，基于RDD 的流式计算任务可描述为：从稳定的物理存储(如分布式文件系统)中加载记录，记录被传入由一组确定性操作构成的DAG，然后写回稳定存储。另外 RDD 还可以将数据集缓存到内存中，使得在多个操作之间可以重用数据集，基于这个特点可以很方便地构建迭代型应用(图计算、机器学习等)或者交互式数据分析应用。可以说 Spark 最初也就是实现 RDD 的一个分布式系统，后面通过不断发展壮大成为现在较为完善的大数据生态系统，简单来讲，Spark-RDD 的关系类似于Hadoop-MapReduce 关系。

* 1. **RDD 特点**

RDD 表示只读的分区的数据集，对 RDD 进行改动，只能通过 RDD 的转换操作，由一个 RDD 得到一个新的 RDD，新的 RDD 包含了从其他 RDD 衍生所必需的信息。RDDs 之间存在依赖，RDD 的执行是按照血缘关系延时计算的。如果血缘关系较长，可以通过持久化RDD 来切断血缘关系。

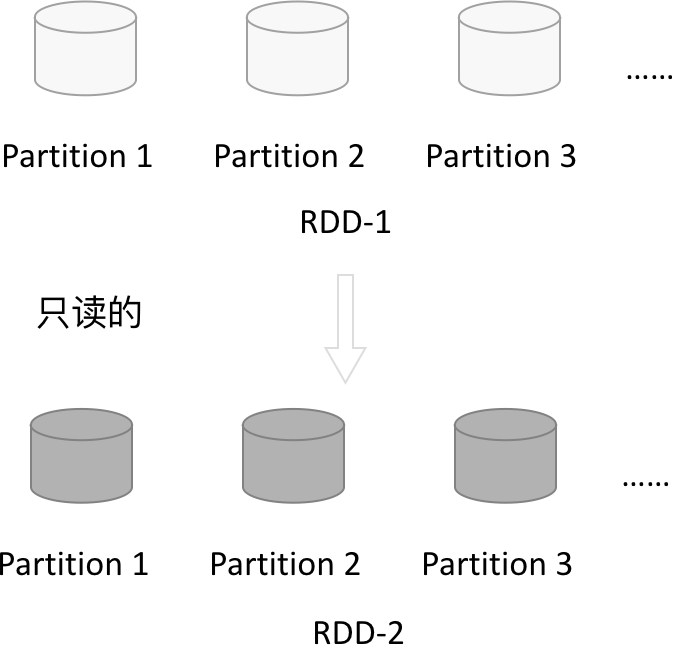
#### 分区

RDD 逻辑上是分区的，每个分区的数据是抽象存在的，计算的时候会通过一个 compute 函数得到每个分区的数据。如果 RDD 是通过已有的文件系统构建，则 compute 函数是读取指定文件系统中的数据，如果 RDD 是通过其他 RDD 转换而来，则 compute 函数是执行转换逻辑将其他 RDD 的数据进行转换。



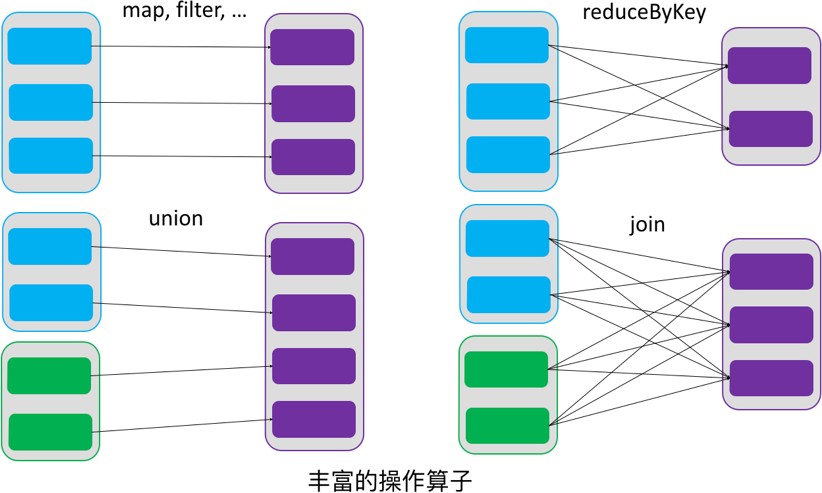
#### 只读

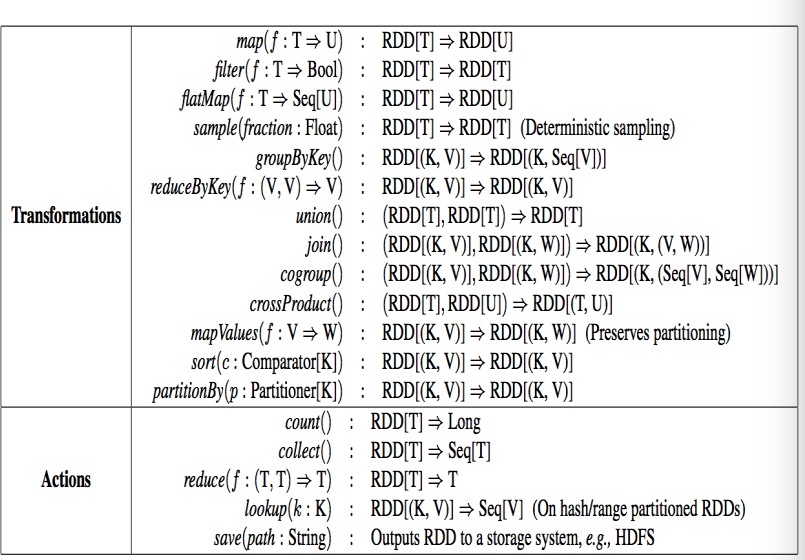
如下图所示，RDD 是只读的，要想改变 RDD 中的数据，只能在现有的 RDD 基础上创建新的 RDD。



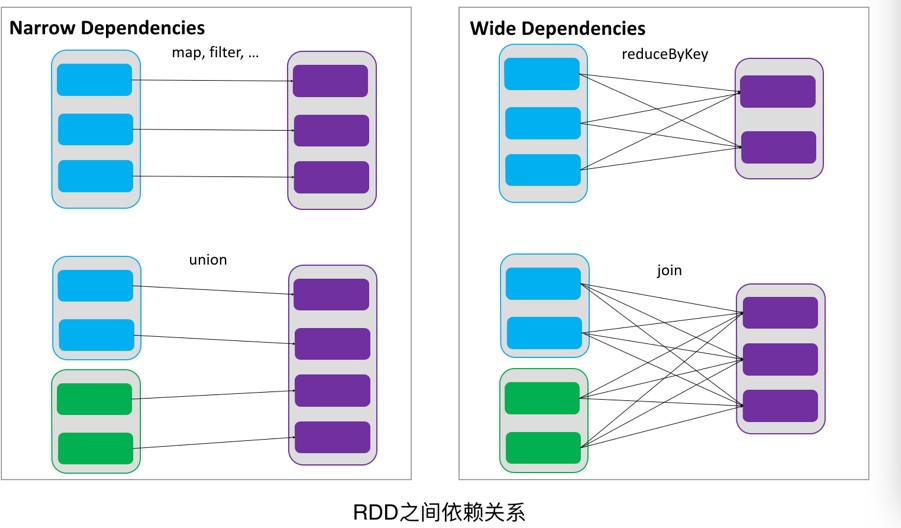
由一个 RDD 转换到另一个 RDD，可以通过丰富的操作算子实现，不再像 MapReduce

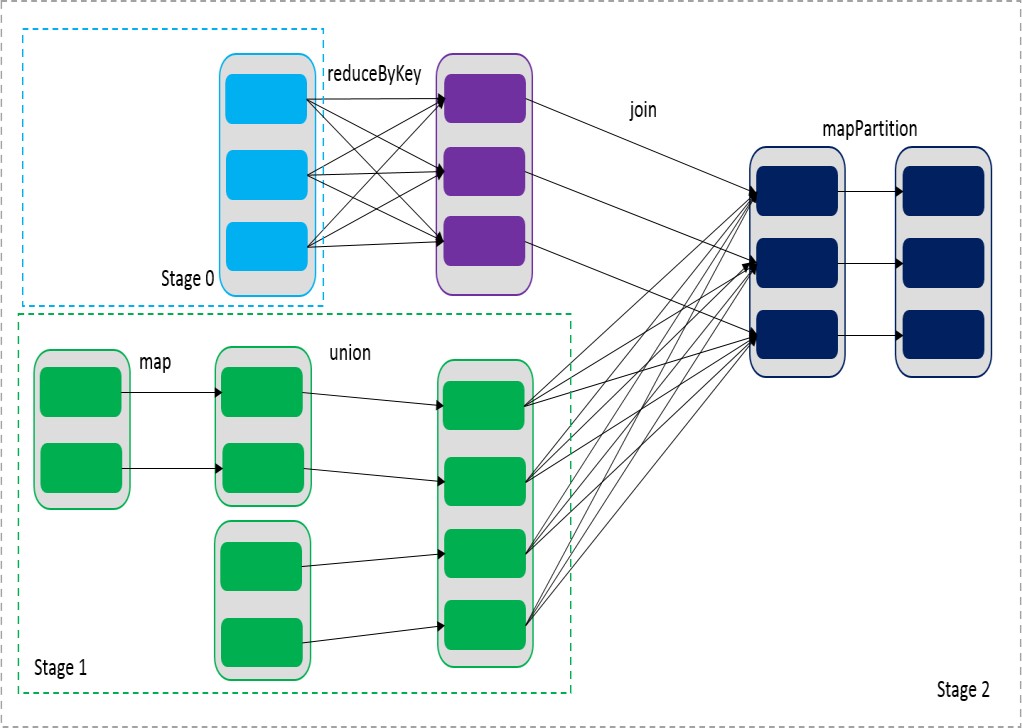
那样只能写 map 和 reduce 了，如下图所示。



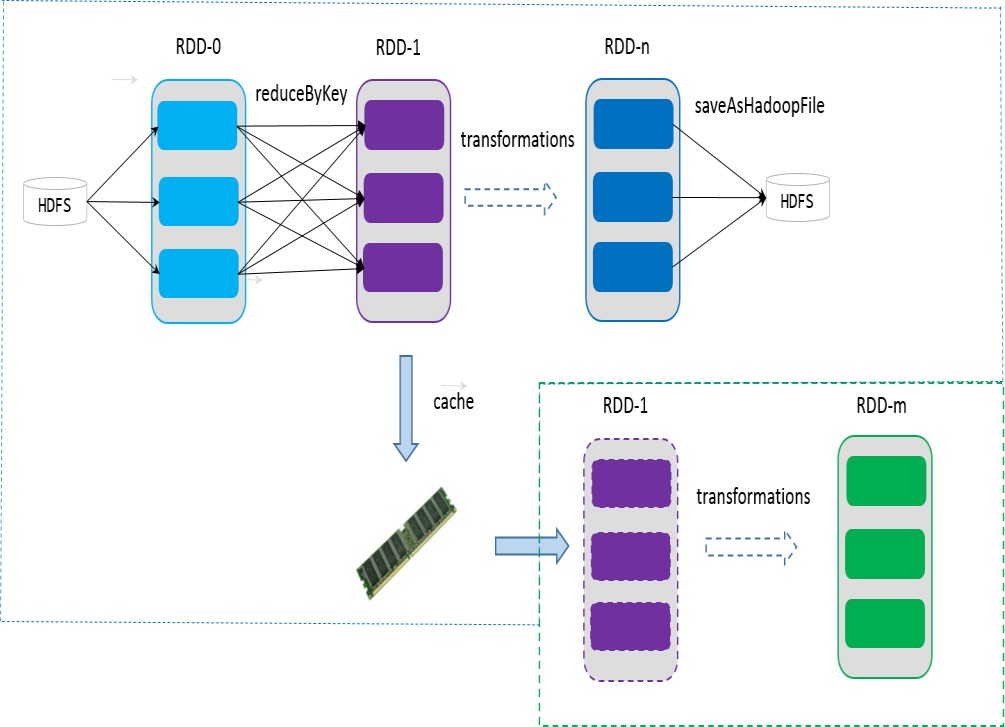
RDD 的操作算子包括两类，一类叫做 transformations，它是用来将 RDD 进行转化，构建 RDD 的血缘关系；另一类叫做 actions，它是用来触发 RDD 的计算，得到 RDD 的相关计算结果或者将 RDD 保存的文件系统中。下图是 RDD 所支持的操作算子列表。

#### 依赖

RDDs 通过操作算子进行转换，转换得到的新 RDD 包含了从其他 RDDs 衍生所必需的信息，RDDs 之间维护着这种血缘关系，也称之为依赖。如下图所示，依赖包括两种，一种是窄依赖，RDDs 之间分区是一一对应的，另一种是宽依赖，下游 RDD 的每个分区与上游RDD(也称之为父 RDD)的每个分区都有关，是多对多的关系。

通过 RDDs 之间的这种依赖关系，一个任务流可以描述为 DAG(有向无环图)，如下图所示，在实际执行过程中宽依赖对应于 Shuffle(图中的 reduceByKey 和 join)，窄依赖中的所有转换操作可以通过类似于管道的方式一气呵成执行(图中 map 和 union 可以一起执行)。

* + 1. 缓存（**cache**）

如果在应用程序中多次使用同一个 RDD，可以将该 RDD 缓存起来，该 RDD 只有在第一次计算的时候会根据血缘关系得到分区的数据，在后续其他地方用到该 RDD 的时候，会直接从缓存处取而不用再根据血缘关系计算，这样就加速后期的重用。如下图所示，RDD- 1 经过一系列的转换后得到 RDD-n 并保存到 hdfs，RDD-1 在这一过程中会有个中间结果， 如果将其缓存到内存，那么在随后的 RDD-1 转换到 RDD-m 这一过程中，就不会计算其之前的 RDD-0 了。

* + 1. **CheckPoint(**检测点**)**

虽然 RDD 的血缘关系天然地可以实现容错，当 RDD 的某个分区数据失败或丢失，可以通过血缘关系重建。但是对于长时间迭代型应用来说，随着迭代的进行，RDDs 之间的血缘关系会越来越长，一旦在后续迭代过程中出错，则需要通过非常长的血缘关系去重建，势必影响性能。为此，RDD 支持 checkpoint 将数据保存到持久化的存储中，这样就可以切断之前的血缘关系，因为checkpoint 后的RDD 不需要知道它的父RDDs 了，它可以从checkpoint 处拿到数据。

给定一个 RDD 我们至少可以知道如下几点信息：1、分区数以及分区方式；2、由父 RDDs衍生而来的相关依赖信息；3、计算每个分区的数据，计算步骤为：1）如果被缓存，则从缓存中取的分区的数据；2）如果被 checkpoint，则从 checkpoint 处恢复数据；3）根据血缘关系计算分区的数据。

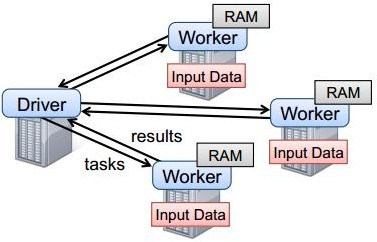
第 **2** 章 **RDD** 编程

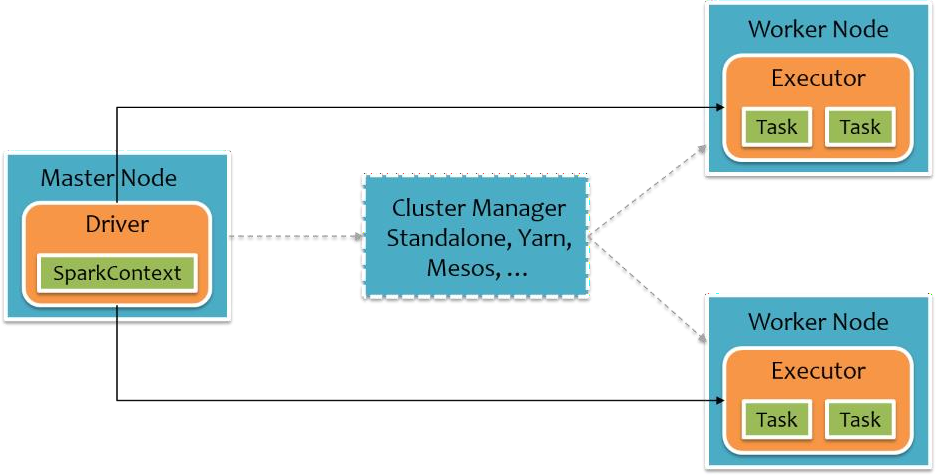
* 1. **编程模型**

在 Spark 中，RDD 被表示为对象，通过对象上的方法调用来对 RDD 进行转换。经过一系列的 transformations 定义 RDD 之后，就可以调用 actions 触发 RDD 的计算，action 可以是向应用程序返回结果(count, collect 等)，或者是向存储系统保存数据(saveAsTextFile

等)。在 Spark 中，只有遇到 action，才会执行 RDD 的计算(即延迟计算)，这样在运行时可以通过管道的方式传输多个转换。

要使用 Spark，开发者需要编写一个 Driver 程序，它被提交到集群以调度运行Worker，如下图所示。Driver 中定义了一个或多个 RDD，并调用 RDD 上的action，Worker 则执行 RDD 分区计算任务。





* 1. **RDD 创建**

在 Spark 中创建 RDD 的创建方式大概可以分为三种：从集合中创建 RDD；从外部存储创建 RDD；从其他 RDD 创建。

由一个已经存在的 Scala 集合创建，集合并行化。

val rdd1 = sc.parallelize(Array(1,2,3,4,5,6,7,8))

而从集合中创建 RDD，[Spark](https://www.iteblog.com/archives/tag/spark/) 主要提供了两种函数：parallelize 和 makeRDD。我们可以先看看这两个函数的声明：

我们可以从上面看出 makeRDD 有两种实现，而且第一个 makeRDD 函数接收的参数和 parallelize 完全一致。其实第一种 makeRDD 函数实现是依赖了 parallelize 函数的实现，来看看 Spark 中是怎么实现这个 makeRDD 函数的：

def makeRDD[T: ClassTag]( seq: Seq[T],

numSlices: Int = defaultParallelism): RDD[T] = withScope { parallelize(seq, numSlices)

}

def parallelize[T: ClassTag]( seq: Seq[T],

numSlices: Int = defaultParallelism): RDD[T]

def makeRDD[T: ClassTag]( seq: Seq[T],

numSlices: Int = defaultParallelism): RDD[T]

def makeRDD[T: ClassTag](seq: Seq[(T, Seq[String])]): RDD[T]

我们可以看出，这个 makeRDD 函数完全和 parallelize 函数一致。但是我们得看看第二种 makeRDD 函数函数实现了，它接收的参数类型是 Seq[(T, Seq[String])]，Spark 文档的说明是：

Distribute a local Scala collection to form an RDD, with one or more location preferences (hostnames of Spark nodes) for each object. Create a new partition for each collection item.

原来，这个函数还为数据提供了位置信息，来看看我们怎么使用：

scala> val guigu1= sc.parallelize(List(1,2,3))

guigu1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[10] at parallelize at <console>:21

scala> val guigu2 = sc.makeRDD(List(1,2,3))

guigu2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[11] at makeRDD at <console>:21

scala> val seq = List((1, List("slave01")),| (2, List("slave02")))

seq: List[(Int, List[String])] = List((1,List(slave01)), (2,List(slave02)))

scala> val guigu3 = sc.makeRDD(seq)

guigu3: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[12] at makeRDD at <console>:23

scala> guigu3.preferredLocations(guigu3.partitions(1)) res26: Seq[String] = List(slave02)

scala> guigu3.preferredLocations(guigu3.partitions(0)) res27: Seq[String] = List(slave01)

scala> guigu1.preferredLocations(guigu1.partitions(0)) res28: Seq[String] = List()

我们可以看到，makeRDD 函数有两种实现，第一种实现其实完全和 parallelize 一致； 而第二种实现可以为数据提供位置信息，而除此之外的实现和 parallelize 函数也是一致

的，如下：

def parallelize[T: ClassTag]( seq: Seq[T],

numSlices: Int = defaultParallelism): RDD[T] = withScope { assertNotStopped()

new ParallelCollectionRDD[T](this, Seq[String]]())

}

seq,

numSlices, Map[Int,

def makeRDD[T: ClassTag](seq: Seq[(T, Seq[String])]): RDD[T] = withScope {

assertNotStopped()

val indexToPrefs = seq.zipWithIndex.map(t => (t.\_2,

t.\_1.\_2)).toMap

new ParallelCollectionRDD[T](this, indexToPrefs)

}

seq.map(\_.\_1),

seq.size,

都是返回 ParallelCollectionRDD，而且这个 makeRDD 的实现不可以自己指定分区的数量，而是固定为 seq 参数的 size 大小。

由外部存储系统的数据集创建，包括本地的文件系统，还有所有Hadoop 支持的数据集，比如HDFS、Cassandra、HBase 等

scala> val atguigu = sc.textFile("hdfs://hadoop102:9000/RELEASE") atguigu: org.apache.spark.rdd.RDD[String] = hdfs://

hadoop102:9000/RELEASE

<console>:24

MapPartitionsRDD[4]

at textFile

at

* 1. **TransFormation**
     1. **map(func)**

返回一个新的 RDD，该 RDD 由每一个输入元素经过func 函数转换后组成

scala> var source = sc.parallelize(1 to 10)

source: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[8] at parallelize at <console>:24

scala> source.collect()

res7: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> val mapadd = source.map(\_ \* 2)

mapadd: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[9] at map at <console>:26

scala> mapadd.collect()

res8: Array[Int] = Array(2, 4, 6, 8, 10, 12, 14, 16, 18, 20)

* + 1. **mapPartitions(func)**

类似于 map，但独立地在 RDD 的每一个分片上运行，因此在类型为 T 的 RDD 上运行时， func 的函数类型必

须是 Iterator[T] => Iterator[U]。假设有N 个元素，有M 个分区，那么 map 的函数的将被调用 N 次,而 mapPartitions 被调用 M 次,一个函数一次处理所有分区

def mappatitions(): Unit = {

val conf = new SparkConf().setAppName("cogroup").setMaster("local[\*]")

val sc = new SparkContext(conf)

val a = sc.parallelize(1 to 9, 3)

def doubleFunc(iter: Iterator[Int]): Iterator[Int] = {

var res = List[Int]()

while (iter.hasNext) {

val cur = iter.next;

res.::=((cur \* 2))

}

res.iterator

}

val result = a.mapPartitions(doubleFunc)

result.collect().foreach(println)

}

* + 1. **glom**

将每一个分区形成一个数组，形成新的 RDD 类型时RDD[Array[T]]

scala> val rdd = sc.parallelize(1 to 16,4)

rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[65] at parallelize at <console>:24

scala> rdd.glom().collect()

res25: Array[Array[Int]] = Array(Array(1, 2, 3, 4), Array(5, 6, 7,

8), Array(9, 10, 11, 12), Array(13, 14, 15, 16))

* + 1. **flatMap(func)**

类似于 map，但是每一个输入元素可以被映射为 0 或多个输出元素（所以 func 应该返回一个序列，而不是单一元素）

scala> val sourceFlat = sc.parallelize(1 to 5) sourceFlat: org.apache.spark.rdd.RDD[Int]

=

ParallelCollectionRDD[12] at parallelize at <console>:24

scala> sourceFlat.collect()

res11: Array[Int] = Array(1, 2, 3, 4, 5)

scala> val flatMap = sourceFlat.flatMap(1 to \_)

flatMap: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[13] at flatMap at <console>:26

scala> flatMap.collect()

res12: Array[Int] = Array(1, 1, 2, 1, 2, 3, 1, 2, 3, 4, 1, 2, 3, 4,

5)

* + 1. **filter(func)**

返回一个新的 RDD，该 RDD 由经过 func 函数计算后返回值为 true 的输入元素组成

// 创建 RDD

val rdd: RDD[Int] = sc.makeRDD(Array(1, 2, 3, 4, 5))

// 获得所有的偶数。

val num: RDD[Int] = rdd.filter(aaa => aaa % 2==0 )

num.foreach(line => println(line))

* + 1. **mapPartitionsWithIndex(func)**

类似于 mapPartitions，但 func 带有一个整数参数表示分片的索引值，因此在类型为 T 的

RDD 上运行时，func 的函数类型必须是(Int, Interator[T]) => Iterator[U]

def mappatitionswithindex() {

val conf = new SparkConf().setAppName("mappatitionswithindex").setMaster("local[\*]"

)

val sc = new SparkContext(conf)

val rdd1: RDD[Int] = sc.makeRDD(1 to 10, 2)

val rdd2: RDD[String] = rdd1.mapPartitionsWithIndex((x, iter) => {

var result = List[String]()

result.::=(x + "|" + iter.toList)

result.iterator

})

val res: Array[String] = rdd2.collect()

for (x <- res) {

println(x)

}

}

* + 1. **sample(withReplacement, fraction, seed)**

以指定的随机种子随机抽样出数量为 fraction 的数据，withReplacement 表示是抽出的数据是否放回，true 为有放回的抽样，false 为无放回的抽样，seed 用于指定随机数生成器种子。例子从 RDD 中随机且有放回的抽出 50%的数据，随机种子值为 3（即可能以 1 2 3 的其中一个起始值）

scala> val rdd = sc.parallelize(1 to 10)

rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[20] at parallelize at <console>:24

scala> rdd.collect()

res15: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> var sample1 = rdd.sample(true,0.4,2)

sample1: org.apache.spark.rdd.RDD[Int] =

PartitionwiseSampledRDD[21] at sample at <console>:26

scala> sample1.collect()

res16: Array[Int] = Array(1, 2, 2, 7, 7, 8, 9)

scala> var sample2 = rdd.sample(false,0.2,3)

sample2: org.apache.spark.rdd.RDD[Int] =

PartitionwiseSampledRDD[22] at sample at <console>:26

scala> sample2.collect()

res17: Array[Int] = Array(1, 9)

* + 1. **distinct([numTasks]))**

对源 RDD 进行去重后返回一个新的 RDD. 默认情况下，只有 8 个并行任务来操作，但是可以传入一个可选的 numTasks 参数改变它。

def distinct(): Unit = {

val conf = new SparkConf().setAppName("distinct").setMaster("local[\*]")

val sc = new SparkContext(conf)

val distinctRdd = sc.parallelize(List(1, 2, 1, 5, 2, 9, 6, 1))

val res: RDD[Int] = distinctRdd.distinct(2)

// distinct 中的参数 numTask 表示 数据先对 2 整除，其他的依次余 1，余2，余 3 。局部无序，整体有序。

res.foreach(println)

}

* + 1. **partitionBy**

对 RDD 进行分区操作，如果原有的 partionRDD 和现有的 partionRDD 是一致的话就不进行分区， 否则会生成 ShuffleRDD。

def partitonBy2repatition(): Unit = {

val conf = new SparkConf().setAppName("localTest").setMaster("local[4]")

val sc = new SparkContext(conf)

###### //设置 4 个分区;

val rdd = sc.parallelize(List("hello", "jason", "what", "are", "you", "doing", "hi", "jason", "do", "you", "eat", "dinner",

"hello", "jason", "do", "you", "have", "some", "time", "hello",

"jason", "time", "do", "you", "jason", "jason"), 4)

val word\_count = rdd.flatMap(\_.split(",")).map((\_, 1))

###### //重分区为 10 个;

val rep = word\_count.repartition(10)

rep.foreachPartition(pair => {

println("第几个分区 " + TaskContext.get.partitionId)

pair.foreach(p => {

println(p)

})

})

println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

###### //重分区为 10;

val parby = word\_count.partitionBy(new HashPartitioner(10))

parby.foreachPartition(pair => {

println("第几个分区 " + TaskContext.get.partitionId)

pair.foreach(p => {

println(p)

})

})

}

自定义分区:

class MyPartition(var num:Int) extends Partitioner{

def numPartitions: Int= num

def getPartition(key: Any): Int={

if(key.toString().length()<=2)

0

else

1

}

}

* + 1. **coalesce(numPartitions)**

缩减分区数，用于大数据集过滤后，提高小数据集的执行效率。

scala> val rdd = sc.parallelize(1 to 16,4)

rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[54] at parallelize at <console>:24

scala> rdd.partitions.size res20: Int = 4

scala> val coalesceRDD = rdd.coalesce(3)

coalesceRDD: org.apache.spark.rdd.RDD[Int] = CoalescedRDD[55] at coalesce at <console>:26

scala> coalesceRDD.partitions.size res21: Int = 3

* + 1. **repartition(numPartitions)**

根据分区数，从新通过网络随机洗牌所有数据。

scala> val rdd = sc.parallelize(1 to 16,4)

rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[56] at parallelize at <console>:24

scala> rdd.partitions.size res22: Int = 4

scala> val rerdd = rdd.repartition(2)

rerdd: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[60] at repartition at <console>:26

scala> rerdd.partitions.size res23: Int = 2

scala> val rerdd = rdd.repartition(4)

rerdd: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[64] at repartition at <console>:26

scala> rerdd.partitions.size res24: Int = 4

* + 1. **repartitionAndSortWithinPartitions(partitioner)**

repartitionAndSortWithinPartitions 函数是 repartition 函数的变种，与 repartition 函数不同的是，repartitionAndSortWithinPartitions 在给定的 partitioner 内部进行排序，性能比 repartition 要高。

* + 1. **sortBy(func,[ascending], [numTasks])**

用 func 先对数据进行处理，按照处理后的数据比较结果排序。

|  |
| --- |
| scala> val rdd = sc.parallelize(List(1,2,3,4))  rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[21] at parallelize at <console>:24  scala> rdd.sortBy(x => x).collect() res11: Array[Int] = Array(1, 2, 3, 4)  scala> rdd.sortBy(x => x%3).collect() res12: Array[Int] = Array(3, 4, 1, 2) |
| 自定义排序:  val rdd1: RDD[(String,Int)] = sc.parallelize(List(("baihuaqiang",6),("hello",12),("tom",2),("j ack",5)))  val rdd2: RDD[(String,Int)] = rdd1.sortBy(x =>new Student(x.\_1,x.\_2),false,2)  println(rdd2.collect().toBuffer)  }  }  class Student(val name:String,var age:Int) extends Ordered[Student] with Serializable{  override def compare(that: Student): Int = { this.age - that.age  }  }  采用隐士转换模式:  implicit object MyPredef extends Ordering[Boy]{ override def compare(x: Boy, y: Boy): Int = {  x.age - y.age  }  }  //第二种方式通过实现隐式转换实现 |

|  |
| --- |
| class Boy(val name:String,var age:Int) extends Serializable {  } |
|  |

* + 1. **union(otherDataset)**

对源 RDD 和参数 RDD 求并集后返回一个新的 RDD 不去重

scala> val rdd1 = sc.parallelize(1 to 5)

rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[23] at parallelize at <console>:24

scala> val rdd2 = sc.parallelize(5 to 10)

rdd2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[24] at parallelize at <console>:24

scala> val rdd3 = rdd1.union(rdd2)

rdd3: org.apache.spark.rdd.RDD[Int] = UnionRDD[25] at union at

<console>:28

scala> rdd3.collect()

res18: Array[Int] = Array(1, 2, 3, 4, 5, 5, 6, 7, 8, 9, 10)

* + 1. **subtract (otherDataset)**

计算差的一种函数，去除两个 RDD 中相同的元素，不同的 RDD 将保留下来

scala> val rdd = sc.parallelize(3 to 8)

rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[70] at parallelize at <console>:24 3 4 5 6 7 8

1 2 3 4 5

scala> val rdd1 = sc.parallelize(1 to 5)

rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[71] at parallelize at <console>:24

scala> rdd.subtract(rdd1).collect() res27: Array[Int] = Array(8, 6, 7)

* + 1. **intersection(otherDataset)**

对源 RDD 和参数 RDD 求交集后返回一个新的 RDD

scala> val rdd1 = sc.parallelize(1 to 7)

rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[26] at parallelize at <console>:24

scala> val rdd2 = sc.parallelize(5 to 10)

rdd2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[27] at parallelize at <console>:24

scala> val rdd3 = rdd1.intersection(rdd2)

rdd3: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[33] at intersection at <console>:28

scala> rdd3.collect()

res19: Array[Int] = Array(5, 6, 7)

* + 1. **cartesian(otherDataset)**

笛卡尔积

scala> val rdd1 = sc.parallelize(1 to 3)

rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[47] at parallelize at <console>:24

scala> val rdd2 = sc.parallelize(2 to 5)

rdd2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[48] at parallelize at <console>:24

scala> rdd1.cartesian(rdd2).collect()

res17: Array[(Int, Int)] = Array((1,2), (1,3), (1,4), (1,5), (2,2),

(2,3), (2,4), (2,5), (3,2), (3,3), (3,4), (3,5))

* + 1. **pipe(command, [envVars])**

管道，对于每个分区，都执行一个 perl 或者 shell 脚本，返回输出的 RDD

Shell 脚本

#!/bin/sh echo "AA"

while read LINE; do echo ">>>"${LINE}

done scala>

val

rdd

=

sc.parallelize(List("hi","Hello","how","are","you"),1)

rdd: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[50] at parallelize at <console>:24

scala> rdd.pipe("/home/bigdata/pipe.sh").collect()

res18: Array[String] = Array(AA, >>>hi, >>>Hello, >>>how, >>>are, >>>you)

scala>

val

rdd

=

sc.parallelize(List("hi","Hello","how","are","you"),2)

rdd: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[52] at parallelize at <console>:24

scala> rdd.pipe("/home/bigdata/pipe.sh").collect()

res19: Array[String] = Array(AA, >>>hi, >>>Hello, AA, >>>how, >>>are, >>>you)

pipe.sh: #!/bin/sh echo "AA"

while read LINE; do echo ">>>"${LINE}

done

* + 1. **join(otherDataset, [numTasks])**

在类型为(K,V)和(K,W)的 RDD 上调用，返回一个相同 key 对应的所有元素对在一起的

def join() {

val conf = new SparkConf()

.setAppName("join")

.setMaster("local")

val sc = new SparkContext(conf)

val studentList = Array(

Tuple2(1, "leo"),

Tuple2(2, "jack"),

Tuple2(3, "tom"));

val scoreList = Array(

Tuple2(1, 100),

Tuple2(2, 90),

Tuple2(3, 60));

val students = sc.parallelize(studentList);

val scores = sc.parallelize(scoreList);

val studentScores = students.join(scores)

studentScores.foreach(studentScore => {

println("student id: " + studentScore.\_1);

println("student name: " + studentScore.\_2.\_1)

println("student socre: " + studentScore.\_2.\_2)

println("=======================================")

})

}

def join2(): Unit = {

val conf = new SparkConf()

.setAppName("join2")

.setMaster("local[\*]")

val sc = new SparkContext(conf)

val idName = sc.parallelize(Array((1, "zhangsan"), (2, "lisi"), (3, "wangwu")))

val idAge = sc.parallelize(Array((1, 30), (2, 29), (4, 21)))

//idName.join(idAge).collect().foreach(println)

//val result: RDD[(Int, (String, Int))] = idName.join(idAge)

//左外连接 idName.leftOuterJoin(idAge).collect().foreach(println)

//右外连接 idName.rightOuterJoin(idAge).collect().foreach(println)

//全连接

idName.fullOuterJoin(idAge).collect().foreach(println)

// result.collect().foreach(res => {

// println("学生 ID"+res.\_1)

// println("学生姓名："+res.\_2.\_1)

// println("学生成绩："+res.\_2.\_2)

// println("=============================================")

// })

}

* + 1. **cogroup(otherDataset, [numTasks])**

在类型为(K,V)和(K,W)的 RDD 上调用，返回一个(K,(Iterable<V>,Iterable<W>))类型的 RDD

def cogroup(): Unit = {

val conf = new SparkConf()

.setAppName("cogroup")

.setMaster("local[\*]")

val sc = new SparkContext(conf)

val idName = sc.parallelize(Array((1, "zhangsan"), (2, "lisi")))

val idScore = sc.parallelize(Array((1, 100), (2, 90), (2, 95)))

idName.cogroup(idScore).foreach(

res => {

println("id\t" + res.\_1)

println("name\t" + res.\_2.\_1)

println("score\t" + res.\_2.\_2)

println("=============================")

}

)

}

* + 1. **reduceByKey(func, [numTasks])**

在一个(K,V)的 RDD 上调用，返回一个(K,V)的 RDD，使用指定的 reduce 函数，将相同 key

的值聚合到一起，reduce 任务的个数可以通过第二个可选的参数来设置。

def reduceByKey(): Unit ={

val conf = new SparkConf()

.setAppName("cogroup")

.setMaster("local[\*]")

val sc = new SparkContext(conf)

val rdd = sc.parallelize(List(("jack",1),("tom",5),("jack",5),("tom",6),("jack ",7)))

val result: Array[(String, Int)] = rdd.reduceByKey((x,y)=>(x+y)).collect()

for((x:String,y:Int) <- result){

println(x+" "+y)

}

}

* + 1. **groupByKey**

groupByKey 也是对每个 key 进行操作，但只生成一个 sequence。

def groupByKey(): Unit ={

val conf = new SparkConf()

.setAppName("groupByKey")

.setMaster("local[\*]")

val sc = new SparkContext(conf)

val rdd = sc.parallelize(List(("jack",1),("tom",5),("jack",5),("tom",6),("jack",7)))

rdd.groupByKey().collect().foreach(x=>{

println("姓名："+x.\_1+" 分数"+x.\_2)

})

rdd.groupByKey().map(x=>{

val name = x.\_1

var sum = 0

for(y<- x.\_2 ){

sum += y

}

(name,sum)

}).collect().foreach(println)

}

* + 1. **combineByKey[C]**

( createCombiner: V => C, mergeValue: (C, V) => C, mergeCombiners: (C, C) => C)

对相同 K，把 V 合并成一个集合。

createCombiner: combineByKey() 会遍历分区中的所有元素，因此每个元素的键要么还没有遇到过，要么就 和之前的某个元素的键相同。如果这是一个新的元素,combineByKey() 会使用一个叫作 createCombiner() 的函数来创建

那个键对应的累加器的初始值

mergeValue: 如果这是一个在处理当前分区之前已经遇到的键， 它会使用 mergeValue() 方法将该键的累加器对应的当前值与这个新的值进行合并

mergeCombiners: 由于每个分区都是独立处理的， 因此对于同一个键可以有多个累加器。如果有两个或者更多的分区都有对应同一个键的累加器， 就需要使用用户提供的mergeCombiners() 方法将各个分区的结果进行合并。

def combineByKey(): Unit ={

val conf = new SparkConf().setAppName("combinerByKey").setMaster("local[\*]")

val sc = new SparkContext(conf)

// 求出每个字母的平均值。

/\*

createCombiner: V => C, 创建一个合并对象。a:（90,1）

mergeValue: (C, V) => C, a（170,2） 区内合并

mergeCombiners: (C, C) => C, a(76 1) a(246,3) 全区合并。

\*/

val rdd1: RDD[(String, Int)] = sc.makeRDD(Array(("a",90),("a",80),("b",46),("b",58),("b",29),("c",58),("c"

,90),("d",91),("a",76)))

// key value 和 累加器和

val rdd3: RDD[(String, (Int, Int))] = rdd1.combineByKey(

v => (v, 1), // （90,1）

(c: (Int, Int), v) => (c.\_1 + v, c.\_2 + 1),

(c1: (Int, Int), c2: (Int, Int)) => (c1.\_1 + c2.\_1, c1.\_2 + c2.\_2)

)

//rdd3.map(x=>(x.\_1+"平均值"+(x.\_2.\_1/x.\_2.\_2))).collect().foreach(println)

rdd3.map{case(x:String,(y:Int,z:Int))=>(x,(y/z))}.collect().foreach(println

)

* + 1. **aggregateByKey**

(zeroValue:U,[partitioner: Partitioner]) (seqOp: (U, V) => U,combOp: (U, U) => U)

在 kv 对的 RDD 中，，按 key 将 value 进行分组合并，合并时，将每个 value 和初始值作为seq 函数的参数，进行计算，返回的结果作为一个新的 kv 对，然后再将结果按照 key 进行合并，最后将每个分组的 value 传递给 combine 函数进行计算（先将前两个 value 进行计 算，将返回结果和下一个 value 传给 combine 函数，以此类推），将 key 与计算结果作为一个新的 kv 对输出。

seqOp 函数用于在每一个分区中用初始值逐步迭代 value，combOp 函数用于合并每个分区中的结果。

scala>

val

rdd

=

sc.parallelize(List((1,3),(1,2),(1,4),(2,3),(3,6),(3,8)),3)

rdd: org.apache.spark.rdd.RDD[(Int, Int)] =

ParallelCollectionRDD[12] at parallelize at <console>:24

scala> val agg = rdd.aggregateByKey(0)(math.max(\_,\_),\_+\_)

agg: org.apache.spark.rdd.RDD[(Int, Int)] = ShuffledRDD[13] at aggregateByKey at <console>:26

scala> agg.collect()

res7: Array[(Int, Int)] = Array((3,8), (1,7), (2,3))

scala> agg.partitions.size res8: Int = 3

scala>

val

rdd

=

sc.parallelize(List((1,3),(1,2),(1,4),(2,3),(3,6),(3,8)),1)

rdd: org.apache.spark.rdd.RDD[(Int, Int)] =

ParallelCollectionRDD[10] at parallelize at <console>:24

scala> val agg = rdd.aggregateByKey(0)(math.max(\_,\_),\_+\_).collect() agg: Array[(Int, Int)] = Array((1,4), (3,8), (2,3))

* + 1. **foldByKey**

(zeroValue: V)(func: (V, V) => V): RDD[(K, V)] aggregateByKey 的简化操作，seqop 和 combop 相同

scala>

val

rdd

=

sc.parallelize(List((1,3),(1,2),(1,4),(2,3),(3,6),(3,8)),3)

rdd: org.apache.spark.rdd.RDD[(Int, Int)] =

ParallelCollectionRDD[91] at parallelize at <console>:24

scala> val agg = rdd.foldByKey(0)(\_+\_)

agg: org.apache.spark.rdd.RDD[(Int, Int)] = ShuffledRDD[92] at foldByKey at <console>:26

scala> agg.collect()

res61: Array[(Int, Int)] = Array((3,14), (1,9), (2,3))

* + 1. **sortByKey([ascending], [numTasks])**

在一个(K,V)的 RDD 上调用，K 必须实现Ordered 接口，返回一个按照 key 进行排序的

(K,V)的 RDD 对 key 进行排序. (Student,23)(Student,14) map(k1 <- val) map(k1 <- val)

map(k4 <- val) map(k3 <- val) map(k3 <- val) map(k4 <- val) map(k5 <- val) map(k5 <- val)

scala>

val

rdd

=

sc.parallelize(Array((3,"aa"),(6,"cc"),(2,"bb"),(1,"dd")))

rdd: org.apache.spark.rdd.RDD[(Int, String)] =

ParallelCollectionRDD[14] at parallelize at <console>:24

scala> rdd.sortByKey(true).collect()

res9: Array[(Int, String)] = Array((1,dd), (2,bb), (3,aa), (6,cc))

scala> rdd.sortByKey(false).collect()

res10: Array[(Int, String)] = Array((6,cc), (3,aa), (2,bb), (1,dd))

* + 1. **mapValues**

针对于(K,V)形式的类型只对 V 进行操作

scala>

val

rdd3

=

sc.parallelize(Array((1,"a"),(1,"d"),(2,"b"),(3,"c")))

rdd3: org.apache.spark.rdd.RDD[(Int, String)] =

ParallelCollectionRDD[67] at parallelize at <console>:24

scala> rdd3.mapValues(\_+"|||").collect()

res26: Array[(Int, String)] = Array((1,a|||), (1,d|||), (2,b|||), (3,c|||))

* 1. **Action**
     1. **reduce(func)**

通过 func 函数聚集 RDD 中的所有元素，这个功能必须是可交换且可并联的

scala> val rdd1 = sc.makeRDD(1 to 10,2)

rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[85] at makeRDD at <console>:24

scala> rdd1.reduce(\_+\_) res50: Int = 55

scala> val

rdd2

=

sc.makeRDD(Array(("a",1),("a",3),("c",3),("d",5)))

rdd2: org.apache.spark.rdd.RDD[(String, Int)] =

ParallelCollectionRDD[86] at makeRDD at <console>:24

scala> rdd2.reduce((x,y)=>(x.\_1 + y.\_1,x.\_2 + y.\_2)) res51: (String, Int) = (adca,12)

* + 1. **collect()**

在驱动程序中，以数组的形式返回数据集的所有元素

* + 1. **count()**

返回 RDD 的元素个数

* + 1. **first()**

返回 RDD 的第一个元素（类似于 take(1)）

* + 1. **take(n) top(3)**

返回一个由数据集的前 n 个元素组成的数组

* + 1. **takeSample(withReplacement,num, [seed])**

返回一个数组，该数组由从数据集中随机采样的 num 个元素组成，可以选择是否用随机数替换不足的部分，seed 用于指定随机数生成器种子

* + 1. **takeOrdered(n)**

返回前几个的排序

* + 1. **aggregate**

(zeroValue: U)(seqOp: (U, T) ⇒ U, combOp: (U, U) ⇒ U)

aggregate 函数将每个分区里面的元素通过 seqOp 和初始值进行聚合，然后用 combine 函数将每个分区的结果和初始值(zeroValue)进行 combine 操作。这个函数最终返回的类型不需要和 RDD 中元素类型一致。

scala> var rdd1 = sc.makeRDD(1 to 10,2)

rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[88] at makeRDD at <console>:24

scala> rdd.aggregate(1)(

| {(x : Int,y : Int) => x + y},

| {(a : Int,b : Int) => a + b}

| )

res56: Int = 58

scala> rdd1.aggregate(1)(

| {(x : Int,y : Int) => x \* y},

| {(a : Int,b : Int) => a + b}

| )

res57: Int = 30361

* + 1. **fold(num)(func)**

折叠操作，aggregate 的简化操作，seqop 和 combop 一样。

scala> var rdd1 = sc.makeRDD(1 to 4,2)

rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[90] at makeRDD at <console>:24

scala> rdd1.aggregate(1)(

| {(x : Int,y : Int) => x + y},

| {(a : Int,b : Int) => a + b}

| )

res59: Int = 13

scala> rdd1.fold(1)(\_+\_) res60: Int = 13

(1 2 +1) (3 4 +1) 1

* + 1. **saveAsTextFile(path)**

将数据集的元素以 textfile 的形式保存到 HDFS 文件系统或者其他支持的文件系统，对于每个元素，Spark 将会调用 toString 方法，将它装换为文件中的文本

* + 1. **saveAsSequenceFile(path)**

将数据集中的元素以Hadoop sequencefile 的格式保存到指定的目录下，可以使 HDFS 或者其他 Hadoop 支持的文件系统。

* + 1. **saveAsObjectFile(path)**

用于将 RDD 中的元素序列化成对象，存储到文件中。

* + 1. **countByKey()**

针对(K,V)类型的 RDD，返回一个(K,Int)的 map，表示每一个 key 对应的元素个数。

scala>

val

rdd

=

sc.parallelize(List((1,3),(1,2),(1,4),(2,3),(3,6),(3,8)),3)

rdd: org.apache.spark.rdd.RDD[(Int, Int)] =

ParallelCollectionRDD[95] at parallelize at <console>:24

scala> rdd.countByKey()

res63: scala.collection.Map[Int,Long] = Map(3 -> 2, 1 -> 3, 2 -> 1)

* + 1. **foreach(func)**

在数据集的每一个元素上，运行函数 func 进行更新。

scala> var rdd = sc.makeRDD(1 to 10,2)

rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[107] at makeRDD at <console>:24

scala> var sum = sc.accumulator(0)

warning: there were two deprecation warnings; re-run with - deprecation for details

sum: org.apache.spark.Accumulator[Int] = 0 scala> rdd.foreach(sum+=\_)

scala> sum.value res68: Int = 55

scala> rdd.collect().foreach(println) 1

2

3

4

5

6

7

8

9

10

* 1. **RDD 持久化**
     1. **RDD** 的缓存

Spark 速度非常快的原因之一，就是在不同操作中可以在内存中持久化或缓存个数据 集。当持久化某个 RDD 后，每一个节点都将把计算的分片结果保存在内存中，并在对此RDD 或衍生出的 RDD 进行的其他动作中重用。这使得后续的动作变得更加迅速。RDD 相关的持久化和缓存(cache)，是 Spark 最重要的特征之一。可以说，缓存是 Spark 构建迭代式算法和快速交互式查询的关键。如果一个有持久化数据的节点发生故障，Spark 会在需要用到缓存的数据时重算丢失的数据分区。如果 希望节点故障的情况不会拖累我们的执行速度，也可以把数据备份到多个节点上。（监测点：checkpoint）

* + 1. **RDD** 缓存方式

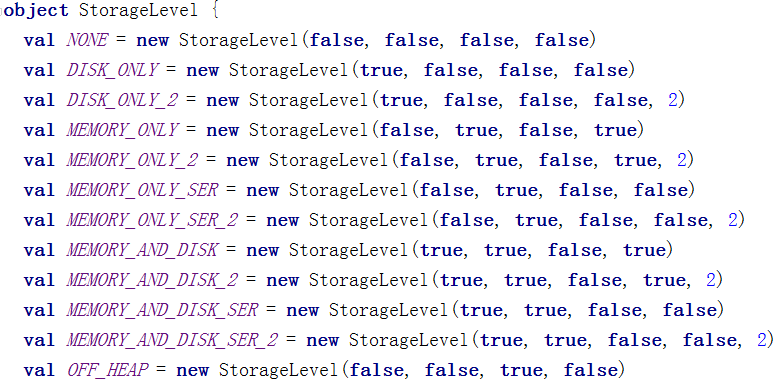
RDD 通过 persist 方法或 cache 方法可以将前面的计算结果缓存，默认情况下 persist()

会把数据以序列化的形式缓存在 JVM 的堆空 间中。

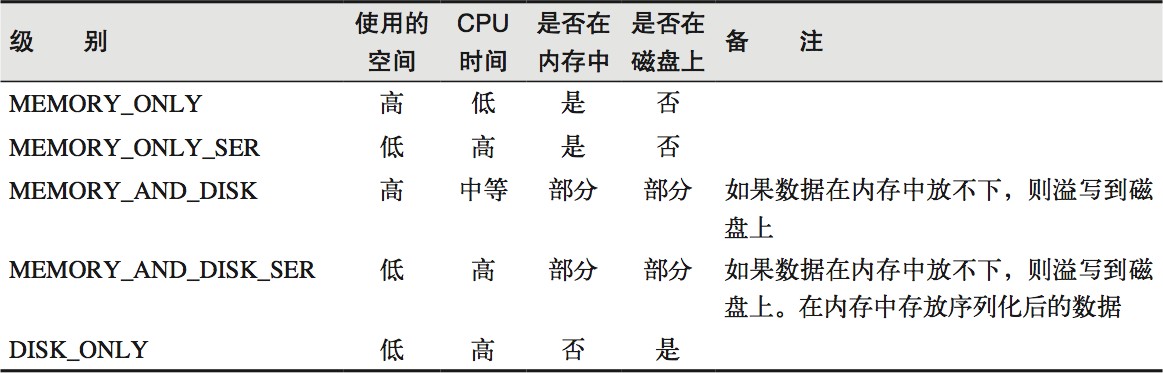
但是并不是这两个方法被调用时立即缓存，而是触发后面的 action 时，该 RDD 将会被缓存在计算节点的内存中，并供后面重用。



通过查看源码发现 cache 最终也是调用了 persist 方法，默认的存储级别都是仅在内存存储一份，Spark 的存储级别还有好多种，存储级别在 object StorageLevel 中定义的。



在存储级别的末尾加上“\_2”来把持久化数据存为两份



缓存有可能丢失，或者存储存储于内存的数据由于内存不足而被删除，RDD 的缓存容错机制保证了即使缓存丢失也能保证计算的正确执行。通过基于 RDD 的一系列转换，丢失的数据会被重算，由于 RDD 的各个 Partition 是相对独立的，因此只需要计算丢失的部分即可，并不需要重算全部 Partition。

注意：使用 Tachyon 可以实现堆外缓存

scala> val rdd = sc.makeRDD(1 to 10)

rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[19] at makeRDD at <console>:25

scala> val nocache = rdd.map(\_.toString+"["+System.currentTimeMillis+"]")

nocache: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[20] at map at <console>:27

scala> val cache = rdd.map(\_.toString+"["+System.currentTimeMillis+"]")

cache: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[21] at map at <console>:27

scala> cache.cache

res24: cache.type = MapPartitionsRDD[21] at map at <console>:27

scala> nocache.collect

res25: Array[String] = Array(1[1505479375155], 2[1505479374674],

|  |  |  |
| --- | --- | --- |
| 3[1505479374674], | 4[1505479375153], | 5[1505479375153], |
| 6[1505479374675], | 7[1505479375154], | 8[1505479375154], |

9[1505479374676], 10[1505479374676])

scala> nocache.collect

res26: Array[String] = Array(1[1505479375679], 2[1505479376157],

|  |  |  |
| --- | --- | --- |
| 3[1505479376157], | 4[1505479375680], | 5[1505479375680], |
| 6[1505479376159], | 7[1505479375680], | 8[1505479375680], |

9[1505479376158], 10[1505479376158])

scala> nocache.collect

res27: Array[String] = Array(1[1505479376743], 2[1505479377218],

|  |  |  |
| --- | --- | --- |
| 3[1505479377218], | 4[1505479376745], | 5[1505479376745], |
| 6[1505479377219], | 7[1505479376747], | 8[1505479376747], |

9[1505479377218], 10[1505479377218])

scala> cache.collect

res28: Array[String] = Array(1[1505479382745], 2[1505479382253],

|  |  |  |
| --- | --- | --- |
| 3[1505479382253], | 4[1505479382748], | 5[1505479382748], |
| 6[1505479382257], | 7[1505479382747], | 8[1505479382747], |
| 9[1505479382253], 10[1505479382253])  scala> cache.collect  res29: Array[String] = Array(1[1505479382745], 2[1505479382253], | | |
| 3[1505479382253], | 4[1505479382748], | 5[1505479382748], |
| 6[1505479382257], | 7[1505479382747], | 8[1505479382747], |
| 9[1505479382253], 10[1505479382253])  scala> cache.collect  res30: Array[String] = Array(1[1505479382745], 2[1505479382253], | | |
| 3[1505479382253], | 4[1505479382748], | 5[1505479382748], |
| 6[1505479382257], | 7[1505479382747], | 8[1505479382747], |
| 9[1505479382253], 10[1505479382253])  cache.persist(org.apache.spark.storage.StorageLevel.MEMORY\_ONLY) | | |

* 1. **RDD 检查点机制**

Spark 中对于数据的保存除了持久化操作之外，还提供了一种检查点的机制，检查点

（本质是通过将 RDD 写入Disk 做检查点）是为了通过 lineage 做容错的辅助，lineage 过长

会造成容错成本过高，这样就不如在中间阶段做检查点容错，如果之后有节点出现问题而丢失分区，从做检查点的 RDD 开始重做 Lineage，就会减少开销。检查点通过将数据写入到 HDFS 文件系统实现了 RDD 的检查点功能。

cache 和 checkpoint 是有显著区别的， 缓存把 RDD 计算出来然后放在内存中，但是 RDD 的依赖链（相当于数据库中的redo 日志）， 也不能丢掉， 当某个点某个executor 宕了，上面 cache 的 RDD 就会丢掉， 需要通过 依赖链重放计算出来， 不同的是， checkpoint 是把 RDD 保存在 HDFS 中， 是多副本可靠存储，所以依赖链就可以丢掉了，就斩断了依赖链， 是通过复制实现的高容错。

如果存在以下场景，则比较适合使用检查点机制：

1. DAG 中的 Lineage 过长，如果重算，则开销太大（如在 PageRank 中）。
2. 在宽依赖上做 Checkpoint 获得的收益更大。

为当前 RDD 设置检查点。该函数将会创建一个二进制的文件，并存储到 checkpoint 目录中，该目录是用 [Spark](https://www.iteblog.com/archives/tag/spark/)Context.setCheckpointDir()设置的。在 checkpoint 的过程中，该RDD 的所有依赖于父 RDD 中的信息将全部被移出。对 RDD 进行 checkpoint 操作并不会马上被执行，必须执行 Action 操作才能触发。

scala> val data = sc.parallelize(1 to 100 , 5) data: org.apache.spark.rdd.RDD[Int] =

ParallelCollectionRDD[12] at parallelize at <console>:12

scala> sc.setCheckpointDir("hdfs://192.168.137.51:8020/checkpoint") scala> data.checkpoint

scala> data.count

scala> val ch1 = sc.parallelize(1 to 2)

ch1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[33] at parallelize at <console>:25

scala>

val

ch2

=

ch1.map(\_.toString+"["+System.currentTimeMillis+"]")

ch2: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[36] at map at <console>:27

scala>

val

ch3

=

ch1.map(\_.toString+"["+System.currentTimeMillis+"]")

ch3: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[37] at map at <console>:27

scala> ch3.checkpoint scala> ch2.collect

res62: Array[String] = Array(1[1505480940726], 2[1505480940243])

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| scala> res63: | ch2.collect Array[String] | = | Array(1[1505480941957], | 2[1505480941480]) |
| scala> res64: | ch2.collect Array[String] | = | Array(1[1505480942736], | 2[1505480942257]) |
| scala> res65: | ch3.collect Array[String] | = | Array(1[1505480949080], | 2[1505480948603]) |
| scala> res66: | ch3.collect Array[String] | = | Array(1[1505480948683], | 2[1505480949161]) |
| scala> res67: | ch3.collect Array[String] | = | Array(1[1505480948683], | 2[1505480949161]) |

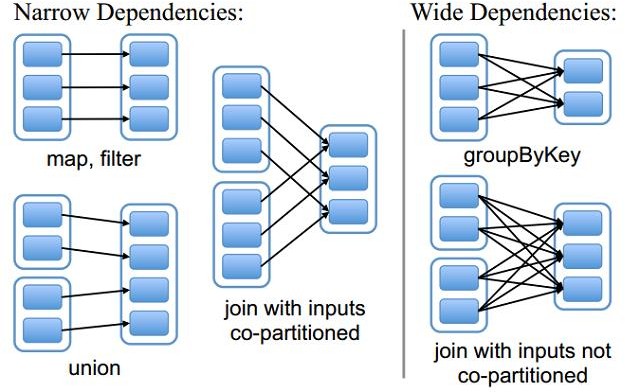
* 1. **RDD 依赖关系**

#### sc.makeRDD("").flatMap("").map().reduceBykey().collect()

#### new SparkConf().setMaster("local[\*]") sc.textFile("")

RDD 和它依赖的父 RDD（s）的关系有两种不同的类型，即窄依赖（narrow

dependency）和宽依赖（wide dependency）。



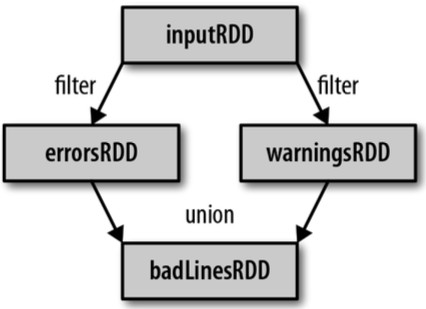
#### 窄依赖

窄依赖指的是每一个父 RDD 的 Partition 最多被子 RDD 的一个 Partition 使用,窄依赖我们形象的比喻为独生子女

#### 宽依赖

宽依赖指的是多个子 RDD 的 Partition 会依赖同一个父 RDD 的 Partition，会引起shuffle,总结：宽依赖我们形象的比喻为超生

* + 1. **Lineage**

RDD 只支持粗粒度转换，即在大量记录上执行的单个操作。将创建 RDD 的一系列Lineage（即血统）记录下来，以便恢复丢失的分区。RDD 的 Lineage 会记录 RDD 的元数据信息和转换行为，当该 RDD 的部分分区数据丢失时，它可以根据这些信息来重新运算和恢复丢失的数据分区。

scala> val text = sc.textFile("README.md") text: org.apache.spark.rdd.RDD[String]

=

README.md

MapPartitionsRDD[1] at textFile at <console>:24

scala> val words = text.flatMap(\_.split) split splitAt

scala> val words = text.flatMap(\_.split(" "))

words: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[2] at flatMap at <console>:26

scala> words.map((\_,1))

res0: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[3] at map at <console>:29

scala> res0.reduceByKey reduceByKey reduceByKeyLocally

scala> res0.reduceByKey(\_+\_)

res1: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[4] at reduceByKey at <console>:31

scala> res1.dependencies res2:

Seq[org.apache.spark.Dependency[\_]]

=

List(org.apache.spark.ShuffleDependency@6cfe48a4)

scala> res0.dependencies

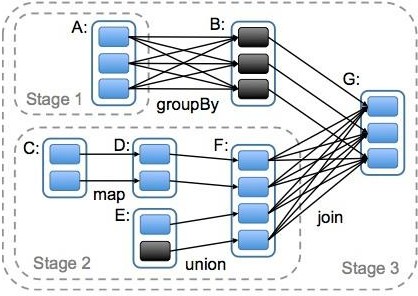
res3:

Seq[org.apache.spark.Dependency[\_]]

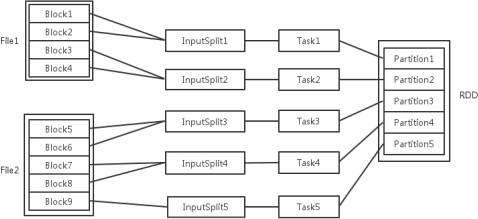
=

List(org.apache.spark.OneToOneDependency@6c9e24c4)

* 1. **DAG 的生成**

DAG(Directed Acyclic Graph)叫做有向无环图，原始的 RDD 通过一系列的转换就就形成了 DAG，根据 RDD 之间的依赖关系的不同将 DAG 划分成不同的 Stage，对于窄依赖，partition 的转换处理在 Stage 中完成计算。对于宽依赖，由于有 Shuffle 的存在，只能在 parent RDD 处理完成后，才能开始接下来的计算，因此宽依赖是划分 Stage 的依据。

* 1. **RDD 相关概念关系**



输入可能以多个文件的形式存储在 HDFS 上，每个 File 都包含了很多块，称为Block。当 Spark 读取这些文件作为输入时，会根据具体数据格式对应的 InputFormat 进行解析，一般是将若干个 Block 合并成一个输入分片，称为 InputSplit，注意 InputSplit 不能

跨越文件。随后将为这些输入分片生成具体的 Task。InputSplit 与 Task 是一一对应的关系。随后这些具体的 Task 每个都会被分配到集群上的某个节点的某个 Executor 去执行。

1. 每个节点可以起一个或多个 Executor。
2. 每个Executor 由若干core 组成，每个Executor 的每个core 一次只能执行一个Task。
3. 每个 Task 执行的结果就是生成了目标 RDD 的一个 partiton。

注意: `这里的 core 是虚拟的 core 而不是机器的物理 CPU 核，可以理解为就是Executor 的一个工作线程。而 Task 被执行的并发度 = Executor 数目 \* 每个 Executor 核数。至于 partition 的数目：

1. 对于数据读入阶段，例如 sc.textFile，输入文件被划分为多少 InputSplit 就会需要多少初始 Task。
2. 在 Map 阶段 partition 数目保持不变。
3. 在 Reduce 阶段，RDD 的聚合会触发 shuffle 操作，聚合后的 RDD 的 partition 数目跟具体操作有关，例如 repartition 操作会聚合成指定分区数，还有一些算子是可配置的。

RDD 在计算的时候，每个分区都会起一个 task，所以 rdd 的分区数目决定了总的的task 数目。申请的计算节点（Executor）数目和每个计算节点核数，决定了你同一时刻可以并行执行的 task。

比如的 RDD 有 100 个分区，那么计算的时候就会生成 100 个 task，你的资源配置为10 个计算节点，每个两 2 个核，同一时刻可以并行的 task 数目为 20，计算这个 RDD 就需要 5 个轮次。如果计算资源不变，你有 101 个 task 的话，就需要 6 个轮次，在最后一轮中，只有一个 task 在执行，其余核都在空转。如果资源不变，你的 RDD 只有 2 个分区， 那么同一时刻只有 2 个 task 运行，其余 18 个核空转，造成资源浪费。这就是在 spark 调优中，增大 RDD 分区数目，增大任务并行度的做法。

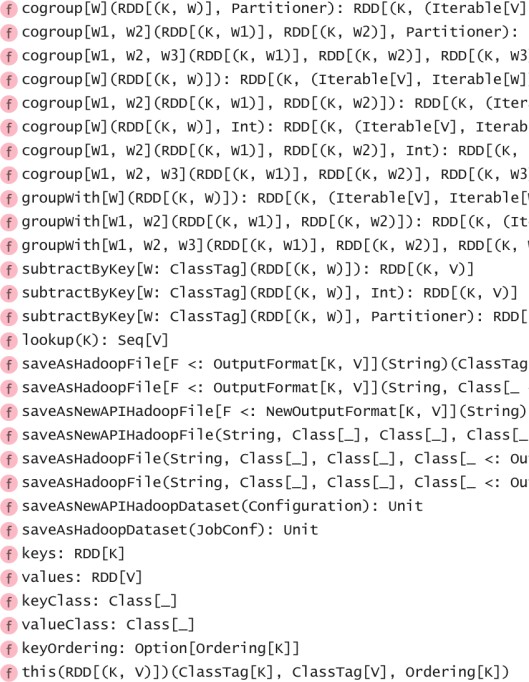
第 **3** 章 键值对 **RDD**

键值对 RDD 是 Spark 中许多操作所需要的常见数据类型。 本章做特别讲解。除了在基础 RDD 类中定义的操作之外，Spark 为包含键值对类型的 RDD 提供了一些专有的操作 在 PairRDDFunctions 专门进行了定义。这些 RDD 被称为 pair RDD。

有很多种方式创建 pair RDD，在输入输出章节会讲解。一般如果从一个普通的 RDD

转 为 pair RDD 时，可以调用 map()函数来实现，传递的函数需要返回键值对。

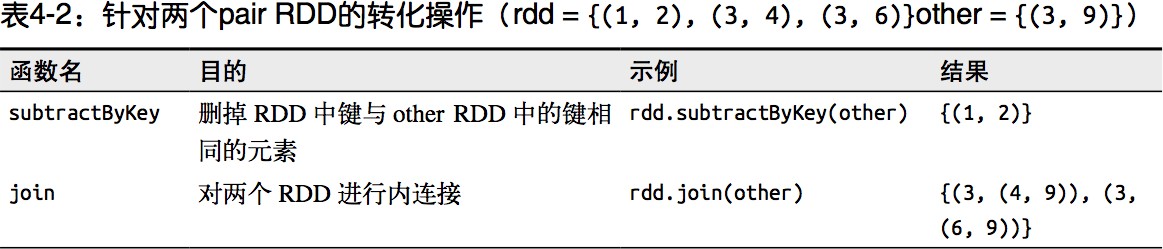
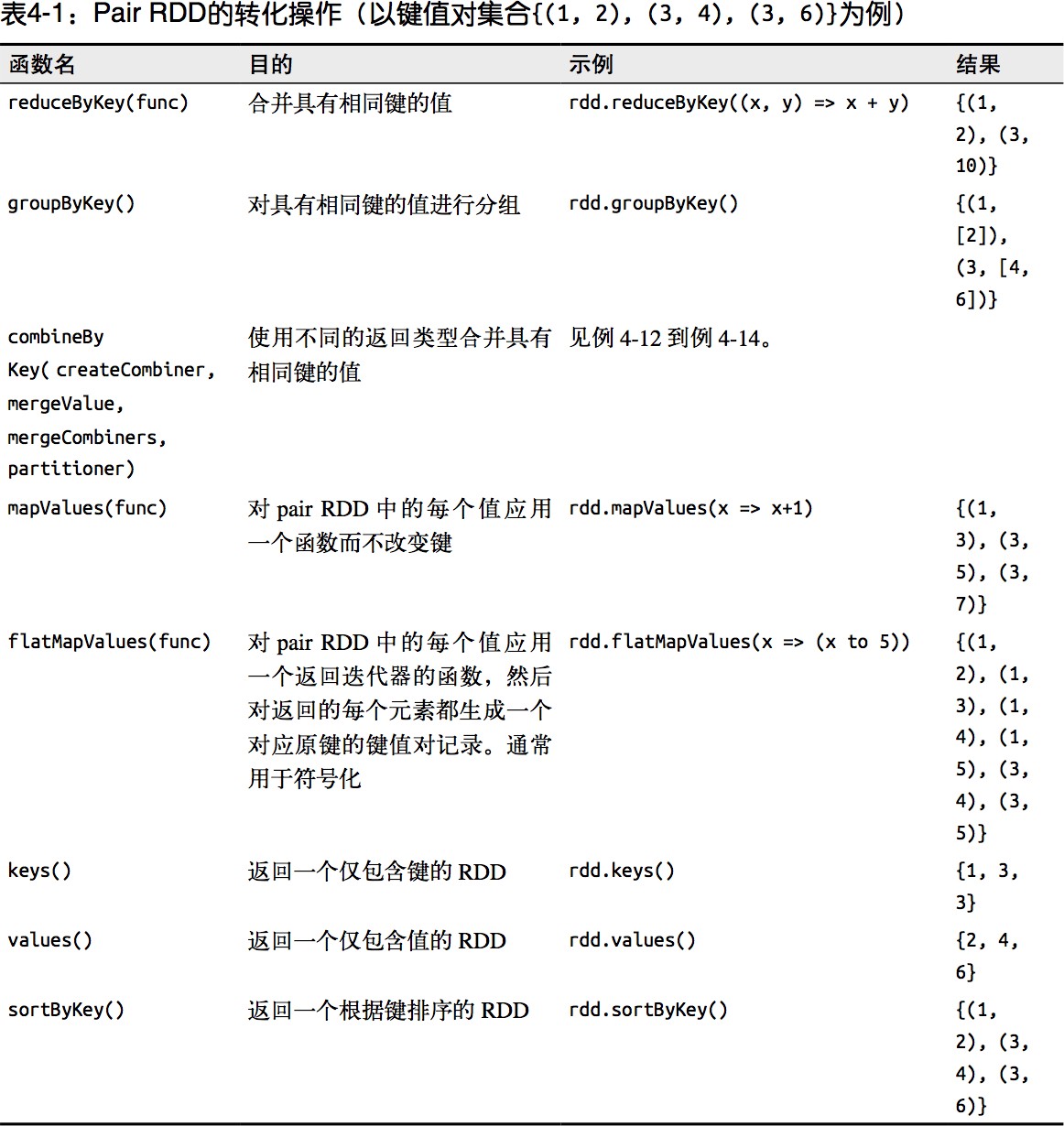
val pairs = lines.map(x => (x.split(" ")(0), x))



* 1. **键值对 RDD 的转化操作**

#### 转化操作列表

上一章进行了练习，这一章会重点讲解。针对一个 Pair RDD 的转化操作





#### 聚合操作

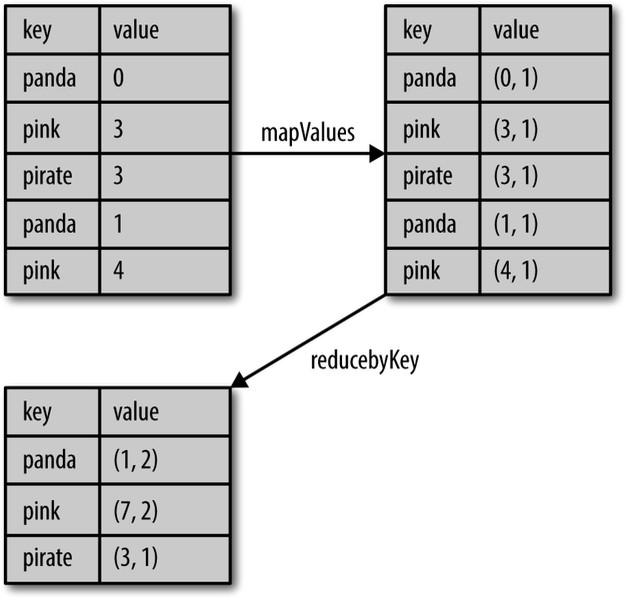
当数据集以键值对形式组织的时候，聚合具有相同键的元素进行一些统计是很常见的操 作。之前讲解过基础 RDD 上的 fold()、combine()、reduce()等行动操作，pair RDD 上则 有相应的针对键的转化操作。Spark 有一组类似的操作，可以组合具有相同键的值。这些 操作返回 RDD，因此它们是转化操作而不是行动操作。

reduceByKey() 与 reduce() 相当类似;它们都接收一个函数，并使用该函数对值进行合并。 reduceByKey() 会为数据集中的每个键进行并行的归约操作，每个归约操作会将键相同的值合 并起来。因为数据集中可能有大量的键，所以 reduceByKey() 没有被实现为向用户程序返回一 个值的行动操作。实际上，它会返回一个由各键和对应键归约出来的结果值组成的新的 RDD。

foldByKey() 则与 fold() 相当类似;它们都使用一个与 RDD 和合并函数中的数据类型相 同的零值作为初始值。与 fold() 一样，foldByKey() 操作所使用的合并函数对零值与另一 个元素进行合并，结果仍为该元素。

求均值操作：版本一

input.mapValues(x => (x, 1)).reduceByKey((x, y) => (x.\_1 + y.\_1, x.\_2 + y.\_2)).map{ case (key, value) => (key, value.\_1 / value.\_2.toFloat) }



combineByKey() 是最为常用的基于键进行聚合的函数。大多数基于键聚合的函数都是用它 实现的。和 aggregate() 一样，combineByKey() 可以让用户返回与输入数据的类型不同的 返回值。

要理解 combineByKey()，要先理解它在处理数据时是如何处理每个元素的。由于combineByKey() 会遍历分区中的所有元素，因此每个元素的键要么还没有遇到过，要么就和之前的某个元素的键相同。

如果这是一个新的元素，combineByKey() 会使用一个叫作 createCombiner() 的函数来创建 那个键对应的累加器的初始值。需要注意的是，这一过程会在每个分区中第一次出现各个 键时发生，而不是在整个 RDD 中第一次出现一个键时发生。

如果这是一个在处理当前分区之前已经遇到的键，它会使用 mergeValue() 方法将该键的累 加器对应的当前值与这个新的值进行合并。

由于每个分区都是独立处理的，因此对于同一个键可以有多个累加器。如果有两个或者更 多的分区都有对应同一个键的累加器，就需要使用用户提供的 mergeCombiners() 方法将各 个分区的结果进行合并。

求均值：版本二

val result = input.combineByKey( (v) => (v, 1),

(acc: (Int, Int), v) => (acc.\_1 + v, acc.\_2 + 1),

(acc1: (Int, Int), acc2: (Int, Int)) => (acc1.\_1 + acc2.\_1, acc1.\_2 + acc2.\_2)

).map{ case (key, value) => (key, value.\_1 / value.\_2.toFloat) }

result.collectAsMap().map(println(\_))

#### 数据分组

如果数据已经以预期的方式提取了键，groupByKey() 就会使用 RDD 中的键来对数据进行 分组。对于一个由类型 K 的键和类型 V 的值组成的 RDD，所得到的结果 RDD 类型会是 [K, Iterable[V]]。

groupBy() 可以用于未成对的数据上，也可以根据除键相同以外的条件进行分组。它可以 接收一个函数，对源 RDD 中的每个元素使用该函数，将返回结果作为键再进行分组。

多个 RDD 分组，可以使用 cogroup 函数，cogroup() 的函数对多个共享同 一个键的RDD 进行分组。对两个键的类型均为 K 而值的类型分别为 V 和 W 的 RDD 进行cogroup() 时，得到的结果 RDD 类型为 [(K, (Iterable[V], Iterable[W]))]。如果其中的 一个RDD 对于另一个 RDD 中存在的某个键没有对应的记录，那么对应的迭代器则为空。cogroup() 提供了为多个 RDD 进行数据分组的方法。

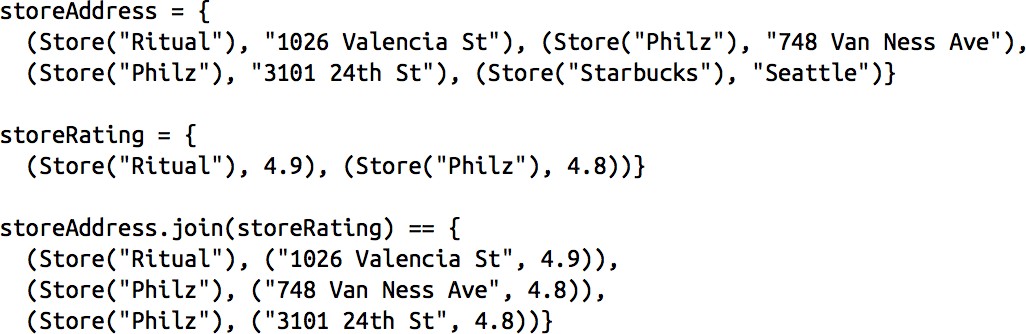
#### 连接

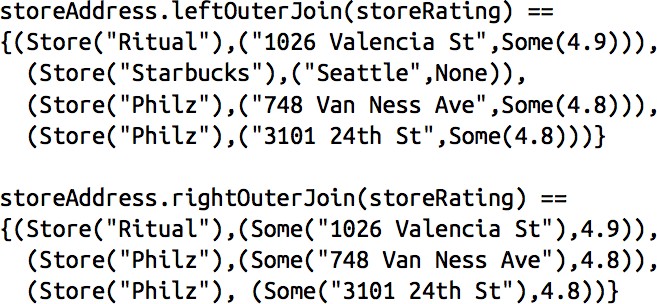
连接主要用于多个 Pair RDD 的操作，连接方式多种多样:右外连接、左外连接、交叉连接以及内连接。

普通的 join 操作符表示内连接 2。只有在两个 pair RDD 中都存在的键才叫输出。当一个输 入对应的某个键有多个值时，生成的 pair RDD 会包括来自两个输入 RDD 的每一组相对应 的记录。

leftOuterJoin()产生的 pair RDD 中，源 RDD 的每一个键都有对应的记录。每个 键相应的值是由一个源 RDD 中的值与一个包含第二个 RDD 的值的 Option(在 Java 中为Optional)对象组成的二元组。

rightOuterJoin() 几乎与 leftOuterJoin() 完全一样，只不过预期结果中的键必须出现在 第二个 RDD 中，而二元组中的可缺失的部分则来自于源 RDD 而非第二个 RDD。

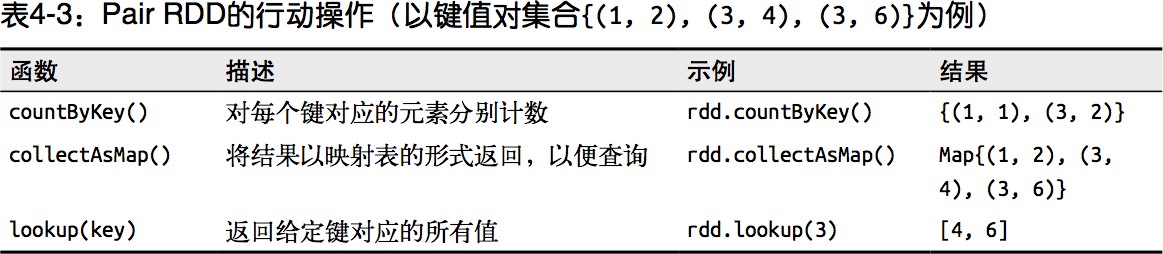




#### 数据排序

sortByKey() 函数接收一个叫作 ascending 的参数，表示我们是否想要让结果按升序排序(默认值为 true)。

* 1. **键值对 RDD 的行动操作**



* 1. **键值对 RDD 的数据分区**

Spark 目前支持 Hash 分区和 Range 分区，用户也可以自定义分区，Hash 分区为当前的默认分区，Spark 中分区器直接决定了 RDD 中分区的个数、RDD 中每条数据经过Shuffle 过程属于哪个分区和 Reduce 的个数，注意：

1. 只有 Key-Value 类型的 RDD 才有分区的，非 Key-Value 类型的 RDD 分区的值是

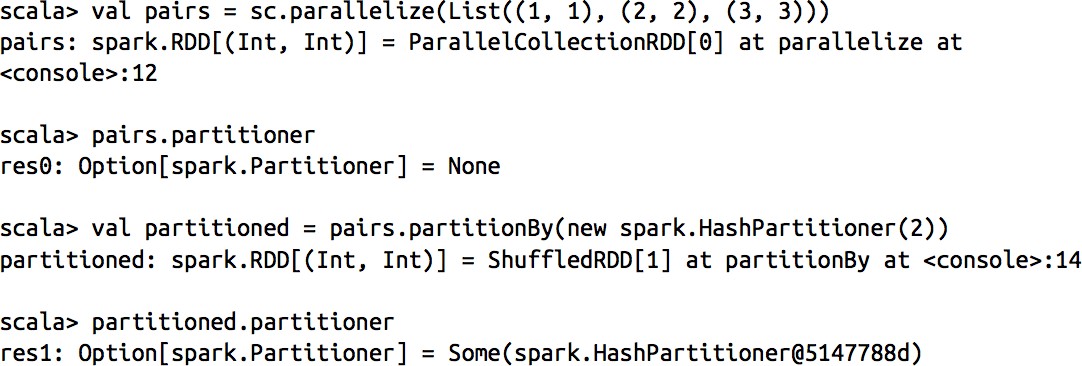
None

1. 每个 RDD 的分区 ID 范围：0~numPartitions-1，决定这个值是属于那个分区的。
   * 1. 获取 **RDD** 分区的方式

spark.default.parallelism 默认分区。

可以通过使用 RDD 的 partitioner 属性来获取 RDD 的分区方式。它会返回一个

scala.Option 对象， 通过 get 方法获取其中的值。



* + 1. **Hash** 分区方式

HashPartitioner 分区的原理：对于给定的 key，计算其 hashCode，并除于分区的个数取余，如果余数小于 0，则用余数+分区的个数，最后返回的值就是这个 key 所属的分区ID。

scala> nopar.partitioner

res20: Option[org.apache.spark.Partitioner] = None

scala>

val

nopar

=

sc.parallelize(List((1,3),(1,2),(2,4),(2,3),(3,6),(3,8)),8)

nopar: org.apache.spark.rdd.RDD[(Int, Int)] = ParallelCollectionRDD[10] at parallelize at <console>:24

scala>nopar.mapPartitionsWithIndex((index,iter)=>{ Iterator(index

.toString+" : "+iter.mkString("|")) }).collect

res0: Array[String] = Array("0 : ", 1 : (1,3), 2 : (1,2), 3 : (2,4),

"4 : ", 5 : (2,3), 6 : (3,6), 7 : (3,8))

scala> val hashpar = nopar.partitionBy(new org.apache.spark.HashPartitioner(7))

hashpar: org.apache.spark.rdd.RDD[(Int, Int)] = ShuffledRDD[12] at partitionBy at <console>:26

scala> hashpar.count res18: Long = 6

scala> hashpar.partitioner res21:

Option[org.apache.spark.Partitioner]

=

Some(org.apache.spark.HashPartitioner@7)

scala>

hashpar.mapPartitions(iter

=>

Iterator(iter.length)).collect()

res19: Array[Int] = Array(0, 3, 1, 2, 0, 0, 0)

* + 1. **Ranger** 分区方式

HashPartitioner 分区弊端：可能导致每个分区中数据量的不均匀，极端情况下会导致某些分区拥有 RDD 的全部数据。

RangePartitioner 分区优势：尽量保证每个分区中数据量的均匀，而且分区与分区之间是有序的，一个分区中的元素肯定都是比另一个分区内的元素小或者大；

但是分区内的元素是不能保证顺序的。简单的说就是将一定范围内的数映射到某一个分区内。

RangePartitioner 作用：将一定范围内的数映射到某一个分区内，在实现中，分界的[算法](http://lib.csdn.net/base/datastructure)尤为重要。用到了水塘抽样算法。

#### 自定义分区方式

要实现自定义的分区器，你需要继承 org.apache.spark.Partitioner 类并实现下面三个方法。

numPartitions: Int:返回创建出来的分区数。

getPartition(key: Any): Int:返回给定键的分区编号(0 到numPartitions-1)。equals():Java 判断相等性的标准方法。这个方法的实现非常重要，Spark 需要用这个

方法来检查你的分区器对象是否和其他分区器实例相同，这样 Spark 才可以判断两个

RDD 的分区方式是否相同。

假设我们需要将相同后缀的数据写入相同的文件，我们通过将相同后缀的数据分区到相同的分区并保存输出来实现。

package com.atguigu.spark

import org.apache.spark.{Partitioner, SparkConf, SparkContext} class CustomerPartitioner(numParts:Int) extends Partitioner {

//覆盖分区数

override def numPartitions: Int = numParts

//覆盖分区号获取函数

override def getPartition(key: Any): Int = { val ckey: String = key.toString

ckey.substring(ckey.length-1).toInt%numParts

}

}

object CustomerPartitioner {

def main(args: Array[String]) {

val conf=new SparkConf().setAppName("partitioner") val sc=new SparkContext(conf)

val data=sc.parallelize(List("aa.2","bb.2","cc.3","dd.3","ee.5"))

data.map((\_,1)).partitionBy(new CustomerPartitioner(5)).keys.saveAsTextFile("hdfs://master01:9000

/partitioner")

}

}

scala> val

data=sc.parallelize(List("aa.2","bb.2","cc.3","dd.3","ee.5").zipW ithIndex,2)

data: org.apache.spark.rdd.RDD[(String, Int)] =

ParallelCollectionRDD[7] at parallelize at <console>:27

scala> data.collect

res4: Array[(String, Int)] = Array((aa.2,0), (bb.2,1), (cc.3,2), (dd.3,3), (ee.5,4))

scala> data.mapPartitionsWithIndex((index,iter)=>Iterator(index.toString

+" : "+ iter.mkString("|"))).collect

res5: Array[String] = Array(0 : (aa.2,0)|(bb.2,1), 1 :

(cc.3,2)|(dd.3,3)|(ee.5,4))

scala> :paste

// Entering paste mode (ctrl-D to finish)

class CustomerPartitioner(numParts:Int) extends org.apache.spark.Partitioner{

//覆盖分区数

override def numPartitions: Int = numParts

//覆盖分区号获取函数

override def getPartition(key: Any): Int = { val ckey: String = key.toString ckey.substring(ckey.length-1).toInt%numParts

}

}

// Exiting paste mode, now interpreting. defined class CustomerPartitioner

scala> data.partitionBy(new CustomerPartitioner(4))

res7: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[9] at partitionBy at <console>:31

scala>

res7.mapPartitionsWithIndex((index,iter)=>Iterator(index.toString

+" : "+ iter.mkString("|"))).collect

res8: Array[String] = Array("0 : ", 1 : (ee.5,4), 2 :

(aa.2,0)|(bb.2,1), 3 : (cc.3,2)|(dd.3,3))

使用自定义的 Partitioner 是很容易的:只要把它传给 partitionBy() 方法即可。Spark 中有许多依赖于数据混洗的方法，比如 join() 和 groupByKey()，它们也可以接收一个可选的 Partitioner 对象来控制输出数据的分区方式。

* + 1. 分区 **shuffle** 优化

在分布式程序中， 通信的代价是很大的，因此控制数据分布以获得最少的网络传输可以极大地提升整体性能。

Spark 中所有的键值对 RDD 都可以进行分区。系统会根据一个针对键的函数对元素进行分 组。 主要有哈希分区和范围分区，当然用户也可以自定义分区函数。

通过分区可以有效提升程序性能。如下例子：

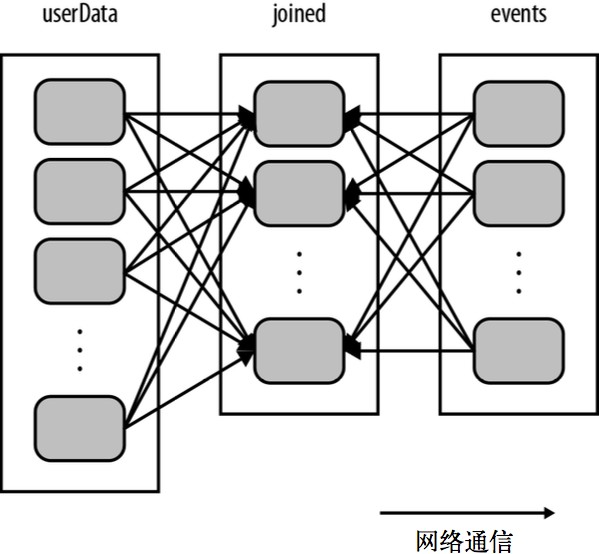
分析这样一个应用，它在内存中保存着一张很大的用户信息表—— 也就是一个由

(UserID, UserInfo) 对组成的 RDD，其中 UserInfo 包含一个该用户所订阅 的主题的列

表。该应用会周期性地将这张表与一个小文件进行组合，这个小文件中存着过 去五分钟内发生的事件——其实就是一个由 (UserID, LinkInfo) 对组成的表，存放着过去 五分钟内某网站各用户的访问情况。例如，我们可能需要对用户访问其未订阅主题的页面 的情况进行统计。

解决方案一：



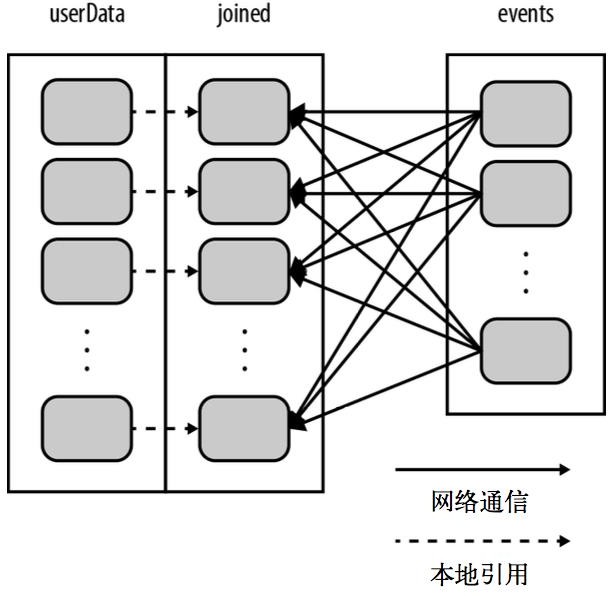


这段代码可以正确运行，但是不够高效。这是因为在每次调用 processNewLogs() 时都会用 到 join() 操作，而我们对数据集是如何分区的却一无所知。默认情况下，连接操作会将两 个数据集中的所有键的哈希值都求出来，将该哈希值相同的记录通过网络传到同一台机器 上，然后在那台机器上对所有键相同的记录进行连接操作。因为 userData 表比每五分钟出现的访问日志表 events 要大得多，所以要浪费时间做很多额外工作:在每次调用时都对 userData 表进行哈希值计算和跨节点数据混洗，降低了程序的执行效率。

优化方法：



我们在构 建 userData 时调用了 partitionBy()，Spark 就知道了该 RDD 是根据键的哈希值来分 区的，这样在调用 join() 时，Spark 就会利用到这一点。具体来说，当调用userData. join(events) 时，Spark 只会对 events 进行数据混洗操作，将 events 中特定UserID 的记 录发送到 userData 的对应分区所在的那台机器上。这样，需要通过网络传输的 数据就大大减少了，程序运行速度也可以显著提升了。



#### 基于分区进行操作

基于分区对数据进行操作可以让我们避免为每个数据元素进行重复的配置工作。诸如打开 数据库连接或创建随机数生成器等操作，都是我们应当尽量避免为每个元素都配置一次的 工作。Spark 提供基于分区的 mapPartition 和 foreachPartition，让你的部分代码只对 RDD 的每个分区运行 一次，这样可以帮助降低这些操作的代价。、

map 每行数据执行函数 1 个 task

mapPartition 每个分区执行函数 每个分区一个 task.

foreache 每行数据遍历。

foreachPartition 每个分区遍历。

#### 从分区中获益的操作

能够从数据分区中获得性能提升的操作有 cogroup()、 groupWith()、join()、leftOuterJoin()、rightOuterJoin()、groupByKey()、reduceByKey()、 combineByKey() 以及lookup()等。

第 **4** 章 数据读取与保存

* 1. **文本文件**

当我们将一个文本文件读取为 RDD 时，输入的每一行 都会成为 RDD 的一个元素。也可以将多个完整的文本文件一次性读取为一个pair RDD， 其中键是文件名，值是文件内容。

val input = sc.textFile("./README.md")

如果传递目录，则将目录下的所有文件读取作为 RDD。文件路径支持通配符。

通过 wholeTextFiles()对于大量的小文件读取效率比较高，大文件效果没有那么高。Spark 通过 saveAsTextFile() 进行文本文件的输出，该方法接收一个路径，并将 RDD

中的内容都输入到路径对应的文件中。Spark 将传入的路径作为目录对待，会在那个 目录下输出多个文件。这样，Spark 就可以从多个节点上并行输出了。

result.saveAsTextFile(outputFile)

scala> sc.textFile("./README.md")

res6: org.apache.spark.rdd.RDD[String]

=

./README.md

MapPartitionsRDD[7] at textFile at <console>:25

scala> val readme = sc.textFile("./README.md")

readme: org.apache.spark.rdd.RDD[String] = ./README.md MapPartitionsRDD[9] at textFile at <console>:24

scala> readme.collect()

res7: Array[String] = Array(# Apache Spark, "", Spark is a fast and general cluster...

scala> readme.saveAsTextFile("hdfs://master01:9000/test")

* 1. **JSON 文件**

如果JSON 文件中每一行就是一个JSON 记录，那么可以通过将JSON 文件当做文本文件来读取，然后利用相关的JSON 库对每一条数据进行JSON 解析。

scala> import org.json4s.\_ import org.json4s.\_

scala> import org.json4s.jackson.JsonMethods.\_ import org.json4s.jackson.JsonMethods.\_

scala> import org.json4s.jackson.Serialization import org.json4s.jackson.Serialization

scala>

var

result

=

sc.textFile("examples/src/main/resources/people.json") result: org.apache.spark.rdd.RDD[String]

examples/src/main/resources/people.json MapPartitionsRDD[7]

=

at

textFile at <console>:47

scala>

implicit

val

formats

=

Serialization.formats(ShortTypeHints(List()))

formats: org.json4s.Formats{val dateFormat: org.json4s.DateFormat; val typeHints: org.json4s.TypeHints} = org.json4s.Serialization$$anon$1@61f2c1da

scala>

result.collect().foreach(x

=>

{var

c

=

parse(x).extract[Person];println(c.name + "," + c.age)}) Michael,30

Andy,30 Justin,19

* 1. **CSV 文件**

读取 CSV/TSV 数据和读取 JSON 数据相似，都需要先把文件当作普通文本文件来读取数据，然后通过将每一行进行解析实现对 CSV 的读取。

CSV/TSV 数据的输出也是需要将结构化 RDD 通过相关的库转换成字符串 RDD，然后使用 Spark 的文本文件 API 写出

去。·····································

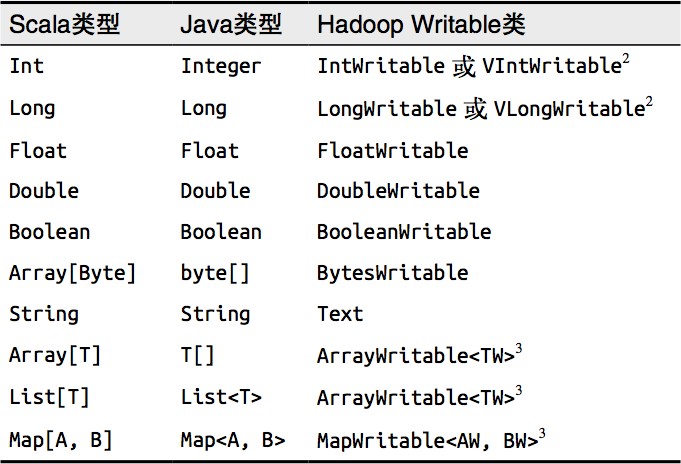
·······

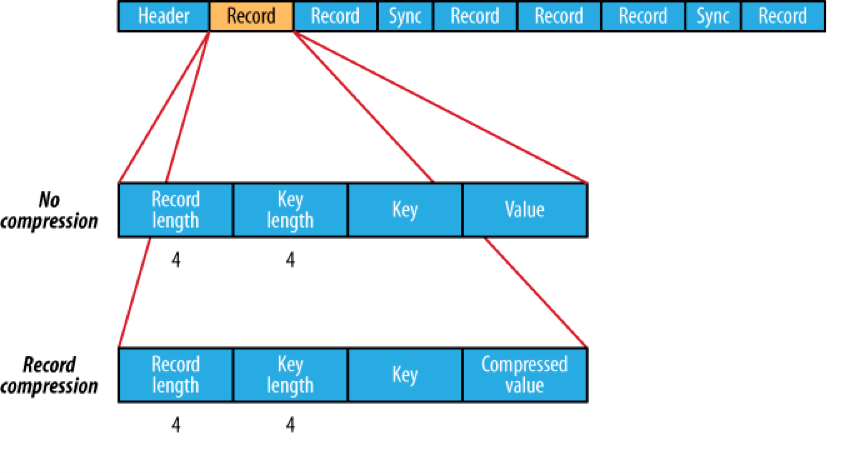
* 1. **Sequence 文件**

SequenceFile 文件是 [Hadoop](http://lib.csdn.net/base/hadoop) 用来存储二进制形式的 key-value 对而设计的一种平面文件(Flat File)。

Spark 有专门用来读取 SequenceFile 的接口。在 SparkContext 中，可以调用

sequenceFile[ keyClass, valueClass](path)。





* 1. **对象文件**

对象文件是将对象序列化后保存的文件，采用 Java 的序列化机制。可以通过objectFile[k,v](path) 函数接收一个路径，读取对象文件，返回对应的 RDD，也可以通过调用 saveAsObjectFile() 实现对对象文件的输出。因为是序列化所以要指定类型。

scala>

val

data=sc.parallelize(List((2,"aa"),(3,"bb"),(4,"cc"),(5,"dd"),(6," ee")))

data: org.apache.spark.rdd.RDD[(Int, String)] =

ParallelCollectionRDD[20] at parallelize at <console>:24

scala> data.saveAsObjectFile("hdfs://master01:9000/objfile") scala> import org.apache.spark.rdd.RDD

import org.apache.spark.rdd.RDD

scala>

val

objrdd:RDD[(Int,String)]

=

sc.objectFile[(Int,String)]("hdfs://master01:9000/objfile/p\*") objrdd: org.apache.spark.rdd.RDD[(Int, String)] =

MapPartitionsRDD[28] at objectFile at <console>:25

scala> objrdd.collect()

res20: Array[(Int, String)] = Array((2,aa), (3,bb), (4,cc), (5,dd), (6,ee))

* 1. **HDFS**

Spark 的整个生态系统与Hadoop 是完全兼容的,所以对于 Hadoop 所支持的文件类型或者数据库类型,Spark 也同样支持.另外,由于 Hadoop 的 API 有新旧两个版本,所以 Spark 为了能够兼容Hadoop 所有的版本,也提供了两套创建操作接口.对于外部存储创建操作而

言,hadoopRDD 和 newHadoopRDD 是最为抽象的两个函数接口,主要包含以下四个参数.

* + 1. 输入格式(InputFormat): 制定数据输入的类型,如 TextInputFormat 等,新旧两个版本所引用的版本分别是 org.apache.hadoop.mapred.InputFormat 和org.apache.hadoop.mapreduce.InputFormat(NewInputFormat)
    2. 键类型: 指定[K,V]键值对中 K 的类型
    3. 值类型: 指定[K,V]键值对中 V 的类型
    4. 分区值: 指定由外部存储生成的 RDD 的 partition 数量的最小值,如果没有指定,系统会使用默认值 defaultMinSplits

其他创建操作的 API 接口都是为了方便最终的 Spark 程序开发者而设置的,是这两个接口的高效实现版本.例如,对于 textFile 而言,只有 path 这个指定文件路径的参数,其他参数在系统内部指定了默认值

兼容旧版本 **HadoopAPI** 的创建操作

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **文件路径** | 输入格**式** | 键类**型** | 值类**型** | **分区**值 |
| textFile(path: String, minPartitions: Int = defaultMinPartitions) | path | TextInputFormat | LongWritabl e | Text | minSplits |
| hadoopFile[K, V, F <: InputFormat[K, V]](path: String, minPartitions: Int) (implicit km: ClassTag[K], vm: ClassTag[V], fm:  ClassTag[F]): RDD[(K, V)] | path | F | K | V | minSplits |
| hadoopFile[K, V, F <: [K, V]](path: String)  (implicit km: ClassTag[K], vm: ClassTag[V], fm:  ClassTag[F]): RDD[(K, V)] | path | F | K | V | DefaultM inSplits |
| hadoopFile[K, V](path: String, inputFormatClass: Class[\_ <: InputFormat[K, V]], keyClass: Class[K], valueClass: Class[V], minPartitions: Int = defaultMinPartitions):  RDD[(K, V)] | path | inputFormatClass | keyClass | valueCl ass | defaultMi nPartition s |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| hadoopRDD[K, V](conf: JobConf, inputFormatClass: Class[\_ <: InputFormat[K, V]], keyClass: Class[K], valueClass: Class[V], minPartitions: Int = defaultMinPartitions):  RDD[(K, V)] | n/a | inputFormatClass | keyClass | valueCl ass | defaultMi nPartition s |
| sequenceFile[K, V](path: String, minPartitions: Int = defaultMinPartitions) (implicit km: ClassTag[K],  vm: ClassTag[V], kcf: () ⇒  WritableConverter[K], vcf:  () ⇒ WritableConverter[V]): RDD[(K, V)] | path | SequenceFileInputF ormat[K,V] | K | V | defaultMi nPartition s |
| objectFile[T](path: String, minPartitions: Int = defaultMinPartitions)(implic it arg0: ClassTag[T]): RDD[T] | path | SequenceFileInputF ormat[NullWritable  ,BytesWritable] | NullWritable | BytesW ritable | minSplits |

兼容新版本 **HadoopAPI** 的创建操作

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 文件路径 | 输入格式 | 键类  型 | 值类  型 | 分区值 |
| newAPIHadoopFile[K, V, F <: InputFormat[K, V]](path: String, fClass: Class[F], kClass: Class[K], vClass: Class[V], conf: Configuration = hadoopConfiguration): RDD[(K,  V)] | path | F | K | V | n/a |
| newAPIHadoopFile[K, V, F <: InputFormat[K, V]](path: String)(implicit km: ClassTag[K], vm: ClassTag[V], fm:  ClassTag[F]): RDD[(K, V)] | path | F | K | V | n/a |
| newAPIHadoopRDD[K, V, F <: InputFormat[K, V]](conf: | n/a | F | K | V | n/a |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Configuration = hadoopConfiguration, fClass: Class[F], kClass: Class[K], vClass:  Class[V]): RDD[(K, V)] |  |  |  |  |  |

注意**:**

1. 在 Hadoop 中以压缩形式存储的数据,不需要指定解压方式就能够进行读取,因为

Hadoop 本身有一个解压器会根据压缩文件的后缀推断解压算法进行解压.

1. 如果用 Spark 从 Hadoop 中读取某种类型的数据不知道怎么读取的时候,上网查找一个使用 map-reduce 的时候是怎么读取这种这种数据的,然后再将对应的读取方式改写成上面的hadoopRDD 和 newAPIHadoopRDD 两个类就行了

读取示例：

scala> val data = sc.parallelize(Array((30,"hadoop"), (71,"hive"), (11,"cat")))

data: org.apache.spark.rdd.RDD[(Int, String)] =

ParallelCollectionRDD[47] at parallelize at <console>:35

scala> data.saveAsNewAPIHadoopFile("hdfs://hadoop102:9000/output4/",clas sOf[LongWritable] ,classOf[Text] ,classOf[org.apache.hadoop.mapre duce.lib.output.TextOutputFormat[LongWritable, Text]])

对于 RDD 最后的归宿除了返回为集合和标量,也可以将 RDD 存储到外部文件系统或者数据库中,Spark 系统与 Hadoop 是完全兼容的,所以 MapReduce 所支持的读写文件或者数据库类型,Spark 也同样支持.另外,由于 Hadoop 的 API 有新旧两个版本,所以 Spark 为了能够兼容 Hadoop 所有的版本,也提供了两套 API。

将 RDD 保存到 HDFS 中在通常情况下需要关注或者设置五个参数,即文件保存的路径,key 值的 class 类型,Value 值的 class 类型,RDD 的输出格式(OutputFormat,如TextOutputFormat/SequenceFileOutputFormat),以及最后一个相关的参数 codec(这个参数表示压缩存储的压缩形式,如 DefaultCodec,Gzip,Codec 等等)

兼容旧版 **API**

|  |
| --- |
| saveAsObjectFile(path: String): Unit |
| saveAsTextFile(path: String, codec: Class[\_ <: CompressionCodec]): Unit |
| saveAsTextFile(path: String): Unit |
| saveAsHadoopFile[F <: OutputFormat[K, V]](path: String)(implicit fm: ClassTag[F]): Unit |
| saveAsHadoopFile[F <: OutputFormat[K, V]](path: String, codec: Class[\_ <: CompressionCodec])(implicit fm: ClassTag[F]): Unit |

|  |
| --- |
| saveAsHadoopFile(path: String, keyClass: Class[\_], valueClass: Class[\_], outputFormatClass: Class[\_ <: OutputFormat[\_, \_]], codec: Class[\_ <: CompressionCodec]): Unit |
| saveAsHadoopDataset(conf: JobConf): Unit |

这里列出的 API,前面 6 个都是 saveAsHadoopDataset 的简易实现版本,仅仅支持将 RDD 存储到 HDFS 中,而 saveAsHadoopDataset 的参数类型是 JobConf,所以其不仅能够将 RDD 存储到 HDFS 中,也可以将 RDD 存储到其他数据库中,如 Hbase,MangoDB,Cassandra 等。

兼容新版 **API**

|  |
| --- |
| saveAsNewAPIHadoopFile(path: String, keyClass: Class[\_], valueClass: Class[\_], outputFormatClass: Class[\_ <: OutputFormat[\_, \_]], conf: Configuration = self.context.hadoopConfiguration): Unit |
| saveAsNewAPIHadoopFile[F <: OutputFormat[K, V]](path: String)(implicit fm: ClassTag[F]): Unit |
| saveAsNewAPIHadoopDataset(conf: Configuration): Unit |

同样的,前 2 个 API 是 saveAsNewAPIHadoopDataset 的简易实现,只能将 RDD 存到HDFS 中,而 saveAsNewAPIHadoopDataset 比较灵活.新版的 API 没有 codec 的参数,所以要压缩存储文件到 HDFS 中每需要使用 hadoopConfiguration 参数,设置对应mapreduce.map.output.compress.codec 参数和 mapreduce.map.output.compress 参数.

注意:如果不知道怎么将 RDD 存储到 Hadoop 生态的系统中,主要上网搜索一下对应的 map- reduce 是怎么将数据存储进去的,然后改写成对应的 saveAsHadoopDataset 或saveAsNewAPIHadoopDataset 就可以了。

* 1. **文件系统**

Spark 支持读写很多种文件系统， 像本地文件系统、Amazon S3、HDFS 等。

* 1. **数据库**

#### 关系型数据库连接

支持通过 Java JDBC 访问关系型数据库。需要通过JdbcRDD 进行，示例如下: Mysql 读取：

def main (args: Array[String] ) { val sparkConf = new

SparkConf

().setMaster

("local[2]").setAppName ("JdbcApp") val sc = new SparkContext (sparkConf)

val rdd = new org.apache.spark.rdd.JdbcRDD ( sc,

() => {

Class.forName ("com.mysql.jdbc.Driver").newInstance() java.sql.DriverManager.getConnection

("jdbc:mysql://master01:3306/rdd", "root", "hive")

},

"select \* from rddtable where id >= ? and id <= ?;", 1,

10,

1,

r => (r.getInt(1), r.getString(2)))

println (rdd.count () ) rdd.foreach (println (\_) ) sc.stop ()

}

Mysql 写入：

def main(args: Array[String]) { val sparkConf

=

new

SparkConf().setMaster("local[2]").setAppName("HBaseApp") val sc = new SparkContext(sparkConf)

val data = sc.parallelize(List("Female", "Male","Female"))

data.foreachPartition(insertData)

}

def insertData(iterator: Iterator[String]): Unit = { Class.forName ("com.mysql.jdbc.Driver").newInstance()

val conn = java.sql.DriverManager.getConnection("jdbc:mysql://master01:3306/ rdd", "root", "hive")

iterator.foreach(data => {

val ps = conn.prepareStatement("insert into rddtable(name) values (?)")

ps.setString(1, data) ps.executeUpdate()

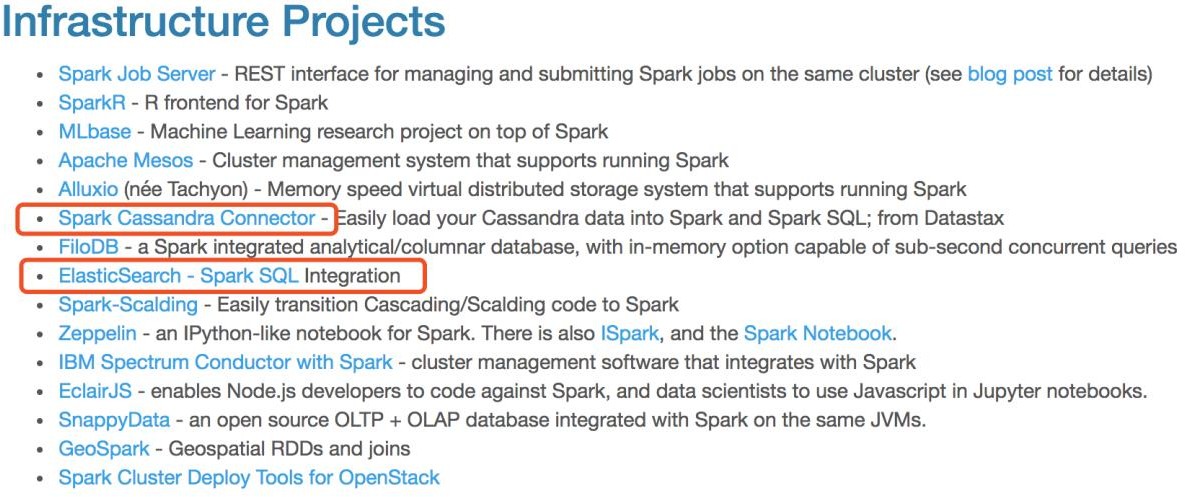
})

}

JdbcRDD 接收这样几个参数。

* 首先，要提供一个用于对数据库创建连接的函数。这个函数让每个节点在连接必要的配 置后创建自己读取数据的连接。
* 接下来，要提供一个可以读取一定范围内数据的查询，以及查询参数中 lowerBound 和upperBound 的值。这些参数可以让 Spark 在不同机器上查询不同范围的数据，这样就不 会因尝试在一个节点上读取所有数据而遭遇性能瓶颈。
* 这个函数的最后一个参数是一个可以将输出结果从转为对操作数据有用的格式的函数。如果这个参数空缺，Spark 会自动将每行结果转为一个对象数组。

Cassandra 数据库和ElasticSearch 集成：



* + 1. **HBase** 数据库

由于 org.apache.hadoop.hbase.mapreduce.TableInputFormat 类的实现，Spark 可以通过Hadoop 输入格式访问 HBase。这个输入格式会返回键值对数据，其中键的类型为 org. apache.hadoop.hbase.io.ImmutableBytesWritable ， 而 值 的 类 型 为org.apache.hadoop.hbase.client.Result。

HBase 读取：

def main(args: Array[String]) { val sparkConf

=

new

SparkConf().setMaster("local[2]").setAppName("HBaseApp") val sc = new SparkContext(sparkConf)

val conf = HBaseConfiguration.create()

//HBase 中的表名

conf.set(TableInputFormat.INPUT\_TABLE, "fruit")

val hBaseRDD = sc.newAPIHadoopRDD(conf, classOf[TableInputFormat], classOf[org.apache.hadoop.hbase.io.ImmutableBytesWritable], classOf[org.apache.hadoop.hbase.client.Result])

val count = hBaseRDD.count() println("hBaseRDD RDD Count:"+ count) hBaseRDD.cache()

hBaseRDD.foreach { case (\_, result) =>

val key = Bytes.toString(result.getRow)

val name = Bytes.toString(result.getValue("info".getBytes, "name".getBytes))

val color = Bytes.toString(result.getValue("info".getBytes, "color".getBytes))

println("Row key:" + key + " Name:" + name + " Color:" + color)

}

sc.stop()

}

HBase 写入：

def main(args: Array[String]) {

val sparkConf = new SparkConf().setMaster("local[2]").setAppName("HBaseApp")

val sc = new SparkContext(sparkConf)

val conf = HBaseConfiguration.create() val jobConf = new JobConf(conf)

jobConf.setOutputFormat(classOf[TableOutputFormat]) jobConf.set(TableOutputFormat.OUTPUT\_TABLE, "fruit\_spark")

val fruitTable = TableName.valueOf("fruit\_spark") val tableDescr = new HTableDescriptor(fruitTable)

tableDescr.addFamily(new HColumnDescriptor("info".getBytes))

val admin = new HBaseAdmin(conf)

if (admin.tableExists(fruitTable)) { admin.disableTable(fruitTable) admin.deleteTable(fruitTable)

}

admin.createTable(tableDescr)

def convert(triple: (Int, String, Int)) = { val put = new Put(Bytes.toBytes(triple.\_1))

put.addImmutable(Bytes.toBytes("info"), Bytes.toBytes("name"), Bytes.toBytes(triple.\_2))

put.addImmutable(Bytes.toBytes("info"), Bytes.toBytes("price"), Bytes.toBytes(triple.\_3))

(new ImmutableBytesWritable, put)

}

val initialRDD = sc.parallelize(List((1,"apple",11), (2,"banana",12), (3,"pear",13)))

val localData = initialRDD.map(convert)

localData.saveAsHadoopDataset(jobConf)

}

第 **5** 章 **RDD** 编程进阶

* 1. **累加器**

累加器用来对信息进行聚合，通常在向 Spark 传递函数时，比如使用 map() 函数或者用 filter() 传条件时，可以使用驱 动器程序中定义的变量，但是集群中运行的每个任务都会得到这些变量的一份新的副本， 更新这些副本的值也不会影响驱动器中的对应变量。 如果我们想实现所有分片处理时更新共享变量的功能，那么累加器可以实现我们想要的效果。

#### 系统累加器

针对一个输入的日志文件，如果我们想计算文件中所有空行的数量，我们可以编写以下程序：

scala> val notice = sc.textFile("./NOTICE") notice: org.apache.spark.rdd.RDD[String]

=

./NOTICE

MapPartitionsRDD[40] at textFile at <console>:32

scala> val blanklines = sc.accumulator(0)

warning: there were two deprecation warnings; re-run with - deprecation for details

blanklines: org.apache.spark.Accumulator[Int] = 0

scala> val tmp = notice.flatMap(line => {

| if (line == "") {

| blanklines += 1

| }

| line.split(" ")

| })

tmp: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[41] at flatMap at <console>:36

scala> tmp.count() res31: Long = 3213

scala> blanklines.value res32: Int = 171

累加器的用法如下所示。

通过在驱动器中调用 SparkContext.accumulator(initialValue)方法，创建出存有初始值的累加器。返回值为 org.apache.spark.Accumulator[T] 对象，其中 T 是初始值 initialValue 的类型。Spark 闭包里的执行器代码可以使用累加器的 += 方法(在 Java 中是 add)增加累加器的值。 驱动器程序可以调用累加器的 value 属性(在 Java 中使用 value()或 setValue())来访问累加器的值。

注意：工作节点上的任务不能访问累加器的值。从这些任务的角度来看，累加器是一个只写变量。

对于要在行动操作中使用的累加器，Spark 只会把每个任务对各累加器的修改应用一次。因此，如果想要一个无论在失败还是重复计 算时都绝对可靠的累加器，我们必须把它放在foreach() 这样的行动操作中。转化操作中累加器可 能会发生不止一次更新

#### 自定义累加器

自定义累加器类型的功能在 1.X 版本中就已经提供了，但是使用起来比较麻烦，在 2.0 版本后， 累加器的易用性有了较大的改进， 而且官方还提供了一个新的抽象类： AccumulatorV2 来提供更加友好的自定义类型累加器的实现方式。实现自定义类型累加器需要继承 AccumulatorV2 并至少覆写下例中出现的方法，下面这个累加器可以用于在程序运行过程中收集一些文本类信息，最终以 Set[String]的形式返回。

package com.atguigu.spark

import org.apache.spark.util.AccumulatorV2

import org.apache.spark.{SparkConf, SparkContext} import scala.collection.JavaConversions.\_

class LogAccumulator extends org.apache.spark.util.AccumulatorV2[String, java.util.Set[String]]

{

private val \_logArray: java.util.Set[String] = new java.util.HashSet[String]()

override def isZero: Boolean = {

\_logArray.isEmpty

}

override def reset(): Unit = {

\_logArray.clear()

}

override def add(v: String): Unit = {

\_logArray.add(v)

}

override def merge(other: org.apache.spark.util.AccumulatorV2[String, java.util.Set[String]]): Unit = {

other match {

case o: LogAccumulator => \_logArray.addAll(o.value)

}

}

override def value: java.util.Set[String] = { java.util.Collections.unmodifiableSet(\_logArray)

}

override def copy():org.apache.spark.util.AccumulatorV2[String, java.util.Set[String]] = {

val newAcc = new LogAccumulator()

\_logArray.synchronized{ newAcc.\_logArray.addAll(\_logArray)

}

newAcc

}

}

// 过滤掉带字母的

object LogAccumulator {

def main(args: Array[String]) {

val conf=new SparkConf().setAppName("LogAccumulator") val sc=new SparkContext(conf)

val accum = new LogAccumulator sc.register(accum, "logAccum")

val sum = sc.parallelize(Array("1", "2a", "3", "4b", "5", "6",

"7cd", "8", "9"), 2).filter(line => {

val pattern = """^-?(\d+)""" val flag = line.matches(pattern) if (!flag) {

accum.add(line)

}

flag

}).map(\_.toInt).reduce(\_ + \_)

println("sum: " + sum)

for (v <- accum.value) print(v + "") println()

sc.stop()

}

}

* 1. **广播变量**

广播变量用来高效分发较大的对象。向所有工作节点发送一个 较大的只读值，以供一个或多个 Spark 操作使用。比如，如果你的应用需要向所有节点发 送一个较大的只读查询表，甚至是机器学习算法中的一个很大的特征向量，广播变量用起 来都很顺手。

传统方式下，Spark 会自动把闭包中所有引用到的变量发送到工作节点上。虽然这很方便，但也很低效。原因有二:首先，默认的任务发射机制是专门为小任务进行优化的；其次，事实上你可能会在多个并行操作中使用同一个变量，但是 Spark 会为每个任务分别发送。

scala> val broadcastVar = sc.broadcast(Array(1, 2, 3)) broadcastVar: org.apache.spark.broadcast.Broadcast[Array[Int]] = Broadcast(35)

scala> broadcastVar.value

res33: Array[Int] = Array(1, 2, 3)

使用广播变量的过程如下：

1. 通过对一个类型 T 的对象调用 SparkContext.broadcast 创建出一个 Broadcast[T]

对象。 任何可序列化的类型都可以这么实现。

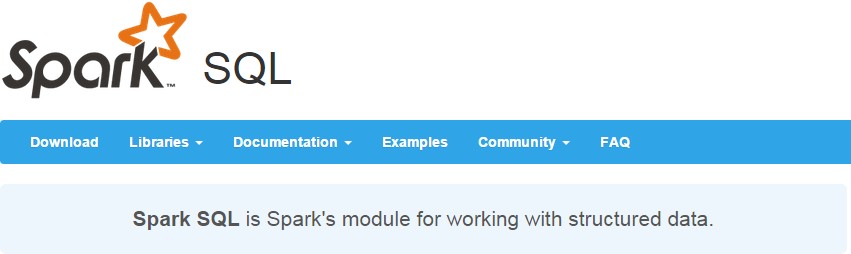
1. 通过 value 属性访问该对象的值(在 Java 中为 value() 方法)。
2. 变量只会被发到各个节点一次，应作为只读值处理(修改这个值不会影响到别的节点)。

大数据技术之 SparkSQL

版本：V1.1

第 **1** 章 **Spark SQL** 概述

* 1. **什么是 Spark SQL**

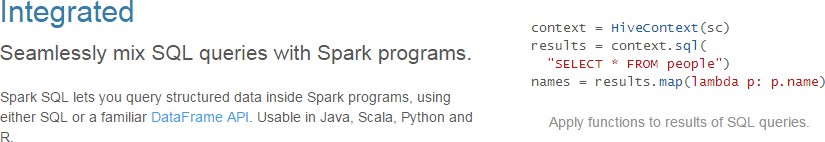


Spark SQL 是 Spark 用来处理（` ETL）数据的一个模块，它提供了一个编程抽象叫做 DataFrame

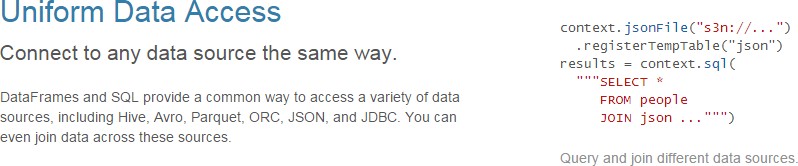
并且作为分布式 SQL 查询引擎的作用。

我们已经学习了 Hive，它是将Hive SQL 转换成 MapReduce 然后提交到集群上执行，大大简化了编写 MapReduce 的程序的复杂性，由于 MapReduce 这种计算模型执行效率比较慢。所有Spark SQL 的应运而生，它是将 Spark SQL 转换成 RDD，然后提交到集群执行，执行效率非常快！

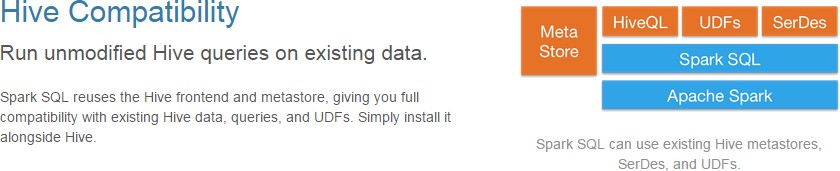
* + 1. 易整合



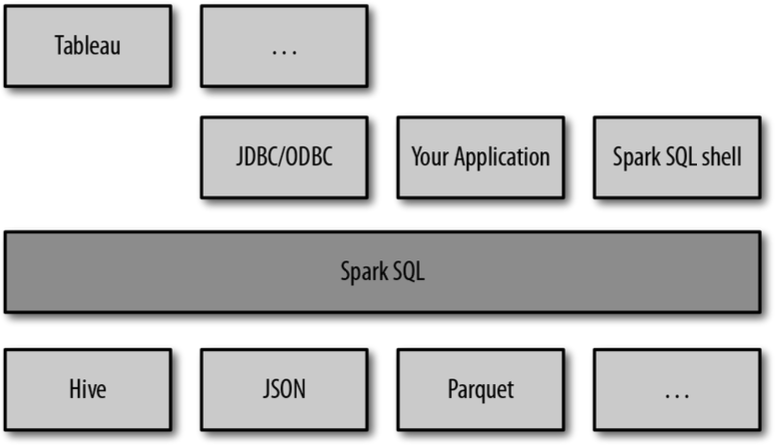
* + 1. 统一的数据访问方式



* + 1. 兼容 Hive



* + 1. 标准的数据连接



SparkSQL 可以看做是一个转换层，向下对接各种不同的结构化数据源，向上提供不同的数据访问方式。

* 1. **RDD vs DataFrames vs DataSet**



在 SparkSQL 中 Spark 为我们提供了两个新的抽象，分别是 DataFrame 和 DataSet。他们和RDD 有什么区别呢？首先从版本的产生上来看：

RDD (Spark1.0) —> Dataframe(Spark1.3) —> Dataset(Spark1.6)

如果同样的数据都给到这三个数据结构，他们分别计算之后，都会给出相同的结果。不同是的他们的执行效率和执行方式。

在后期的 Spark 版本中，DataSet 会逐步取代 RDD 和DataFrame 成为唯一的 API 接口。

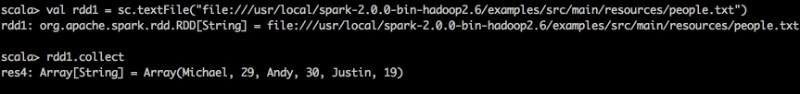
* + 1. **RDD**

RDD 是一个懒执行的不可变的可以支持 Lambda 表达式的并行数据集合。

RDD 的最大好处就是简单，API 的人性化程度很高。

RDD 的劣势是性能限制，它是一个JVM 驻内存对象，这也就决定了存在 GC 的限制和数据增加时Java 序列化成本的升高。

RDD 例子如下:



* + 1. **Dataframe**

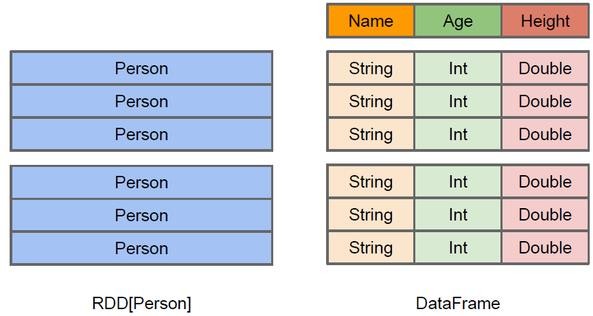
var rdd = sc.textFile("")

var dataframe = spark.read.json("\*\*\*.json") dataframe.rdd;

import spark.implicits.\_

rdd.toDF：

与 RDD 类似，DataFrame 也是一个分布式数据容器。然而 DataFrame 更像传统数据库的二维表格，除了数据以外，还记录数据的结构信息，即 schema。同时，与 Hive 类似，DataFrame 也支持嵌套数据类型（struct、array 和 map）。从 API 易用性的角度上看，DataFrame API 提供的是一套高层的关系操作，比函数式的 RDD API 要更加友好，门槛更低。由于与 R 和Pandas 的DataFrame 类似，Spark DataFrame 很好地继承了传统单机数据分析的开发体验。

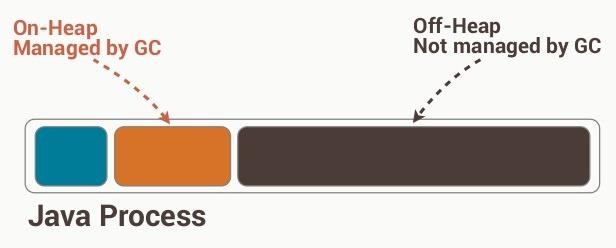


上图直观地体现了DataFrame 和 RDD 的区别。左侧的 RDD[Person]虽然以 Person 为类型参数，但 Spark 框架本身不了解 Person 类的内部结构。而右侧的DataFrame 却提供了详细的结构信息，使得 Spark SQL 可以清楚地知道该数据集中包含哪些列，每列的名称和类型各是什么。DataFrame 多了数据的结构信息，即 schema。RDD 是分布式的 Java 对象的集合。DataFrame 是分布式的 Row 对象的集合。DataFrame 除了提供了比 RDD 更丰富的算子以外，更重要的特点是提升执行效率、减少数据读取以及执行计划的优化，比如 filter 下推、裁剪等。

DataFrame 是为数据提供了 Schema 的视图。可以把它当做数据库中的一张表来对待

DataFrame 也是懒执行的。

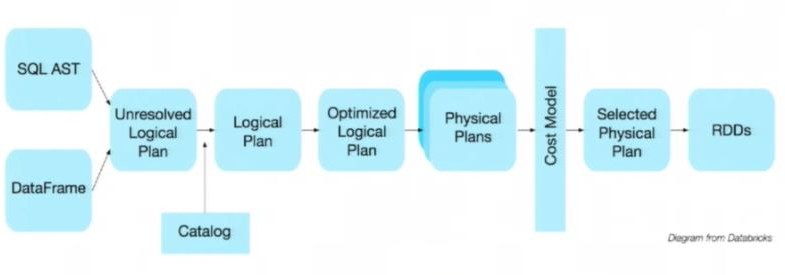
性能上比 RDD 要高，主要有两方面原因： 定制化内存管理

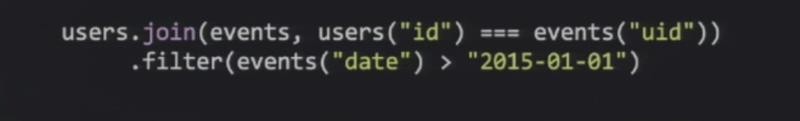
数据以二进制的方式存在于非堆内存，节省了大量空间之外，还摆脱了 GC 的限制。

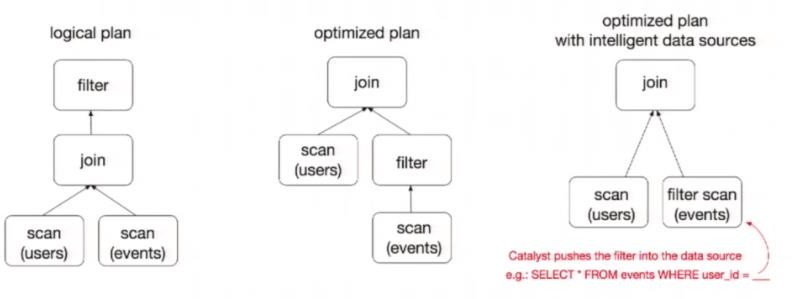
优化的执行计划

Json 连接两个文件。Filter("过滤") RDD: 两个大文件。 再过滤。比较消耗内存。查询计划通过 Spark catalyst optimiser 进行优化.

有一个需求:两个大表.两个超大数据文件.

两个表关联查询,其中一个表中,要通过条件,过滤掉一部分数据. rdd1.join(rdd2).filter(rdd2)

比如下面一个例子：



select \* from table t,emp e where t.id = e.id and e.empno>100

为了说明查询优化，我们来看上图展示的人口数据分析的示例。图中构造了两个 DataFrame， 将它们 join 之后又做了一次 filter 操作。如果原封不动地执行这个执行计划，最终的执行效率是不高的。因为 join 是一个代价较大的操作，也可能会产生一个较大的数据集。如果我们能将 filter 下推到 join 下方，先对 DataFrame 进行过滤，再 join 过滤后的较小的结果集，便可以有效缩短执行时间。而 Spark SQL 的查询优化器正是这样做的。简而言之，逻辑查询计划优化就是一个利用基于关系代数的等价变换，将高成本的操作替换为低成本操作的过程。

得到的优化执行计划在转换成物 理执行计划的过程中，还可以根据具体的数据源的特性将过滤条件下推至数据源内。最右侧的物理执行计划中 Filter 之所以消失不见，就是因为溶入了用于执行最终的读取操作的表扫描节点内。

对于普通开发者而言，查询优化器的意义在于，即便是经验并不丰富的程序员写出的次优的查询，也可以被尽量转换为高效的形式予以执行。

Dataframe 的劣势在于在编译期缺少类型安全检查，导致运行时出错.

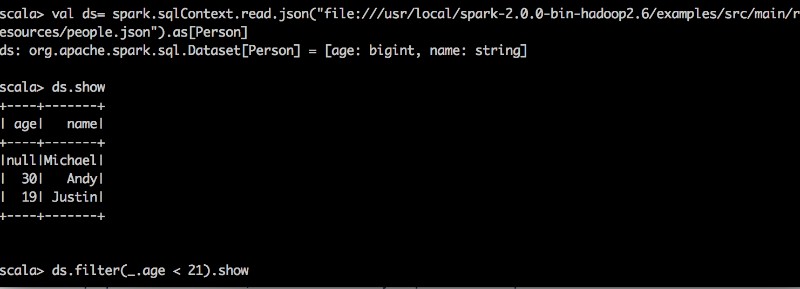
* + 1. **Dataset**
       1. 是 Dataframe API 的一个扩展，是 Spark 最新的数据抽象
       2. 用户友好的 API 风格，既具有类型安全检查也具有 Dataframe 的查询优化特性。
       3. Dataset 支持编解码器，当需要访问非堆上的数据时可以避免反序列化整个对象，提高了效率。

当我需要查询某个对象的某个字段时，不需要加载整个对象(文件)信息。只加载某一列。

* + - 1. 样例类被用来在 Dataset 中定义数据的结构信息，样例类中每个属性的名称直接映射到

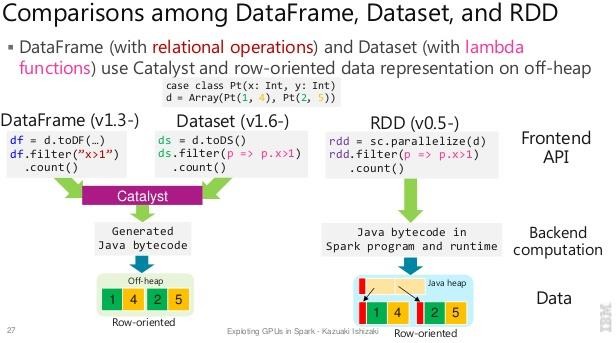
DataSet 中的字段名称。

* + - 1. Dataframe 是Dataset 的特列，DataFrame=Dataset[Row] ，所以可以通过 as 方法将Dataframe转换为 Dataset。Row 是一个类型，跟 Car、Person 这些的类型一样，所有的表结构信息我都用 Row 来表示。
      2. DataSet 是强类型的。比如可以有 Dataset[Car]，Dataset[Person].
      3. DataFrame 只是知道字段，但是不知道字段的类型，所以在执行这些操作的时候是没办法在编译的时候检查是否类型失败的，比如你可以对一个 String 进行减法操作，在执行的时候才报错，而 DataSet 不仅仅知道字段，而且知道字段类型，所以有更严格的错误检查。就跟 JSON 对象和类对象之间的类比。



RDD 让我们能够决定怎么做，而 DataFrame 和 DataSet 让我们决定做什么，控制的粒度不一

样。



#### 三者的共性

1、RDD、DataFrame、Dataset 全都是 spark 平台下的分布式弹性数据集，为处理超大型数据提供便利

2、三者都有惰性机制，在进行创建、转换，如 map 方法时，不会立即执行，只有在遇到 Action

如 foreach 时，三者才会开始遍历运算，极端情况下，如果代码里面有创建、转换，但是后面没有在 Action 中使用对应的结果，在执行时会被直接跳过。

val sparkconf = new SparkConf().setMaster("local").setAppName("test")

.set("spark.port.maxRetries","1000")

val spark = SparkSession.builder().config(sparkconf).getOrCreate() val rdd=spark.sparkContext.parallelize(Seq(("a", 1), ("b", 1),

("a", 1)))

// map 不运行

rdd.map{line=> println("运行") line.\_1

}

3、三者都会根据 spark 的内存情况自动缓存运算，这样即使数据量很大，也不用担心会内存

溢出

4、三者都有 partition 的概念

5、三者有许多共同的函数，如 filter，排序等

6、在对 DataFrame 和 Dataset 进行操作许多操作都需要这个包进行支持

import spark.implicits.\_

7、DataFrame 和 Dataset 均可使用模式匹配获取各个字段的值和类型

DataFrame:

testDF.map{

case Row(col1:String,col2:Int)=> println(col1);println(col2) col1

case \_=> ""

}

Dataset:

case class Coltest(col1:String,col2:Int)extends Serializable //定义字段名和类型

testDS.map{

case Coltest(col1:String,col2:Int)=> println(col1);println(col2)

col1 case \_=>

""

}

#### 三者的区别

RDD:

1、RDD 一般和 spark mlib 同时使用2、RDD 不支持 sparksql 操作DataFrame:

1、与 RDD 和 Dataset 不同，DataFrame 每一行的类型固定为 Row，只有通过解析才能获取

各个字段的值，如

testDF.foreach{ line =>

val col1=line.getAs[String]("col1") val col2=line.getAs[String]("col2")

}

每一列的值没法直接访问

2、DataFrame 与 Dataset 一般不与 spark ml 同时使用

3、DataFrame 与 Dataset 均支持 sparksql 的操作，比如 select，groupby 之类，还能注册临时表/视窗，进行 sql 语句操作，如

dataDF.createOrReplaceTempView("tmp")

spark.sql("select ROW,DATE from tmp where DATE is not null order by DATE").show(100,false)

4、DataFrame 与 Dataset 支持一些特别方便的保存方式，比如保存成 csv，可以带上表头，这样每一列的字段名一目了然

//保存

val saveoptions = Map("header" -> "true", "delimiter" -> "\t", "path"

-> "hdfs://master01:9000/test") datawDF.write.format("com.atguigu.spark.csv").mode(SaveMode.Overwrite

).options(saveoptions).save()

//读取

val options = Map("header" -> "true", "delimiter" -> "\t", "path" -> "hdfs://master01:9000/test")

val datarDF= spark.read.options(options).format("com.atguigu.spark.cs v").load()

利用这样的保存方式，可以方便的获得字段名和列的对应，而且分隔符（delimiter）可以自由指定。

Dataset:

Dataset 和DataFrame 拥有完全相同的成员函数，区别只是每一行的数据类型不同。

DataFrame 也可以叫 Dataset[Row],每一行的类型是 Row，不解析，每一行究竟有哪些字段， 各个字段又是什么类型都无从得知，只能用上面提到的 getAS 方法或者共性中的第七条提到的模式匹配拿出特定字段

而 Dataset 中，每一行是什么类型是不一定的，在自定义了 case class 之后可以很自由的获得每一行的信息

case class Coltest(col1:String,col2:Int)extends Serializable //定义字段名和类型

/\*\*

rdd ("a", 1)

("b", 1)

("a", 1)

\*\*/

val test: Dataset[Coltest]=rdd.map{line=> Coltest(line.\_1,line.\_2)

}.toDS

test.map{

line=>

println(line.col1) println(line.col2)

}

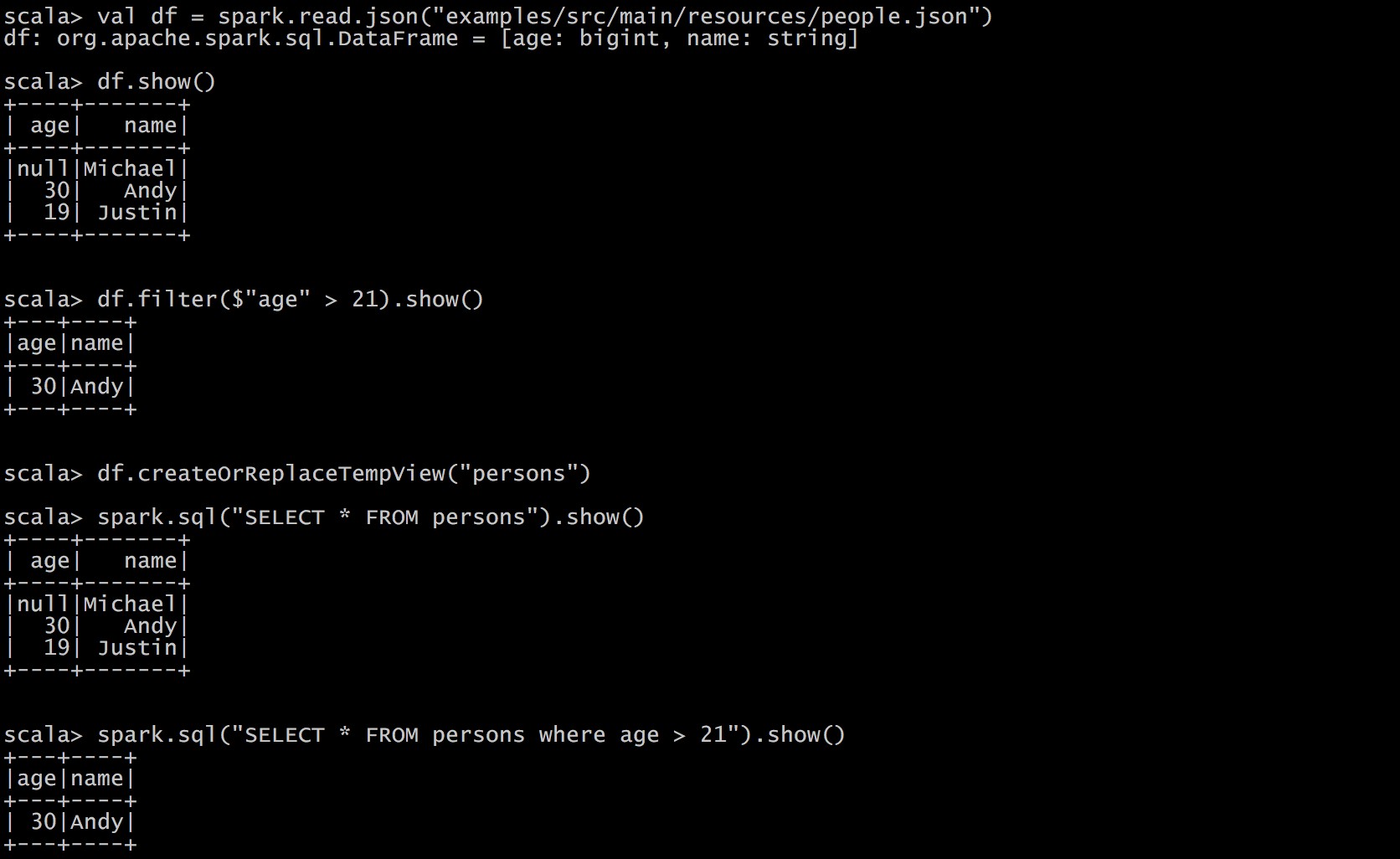
可以看出，Dataset 在需要访问列中的某个字段时是非常方便的，然而，如果要写一些适配性很强的函数时，如果使用Dataset，行的类型又不确定，可能是各种 case class，无法实现适配，这时候用 DataFrame 即 Dataset[Row]就能比较好的解决问题

第 **2** 章 执行 **SparkSQL** 查询

* 1. **命令行查询流程**

打开 Spark shell

例子：查询大于 30 岁的用户

创建如下JSON 文件，注意JSON 的格式：

{"name":"Michael"}

{"name":"Andy", "age":30}

{"name":"Justin", "age":19}

* 1. **IDEA 创建 SparkSQL 程序**

IDEA 中程序的打包和运行方式都和 SparkCore 类似，Maven 依赖中需要添加新的依赖项：

<dependency>

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

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

<version>2.2.0</version>

<scope>provided</scope>

</dependency>

程序如下：

package com.atguigu.sparksql

import org.apache.spark.sql.SparkSession

import org.apache.spark.{SparkConf, SparkContext} import org.slf4j.LoggerFactory

object HelloWorld {

def main(args: Array[String]) {

//创建 SparkConf()并设置 App 名称

val spark = SparkSession

.builder()

.appName("Spark SQL basic example")

.config("spark.some.config.option", "some-value")

.getOrCreate()

// For implicit conversions like converting RDDs to DataFrames import spark.implicits.\_

val df = spark.read.json("examples/src/main/resources/people.json")

// Displays the content of the DataFrame to stdout df.show()

df.filter($"age" > 21).show() df.createOrReplaceTempView("persons")

spark.sql("SELECT \* FROM persons where age > 21").show()

spark.stop()

}

}

第 **3** 章 **SparkSQL** 解析

* 1. **新的起始点 SparkSession**

在老的版本中，SparkSQL 提供两种 SQL 查询起始点，一个叫 SQLContext，用于 Spark 自己提供的 SQL 查询，一个叫 HiveContext，用于连接 Hive 的查询，SparkSession 是 Spark 最新的 SQL 查询起始点，实质上是 SQLContext 和 HiveContext 的组合，所以在 SQLContext 和 HiveContext 上可用的 API 在 SparkSession 上同样是可以使用的。SparkSession 内部封装了 sparkContext，所以计算实际上是由 sparkContext 完成的。

import org.apache.spark.sql.SparkSession

val spark = SparkSession

.builder()

.appName("Spark SQL basic example")

.config("spark.some.config.option", "some-value")

.getOrCreate()

// For implicit conversions like converting RDDs to DataFrames import spark.implicits.\_

SparkSession.builder 用于创建一个 SparkSession。

import spark.implicits.\_的引入是用于将 DataFrames 隐式转换成 RDD，使 df 能够使用 RDD 中的方法。

如果需要Hive 支持，则需要以下创建语句：

import org.apache.spark.sql.SparkSession

val spark = SparkSession

.builder()

.appName("Spark SQL basic example")

.config("spark.some.config.option", "some-value")

.enableHiveSupport()

.getOrCreate()

// For implicit conversions like converting RDDs to DataFrames import spark.implicits.\_

* 1. **创建 DataFrames**

在 Spark SQL 中 SparkSession 是创建 DataFrames 和执行 SQL 的入口，创建 DataFrames 有三种方式，一种是可以从一个存在的 RDD 进行转换，还可以从 Hive Table 进行查询返回，或者通过 Spark 的数据源进行创建。

从 Spark 数据源进行创建：

val df = spark.read.json("examples/src/main/resources/people.json")

// Displays the content of the DataFrame to stdout df.show()

// +- + +

// | age| name|

// +----+-------+

// |null|Michael|

// | 30| Andy|

// | 19| Justin|

// +- + +

从 RDD 进行转换：

/\*\* Michael, 29

Andy, 30

Justin, 19

\*\*/ scala>

val

peopleRdd

=

sc.textFile("examples/src/main/resources/people.txt") peopleRdd: org.apache.spark.rdd.RDD[String]

examples/src/main/resources/people.txt textFile at <console>:24

MapPartitionsRDD[18]

=

at

scala> val peopleDF3 = peopleRdd.map(\_.split(",")).map(paras =>

(paras(0),paras(1).trim().toInt)).toDF("name","age")

peopleDF3: org.apache.spark.sql.DataFrame = [name: string, age: int]

scala> peopleDF.show()

+ + +

| name|age|

+ + +

|Michael| 29|

| Andy| 30|

| Justin| 19|

+ + +

Hive 我们在数据源章节介绍

* 1. **DataFrame 常用操作**
     1. **DSL 风格语法**

// This import is needed to use the $-notation import spark.implicits.\_

// Print the schema in a tree format df.printSchema()

// root

// |-- age: long (nullable = true)

// |-- name: string (nullable = true)

// Select only the "name" column df.select("name").show()

// +- +

// | name|

// +-------+

// |Michael|

// | Andy|

// | Justin|

// +- +

// Select everybody, but increment the age by 1 df.select($"name", $"age" + 1).show()

// +- +- +

// | name|(age + 1)|

// +- +- +

// |Michael| null|

// | Andy| 31|

// | Justin| 20|

// +- +- +

// Select people older than 21 df.filter($"age" > 21).show()

// +- +- +

// |age|name|

// +- +- +

// | 30|Andy|

// +- +- +

// Count people by age df.groupBy("age").count().show()

// +- + -+

// | age|count|

// +----+-----+

// | 19| 1|

// |null| 1|

// | 30| 1|

// +----+-----+

* + 1. **SQL 风格语法**

// Register the DataFrame as a SQL temporary view df.createOrReplaceTempView("people")

val sqlDF = spark.sql("SELECT \* FROM people") sqlDF.show()

// +----+-------+

// | age| name|

// +----+-------+

// |null|Michael|

// | 30| Andy|

// | 19| Justin|

// +- + +

// Register the DataFrame as a global temporary view df.createGlobalTempView("people")

// Global temporary view is tied to a system preserved database

`global\_temp`

spark.sql("SELECT \* FROM global\_temp.people").show()

// +- + +

// | age| name|

// +----+-------+

// |null|Michael|

// | 30| Andy|

// | 19| Justin|

// +- + +

// Global temporary view is cross-session spark.newSession().sql("SELECT \* FROM global\_temp.people").show()

// +- + +

// | age| name|

// +----+-------+

// |null|Michael|

// | 30| Andy|

// | 19| Justin|

// +- + +

临时表是 Session 范围内的，Session 退出后，表就失效了。如果想应用范围内有效，可以使用全局表。注意使用全局表时需要全路径访问，如：global\_temp.people

* 1. **创建 DataSet**

Dataset 是具有强类型的数据集合，需要提供对应的类型信息。

// Note: Case classes in Scala 2.10 can support only up to 22 fields. To work around this limit,

// you can use custom classes that implement the Product interface case class Person(name: String, age: Long)

// Encoders are created for case classes

val caseClassDS = Seq(Person("Andy", 32)).toDS() caseClassDS.show()

// +- + +

// |name|age|

// +- + +

// |Andy| 32|

// +- + +

// Encoders for most common types are automatically provided by importing spark.implicits.\_

val primitiveDS = Seq(1, 2, 3).toDS()

primitiveDS.map(\_ + 1).collect() // Returns: Array(2, 3, 4)

// DataFrames can be converted to a Dataset by providing a class.

Mapping will be done by name

val path = "examples/src/main/resources/people.json" val peopleDS = spark.read.json(path).as[Person] peopleDS.show()

// +- + +

// | age| name|

// +----+-------+

// |null|Michael|

// | 30| Andy|

// | 19| Justin|

// +- + +

* 1. **Dataset 和 RDD 互操作**

Spark SQL 支持通过两种方式将存在的 RDD 转换为 Dataset，转换的过程中需要让 Dataset 获取 RDD 中的 Schema 信息，主要有两种方式，一种是通过反射来获取 RDD 中的 Schema 信息。这种方式适合于列名已知的情况下。第二种是通过编程接口的方式将 Schema 信息应用于 RDD， 这种方式可以处理那种在运行时才能知道列的方式。

* + 1. **通过反射获取 Scheam**

SparkSQL 能够自动将包含有 case 类的 RDD 转换成DataFrame，case 类定义了 table 的结构，

case 类属性通过反射变成了表的列名。Case 类可以包含诸如 Seqs 或者 Array 等复杂的结构。

// For implicit conversions from RDDs to DataFrames import spark.implicits.\_

// Create an RDD of Person objects from a text file, convert it to a Dataframe

val peopleDF = spark.sparkContext

.textFile("examples/src/main/resources/people.txt")

.map(\_.split(","))

.map(attributes => Person(attributes(0), attributes(1).trim.toInt))

.toDF()

// Register the DataFrame as a temporary view peopleDF.createOrReplaceTempView("people")

// SQL statements can be run by using the sql methods provided by Spark val teenagersDF = spark.sql("SELECT name, age FROM people WHERE age BETWEEN 13 AND 19")

// The columns of a row in the result can be accessed by field index ROW object

teenagersDF.map(teenager => "Name: " + teenager(0)).show()

// +- +

// | value|

// +- +

// |Name: Justin|

// +- +

// or by field name

teenagersDF.map(teenager => "Name: " + teenager.getAs[String]("name")).show()

// +- +

// | value|

// +- +

// |Name: Justin|

// +- +

// No pre-defined encoders for Dataset[Map[K,V]], define explicitly implicit val mapEncoder = org.apache.spark.sql.Encoders.kryo[Map[String, Any]]

// Primitive types and case classes can be also defined as

// implicit val stringIntMapEncoder: Encoder[Map[String, Any]] = ExpressionEncoder()

// row.getValuesMap[T] retrieves multiple columns at once into a Map[String, T]

teenagersDF.map(teenager => teenager.getValuesMap[Any](List("name", "age"))).collect()

// Array(Map("name" -> "Justin", "age" -> 19))

* + 1. **通过编程设置 Schema**

如果 case 类不能够提前定义，可以通过下面三个步骤定义一个DataFrame

创建一个多行结构的 RDD;

创建用 StructType 来表示的行结构信息。

通过 SparkSession 提供的 createDataFrame 方法来应用 Schema 。

import org.apache.spark.sql.types.\_

// Create an RDD val

peopleRDD

=

spark.sparkContext.textFile("examples/src/main/resources/people.txt")

// The schema is encoded in a string,应该是动态通过程序生成的val schemaString = "name age"

// Generate the schema based on the string of schema Array[StructFiled] val fields = schemaString.split(" ")

.map(fieldName => StructField(fieldName, StringType, nullable = true))

// val filed = schemaString.split(" ").map(filename=> filename match{ case "name"=> StructField(filename,StringType,nullable = true); case "age"=>StructField(filename, IntegerType,nullable = true)} )

val schema = StructType(fields)

// Convert records of the RDD (people) to Rows import org.apache.spark.sql.\_

val rowRDD = peopleRDD

.map(\_.split(","))

.map(attributes => Row(attributes(0), attributes(1).trim))

// Apply the schema to the RDD

val peopleDF = spark.createDataFrame(rowRDD, schema)

// Creates a temporary view using the DataFrame peopleDF.createOrReplaceTempView("people")

// SQL can be run over a temporary view created using DataFrames val results = spark.sql("SELECT name FROM people")

// The results of SQL queries are DataFrames and support all the normal RDD operations

// The columns of a row in the result can be accessed by field index or by field name

results.map(attributes => "Name: " + attributes(0)).show()

// +-- +

// | value|

// +- +

// |Name: Michael|

// | Name: Andy|

// | Name: Justin|

// +- +

* 1. **类型之间的转换总结**

RDD、DataFrame、Dataset 三者有许多共性，有各自适用的场景常常需要在三者之间转换

DataFrame/Dataset 转 RDD： 这个转换很简单

val rdd1=testDF.rdd val rdd2=testDS.rdd

RDD 转 DataFrame：

import spark.implicits.\_

val testDF = rdd.map {line=> (line.\_1,line.\_2)

}.toDF("col1","col2")

一般用元组把一行的数据写在一起，然后在 toDF 中指定字段名

RDD 转 Dataset：

import spark.implicits.\_

case class Coltest(col1:String,col2:Int)extends Serializable //定义字段名和类型

val testDS = rdd.map {line=> Coltest(line.\_1,line.\_2)

}.toDS

可以注意到，定义每一行的类型（case class）时，已经给出了字段名和类型，后面只要往 case class 里面添加值即可

Dataset 转DataFrame：

这个也很简单，因为只是把 case class 封装成 Row

import spark.implicits.\_ val testDF = testDS.toDF

DataFrame 转 Dataset：

import spark.implicits.\_

case class Coltest(col1:String,col2:Int)extends Serializable //定义字段名和类型

val testDS = testDF.as[Coltest]

这种方法就是在给出每一列的类型后，使用 as 方法，转成 Dataset，这在数据类型是 DataFrame

又需要针对各个字段处理时极为方便。

在使用一些特殊的操作时，一定要加上 import spark.implicits.\_ 不然 toDF、toDS 无法使用

* 1. **用户自定义函数**

通过 spark.udf 功能用户可以自定义函数。

* + 1. **用户自定义 UDF 函数**

scala>

val

df

=

spark.read.json("examples/src/main/resources/people.json")

df: org.apache.spark.sql.DataFrame = [age: bigint, name: string]

scala> df.show()

+ + +

| age| name|

+ + +

|null|Michael|

| 30| Andy|

| 19| Justin|

+ + +

scala> spark.udf.register("addName", (x:String)=> "Name:"+x) res5: org.apache.spark.sql.expressions.UserDefinedFunction

=

UserDefinedFunction(<function1>,StringType,Some(List(StringType))) scala> df.createOrReplaceTempView("people")

scala> spark.sql("Select addName(name), age from people").show()

+ -+ +

|UDF:addName(name)| age|

+ -+ +

| Name:Michael|null|

| Name:Andy| 30|

| Name:Justin| 19|

+ -+ +

* + 1. **用户自定义聚合函数**

强类型的Dataset 和弱类型的 DataFrame 都提供了相关的聚合函数，如 count()，countDistinct()， avg()，max()，min()。除此之外，用户可以设定自己的自定义聚合函数。

弱类型用户自定义聚合函数：通过继承 UserDefinedAggregateFunction 来实现用户自定义聚

合函数。下面展示一个求平均工资的自定义聚合函数。

import org.apache.spark.sql.expressions.MutableAggregationBuffer import org.apache.spark.sql.expressions.UserDefinedAggregateFunction import org.apache.spark.sql.types.\_

import org.apache.spark.sql.Row

import org.apache.spark.sql.SparkSession

object MyAverage extends UserDefinedAggregateFunction {

// 聚合函数输入参数的数据类型

def inputSchema: StructType = StructType(StructField("inputColumn", LongType) :: Nil)

// 聚合缓冲区中值得数据类型

def bufferSchema: StructType = {

StructType(StructField("sum", LongType) :: StructField("count", LongType) :: Nil)

}

// 返回值的数据类型

def dataType: DataType = DoubleType

// 对于相同的输入是否一直返回相同的输出。

def deterministic: Boolean = true

// 初始化

def initialize(buffer: MutableAggregationBuffer): Unit = {

// 存工资的总额

buffer(0) = 0L

// 存工资的个数

buffer(1) = 0L

}

// 相同 Execute 间的数据合并。

def update(buffer: MutableAggregationBuffer, input: Row): Unit = { if (!input.isNullAt(0)) {

buffer(0) = buffer.getLong(0) + input.getLong(0) buffer(1) = buffer.getLong(1) + 1

}

}

// 不同 Execute 间的数据合并

def merge(buffer1: MutableAggregationBuffer, buffer2: Row): Unit = { buffer1(0) = buffer1.getLong(0) + buffer2.getLong(0)

buffer1(1) = buffer1.getLong(1) + buffer2.getLong(1)

}

// 计算最终结果

def evaluate(buffer: Row): Double = buffer.getLong(0).toDouble / buffer.getLong(1)

}

// 注册函数

spark.udf.register("myAverage", MyAverage)

val df = spark.read.json("examples/src/main/resources/employees.json") df.createOrReplaceTempView("employees")

df.show()

// +- +- -+

// | name|salary|

// +- +- -+

// |Michael| 3000|

// | Andy| 4500|

// | Justin| 3500|

// | Berta| 4000|

// +- +- -+

val result = spark.sql("SELECT myAverage(salary) as average\_salary FROM employees")

result.show()

// +-

// |average\_salary|

// +-

-+

-+

// | 3750.0|

// +- -+

强类型用户自定义聚合函数：通过继承 Aggregator 来实现强类型自定义聚合函数，同样是求

平均工资

import org.apache.spark.sql.expressions.Aggregator import org.apache.spark.sql.Encoder

import org.apache.spark.sql.Encoders import org.apache.spark.sql.SparkSession

// 既然是强类型，可能有 case 类

case class Employee(name: String, salary: Long) case class Average(var sum: Long, var count: Long)

object MyAverage extends Aggregator[Employee, Average, Double] {

// 定义一个数据结构，保存工资总数和工资总个数，初始都为 0 def zero: Average = Average(0L, 0L)

// Combine two values to produce a new value. For performance, the function may modify `buffer`

// and return it instead of constructing a new object

def reduce(buffer: Average, employee: Employee): Average = { buffer.sum += employee.salary

buffer.count += 1 buffer

}

// 聚合不同 execute 的结果

def merge(b1: Average, b2: Average): Average = { b1.sum += b2.sum

b1.count += b2.count b1

}

// 计算输出

def finish(reduction: Average): Double = reduction.sum.toDouble / reduction.count

// 设定之间值类型的编码器，要转换成 case 类

// Encoders.product 是进行 scala 元组和 case 类转换的编码器

def bufferEncoder: Encoder[Average] = Encoders.product

// 设定最终输出值的编码器

def outputEncoder: Encoder[Double] = Encoders.scalaDouble

}

import spark.implicits.\_

val ds = spark.read.json("examples/src/main/resources/employees.json").as[Empl oyee]

ds.show()

// +- +- -+

// | name|salary|

// +- +- -+

// |Michael| 3000|

// | Andy| 4500|

// | Justin| 3500|

// | Berta| 4000|

// +- +- -+

// Convert the function to a `TypedColumn` and give it a name val averageSalary = MyAverage.toColumn.name("average\_salary") val result = ds.select(averageSalary)

result.show()

// +-

// |average\_salary|

// +-

-+

// |

-+ 3750.0|

// +- -+

第 **4** 章 **SparkSQL** 数据源

* 1. **通用加载/保存方法**
     1. **手动指定选项**

Spark SQL 的 DataFrame 接口支持多种数据源的操作。一个 DataFrame 可以进行 RDDs 方式的操作，也可以被注册为临时表。把DataFrame 注册为临时表之后，就可以对该 DataFrame 执行SQL 查询。

Spark SQL 的默认数据源为 Parquet 格式。数据源为 Parquet 文件时，Spark SQL 可以方便的执行所有的操作。修改配置项 spark.sql.sources.default，可修改默认数据源格式。

val df = spark.read.load("examples/src/main/resources/users.parquet") df.select("name", "favorite\_color").write.save("namesAndFavColors.parquet")

当数据源格式不是 parquet 格式文件时，需要手动指定数据源的格式。数据源格式需要指定全名（例如：org.apache.spark.sql.parquet），如果数据源格式为内置格式，则只需要指定简称定json, parquet, jdbc, orc, libsvm, csv, text 来指定数据的格式。

可以通过 SparkSession 提供的 read.load 方法用于通用加载数据，使用 write 和 save 保存数

据。

val

peopleDF

=

spark.read.format("json").load("examples/src/main/resources/people.js on") peopleDF.write.format("parquet").save("hdfs://hadoop102:9000/namesAnd Ages.parquet")

除此之外，可以直接运行 SQL 在文件上：

val

sqlDF

=

spark.sql("SELECT

\*

FROM

parquet.`hdfs://hadoop102:9000/namesAndAges.parquet`") sqlDF.show()

scala> val peopleDF =

spark.read.format("json").load("examples/src/main/resources/people.js on")

peopleDF: org.apache.spark.sql.DataFrame = [age: bigint, name: string]

scala> peopleDF.write.format("parquet").save("hdfs://hadoop102:9000/namesAnd Ages.parquet")

scala> peopleDF.show()

+ + +

| age| name|

+ + +

|null|Michael|

| 30| Andy|

| 19| Justin|

+ + +

scala> val sqlDF = spark.sql("SELECT \* FROM parquet.`hdfs://master01:9000/namesAndAges.parquet`")

17/09/05 04:21:11 WARN ObjectStore: Failed to get database parquet, returning NoSuchObjectException

sqlDF: org.apache.spark.sql.DataFrame = [age: bigint, name: string]

scala> sqlDF.show()

+ + +

| age| name|

+ + +

|null|Michael|

| 30| Andy|

| 19| Justin|

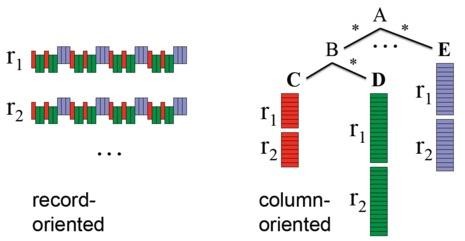
+ + +

* + 1. **文件保存选项**

可以采用 SaveMode 执行存储操作，SaveMode 定义了对数据的处理模式。需要注意的是，这些保存模式不使用任何锁定，不是原子操作。此外，当使用 Overwrite 方式执行时，在输出新数据之前原数据就已经被删除。SaveMode 详细介绍如下表：

|  |  |  |
| --- | --- | --- |
| Scala/Java | Any Language | Meaning |
| SaveMode.ErrorIfExists(default) | "error"(default) | 如果文件存在，则报错 |
| SaveMode.Append | "append" | 追加 |
| SaveMode.Overwrite | "overwrite" | 覆写 |
| SaveMode.Ignore | "ignore" | 数据存在，则忽略 |

* 1. **Parquet 文件**

Parquet 是一种流行的列式存储格式，可以高效地存储具有嵌套字段的记录。

* + 1. **Parquet 读写**

Parquet 格式经常在 Hadoop 生态圈中被使用，它也支持 Spark SQL 的全部数据类型。Spark SQL 提供了直接读取和存储 Parquet 格式文件的方法。

// Encoders for most common types are automatically provided by importing spark.implicits.\_

import spark.implicits.\_

val

peopleDF

=

spark.read.json("examples/src/main/resources/people.json")

// DataFrames can be saved as Parquet files, maintaining the schema information peopleDF.write.parquet("hdfs://master01:9000/people.parquet")

// Read in the parquet file created above

// Parquet files are self-describing so the schema is preserved

// The result of loading a Parquet file is also a DataFrame

val parquetFileDF = spark.read.parquet("hdfs://master01:9000/people.parquet")

// Parquet files can also be used to create a temporary view and then used in SQL statements parquetFileDF.createOrReplaceTempView("parquetFile")

val namesDF = spark.sql("SELECT name FROM parquetFile WHERE age BETWEEN 13 AND 19")

namesDF.map(attributes => "Name: " + attributes(0)).show()

// +- +

// | value|

// +- +

// |Name: Justin|

// +- +

* + 1. **解析分区信息**

对表进行分区是对数据进行优化的方式之一。在分区的表内，数据通过分区列将数据存储在不同的目录下（相当于 hive 的桶）

每天产生的文件。可以通过分区。存放在不同的目录中。如果一天就出现了一个超大文件（128M） 8G（）

。Parquet 数据源现在能够自动发现并解析分区信息。例如，对人口数据进行分区存储，分区列为 gender 和 country，使用下面的目录结构：

**path**

└── **to**

└── **table**

├── **gender=male**

│ ├── **...**

│ │

│ ├── **country=US**

│ │ └── **data.parquet**

│ ├── **country=CN**

│ │ └── **data.parquet**

│ └── **...**

└── **gender=female**

├── **...**

│

├── **country=US**

│ └── **data.parquet**

├── **country=CN**

│ └── **data.parquet**

└── **...**

通过传递 path/to/table 给 SQLContext.read.parquet 或 SQLContext.read.load，Spark SQL 将自动解析分区信息。返回的 DataFrame 的 Schema 如下：

**root**

**|-- name: string (nullable = true)**

**|-- age: long (nullable = true)**

**|-- gender: string (nullable = true)**

**|-- country: string (nullable = true)**

需要注意的是，数据的分区列的数据类型是自动解析的。当前，支持数值类型和字符串类型。自动解析分区类型的参数为：spark.sql.sources.partitionColumnTypeInference.enabled，默认值为 true。如果想关闭该功能，直接将该参数设置为 disabled。此时，分区列数据格式将被默认设置为 string 类型，不再进行类型解析。

* + 1. **Schema 合并**

像 ProtocolBuffer、Avro 和 Thrift 那样，Parquet 也支持 Schema evolution（Schema 演变）。用户可以先定义一个简单的 Schema，然后逐渐的向 Schema 中增加列描述。通过这种方式，用户可以获取多个有不同 Schema 但相互兼容的 Parquet 文件。现在 Parquet 数据源能自动检测这种情况，并合并这些文件的 schemas。

因为 Schema 合并是一个高消耗的操作，在大多数情况下并不需要，所以 Spark SQL 从

1.5.0 开始默认关闭了该功能。可以通过下面两种方式开启该功能：

当数据源为 Parquet 文件时，将数据源选项 mergeSchema 设置为 true

设置全局 SQL 选项 spark.sql.parquet.mergeSchema 为 true

示例如下：

// sqlContext from the previous example is used in this example.

// This is used to implicitly convert an RDD to a DataFrame. import spark.implicits.\_

// Create a simple DataFrame, stored into a partition directory

val df1 = sc.makeRDD(1 to 5).map(i => (i, i \* 2)).toDF("single", "double") df1.write.parquet("hdfs://master01:9000/data/test\_table/key=1")

// Create another DataFrame in a new partition directory,

// adding a new column and dropping an existing column

val df2 = sc.makeRDD(6 to 10).map(i => (i, i \* 3)).toDF("single", "triple") df2.write.parquet("hdfs://master01:9000/data/test\_table/key=2")

// Read the partitioned table val df3 =

spark.read.option("mergeSchema",

"true").parquet("hdfs://master01:9000/data/test\_table") df3.printSchema()

// The final schema consists of all 3 columns in the Parquet files together

// with the partitioning column appeared in the partition directory paths.

// root

// |-- single: int (nullable = true)

// |-- double: int (nullable = true)

// |-- triple: int (nullable = true)

// |-- key : int (nullable = true)

* 1. **Hive 数据库**

Apache Hive 是 Hadoop 上的 SQL 引擎，Spark SQL 编译时可以包含 Hive 支持，也可以不包含。包含 Hive 支持的 Spark SQL 可以支持 Hive 表访问、UDF(用户自定义函数)以及 Hive 查询语言(HiveQL/HQL)等。需要强调的 一点是，如果要在 Spark SQL 中包含 Hive 的库，并不需要事先安装 Hive。一般来说，最好还是在编译 Spark SQL 时引入 Hive 支持，这样就可以使用这些特性了。如果你下载的是二进制版本的 Spark，它应该已经在编译时添加了 Hive 支持。

若要把 Spark SQL 连接到一个部署好的 Hive 上，你必须把 hive-site.xml 复制到 Spark 的配置文件目录中($SPARK\_HOME/conf)。即使没有部署好 Hive，Spark SQL 也可以运行。 需要注意的是，如果你没有部署好 Hive，Spark SQL 会在当前的工作目录中创建出自己的 Hive 元数据仓库，叫作 metastore\_db。此外，如果你尝试使用 HiveQL 中的 CREATE TABLE (并非 CREATE EXTERNAL TABLE)语句来创建表，这些表会被放在你默认的文件系统中的 /user/hive/warehouse 目录中(如果你的 classpath 中有配好的 hdfs-site.xml，默认的文件系统就是 HDFS，否则就是本地文件系统)。

import java.io.File

import org.apache.spark.sql.Row

import org.apache.spark.sql.SparkSession case class Record(key: Int, value: String)

// warehouseLocation points to the default location for managed databases and tables

val warehouseLocation = new File("spark-warehouse").getAbsolutePath

val spark = SparkSession

.builder()

.appName("Spark Hive Example")

.config("spark.sql.warehouse.dir", warehouseLocation)

.enableHiveSupport()

.getOrCreate()

import spark.implicits.\_ import spark.sql

sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING)")

sql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt' INTO TABLE src")

// Queries are expressed in HiveQL sql("SELECT \* FROM src").show()

// +- +- +

// |key| value|

// +- +- +

// |238|val\_238|

// | 86| val\_86|

// |311|val\_311|

// ...

// Aggregation queries are also supported. sql("SELECT COUNT(\*) FROM src").show()

// +- +

// |count(1)|

// +- +

// | 500 |

// +--------+

// The results of SQL queries are themselves DataFrames and support all normal functions.

val sqlDF = sql("SELECT key, value FROM src WHERE key < 10 ORDER BY key")

// The items in DataFrames are of type Row, which allows you to access each column by ordinal.

val stringsDS = sqlDF.map {

case Row(key: Int, value: String) => s"Key: $key, Value: $value"

}

stringsDS.show()

// +- +

// | value|

// +- +

// |Key: 0, Value: val\_0|

// |Key: 0, Value: val\_0|

// |Key: 0, Value: val\_0|

// ...

// You can also use DataFrames to create temporary views within a SparkSession.

val recordsDF = spark.createDataFrame((1 to 100).map(i => Record(i, s"val\_$i")))

recordsDF.createOrReplaceTempView("records")

// Queries can then join DataFrame data with data stored in Hive. sql("SELECT \* FROM records r JOIN src s ON r.key = s.key").show()

// +- +- -+ -+ +

// |key| value|key| value|

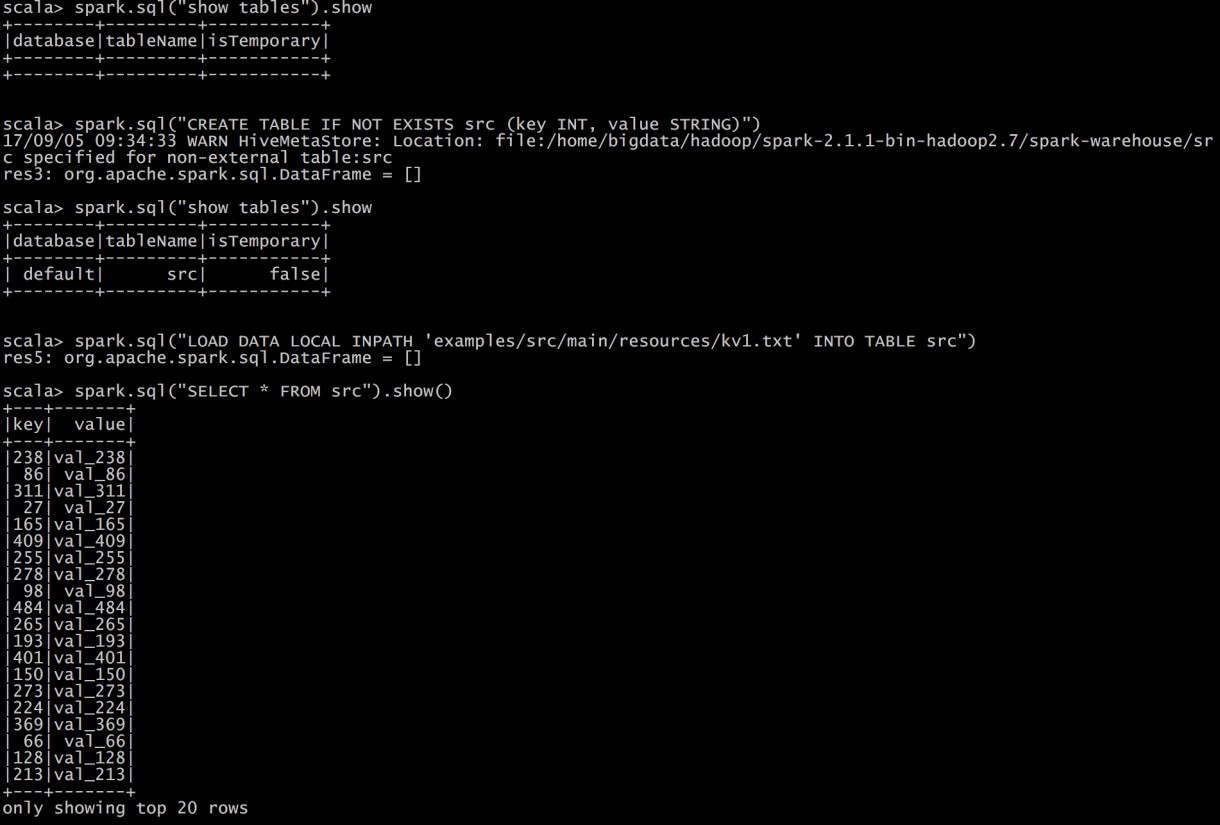
// +- +- -+ -+ +

// | 2| val\_2| 2| val\_2|

// | 4| val\_4| 4| val\_4|

// | 5| val\_5| 5| val\_5|

* + 1. **内嵌 Hive 应用**

如果要使用内嵌的Hive，什么都不用做，直接用就可以了。 --conf spark.sql.warehouse.dir=

注意：如果你使用的是内部的Hive，在Spark2.0 之后，spark.sql.warehouse.dir 用于指定数据仓库的地址，如果你需要是用HDFS 作为路径，那么需要将core- site.xml 和 hdfs-site.xml 加入到Spark conf 目录，否则只会创建 master 节点上的warehouse 目录，查询时会出现文件找不到的问题，这是需要向使用HDFS，则需要将 metastore 删除，重启集群。

* + 1. **外部 Hive 应用**

如果想连接外部已经部署好的Hive，需要通过以下几个步骤。

* + - 1. 将 Hive 中的 hive-site.xml 拷贝或者软连接到 Spark 安装目录下的 conf 目录下。
      2. 打开 spark shell，注意带上访问 Hive 元数据库的 JDBC 客户端

$ bin/spark-shell --master spark://hadoop102:7077 --jars mysql- connector-java-5.1.27-bin.jar

* 1. **JSON 数据集**

Spark SQL 能够自动推测 JSON 数据集的结构，并将它加载为一个 Dataset[Row]. 可以通过SparkSession.read.json()去加载一个 Dataset[String]或者一个 JSON 文件.注意，这个 JSON 文件不是一个传统的JSON 文件，每一行都得是一个JSON 串。

{"name":"Michael"}

{"name":"Andy", "age":30}

{"name":"Justin", "age":19}

// Primitive types (Int, String, etc) and Product types (case classes) encoders are

// supported by importing this when creating a Dataset. import spark.implicits.\_

// A JSON dataset is pointed to by path.

// The path can be either a single text file or a directory storing text files

val path = "examples/src/main/resources/people.json" val peopleDF = spark.read.json(path)

// The inferred schema can be visualized using the printSchema() method peopleDF.printSchema()

// root

// |-- age: long (nullable = true)

// |-- name: string (nullable = true)

// Creates a temporary view using the DataFrame peopleDF.createOrReplaceTempView("people")

// SQL statements can be run by using the sql methods provided by spark val teenagerNamesDF = spark.sql("SELECT name FROM people WHERE age BETWEEN 13 AND 19")

teenagerNamesDF.show()

// +- -+

// | name|

// +- -+

// |J*ustin*|

// +- -+

// Alternatively, a DataFrame can be created for a JSON dataset represented by

// a Dataset[String] storing one JSON object per string val otherPeopleDataset = spark.createDataset(

"""{"name":"Yin","address":{"city":"Columbus","state":"Ohio"}}""" :: Nil)

val otherPeople = spark.read.json(otherPeopleDataset) otherPeople.show()

// +- +- +

// | address|name|

// +- +- +

// |[Columbus,Ohio]| Yin|

// +- +- +

* 1. **JDBC**

Spark SQL 可以通过JDBC 从关系型数据库中读取数据的方式创建 DataFrame，通过对DataFrame 一系列的计算后，还可以将数据再写回关系型数据库中。

注意，需要将相关的数据库驱动放到 spark 的类路径下。

$ bin/spark-shell --master spark://hadoop102:7077 --jars mysql- connector-java-5.1.27-bin.jar

// Note: JDBC loading and saving can be achieved via either the load/save or jdbc methods

// Loading data from a JDBC source

val jdbcDF = spark.read.format("jdbc").option("url", "jdbc:mysql://master01:3306/rdd").option("dbtable", " rddtable").option("user", "root").option("password", "hive").load()

val connectionProperties = new Properties() connectionProperties.put("user", "root") connectionProperties.put("password", "hive") val jdbcDF2 = spark.read

.jdbc("jdbc:mysql://master01:3306/rdd", "rddtable", connectionProperties)

// Saving data to a JDBC source jdbcDF.write

.format("jdbc")

.option("url", "jdbc:mysql://master01:3306/rdd")

.option("dbtable", "rddtable2")

.option("user", "root")

.option("password", "hive")

.save()

jdbcDF2.write

.jdbc("jdbc:mysql://master01:3306/mysql", "db", connectionProperties)

// Specifying create table column data types on write jdbcDF.write

.option("createTableColumnTypes", "name CHAR(64), comments VARCHAR(1024)")

.jdbc("jdbc:mysql://master01:3306/mysql", "db", connectionProperties)

第 **5** 章 **JDBC/ODBC** 服务器

Spark SQL 也提供 JDBC 连接支持，这对于让商业智能(BI)工具连接到 Spark 集群上以 及在多用户间共享一个集群的场景都非常有用。JDBC 服务器作为一个独立的 Spark 驱动 器程序运行，可以在多用户之间共享。任意一个客户端都可以在内存中缓存数据表，对表 进行查询。集群的资源以及缓存数据都在所有用户之间共享。

Spark SQL 的 JDBC 服务器与 Hive 中的 HiveServer2 相一致。由于使用了 Thrift 通信协议， 它也被称为“Thrift server”。

服务器可以通过 Spark 目录中的 sbin/start-thriftserver.sh 启动。这个 脚本接受的参数选项大多与 spark-submit 相同。默认情况下，服务器会在 localhost:10000 上进行监听，我们可以通过环境变量(HIVE\_SERVER2\_THRIFT\_PORT 和 HIVE\_SERVER2\_THRIFT\_BIND\_HOST) 修改这

些设置，也可以通过 Hive 配置选项(hive. server2.thrift.port 和 hive.server2.thrift.bind.host)来修改。你也可以通过命令行参 数--hiveconf property=value 来设置 Hive 选项。

./sbin/start-thriftserver.sh \

--hiveconf hive.server2.thrift.port=<listening-port> \

--hiveconf hive.server2.thrift.bind.host=<listening-host> \

--master <master-uri>

...

./bin/beeline

beeline> !connect jdbc:hive2://hadoop102:10000

在 Beeline 客户端中，你可以使用标准的 HiveQL 命令来创建、列举以及查询数据表。

[bigdata@master01 thriftserver.sh

spark-2.1.1-bin-hadoop2.7]$

./sbin/start-

starting org.apache.spark.sql.hive.thriftserver.HiveThriftServer2, logging to /home/bigdata/hadoop/spark-2.1.1-bin-hadoop2.7/logs/spark- bigdata-org.apache.spark.sql.hive.thriftserver.HiveThriftServer2-1- master01.out

[bigdata@master01 spark-2.1.1-bin-hadoop2.7]$ ./bin/beeline Beeline version 1.2.1.spark2 by Apache Hive

beeline> !connect jdbc:hive2://master01:10000 Connecting to jdbc:hive2://master01:10000

Enter username for jdbc:hive2://master01:10000: bigdata Enter password for jdbc:hive2://master01:10000: \*\*\*\*\*\*\*

log4j:WARN No appenders could (org.apache.hive.jdbc.Utils).

be found for

logger

log4j:WARN Please initialize the log4j system properly.

log4j:WARN See <http://logging.apache.org/log4j/1.2/faq.html#noconfig> for more info.

Connected to: Spark SQL (version 2.1.1) Driver: Hive JDBC (version 1.2.1.spark2)

Transaction isolation: TRANSACTION\_REPEATABLE\_READ 0: jdbc:hive2://master01:10000> show tables;

+ + + - + +

| database | tableName | isTemporary |

+ + + - + +

| default | src | false |

+ + + - + +

1 row selected (0.726 seconds) 0: jdbc:hive2://master01:10000>

第 **6** 章 运行 **Spark SQL CLI**

Spark SQL CLI 可以很方便的在本地运行 Hive 元数据服务以及从命令行执行查询任务。需要注意的是，Spark SQL CLI 不能与 Thrift JDBC 服务交互。

在 Spark 目录下执行如下命令启动 Spark SQL CLI：

./bin/spark-sql

配置 Hive 需要替换 conf/ 下的 hive-site.xml 。

第 **7** 章 **Spark SQL** 的运行原理（了解）

* 1. **Spark SQL 运行架构**

Spark SQL 对 SQL 语句的处理和关系型数据库类似，即词法/语法解析、绑定、优化、执行。Spark SQL 会先将 SQL 语句解析成一棵树，然后使用规则(Rule)对 Tree 进行绑定、优化等处理过程。Spark SQL 由 Core、Catalyst、Hive、Hive-ThriftServer 四部分构成：

Core: 负责处理数据的输入和输出，如获取数据，查询结果输出成 DataFrame 等

Catalyst: 负责处理整个查询过程，包括解析、绑定、优化等

Hive: 负责对 Hive 数据进行处理

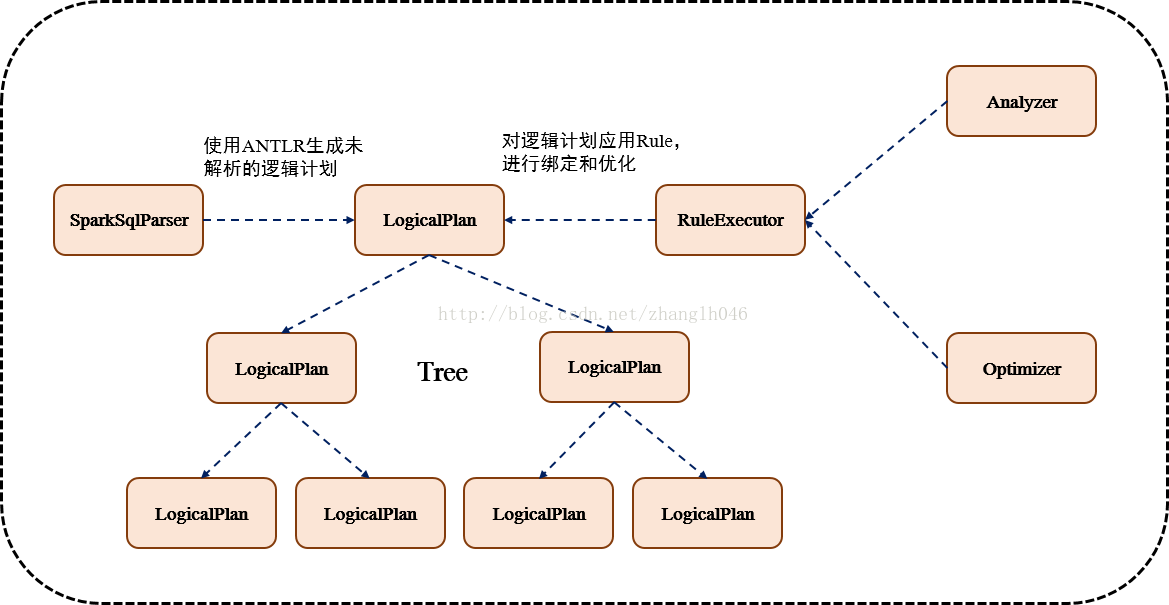
Hive-ThriftServer: 主要用于对 hive 的访问

* + 1. **TreeNode**

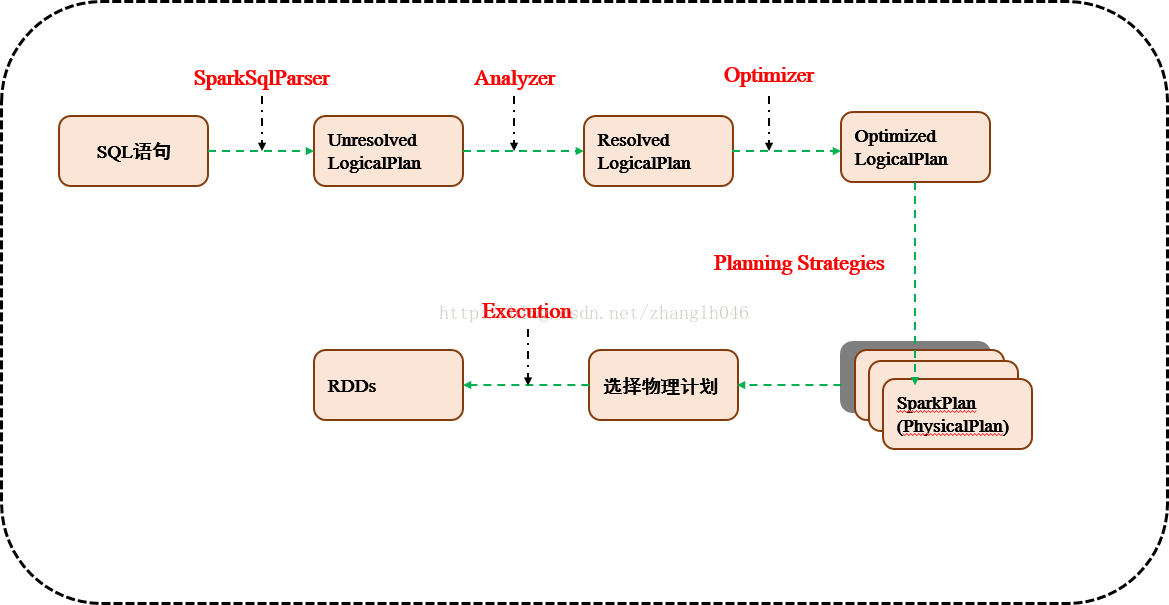
逻辑计划、表达式等都可以用 tree 来表示，它只是在内存中维护，并不会进行磁盘的持久化， 分析器和优化器对树的修改只是替换已有节点。

TreeNode 有2 个直接子类，QueryPlan 和Expression。QueryPlam 下又有LogicalPlan和SparkPlan. Expression 是表达式体系，不需要执行引擎计算而是可以直接处理或者计算的节点，包括投影操作，操作符运算等

* + 1. **Rule & RuleExecutor**

Rule 就是指对逻辑计划要应用的规则，以到达绑定和优化。他的实现类就是 RuleExecutor。优化器和分析器都需要继承 RuleExecutor。每一个子类中都会定义 Batch、Once、FixPoint. 其中每一个 Batch 代表着一套规则，Once 表示对树进行一次操作，FixPoint 表示对树进行多次的迭代操作。RuleExecutor 内部提供一个 Seq[Batch]属性，里面定义的是 RuleExecutor 的处理逻辑，具体的处理逻辑由具体的 Rule 子类实现。

整个流程架构图：



* 1. **Spark SQL 运行原理**
     1. **使用 SessionCatalog 保存元数据**

在解析SQL 语句之前，会创建 SparkSession，或者如果是 2.0 之前的版本初始化SQLContext， SparkSession 只是封装了 SparkContext 和 SQLContext 的创建而已。会把元数据保存在SessionCatalog 中，涉及到表名，字段名称和字段类型。创建临时表或者视图，其实就会往SessionCatalog 注册

* + 1. **解析 SQL,使用 ANTLR 生成未绑定的逻辑计划**

当调用 SparkSession 的 sql 或者 SQLContext 的 sql 方法，我们以 2.0 为准，就会使用SparkSqlParser 进行解析 SQL. 使用的 ANTLR 进行词法解析和语法解析。它分为 2 个步骤来生成Unresolved LogicalPlan：

# 词法分析：Lexical Analysis，负责将 token 分组成符号类

# 构建一个分析树或者语法树 AST

* + 1. **使用分析器 Analyzer 绑定逻辑计划**

在该阶段，Analyzer 会使用 Analyzer Rules，并结合 SessionCatalog，对未绑定的逻辑计划进行解析，生成已绑定的逻辑计划。

* + 1. **使用优化器 Optimizer 优化逻辑计划**

优化器也是会定义一套 Rules，利用这些 Rule 对逻辑计划和 Exepression 进行迭代处理，从而

使得树的节点进行和并和优化

* + 1. **使用 SparkPlanner 生成物理计划**

SparkSpanner 使用 Planning Strategies，对优化后的逻辑计划进行转换，生成可以执行的物理计划 SparkPlan.

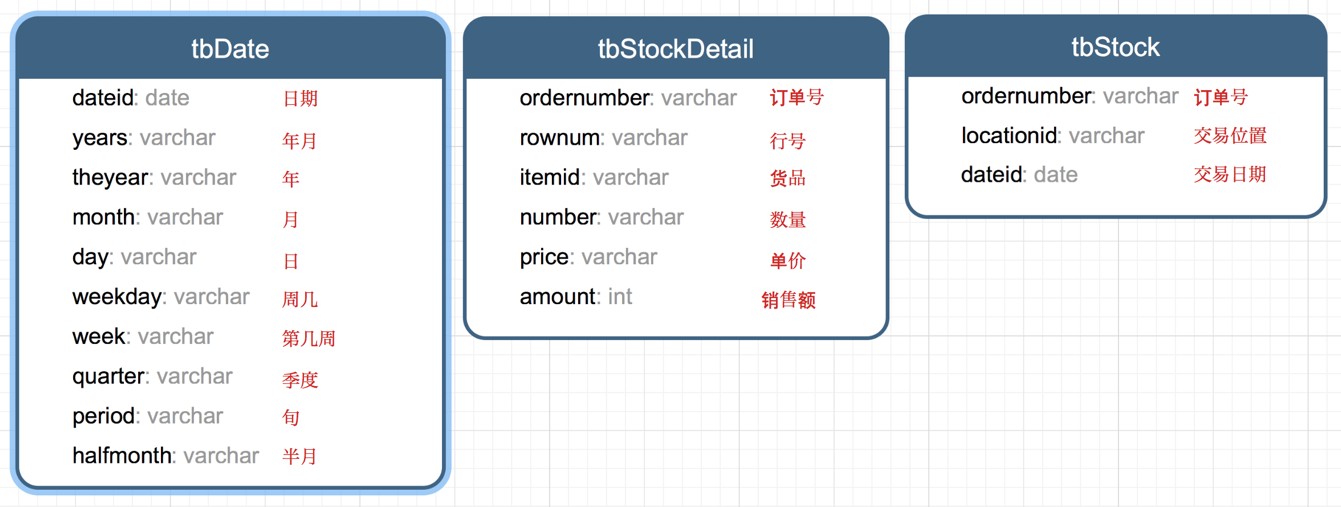
* + 1. **使用 QueryExecution 执行物理计划**

此时调用 SparkPlan 的 execute 方法，底层其实已经再触发JOB 了，然后返回 RDD

第 **8** 章 **Spark SQL** 实战

* 1. **数据说明**

数据集是货品交易数据集。



每个订单可能包含多个货品，每个订单可以产生多次交易，不同的货品有不同的单价。

* 1. **加载数据**

tbStock：

scala> case class tbStock(ordernumber:String,locationid:String,dateid:String) extends Serializable

defined class tbStock

scala> val tbStockRdd = spark.sparkContext.textFile("tbStock.txt")

tbStockRdd: org.apache.spark.rdd.RDD[String] = tbStock.txt MapPartitionsRDD[1] at textFile at

<console>:23

scala> val tbStockDS = tbStockRdd.map(\_.split(",")).map(attr=>tbStock(attr(0),attr(1),attr(2))).toDS

tbStockDS: org.apache.spark.sql.Dataset[tbStock] = [ordernumber: string, locationid: string 1 more field]

scala> tbStockDS.show()

+ + + +

| ordernumber|locationid| dataid|

+ + + +

|BYSL00000893| ZHAO|2007-8-23|

|BYSL00000897| ZHAO|2007-8-24|

|BYSL00000898| ZHAO|2007-8-25|

|BYSL00000899| ZHAO|2007-8-26|

|BYSL00000900| ZHAO|2007-8-26|

|BYSL00000901| ZHAO|2007-8-27|

|BYSL00000902| ZHAO|2007-8-27|

|BYSL00000904| ZHAO|2007-8-28|

|BYSL00000905| ZHAO|2007-8-28|

|BYSL00000906| ZHAO|2007-8-28|

|BYSL00000907| ZHAO|2007-8-29|

|BYSL00000908| ZHAO|2007-8-30|

|BYSL00000909| ZHAO| 2007-9-1|

|BYSL00000910| ZHAO| 2007-9-1|

|BYSL00000911| ZHAO|2007-8-31|

|BYSL00000912| ZHAO| 2007-9-2|

|BYSL00000913| ZHAO| 2007-9-3|

|BYSL00000914| ZHAO| 2007-9-3|

|BYSL00000915| ZHAO| 2007-9-4|

|BYSL00000916| ZHAO| 2007-9-4|

+ + + + only showing top 20 rows

tbStockDetail:

scala> case class tbStockDetail(ordernumber:String, rownum:Int, itemid:String, number:Int, price:Double, amount:Double) extends Serializable

defined class tbStockDetail

scala> val tbStockDetailRdd = spark.sparkContext.textFile("tbStockDetail.txt") tbStockDetailRdd: org.apache.spark.rdd.RDD[String] = tbStockDetail.txt MapPartitionsRDD[13] at textFile at <console>:23

scala> val tbStockDetailDS = tbStockDetailRdd.map(\_.split(",")).map(attr=> tbStockDetail(attr(0),attr(1).trim().toInt,attr(2),attr(3).trim().toInt,attr(4).trim().toDoubl e, attr(5).trim().toDouble)).toDS

tbStockDetailDS: org.apache.spark.sql.Dataset[tbStockDetail] = [ordernumber: string, rownum: int 4 more fields]

scala> tbStockDetailDS.show()

+ + + + + + +

| ordernumber|rownum| itemid|number|price|amount|

+ + + + + + +

|BYSL00000893| 0|FS527258160501| 1|268.0|-268.0|

|BYSL00000893| 1|FS527258169701| 1|268.0| 268.0|

|BYSL00000893| 2|FS527230163001| 1|198.0| 198.0|

|BYSL00000893| 3|24627209125406| 1|298.0| 298.0|

|BYSL00000893| 4|K9527220210202| 1|120.0| 120.0|

|BYSL00000893| 5|01527291670102| 1|268.0| 268.0|

|BYSL00000893| 6|QY527271800242| 1|158.0| 158.0|

|BYSL00000893| 7|ST040000010000| 8| 0.0| 0.0|

|BYSL00000897| 0|04527200711305| 1|198.0| 198.0|

|BYSL00000897| 1|MY627234650201| 1|120.0| 120.0|

|BYSL00000897| 2|01227111791001| 1|249.0| 249.0|

|BYSL00000897| 3|MY627234610402| 1|120.0| 120.0|

|BYSL00000897| 4|01527282681202| 1|268.0| 268.0|

|BYSL00000897| 5|84126182820102| 1|158.0| 158.0|

|BYSL00000897| 6|K9127105010402| 1|239.0| 239.0|

|BYSL00000897| 7|QY127175210405| 1|199.0| 199.0|

|BYSL00000897| 8|24127151630206| 1|299.0| 299.0|

|BYSL00000897| 9|G1126101350002| 1|158.0| 158.0|

|BYSL00000897| 10|FS527258160501| 1|198.0| 198.0|

|BYSL00000897| 11|ST040000010000| 13| 0.0| 0.0|

+ + + + + + + only showing top 20 rows

tbDate:

scala> case class tbDate(dateid:String, years:Int, theyear:Int, month:Int, day:Int, weekday:Int, week:Int, quarter:Int, period:Int, halfmonth:Int) extends Serializable defined class tbDate

scala> val tbDateRdd = spark.sparkContext.textFile("tbDate.txt")

tbDateRdd: org.apache.spark.rdd.RDD[String] = tbDate.txt MapPartitionsRDD[20] at textFile at

<console>:23

scala> val tbDateDS = tbDateRdd.map(\_.split(",")).map(attr=> tbDate(attr(0),attr(1).trim().toInt, attr(2).trim().toInt,attr(3).trim().toInt,

attr(4).trim().toInt, attr(5).trim().toInt, attr(6).trim().toInt, attr(7).trim().toInt, attr(8).trim().toInt, attr(9).trim().toInt)).toDS

tbDateDS: org.apache.spark.sql.Dataset[tbDate] = [dateid: string, years: int 8 more

fields]

scala> tbDateDS.show()

+ + + + + + + + + + +

| dateid| years|theyear|month|day|weekday|week|quarter|period|halfmonth|

+ + + + + + + + + + +

| 2003-1-1|200301| 2003| 1| 1| 3| 1| 1| 1| 1|

| 2003-1-2|200301| 2003| 1| 2| 4| 1| 1| 1| 1|

| 2003-1-3|200301| 2003| 1| 3| 5| 1| 1| 1| 1|

| 2003-1-4|200301| 2003| 1| 4| 6| 1| 1| 1| 1|

| 2003-1-5|200301| 2003| 1| 5| 7| 1| 1| 1| 1|

| 2003-1-6|200301| 2003| 1| 6| 1| 2| 1| 1| 1|

| 2003-1-7|200301| 2003| 1| 7| 2| 2| 1| 1| 1|

| 2003-1-8|200301| 2003| 1| 8| 3| 2| 1| 1| 1|

| 2003-1-9|200301| 2003| 1| 9| 4| 2| 1| 1| 1|

|2003-1-10|200301| 2003| 1| 10| 5| 2| 1| 1| 1|

|2003-1-11|200301| 2003| 1| 11| 6| 2| 1| 2| 1|

|2003-1-12|200301| 2003| 1| 12| 7| 2| 1| 2| 1|

|2003-1-13|200301| 2003| 1| 13| 1| 3| 1| 2| 1|

|2003-1-14|200301| 2003| 1| 14| 2| 3| 1| 2| 1|

|2003-1-15|200301| 2003| 1| 15| 3| 3| 1| 2| 1|

|2003-1-16|200301| 2003| 1| 16| 4| 3| 1| 2| 2|

|2003-1-17|200301| 2003| 1| 17| 5| 3| 1| 2| 2|

|2003-1-18|200301| 2003| 1| 18| 6| 3| 1| 2| 2|

|2003-1-19|200301| 2003| 1| 19| 7| 3| 1| 2| 2|

|2003-1-20|200301| 2003| 1| 20| 1| 4| 1| 2| 2|

+ + + + + + + + + + + only showing top 20 rows

注册表：

scala> tbStockDS.createOrReplaceTempView("tbStock")

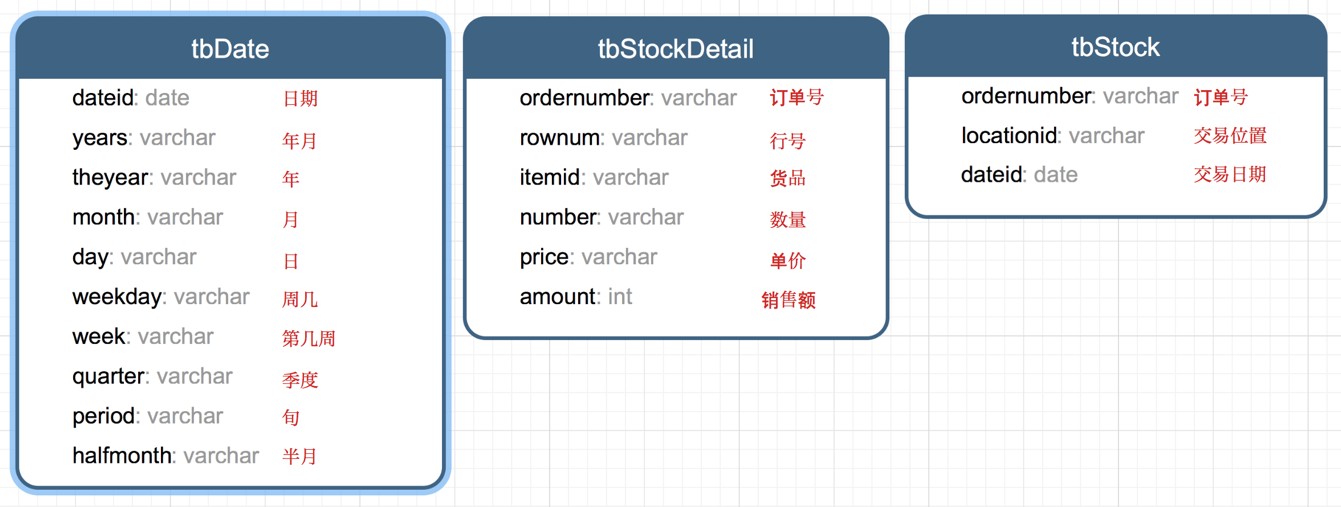
scala> tbDateDS.createOrReplaceTempView("tbDate")

scala> tbStockDetailDS.createOrReplaceTempView("tbStockDetail")

* 1. **计算所有订单中每年的销售单数**、**销售总额**

统计所有订单中每年的销售单数、销售总额

三个表连接后以 count(distinct a.ordernumber)计销售单数，sum(b.amount)计销售总额



SELECT c.theyear, COUNT(DISTINCT a.ordernumber), SUM(b.amount) FROM tbStock a

JOIN tbStockDetail b ON a.ordernumber = b.ordernumber

JOIN tbDate c ON a.dateid = c.dateid GROUP BY c.theyear

ORDER BY c.theyear

spark.sql("SELECT c.theyear, COUNT(DISTINCT a.ordernumber), SUM(b.amount) FROM tbStock a JOIN tbStockDetail b ON a.ordernumber = b.ordernumber JOIN tbDate c ON a.dateid = c.dateid GROUP BY

c.theyear ORDER BY c.theyear").show

结果如下：

+ + + +

|theyear|count(DISTINCT ordernumber)| sum(amount)|

+ + + +

| 2004| 1094| 3268115.499199999|

| 2005| 3828|1.3257564149999991E7|

| 2006| 3772|1.3680982900000006E7|

| 2007| 4885|1.6719354559999993E7|

| 2008| 4861| 1.467429530000001E7|

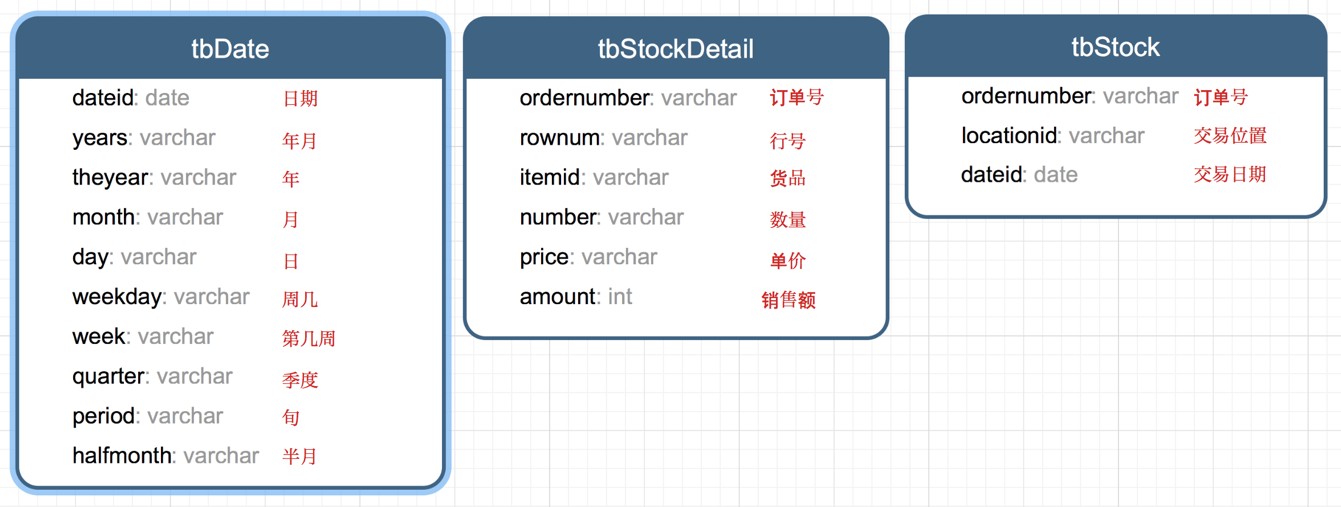
| 2009| 2619| 6323697.189999999|

| 2010| 94| 210949.65999999997|

+ + + +

* 1. **计算所有订单每年最大金额订单的销售额**

目标：统计每年最大金额订单的销售额:



1. 统计每年，每个订单一共有多少销售额

SELECT a.dateid, a.ordernumber, SUM(b.amount) AS SumOfAmount FROM tbStock a

JOIN tbStockDetail b ON a.ordernumber = b.ordernumber

GROUP BY a.dateid, a.ordernumber

spark.sql("SELECT a.dateid, a.ordernumber, SUM(b.amount) AS SumOfAmount FROM tbStock a JOIN

tbStockDetail b ON a.ordernumber = b.ordernumber GROUP BY a.dateid, a.ordernumber").show

结果如下：

+ + + +

| dateid| ordernumber| SumOfAmount|

+ + + +

| 2008-4-9|BYSL00001175| 350.0|

| 2008-5-12|BYSL00001214| 592.0|

| 2008-7-29|BYSL00011545| 2064.0|

| 2008-9-5|DGSL00012056| 1782.0|

| 2008-12-1|DGSL00013189| 318.0|

|2008-12-18|DGSL00013374| 963.0|

| 2009-8-9|DGSL00015223| 4655.0|

| 2009-10-5|DGSL00015585| 3445.0|

| 2010-1-14|DGSL00016374| 2934.0|

| 2006-9-24|GCSL00000673|3556.1000000000004|

| 2007-1-26|GCSL00000826| 9375.199999999999|

| 2007-5-24|GCSL00001020| 6171.300000000002|

| 2008-1-8|GCSL00001217| 7601.6|

| 2008-9-16|GCSL00012204| 2018.0|

| 2006-7-27|GHSL00000603| 2835.6|

|2006-11-15|GHSL00000741| 3951.94|

| 2007-6-6|GHSL00001149| 0.0|

| 2008-4-18|GHSL00001631| 12.0|

| 2008-7-15|GHSL00011367| 578.0|

| 2009-5-8|GHSL00014637| 1797.6|

+ + + +

1. 以上一步查询结果为基础表，和表 tbDate 使用 dateid join，求出每年最大金额订单的销售额

SELECT theyear, MAX(c.SumOfAmount) AS SumOfAmount

FROM (SELECT a.dateid, a.ordernumber, SUM(b.amount) AS SumOfAmount

FROM tbStock a

JOIN tbStockDetail b ON a.ordernumber = b.ordernumber

GROUP BY a.dateid, a.ordernumber

) c

JOIN tbDate d ON c.dateid = d.dateid

GROUP BY theyear

ORDER BY theyear DESC

spark.sql("SELECT theyear, MAX(c.SumOfAmount) AS SumOfAmount FROM (SELECT a.dateid, a.ordernumber, SUM(b.amount) AS SumOfAmount FROM tbStock a JOIN tbStockDetail b ON a.ordernumber = b.ordernumber GROUP BY a.dateid, a.ordernumber ) c JOIN tbDate d ON c.dateid =

d.dateid GROUP BY theyear ORDER BY theyear DESC").show

结果如下：

+-------+------------------+

|theyear| SumOfAmount|

+ + +

| 2010|13065.280000000002|

| 2009|25813.200000000008|

| 2008| 55828.0|

| 2007| 159126.0|

| 2006| 36124.0|

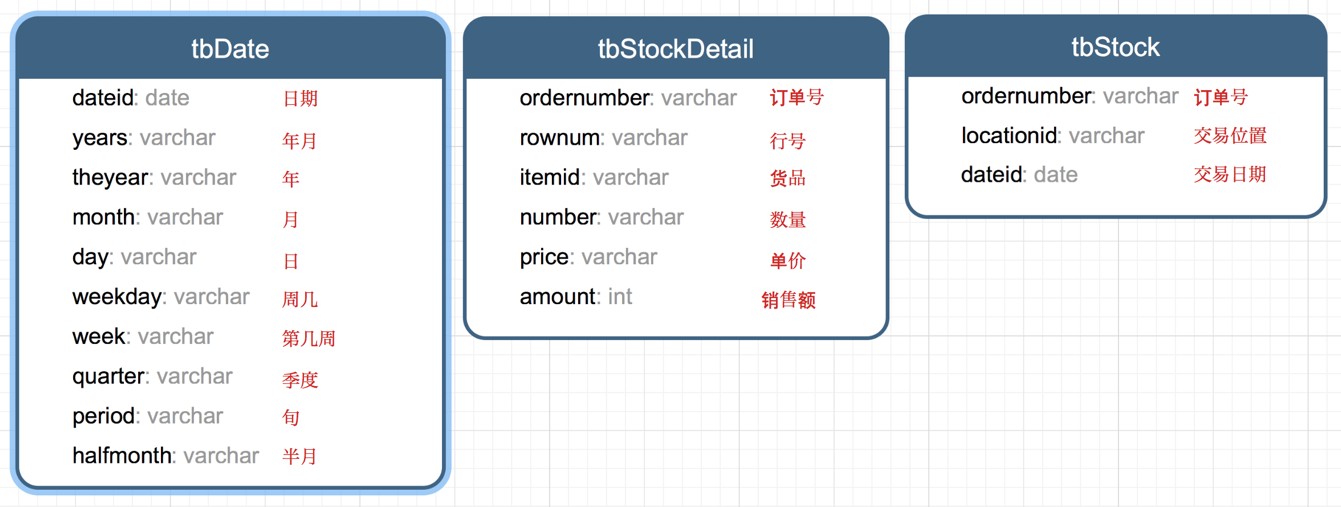
| 2005|38186.399999999994|

| 2004| 23656.79999999997|

+ + +

* 1. **计算所有订单中每年最畅销货品**

目标：统计每年最畅销货品（哪个货品销售额amount 在当年最高，哪个就是最畅销货品）



第一步、求出每年每个货品的销售额

SELECT c.theyear, b.itemid, SUM(b.amount) AS SumOfAmount FROM tbStock a

JOIN tbStockDetail b ON a.ordernumber = b.ordernumber

JOIN tbDate c ON a.dateid = c.dateid GROUP BY c.theyear, b.itemid

spark.sql("SELECT c.theyear, b.itemid, SUM(b.amount) AS SumOfAmount FROM tbStock a JOIN

tbStockDetail b ON a.ordernumber = b.ordernumber JOIN tbDate c ON a.dateid = c.dateid GROUP BY c.theyear, b.itemid").show

结果如下：

+-------+--------------+------------------+

|theyear| itemid| SumOfAmount|

+ + + +

| 2004|43824480810202| 4474.72|

| 2006|YA214325360101| 556.0|

| 2006|BT624202120102| 360.0|

| 2007|AK215371910101|24603.639999999992|

| 2008|AK216169120201|29144.199999999997|

| 2008|YL526228310106|16073.099999999999|

| 2009|KM529221590106| 5124.800000000001|

| 2004|HT224181030201|2898.6000000000004|

| 2004|SG224308320206| 7307.06|

| 2007|04426485470201|14468.800000000001|

| 2007|84326389100102| 9134.11|

| 2007|B4426438020201| 19884.2|

| 2008|YL427437320101|12331.799999999997|

| 2008|MH215303070101| 8827.0|

| 2009|YL629228280106| 12698.4|

| 2009|BL529298020602| 2415.8|

| 2009|F5127363019006| 614.0|

| 2005|24425428180101| 34890.74|

| 2007|YA214127270101| 240.0|

| 2007|MY127134830105| 11099.92|

+ + + +

第二步、在第一步的基础上，统计每年单个货品中的最大金额

SELECT d.theyear, MAX(d.SumOfAmount) AS MaxOfAmount

FROM (SELECT c.theyear, b.itemid, SUM(b.amount) AS SumOfAmount

FROM tbStock a

JOIN tbStockDetail b ON a.ordernumber = b.ordernumber

JOIN tbDate c ON a.dateid = c.dateid

GROUP BY c.theyear, b.itemid

) d

GROUP BY d.theyear

spark.sql("SELECT d.theyear, MAX(d.SumOfAmount) AS MaxOfAmount FROM (SELECT c.theyear, b.itemid, SUM(b.amount) AS SumOfAmount FROM tbStock a JOIN tbStockDetail b ON a.ordernumber = b.ordernumber JOIN tbDate c ON a.dateid = c.dateid GROUP BY c.theyear, b.itemid ) d GROUP BY

d.theyear").show

结果如下：

+-------+------------------+

|theyear| MaxOfAmount|

+ + +

| 2007| 70225.1|

| 2006| 113720.6|

| 2004|53401.759999999995|

| 2009| 30029.2|

| 2005|56627.329999999994|

| 2010| 4494.0|

| 2008| 98003.60000000003|

+ + +

第三步、用最大销售额和统计好的每个货品的销售额join，以及用年 join，集合得到最畅销货品那一行信息

SELECT DISTINCT e.theyear, e.itemid, f.MaxOfAmount

FROM (SELECT c.theyear, b.itemid, SUM(b.amount) AS SumOfAmount

FROM tbStock a

JOIN tbStockDetail b ON a.ordernumber = b.ordernumber

JOIN tbDate c ON a.dateid = c.dateid

GROUP BY c.theyear, b.itemid

) e

JOIN (SELECT d.theyear, MAX(d.SumOfAmount) AS MaxOfAmount

ORDER BY e.theyear

|  |  |  |
| --- | --- | --- |
|  |  | FROM (SELECT c.theyear, b.itemid, SUM(b.amount) AS SumOfAmount |
|  |  | FROM tbStock a |
|  |  | JOIN tbStockDetail b ON a.ordernumber = b.ordernumber |
|  |  | JOIN tbDate c ON a.dateid = c.dateid |
|  |  | GROUP BY c.theyear, b.itemid |
|  |  | ) d |
|  |  | GROUP BY d.theyear |
|  |  | ) f ON e.theyear = f.theyear |
|  |  | AND e.SumOfAmount = f.MaxOfAmount |

spark.sql("SELECT DISTINCT e.theyear, e.itemid, f.maxofamount FROM (SELECT c.theyear, b.itemid, SUM(b.amount) AS sumofamount FROM tbStock a JOIN tbStockDetail b ON a.ordernumber = b.ordernumber JOIN tbDate c ON a.dateid = c.dateid GROUP BY c.theyear, b.itemid ) e JOIN (SELECT d.theyear, MAX(d.sumofamount) AS maxofamount FROM (SELECT c.theyear, b.itemid, SUM(b.amount) AS sumofamount FROM tbStock a JOIN tbStockDetail b ON a.ordernumber = b.ordernumber JOIN tbDate c ON a.dateid = c.dateid GROUP BY c.theyear, b.itemid ) d GROUP BY d.theyear ) f ON e.theyear = f.theyear AND e.sumofamount = f.maxofamount ORDER BY

e.theyear").show

结果如下：

+-------+--------------+------------------+

|theyear| itemid| maxofamount|

+ + + +

| 2004|JY424420810101|53401.759999999995|

| 2005|24124118880102|56627.329999999994|

| 2006|JY425468460101| 113720.6|

| 2007|JY425468460101| 70225.1|

| 2008|E2628204040101| 98003.60000000003|

| 2009|YL327439080102| 30029.2|

| 2010|SQ429425090101| 4494.0|

+ + + +

**SparkStreaming 应用解析**

**教案**

**目 录**

[第 1 章 Spark Streaming 概述 3](#_bookmark0)

* 1. [什么是 Spark Streaming 3](#_bookmark1)
  2. [为什么要学习 Spark Streaming 4](#_bookmark2)
  3. [Spark 与 Storm 的对比 5](#_bookmark3)

[第 2 章 运行 Spark Streaming 6](#_bookmark4)

[2.1 IDEA 编写程序 6](#_bookmark5)

[第 3 章 架构与抽象 7](#_bookmark6)

[第 4 章 Spark Streaming 解析 9](#_bookmark7)

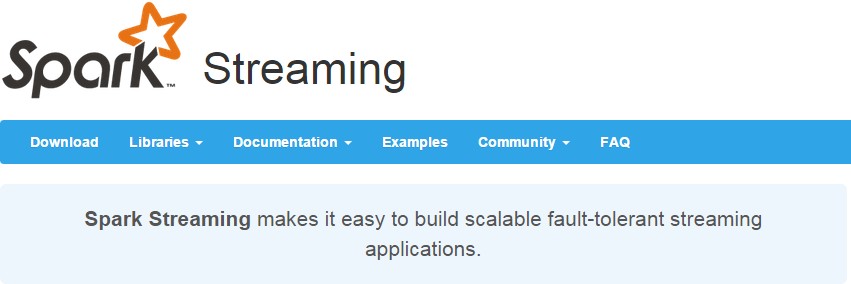
* 1. [初始化 StreamingContext 9](#_bookmark8)
  2. [什么是 DStreams 10](#_bookmark9)
  3. [DStreams 输入 11](#_bookmark10)
     1. [基本数据源 11](#_bookmark11)
     2. [高级数据源 19](#_bookmark12)
  4. [DStreams 转换 33](#_bookmark13)
     1. [无状态转化操作 36](#_bookmark14)
     2. [有状态转化操作 37](#_bookmark15)
     3. [重要操作 46](#_bookmark16)
  5. [DStreams 输出 47](#_bookmark17)
  6. [累加器和广播变量 48](#_bookmark18)
  7. [DataFrame ans SQL Operations 50](#_bookmark19)
  8. [Caching / Persistence 51](#_bookmark20)
  9. [7x24 不间断运行 51](#_bookmark21)
     1. [检查点机制 51](#_bookmark22)
     2. [WAL 预写日志 53](#_bookmark23)
     3. [背压机制 55](#_bookmark24)
     4. [驱动器程序容错 57](#_bookmark25)
     5. [工作节点容错 58](#_bookmark26)
     6. [接收器容错 58](#_bookmark27)
     7. [处理保证 59](#_bookmark28)
  10. [性能考量 60](#_bookmark29)

[第 5 章 高级解析 60](#_bookmark30)

* 1. [DStreamGraph 对象解析 60](#_bookmark31)
  2. [ReceiverTracker 与数据导入 66](#_bookmark32)
  3. [动态生成 JOB 75](#_bookmark33)
  4. [job 的提交与执行 81](#_bookmark34)
  5. [Block 的生成与存储 82](#_bookmark35)

# 第1章 Spark Streaming 概述

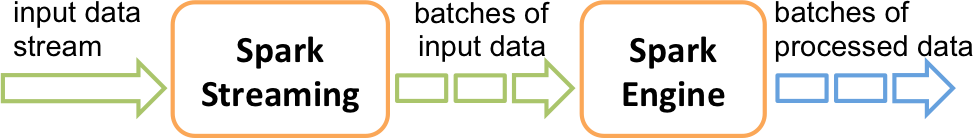
## 什么是 Spark Streaming



Spark Streaming 类似于 Apache Storm，（ flink）用于流式数据的处理。根据其官方文档介绍， Spark Streaming 有高吞吐量和容错能力强等特点。Spark Streaming 支持的数据输入源很多，例如：Kafka、Flume、Twitter、ZeroMQ 和简单的 TCP 套接字等等。数据输入后可以用 Spark 的高度抽象原语如： map、reduce、join、window 等进行运算。而结果也能保存在很多地方，如 HDFS，数据库等。另外 Spark Streaming 也能和 MLlib（机器学习）以及 Graphx 完美融合。



和 Spark 基于 RDD 的概念很相似，Spark Streaming 使用离散化流(discretized stream)作为抽象表示，叫作 DStream。DStream 是随时间推移而收到的数据的序列。在内部，每个时间区间收到的数据都作为 RDD 存在，而 DStream 是由这些 RDD 所组成的序列(因此 得名“离散化”)。

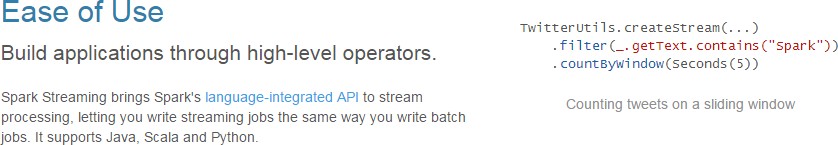


DStream 可以从各种输入源创建，比如 Flume、Kafka 或者 HDFS。创建出来的 DStream 支持两种操作，一种是转化操作(transformation)，会生成一个新的 DStream，另一种是输出操作(output operation)，可以把数据写入外部系统中。DStream 提供了许多与 RDD 所支持的操作相类似的操作支持，还增加了与时间相关的新操作，比如滑动窗口。

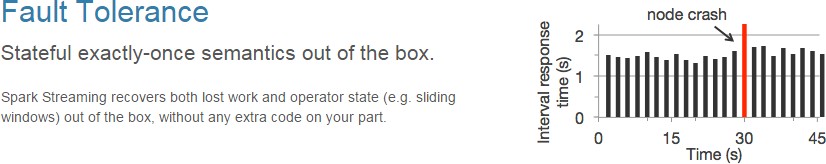
row\_number() over() 开窗。多个 RDD 进行计算。

## 为什么要学习 Spark Streaming

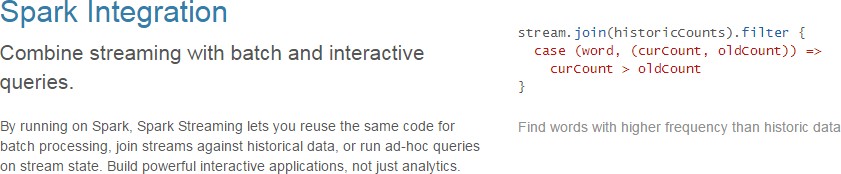
1. 易用



1. 容错



1. 易整合到Spark 体系



如果一个项目除了实时计算，还包括了离线批处理、交互式查询等业务功能，而且实时计算中，可能还会牵扯到高延迟批处理、交互式查询等功能，应首选 Spark 生态，用 Spark Core 开发离线批处理，用 Spark SQL 开发交互式查询，用 Spark Streaming 开发实时计算，三者可以无缝整合，给系统提供非常高的可扩展性。

## Spark 与 Storm 的对比

|  |  |
| --- | --- |
| **Spark** | **Storm** |
| 54fcc92668e64 | 54fcc9176d069 |
| 开发语言：Scala | 开发语言：Clojure |
| 编程模型：DStream | 编程模型：Spout/Bolt |

|  |  |
| --- | --- |
| 52e5f207e3d5b | 20160538_WNgf |

# 第2章 运行 Spark Streaming

2.1 **IDEA 编写程序**

Pom.xml 加入以下依赖：

<**dependency**>

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

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

<**version**>${spark.version}</**version**>

<**scope**>provided</**scope**>

</**dependency**>

**package** com.atguigu.streaming

**import** org.apache.spark.SparkConf

**import** org.apache.spark.streaming.{Seconds, StreamingContext}

*/\*\**

*\* Created by wuyufei on 06/09/2017.*

*\*/*

**object** WorldCount {

**def** main(args: Array[String]) {

**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(**"master01"**, 9999)

*// Split each line into words*

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

*//import org.apache.spark.streaming.StreamingContext.\_ // not necessary since Spark 1.3*

*// 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*

}

}

按照 Spark Core 中的方式进行打包，并将程序上传到 Spark 机器上。并运行：

bin/spark-submit --class com.atguigu.streaming.WorldCount ~/wordcount-jar-with- dependencies.jar

通过 Netcat 发送数据：

# TERMINAL 1:

# Running Netcat #yum install nc

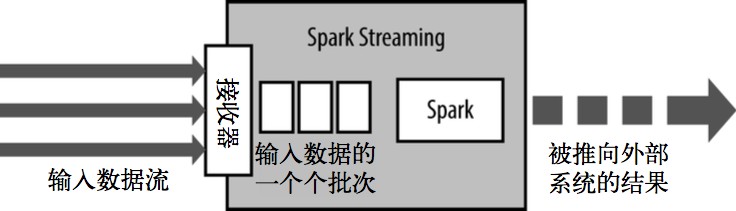
$ nc -lk 9999

hello world

如果程序运行时， log 日志太多， 可以将 spark conf 目录下的 log4j 文件里面的日志级别改成 WARN。

# 第3章 架构与抽象

Spark Streaming 使用“微批次”的架构，把流式计算当作一系列连续的小规

模批处理来对待。Spark Streaming 从各种输入源中读取数据，并把数据分组为小的批次。新的批次按均匀的时间间隔创建出来。在每个时间区间开始的时候， 一个新的批次就创建出来，在该区间内收到的数据都会被添加到这个批次中。在时间区间结束时，批次停止增长。时间区间的大小是由批次间隔这个参数决定的。批次间隔一般设在 500 毫秒到几秒之间，由应用开发者配置。每个输入批次都形成一个 RDD，以 Spark 作业的方式处理并生成其他的 RDD。 处理的结果可以以批处理的方式传给外部系统。高层次的架构如图

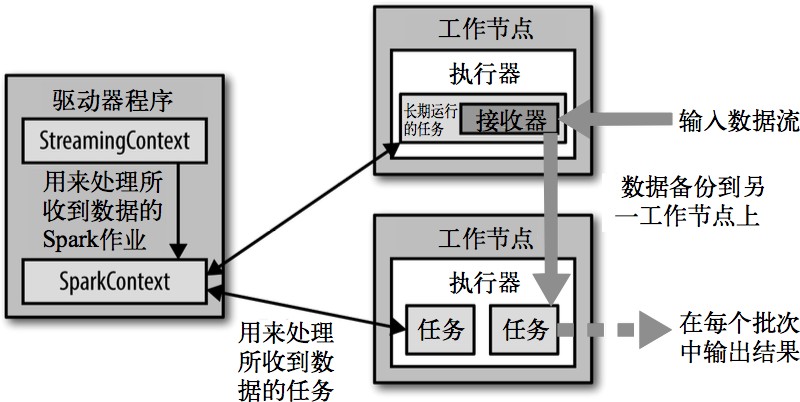
Spark Streaming 的编程抽象是离散化流， 也就是 DStream。它是一个

RDD 序列，每个 RDD 代表数据流中一个时间片内的数据。



Spark Streaming 在 Spark 的驱动器程序—工作节点的结构的执行过程如下图所示。Spark Streaming 为每个输入源启动对 应的接收器。接收器以任务的形式运行在应用的执行器进程中， 从输入源收集数据并保存为 RDD。它们收集到输入数据后会把数据复制到另一个执行器进程来保障容错性(默 认行

为)。数据保存在执行器进程的内存中， 和缓存 RDD 的方式一样。驱动器程序中的 StreamingContext 会周期性地运行 Spark 作业来处理这些数据，把数据与之前时间区间中的 RDD 进行整合。



# 第4章 Spark Streaming 解析

## 初始化 StreamingContext

import org.apache.spark.\_

import org.apache.spark.streaming.\_

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

// 可以通过 ssc.sparkContext 来访问 SparkContext

// 或者通过已经存在的 SparkContext 来创建 StreamingContext

import org.apache.spark.streaming.\_

val sc = // existing SparkContext

val ssc = new StreamingContext(sc, Seconds(1))

初始化完 Context 之后：

* + 1. 定义消息输入源来创建 DStreams.
    2. 定义 DStreams 的转化操作和输出操作。
    3. 通过 streamingContext.start()来启动消息采集和处理.
    4. 等待程序终止，可以通过 streamingContext.awaitTermination()来设置
    5. 通过 streamingContext.stop()来手动终止处理程序。

StreamingContext 和 SparkContext 什么关系？

import org.apache.spark.streaming.\_

val sc = // existing SparkContext

val ssc = new StreamingContext(sc, Seconds(1))

注意：

StreamingContext 一旦启动，对 DStreams 的操作就不能修改了。在同一时间一个 JVM 中只有一个 StreamingContext 可以启动

stop() 方法将同时停止 SparkContext，可以传入参数 stopSparkContext 用于只停止 StreamingContext

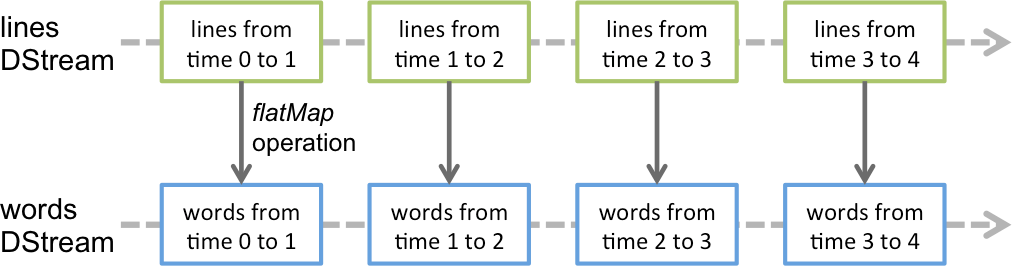
在 Spark1.4 版本后，如何优雅的停止 SparkStreaming 而不丢失数据，通过设置 sparkConf.set("spark.streaming.stopGracefullyOnShutdown","true") 即可。在 StreamingContext 的 start 方法中已经注册了 Hook 方法。

## 什么是 DStreams

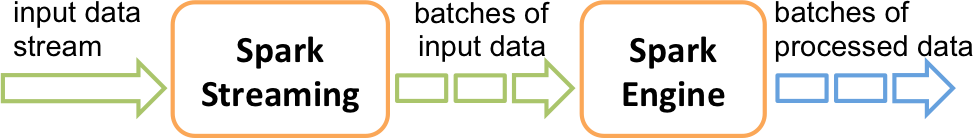
Discretized Stream 是 Spark Streaming 的基础抽象，代表持续性的数据流和经过各种 Spark 原语操作后的结果数据流。在内部实现上， DStream 是一系列连续的 RDD 来表示。每个 RDD 含有一段时间间隔内的数据， 如下图：



对数据的操作也是按照 RDD 为单位来进行的



计算过程由 Spark engine 来完成



## DStreams 输入

Spark Streaming 原生支持一些不同的数据源。一些“核心”数据源已经被打包到 Spark Streaming 的 Maven 工件中， 而其他的一些则可以通过 spark- streaming-kafka 等附加工件获取。每个接收器都以 Spark 执行器程序中一个长期运行的任务的形式运行，因此会占据分配给应用的 CPU 核心。此外， 我们还需要有可用的 CPU 核心来处理数据。这意味着如果要运行多个接收器，就必须至少有和接收器数目相同的核心数，还要加上用来完成计算所需要的核心数。例如， 如果我们想要在流计算应用中运行 10 个接收器，那么至少需要为应用分配 11 个 CPU 核心。所以如果在本地模式运行，不要使用 local 或者 local[1]。

### 基本数据源

* + - 1. 文件数据源

Socket 数据流前面的例子已经看到过。

文件数据流：能够读取所有 HDFS API 兼容的文件系统文件，通过

fileStream 方法进行读取

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

Spark Streaming 将会监控 dataDirectory 目录并不断处理移动进来的文件，记住目前不支持嵌套目录。 像 flume 数据收集。

* + - * 1. 文件需要有相同的数据格式
        2. 文件进入 dataDirectory 的方式需要通过移动或者重命名来实现。
        3. 一旦文件移动进目录，则不能再修改，即便修改了也不会读取新数据。

如果文件比较简单， 则可以使用streamingContext.textFileStream(dataDirectory) 方法来读取文件。文件流不需要接收器，不需要单独分配 CPU 核。

Hdfs 读取实例：

提前需要在 HDFS 上建好目录。

scala> import org.apache.spark.streaming.\_ import org.apache.spark.streaming.\_

scala> val ssc = new StreamingContext(sc, Seconds(10)) ssc: org.apache.spark.streaming.StreamingContext = org.apache.spark.streaming.StreamingContext@4027edeb

scala> val lines = ssc.textFileStream("hdfs://namenode-1:8020/data/") lines: org.apache.spark.streaming.dstream.DStream[String] = org.apache.spark.streaming.dstream.MappedDStream@61d9dd15

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

words: org.apache.spark.streaming.dstream.DStream[String] = org.apache.spark.streaming.dstream.FlatMappedDStream@1e084a26

scala> val wordCounts = words.map(x => (x, 1)).reduceByKey(\_ + \_) wordCounts: org.apache.spark.streaming.dstream.DStream[(String, Int)] = org.apache.spark.streaming.dstream.ShuffledDStream@8947a4b

scala> wordCounts.print()

scala> ssc.start()

上传文件上去：

[bigdata@master01 hadoop-2.7.3]$ ls

bin data etc include lib libexec LICENSE.txt logs NOTICE.txt README.txt sbin sdata share

[bigdata@master01 hadoop-2.7.3]$ bin/hdfs dfs -put ./LICENSE.txt /data/

[bigdata@master01 hadoop-2.7.3]$ bin/hdfs dfs -put ./README.txt /data/

获取计算结果：

Time: 1504665716000 ms

Time: 1504665717000 ms

Time: 1504665718000 ms

(227.7202-1,2)

(created,2) (offer,8)

(BUSINESS,11)

(agree,10) (hereunder,,1) (“control”,1) (Grant,2) (2.2.,2)

(include,11)

...

Time: 1504665719000 ms

Time: 1504665739000 ms

Time: 1504665740000 ms

(under,1)

(Technology,1) (distribution,2)

[(http://hadoop.apache.org/core/,1)](http://hadoop.apache.org/core/%2C1)) (Unrestricted,1)

(740.13),1)

(check,1) (have,1) (policies,1) (uses,1)

...

Time: 1504665741000 ms

* + - 1. 自定义数据源

通过继承 Receiver，并实现 onStart、onStop 方法来自定义数据源采集。

class CustomReceiver(host: String, port: Int)

extends Receiver[String](StorageLevel.MEMORY\_AND\_DISK\_2) with Logging {

def onStart() {

// Start the thread that receives data over a connection

new Thread("Socket Receiver") {

override def run() { receive() }

}.start()

}

def onStop() {

// There is nothing much to do as the thread calling receive()

// is designed to stop by itself if isStopped() returns false

}

/\*\* Create a socket connection and receive data until receiver is stopped \*/

private def receive() {

var socket: Socket = null

var userInput: String = null

try {

// Connect to host:port

socket = new Socket(host, port)

// Until stopped or connection broken continue reading

val reader = new BufferedReader(

new InputStreamReader(socket.getInputStream(), StandardCharsets.UTF\_8))

userInput = reader.readLine()

while(!isStopped && userInput != null) {

store(userInput)

userInput = reader.readLine()

}

reader.close()

socket.close()

// Restart in an attempt to connect again when server is active again

restart("Trying to connect again")

} catch {

case e: java.net.ConnectException =>

// restart if could not connect to server

restart("Error connecting to " + host + ":" + port, e)

case t: Throwable =>

// restart if there is any other error

restart("Error receiving data", t)

}

}

}

可以通过 streamingContext.receiverStream(<instance of custom receiver>)

来使用自定义的数据采集源

// Assuming ssc is the StreamingContext

val customReceiverStream = ssc.receiverStream(new CustomReceiver(host, port))

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

...

模拟Spark 内置的Socket 链接：

**package** com.atguigu.streaming

**import** java.io.{BufferedReader, InputStreamReader} **import** java.net.Socket

**import** java.nio.charset.StandardCharsets

**import** org.apache.spark.SparkConf

**import** org.apache.spark.storage.StorageLevel

**import** org.apache.spark.streaming.{Seconds, StreamingContext} **import** org.apache.spark.streaming.receiver.Receiver

*/\*\**

*\* Created by wuyufei on 06/09/2017.*

*\*/*

**class** CustomReceiver (host: String, port: Int) **extends** Receiver[String](StorageLevel.*MEMORY\_AND\_DISK\_2*) {

**override def** onStart(): Unit = {

*// Start the thread that receives data over a connection*

**new** Thread(**"Socket Receiver"**) { **override def** run() { receive() }

}.start()

}

**override def** onStop(): Unit = {

*// There is nothing much to do as the thread calling receive()*

*// is designed to stop by itself if isStopped() returns false*

}

*/\*\* Create a socket connection and receive data until receiver is stopped \*/*

**private def** receive() { **var** socket: Socket = **null**

**var** userInput: String = **null try** {

*// Connect to host:port*

socket = **new** Socket(host, port)

*// Until stopped or connection broken continue reading*

**val** reader = **new** BufferedReader(**new**

InputStreamReader(socket.getInputStream(), StandardCharsets.*UTF\_8*))

userInput = reader.readLine() **while**(!isStopped && userInput != **null**) {

*//* 传送出来

store(userInput)

userInput = reader.readLine()

}

reader.close() socket.close()

*// Restart in an attempt to connect again when server is active again*

restart(**"Trying to connect again"**)

} **catch** {

**case** e: java.net.ConnectException =>

*// restart if could not connect to server*

restart(**"Error connecting to "** + host + **":"** + port, e) **case** t: Throwable =>

*// restart if there is any other error*

restart(**"Error receiving data"**, t)

}

}

}

**object** CustomReceiver {

**def** main(args: Array[String]) {

**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.receiverStream(**new** CustomReceiver(**"master01"**, 9999))

*// Split each line into words*

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

*//import org.apache.spark.streaming.StreamingContext.\_ // not necessary since Spark 1.3*

*// 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*

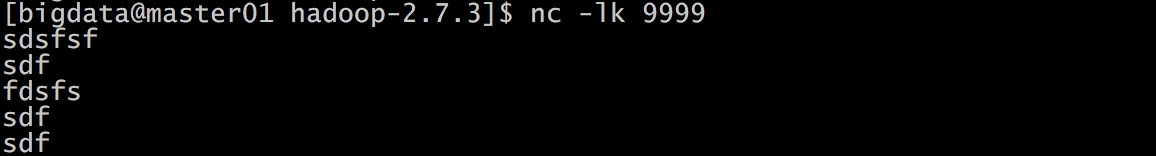
ssc.awaitTermination() *// Wait for the computation to terminate*

*//ssc.stop()*

}

}





* + - 1. RDD 队列

测试过程中，可以通过使用 streamingContext.queueStream(queueOfRDDs) 来创建 DStream， 每一个推送到这个队列中的 RDD，都会作为一个 DStream 处理。

**package** com.atguigu.streaming

**import** org.apache.spark.SparkConf **import** org.apache.spark.rdd.RDD

**import** org.apache.spark.streaming.{Seconds, StreamingContext}

**import** scala.collection.mutable

**object** QueueRdd {

**def** main(args: Array[String]) {

**val** conf = **new** SparkConf().setMaster(**"local[2]"**).setAppName(**"QueueRdd"**) **val** ssc = **new** StreamingContext(conf, *Seconds*(1))

*// Create the queue through which RDDs can be pushed to*

*// a QueueInputDStream*

*//*创建 *RDD* 队列

**val** rddQueue = **new** mutable.SynchronizedQueue[RDD[Int]]()

*// Create the QueueInputDStream and use it do some processing*

*//* 创建*QueueInputDStream*

**val** inputStream = ssc.queueStream(rddQueue)

*//*处理队列中的 *RDD* 数据

**val** mappedStream = inputStream.map(x => (x % 10, 1)) **val** reducedStream = mappedStream.reduceByKey(\_ + \_)

*//*打印结果

reducedStream.print()

*//*启动计算

ssc.start()

*// Create and push some RDDs into*

**for** (i <- 1 to 30) {

rddQueue += ssc.sparkContext.makeRDD(1 to 300, 10) Thread.*sleep*(2000)

*//*通过程序停止*StreamingContext* 的运行

*//ssc.stop()*

}

}

}

[bigdata@master01 spark-2.1.1-bin-hadoop2.7]$ bin/spark-submit --class com.atguigu.streaming.QueueRdd ~/queueRdd-jar-with-dependencies.jar

17/09/05 23:28:03 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable

Time: 1504668485000 ms

(4,30)

(0,30)

(6,30)

(8,30)

(2,30)

(1,30)

(3,30)

(7,30)

(9,30)

(5,30)

Time: 1504668486000 ms

Time: 1504668487000 ms

(4,30)

(0,30)

(6,30)

(8,30)

(2,30)

(1,30)

(3,30)

(7,30)

(9,30)

(5,30)

### 高级数据源

除核心数据源外，还可以用附加数据源接收器来从一些知名数据获取系统中接收的数据，这些接收器都作为 Spark Streaming 的组件进行独立打包了。

它们仍然是 Spark 的一部分，不过你需要在构建文件中添加额外的包才能使用它们。现有的接收器包括 Twitter、Apache Kafka、Amazon Kinesis、Apache Flume，以及 ZeroMQ。可以通过添加与 Spark 版本匹配 的 Maven 工件 spark-streaming-[projectname]\_2.10 来引入这些附加接收器。

* + - 1. Apache Kafka

在工程中需要引入 Maven 工件 spark- streaming-kafka\_2.10 来使用它。包内提供的 KafkaUtils 对象可以在 StreamingContext 和JavaStreamingContext 中以你的 Kafka 消息创建出 DStream。由于 KafkaUtils 可以订阅多个主题，因此它创建出的 DStream 由成对的主题和消息组成。要创建出一个流数据，需 要使用 StreamingContext 实例、一个由逗号隔开的 ZooKeeper 主机列表字符串、消费者组的名字(唯一名字)， 以及一个从主题到针对这个主题的接收器线程数的映射表来调用 createStream()

方法

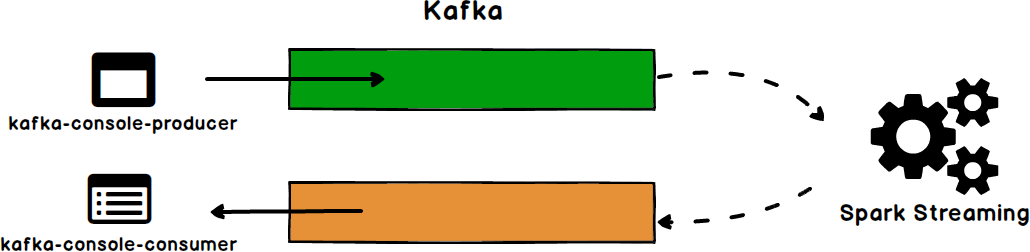
import org.apache.spark.streaming.kafka.\_ ... // 创建一个从主题到接收器线程数的映射表

val topics = List(("pandas", 1), ("logs", 1)).toMap

val topicLines = KafkaUtils.createStream(ssc, zkQuorum, group, topics)

topicLines.map(\_.\_2)

下面我们进行一个实例，演示 SparkStreaming 如何从 Kafka 读取消息，如果通过连接池方法把消息处理完成后再写会 Kafka：



kafka Connection Pool 程序：

**package** com.atguigu.streaming **import** java.util.Properties

**import** org.apache.commons.pool2.impl.DefaultPooledObject

**import** org.apache.commons.pool2.{BasePooledObjectFactory, PooledObject} **import** org.apache.kafka.clients.producer.{KafkaProducer, ProducerRecord}

*/\*\**

*\* Created by wuyufei on 06/09/2017.*

*\*/*

**case class** KafkaProducerProxy(brokerList: String,

producerConfig: Properties = **new** Properties, defaultTopic: Option[String] = None,

producer: Option[KafkaProducer[String, String]] = None)

{

**type** Key = String **type** Val = String

*require*(brokerList == **null** || !brokerList.isEmpty, **"Must set broker list"**)

**private val** *p* = producer getOrElse {

**var** props:Properties= **new** Properties(); props.put(**"bootstrap.servers"**, brokerList); props.put(**"key.serializer"**,

**"org.apache.kafka.common.serialization.StringSerializer"**); props.put(**"value.serializer"**,

**"org.apache.kafka.common.serialization.StringSerializer"**);

**new** KafkaProducer[String,String](props)

}

**private def** toMessage(value: Val, key: Option[Key] = None, topic: Option[String] = None): ProducerRecord[Key, Val] = {

**val** t = topic.getOrElse(defaultTopic.getOrElse(**throw new** IllegalArgumentException(**"Must provide topic or default topic"**)))

*require*(!t.isEmpty, **"Topic must not be empty"**) key **match** {

**case** *Some*(k) => **new** ProducerRecord(t, k, value) **case** \_ => **new** ProducerRecord(t, value)

}

}

**def** send(key: Key, value: Val, topic: Option[String] = None) {

*p*.send(toMessage(value, *Option*(key), topic))

}

**def** send(value: Val, topic: Option[String]) { send(**null**, value, topic)

}

**def** send(value: Val, topic: String) { send(**null**, value, *Option*(topic))

}

**def** send(value: Val) { send(**null**, value, None)

}

**def** shutdown(): Unit = *p*.close()

}

**abstract class** KafkaProducerFactory(brokerList: String, config: Properties, topic: Option[String] = None) **extends** Serializable {

**def** newInstance(): KafkaProducerProxy

}

**class** BaseKafkaProducerFactory(brokerList: String,

config: Properties = **new** Properties, defaultTopic: Option[String] = None)

**extends** KafkaProducerFactory(brokerList, config, defaultTopic) {

**override def** newInstance() = **new** KafkaProducerProxy(brokerList, config, defaultTopic)

}

**class** PooledKafkaProducerAppFactory(**val** factory: KafkaProducerFactory) **extends** BasePooledObjectFactory[KafkaProducerProxy] **with** Serializable {

**override def** create(): KafkaProducerProxy = factory.newInstance()

**override def** wrap(obj: KafkaProducerProxy): PooledObject[KafkaProducerProxy]

= **new** DefaultPooledObject(obj)

**override def** destroyObject(p: PooledObject[KafkaProducerProxy]): Unit = { p.getObject.shutdown()

**super**.destroyObject(p)

}

}

KafkaStreaming main:

**package** com.atguigu.streaming

**import** org.apache.commons.pool2.impl.{GenericObjectPool, GenericObjectPoolConfig}

**import** org.apache.kafka.clients.consumer.ConsumerRecord

**import** org.apache.kafka.common.serialization.StringDeserializer **import** org.apache.spark.SparkConf

**import** org.apache.spark.api.java.function.VoidFunction **import** org.apache.spark.rdd.RDD

**import** org.apache.spark.streaming.kafka010.{ConsumerStrategies, KafkaUtils, LocationStrategies}

**import** org.apache.spark.streaming.{Seconds, StreamingContext}

*/\*\**

*\* Created by wuyufei on 06/09/2017.*

*\*/*

**object** createKafkaProducerPool{

**def** apply(brokerList: String, topic: String): GenericObjectPool[KafkaProducerProxy] = {

**val** producerFactory = **new** BaseKafkaProducerFactory(brokerList, defaultTopic

= *Option*(topic))

**val** pooledProducerFactory = **new** PooledKafkaProducerAppFactory(producerFactory)

**val** poolConfig = {

**val** c = **new** GenericObjectPoolConfig **val** maxNumProducers = 10 c.setMaxTotal(maxNumProducers) c.setMaxIdle(maxNumProducers)

c

}

**new** GenericObjectPool[KafkaProducerProxy](pooledProducerFactory, poolConfig)

}

}

**object** KafkaStreaming{

**def** main(args: Array[String]) {

**val** conf = **new** SparkConf().setMaster(**"local[4]"**).setAppName(**"NetworkWordCount"**)

**val** ssc = **new** StreamingContext(conf, *Seconds*(1))

*//*创建 *topic*

**val** brobrokers = **"172.16.148.150:9092,172.16.148.151:9092,172.16.148.152:9092"**

**val** sourcetopic=**"source"**; **val** targettopic=**"target"**;

*//*创建消费者组

**var** group=**"con-consumer-group"**

*//*消费者配置

**val** kafkaParam = *Map*(

**"bootstrap.servers"** -> brobrokers,*//*用于初始化链接到集群的地址**"key.deserializer"** -> *classOf*[StringDeserializer], **"value.deserializer"** -> *classOf*[StringDeserializer],

*//*用于标识这个消费者属于哪个消费团体

**"group.id"** -> group,；

*//*如果没有初始化偏移量或者当前的偏移量不存在任何服务器上，可以使用这个配置属性

*//*可以使用这个配置，*latest* 自动重置偏移量为最新的偏移量

**"auto.offset.reset"** -> **"latest"**,

*//*如果是*true*，则这个消费者的偏移量会在后台自动提交

**"enable.auto.commit"** -> (**false**: java.lang.Boolean)

);

*//ssc.sparkContext.broadcast(pool)*

*//*创建 *DStream*，返回接收到的输入数据

**var** stream=KafkaUtils.*createDirectStream*[String,String](ssc, LocationStrategies.*PreferConsistent*,ConsumerStrategies.*Subscribe*[String,String

](*Array*(sourcetopic),kafkaParam))

*//*每一个 *stream* 都是一个*ConsumerRecord*

stream.map(s =>(**"id:"** + s.key(),**">>>>:"**+s.value())).foreachRDD(rdd => { rdd.foreachPartition(partitionOfRecords => {

*// Get a producer from the shared pool*

**val** pool = createKafkaProducerPool(brobrokers, targettopic) **val** p = pool.borrowObject()

partitionOfRecords.foreach {message => System.*out*.println(message.\_2);p.send(message.\_2,*Option*(targettopic))}

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

pool.returnObject(p)

})

})

ssc.start() ssc.awaitTermination()

}

}

程序部署：

1、启动zookeeper 和kafka。

bin/kafka-server-start.sh -deamon ./config/server.properties

2、创建两个 topic， 一个为 source，一个为 target

bin/kafka-topics.sh --create --zookeeper 192.168.56.150:2181,192.168.56.151:2181,192.168.56.152:2181 --replication-factor 2 --

partitions 2 --topic source

bin/kafka-topics.sh --create --zookeeper 172.16.148.150:2181,172.16.148.151:2181,172.16.148.152:2181 --replication-factor 2 --

partitions 2 --topic target

3、启动 kafka console producer 写入 source topic

bin/kafka-console-producer.sh --broker-list 192.168.56.150:9092, 192.168.56.151:9092, 192.168.56.152:9092 --topic source

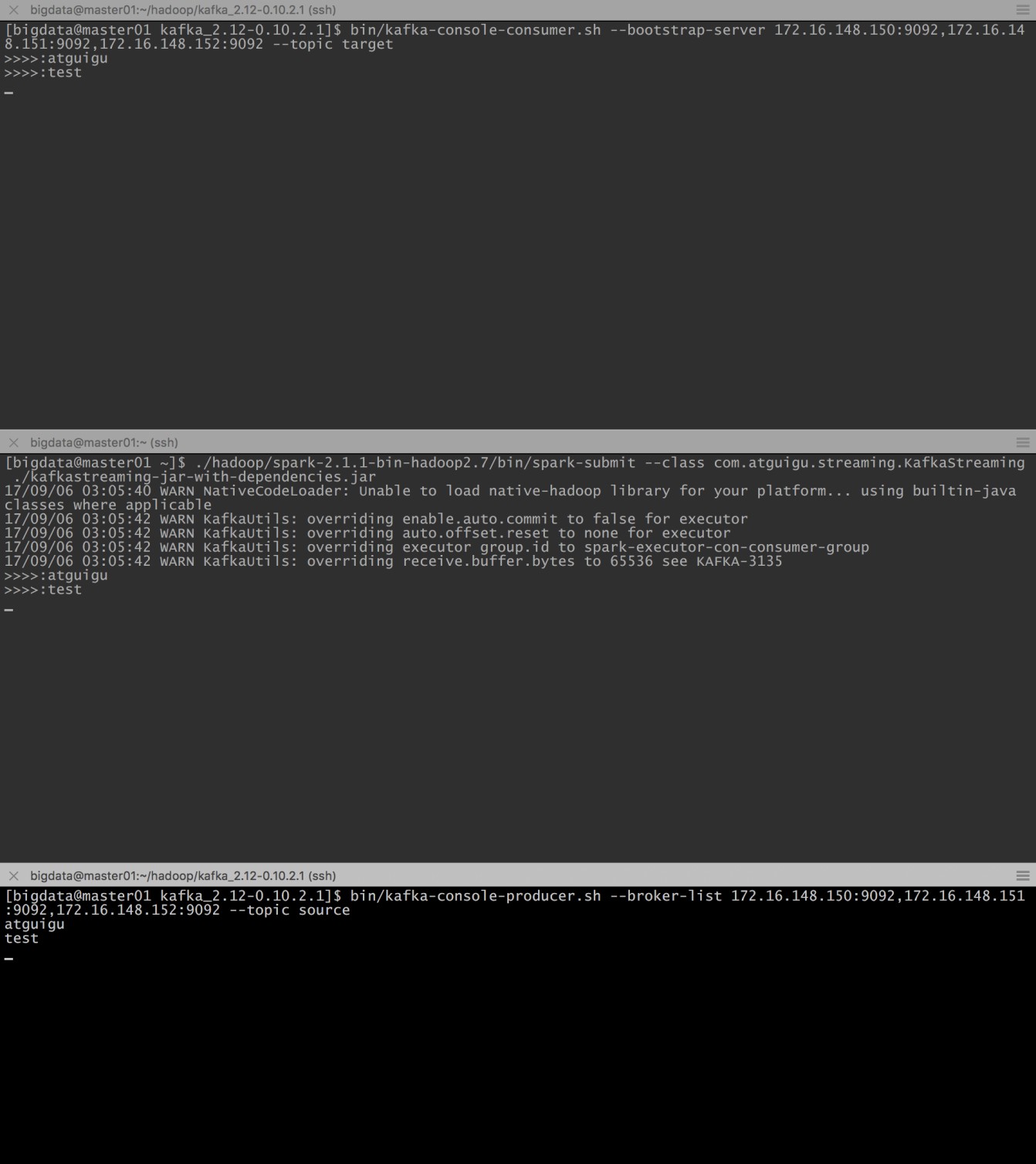
4、启动 kafka console consumer 监听 target topic

bin/kafka-console-consumer.sh --bootstrap-server 192.168.56.150:9092, 192.168.56.151:9092, 192.168.56.152:9092 --topic source

5、启动 kafkaStreaming 程序：

[bigdata@master01 ~]$ ./hadoop/spark-2.1.1-bin-hadoop2.7/bin/spark-submit --class com.atguigu.streaming.KafkaStreaming ./kafkastreaming-jar-with-dependencies.jar

6、程序运行截图：



* + - 1. Spark 对 Kafka 两种连接方式的对比

Spark 对于 Kafka 的连接主要有两种方式， 一种是（直连方式）

DirectKafkaInputDStream，另外一种是 KafkaInputDStream。DirectKafkaInputDStream 只在 driver 端接收数据，所以继承了InputDStream， 是没有 receivers（接收器方式） 的。

主要通过 KafkaUtils#createDirectStream 以及 KafkaUtils#createStream 这两个 API 来创建，除了要传入的参数不同外，接收 kafka 数据的节点、拉取数据的时机也完全不同。

KafkaUtils#createStream【Receiver-based】（接收器模式）

这种方法使用一个 Receiver 来接收数据。在该 Receiver 的实现中使用了Kafka high-level consumer API 。Receiver 从 kafka 接收的数据将被存储到Spark executor 中，随后启动的 job 将处理这些数据。

在默认配置下，该方法失败后会丢失数据（保存在 executor 内存里的数据在 application 失败后就没了），若要保证数据不丢失，需要启用 WAL（即预写日志至 HDFS、S3 等），这样再失败后可以从日志文件中恢复数据。

在该函数中，会新建一个 KafkaInputDStream 对象， KafkaInputDStream 继承于 ReceiverInputDStream。KafkaInputDStream 实现了 getReceiver 方法，返回接收器的实例：

def getReceiver(): Receiver[(K, V)] = {

if (!useReliableReceiver) {

//< 不启用 WAL

new KafkaReceiver[K, V, U, T](kafkaParams, topics, storageLevel)

} else {

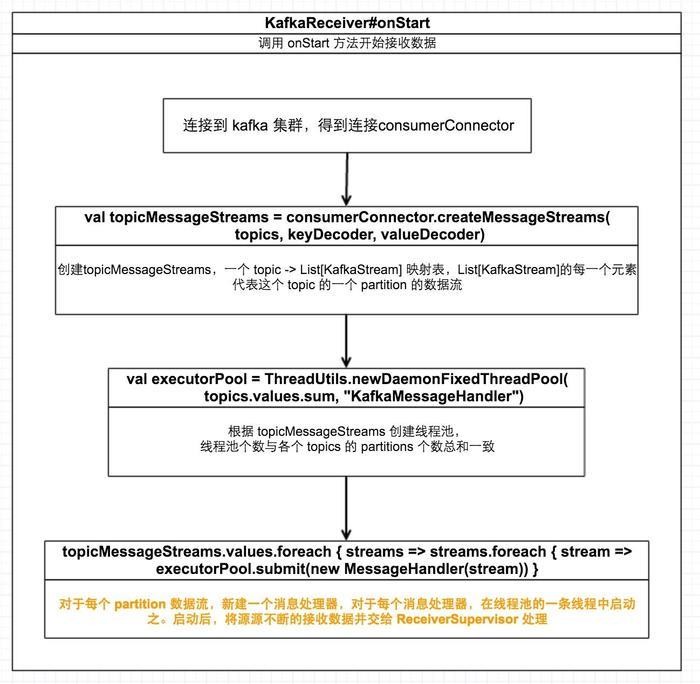
//< 启用 WAL

new ReliableKafkaReceiver[K, V, U, T](kafkaParams, topics, storageLevel)

}

}

根据是否启用 WAL，receiver 分为 KafkaReceiver 和 ReliableKafkaReceiver。下图描述了 KafkaReceiver 接收数据的具体流程：



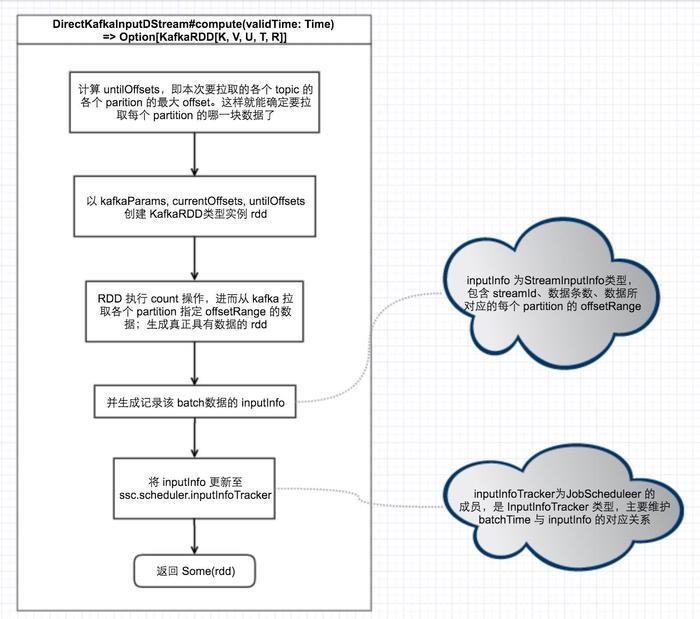
需要注意的点：

* Kafka Topic 的 partitions 与 RDD 的 partitions 没有直接关系，不能一一对应。如果增加 topic 的 partition 个数的话仅仅会增加单个 Receiver 接收数据的线程数。事实上，使用这种方法只会在一个 executor 上启用一个 Receiver，该 Receiver 包含一个线程池，线程池的线程个数与所有topics 的 partitions 个数总和一致，每条线程接收一个 topic 的一个partition 的数据。而并不会增加处理数据时的并行度。
* 对于一个 topic，可以使用多个 groupid 相同的 input DStream 来使用多个 Receivers 来增加并行度，然后 union 他们；对于多个 topics，除了可以用上个办法增加并行度外，还可以对不同的 topic 使用不同的 input DStream 然后 union 他们来增加并行度

* 如果你启用了 WAL，为能将接收到的数据将以 log 的方式在指定的存储系统备份一份，需要指定输入数据的存储等级为StorageLevel.MEMORY\_AND\_DISK\_SER 或StorageLevel.MEMORY\_AND\_DISK\_SER\_2

**KafkaUtils#createDirectStream**【**WithOut Receiver**】（ 不是接收器模式）

自 Spark-1.3.0 起，提供了不需要 Receiver 的方法。替代了使用receivers 来接收数据，该方法定期查询每个 topic+partition 的 lastest offset，并据此决定每个 batch 要接收的 offsets 范围。

KafkaUtils#createDirectStream 调用中，会新建 DirectKafkaInputDStream， DirectKafkaInputDStream#compute(validTime: Time) 会从 kafka 拉取数据并生成 RDD，流程如下：

如上图所示，该函数主要做了以下三个事情：

1. 确定要接收的 partitions 的 offsetRange，以作为第 2 步创建的 RDD 的数据来源
2. 创建 RDD 并执行 count 操作，使 RDD 真实具有数据
3. 以 streamId、数据条数，offsetRanges 信息初始化 inputInfo 并添加到JobScheduler 中

进一步看 KafkaRDD 的 getPartitions 实现：

override def getPartitions: Array[Partition] = {

offsetRanges.zipWithIndex.map { case (o, i) =>

val (host, port) = leaders(TopicAndPartition(o.topic, o.partition))

new KafkaRDDPartition(i, o.topic, o.partition, o.fromOffset, o.untilOffset, host, port)

}.toArray

}

从上面的代码可以很明显看到，KafkaRDD 的 partition 数据与 Kafka topic 的某个 partition 的 o.fromOffset 至 o.untilOffset 数据是相对应的，也就是说sparkStreamRDD 的 partition 与 Kafka partition 是一一对应的

该方式相比使用 Receiver 的方式有以下好处：

* 简化并行：不再需要创建多个 kafka input DStream 然后再 union 这些input DStream。使用 directStream，Spark Streaming 会创建与 Kafka partitions 相同数量的 paritions 的 RDD，RDD 的 partition 与 Kafka 的 partition 一一对应，这样更易于理解及调优
* 高效：在方式一中要保证数据零丢失需要启用 WAL（预写日志），这会占用更多空间。而在方式二中，可以直接从 Kafka 指定的 topic 的指定offsets 处恢复数据，不需要使用 WAL
* 恰好一次语义保证：基于 Receiver 方式使用了 Kafka 的 high level API 来在 Zookeeper 中存储已消费的 offsets。这在某些情况下会导致一些数据被消费两次，比如 streaming app 在处理某个 batch 内已接受到的数据的过程中挂掉，但是数据已经处理了一部分，但这种情况下无法将已处理数据的 offsets 更新到 Zookeeper 中，下次重启时，这批数据将再次被消费且处理。基于 direct 的方式，使用 kafka 的简单 api，Spark Streaming 自己就负责追踪消费的 offset，并保存在 checkpoint 中。Spark 自己一定是同步的，因此可以保证数据是消费一次且仅消费一次。这种方式中，只要将output 操作和保存 offsets 操作封装成一个原子操作就能避免失败后的重复消费和处理，从而达到恰好一次的语义（Exactly-once）

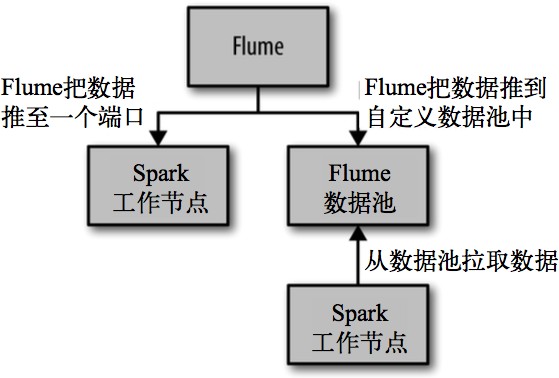
通过以上分析，我们可以对这两种方式的区别做一个总结：

1. createStream 会使用 Receiver；而 createDirectStream 不会
2. createStream 使用的 Receiver 会分发到某个 executor 上去启动并接受数据；而 createDirectStream 直接在 driver 上接收数据（直连方式）
3. createStream 使用 Receiver 源源不断的接收数据并把数据交给ReceiverSupervisor 处理最终存储为 blocks 作为 RDD 的输入，从 kafka 拉取数据与计算消费数据相互独立；而 createDirectStream 会在每个 batch 拉取数据并就地消费，到下个 batch 再次拉取消费，周而复始，从 kafka 拉取数据与计算消费数据是连续的，没有独立开
4. createStream 中创建的 KafkaInputDStream 每个 batch 所对应的 RDD 的partition 不与 Kafka partition 一一对应；而 createDirectStream 中创建的 DirectKafkaInputDStream 每个 batch 所对应的 RDD 的 partition 与 Kafka partition 一一对应
   * + 1. Flume-ng

Spark 提供两个不同的接收器来使用 Apache Flume(<http://flume.apache.org/>，见图 10-8)。 两个接收器简介如下。

* 推式接收器该接收器以 Avro 数据池的方式工作，由 Flume 向其中推数据。
* 拉式接收器该接收器可以从自定义的中间数据池中拉数据， 而其他进程可以使用 Flume 把数据推进 该中间数据池。

两种方式都需要重新配置 Flume，并在某个节点配置的端口上运行接收器(不是已有的 Spark 或者 Flume 使用的端口)。要使用其中任何一种方法，都需要在工程中引入 Maven 工件 spark-streaming-flume\_2.10。



推式接收器的方法设置起来很容易，但是它不使用事务来接收数据。在这种方式中，接收 器以 Avro 数据池的方式工作，我们需要配置 Flume 来把数据发到 Avro 数据池。我们提供的 FlumeUtils 对象会把接收器配置在一个

特定的工作节点的主机名及端口号上。这些设置必须和 Flume 配置相匹配。

虽然这种方式很简洁，但缺点是没有事务支持。这会增加运行接收器的工作节点发生错误 时丢失少量数据的几率。不仅如此，如果运行接收器的工作节点发生故障，系统会尝试从 另一个位置启动接收器，这时需要重新配置Flume 才能将数据发给新的工作节点。这样配 置会比较麻烦。

较新的方式是拉式接收器(在 Spark 1.1 中引入)，它设置了一个专用的Flume 数据池供 Spark Streaming 读取， 并让接收器主动从数据池中拉取数据。这种方式的优点在于弹性较 好，Spark Streaming 通过事务从数据池中读取并复制数据。在收到事务完成的通知前，这 些数据还保留在数据池中。

我们需要先把自定义数据池配置为 Flume 的第三方插件。安装插件的最新方法请参考 Flume 文档的相关部分(https://flume.apache.org/FlumeUserGuide.html#installing-third-party- plugins)。由于插件是用 Scala 写的，因此需要把插件本身以及 Scala 库都添加到Flume 插件 中。Spark 1.1 中对应的 Maven 索引如例 10-37 所示。

<dependency>

org.apache.spark</groupId>

<groupId>

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

<version>1.2.0</version>

</dependency>

<dependency>

<groupId>org.scala-lang</groupId>

<artifactId>scala-library</artifactId>

<version>2.11.11</version>

</dependency>

当你把自定义 Flume 数据池添加到一个节点上之后，就需要配置 Flume

来把数据推送到这个数据池中，

a1.sinks = spark

a1.sinks.spark.type = org.apache.spark.streaming.flume.sink.SparkSink a1.sinks.spark.hostname = 192.168.137.51

a1.sinks.spark.port = 8888

a1.sinks.spark.channel = memoryChannel

等到数据已经在数据池中缓存起来，就可以调用 FlumeUtils 来读取数据

了



## DStreams 转换

每个时间片（批次）中存储的就是RDD。

DStream 上的原语与 RDD 的类似，分为 Transformations（ 转换）和Output Operations（输出）两种，此外转换操作中还有一些比较特殊的原语， 如：updateStateByKey()、transform()以及各种 Window 相关的原语。

|  |  |
| --- | --- |
| **Transformation** | **Meaning** |
| map(func) | 将源 DStream 中的每个元素通过一个函数 func 从而得到新的 DStreams。 |
| flatMap(func) | 和 map 类似，但是每个输入的项可以被映射为 0 或更多项。 |
| filter(func) | 选择源 DStream 中函数 func 判为 true 的记录作为新 DStreams |
| repartition(numPartitions) | 通过创建更多或者更少的 partition 来改变此 DStream 的并行级别。 |
| union(otherStream) | 联合源 DStreams 和其他 DStreams 来得到新 DStream |
| count() | 统计源 DStreams 中每个 RDD 所含元素的个数得到单元素 RDD 的新 DStreams。 |
| reduce(func) | 通过函数 func(两个参数一个输出)来整合源 DStreams 中每个 RDD 元素得到单元素RDD 的 DStreams。这个函数需要关联从而可以被并行计算。 |
| countByValue() | 对于 DStreams 中元素类型为 K 调用此函数，得到包含(K,Long)对的新 DStream，其中Long 值表明相应的 K 在源 DStream 中每个RDD 出现的频率。 |
| reduceByKey(func, [numTasks]) | 对(K,V)对的 DStream 调用此函数，返回同样（ K,V)对的新 DStream，但是新DStream 中的对应 V 为使用 reduce 函数整合  而来。*Note*： 默认情况下，这个操作使用 |

|  |  |
| --- | --- |
|  | Spark 默认数量的并行任务（本地模式为 2， 集群模式中的数量取决于配置参数spark.default.parallelism）。你也可以传入可选的参数 numTaska 来设置不同数量的任  务。 |
| join(otherStream, [numTasks]) | 两 DStream 分别为(K,V)和(K,W)对， 返回(K,(V,W))对的新 DStream。 |
| cogroup(otherStream, [numTasks]) | 两 DStream 分别为(K,V)和(K,W)对， 返回(K,(Seq[V],Seq[W])对新 DStreams |
| transform(func) | 将 RDD 到 RDD 映射的函数 func 作用于源 DStream 中每个 RDD 上得到新 DStream。这个可用于在 DStream 的 RDD 上做任意操作。 |
| updateStateByKey(func) | 得到”状态”DStream， 其中每个 key 状态的更新是通过将给定函数用于此 key 的上一个状态和新值而得到。这个可用于保存每个 key 值的任意状态数据。 |

DStream 的转化操作可以分为无状态(stateless)和有状态(stateful)两种。

* 在无状态转化操作中， 每个批次的处理不依赖于之前批次的数据。常见的 RDD 转化操作，例如 map()、filter()、reduceByKey() 等，都是无状态转化操作。

------------------------------------------- Time: 1586396145000 ms

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

hello 3 你再第一批次。输入了 3 个 hello

------------------------------------------- Time: 1586396145000 ms

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

hello 2 第二批次，输入两个 hello .

* 相对地， 有状态转化操作需要使用之前批次的数据或者是中间结果来计算当前批次的数据。有状态转化操作包括基于滑动窗口的转化操作和追踪状态变化的转化操作。

------------------------------------------- Time: 1586396145000 ms

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

hello 3 你再第一批次。输入了 3 个 hello

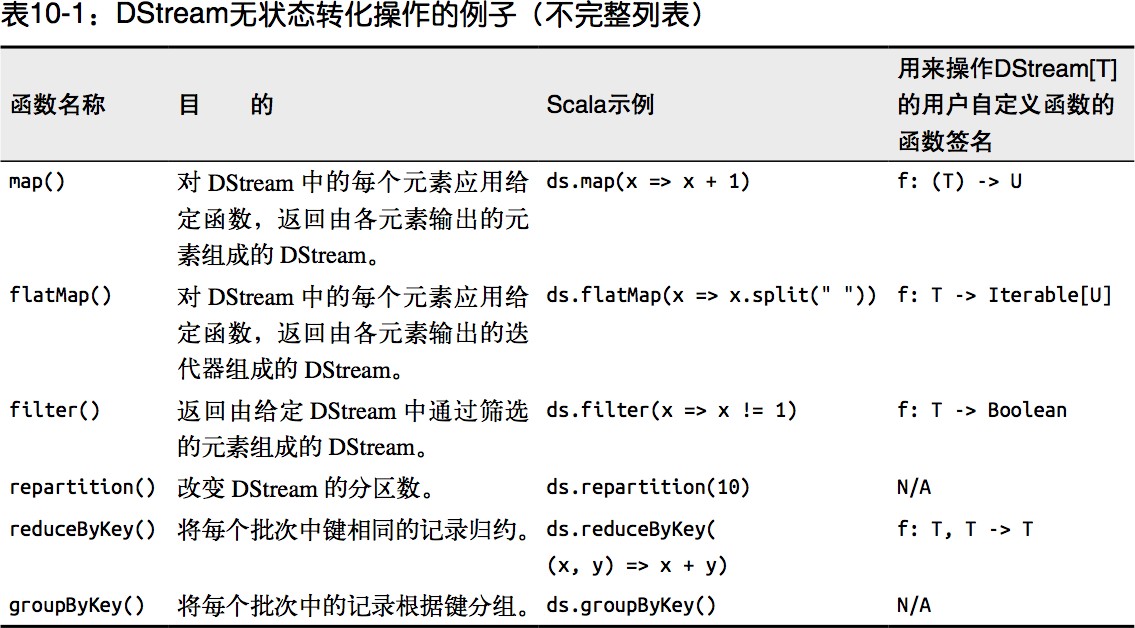
------------------------------------------- Time: 1586396145000 ms

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

hello 5 第二批次，输入 2 个 hello . 经过状态转化，结果为 5

### 无状态转化操作

无状态转化操作就是把简单的 RDD 转化操作应用到每个批次上，也就是转化 DStream 中的每一个 RDD。部分无状态转化操作列在了下表中。注意， 针对键值对的 DStream 转化操作 ( 比如 reduceByKey()) 要添加 import StreamingContext.\_ 才能在 Scala 中使用。



需要记住的是， 尽管这些函数看起来像作用在整个流上一样， 但事实上每个 DStream 在内部是由许多 RDD(批次)组成，且无状态转化操作是分别应用到每个 RDD 上的。例如， reduceByKey() 会归约每个时间区间中的数据，但不会归约不同区间之间的数据。

举个例子，在之前的 wordcount 程序中，我们只会统计 1 秒内接收到的数据的单词个数，而不会累加。

无状态转化操作也能在多个 DStream 间整合数据，不过也是在各个时间区间内。例如，键 值对 DStream 拥有和 RDD 一样的与连接相关的转化操作， 也就是 cogroup()、join()、 leftOuterJoin() 等。我们可以在 DStream 上使用这些操作，这样就对每个批次分别执行了对应的 RDD 操作。

我们还可以像在常规的 Spark 中一样使用 DStream 的 union() 操作将它和另一个 DStream 的内容合并起来，也可以使用 StreamingContext.union() 来合并多个流。

### 有状态转化操作

特殊的 **Transformations**

* + - 1. 追踪状态变化 UpdateStateByKey

UpdateStateByKey 原语用于记录历史记录，有时，我们需要在 DStream 中跨批次维护状态 ( 例如 流 计 算 中 累 加 wordcount) 。针对这种情况， updateStateByKey() 为我们提供了对一个状态变量的访问，用于键值对形式的

DStream。给定一个由(键，事件)对构成的 DStream，并传递一个指定如何根据新的事件 更新每个键对应状态的函数， 它可以构建出一个新的 DStream，其内部数据为(键，状态) 对。

updateStateByKey() 的结果会是一个新的 DStream，其内部的 RDD 序列是由每个时间区间对应的(键，状态)对组成的。

updateStateByKey 操作使得我们可以在用新信息进行更新时保持任意的状态。为使用这个功能，你需要做下面两步：

1. 定义状态，状态可以是一个任意的数据类型。
2. 定义状态更新函数，用此函数阐明如何使用之前的状态和来自输入流的新值对状态进行更新。

使用 updateStateByKey 需要对检查点目录进行配置，会使用检查点来保存状态。

更新版的 wordcount：

**package** com.atguigu.streaming

**import** org.apache.spark.SparkConf

**import** org.apache.spark.streaming.{Seconds, StreamingContext}

*/\*\**

*\* Created by wuyufei on 06/09/2017.*

*\*/*

**object** WorldCount {

**def** main(args: Array[String]) {

*//* 定义更新状态方法，参数*values* 为当前批次单词频度，*state* 为以往批次单词频度

**val** updateFunc = (values: Seq[Int], state: Option[Int]) => { **val** currentCount = values.foldLeft(0)(\_ + \_)

**val** previousCount = state.getOrElse(0)

*Some*(currentCount + previousCount)

}

**val** conf = **new** SparkConf().setMaster(**"local[2]"**).setAppName(**"NetworkWordCount"**)

**val** ssc = **new** StreamingContext(conf, *Seconds*(3)) ssc.checkpoint(**"."**)

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

**val** lines = ssc.socketTextStream(**"master01"**, 9999)

*// Split each line into words*

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

*//import org.apache.spark.streaming.StreamingContext.\_ // not necessary since Spark 1.3*

*// Count each word in each batch*

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

*//* 使用*updateStateByKey* 来更新状态，统计从运行开始以来单词总的次数

**val** stateDstream = pairs.updateStateByKey[Int](updateFunc)

stateDstream.print()

*//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*

*//ssc.stop()*

}

}

启动 nc –lk 9999

[bigdata@master01 ~]$ nc -lk 9999 ni shi shui

ni hao ma

启动统计程序：

[bigdata@master01 ~]$ ./hadoop/spark-2.1.1-bin-hadoop2.7/bin/spark-submit --class com.atguigu.streaming.WorldCount ./statefulwordcount-jar-with-dependencies.jar 17/09/06 04:06:09 WARN NativeCodeLoader: Unable to load native-hadoop library for your

platform... using builtin-java classes where applicable

Time: 1504685175000 ms

Time: 1504685181000 ms

(shi,1)

(shui,1)

(ni,1)

Time: 1504685187000 ms

(shi,1)

(ma,1)

(hao,1)

(shui,1)

(ni,2)

[bigdata@master01 ~]$ ls

2df8e0c3-174d-401a-b3a7-f7776c3987db checkpoint-1504685205000 data backup checkpoint-1504685205000.bk debug.log checkpoint-1504685199000 checkpoint-1504685208000 hadoop

checkpoint-1504685199000.bk checkpoint-1504685208000.bk receivedBlockMetadata checkpoint-1504685202000 checkpoint-1504685211000 software

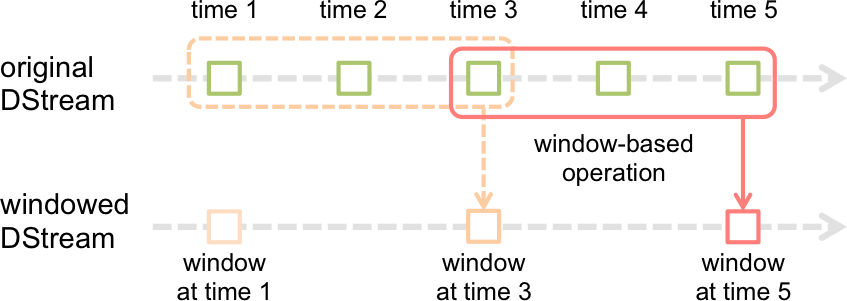
checkpoint-1504685202000.bk checkpoint-1504685211000.bk statefulwordcount-jar-with-

dependencies.jar

* + - 1. Window Operations

Window Operations 有点类似于 Storm 中的 State，可以设置窗口的大小和滑动窗口的间隔来动态的获取当前 Steaming 的允许状态。

基于窗口的操作会在一个比 StreamingContext 的批次间隔更长的时间范围内，通过整合多个批次的结果， 计算出整个窗口的结果。

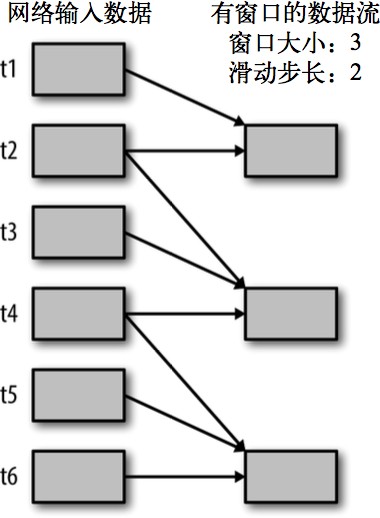


所有基于窗口的操作都需要两个参数， 分别为窗口时长以及滑动步长，两者都必须是 StreamContext 的批次间隔的整数倍。窗口时长控制每次计算最近的多少个批次的数据，其实就是最近的 windowDuration/batchInterval 个批次。如果有一个以 10 秒为批次间隔的源 DStream，要创建一个最近 30 秒的时间窗口(即最近 3 个批次)，就应当把 windowDuration 设为 30 秒。而滑动步长的默认值与批次间隔相等，用来控制对新的 DStream 进行计算的间隔。如果源 DStream 批次间隔为 10 秒，并且我们只希望每两个批次计算一次窗口结果， 就应该把滑动步长设置为 20 秒。

假设，你想拓展前例从而每隔十秒对持续 30 秒的数据生成 word count。为做到这个，我们需要在持续 30 秒数据的(word,1)对DStream 上应用reduceByKey。使用操作 reduceByKeyAndWindow.

# reduce last 30 seconds of data, every 10 second

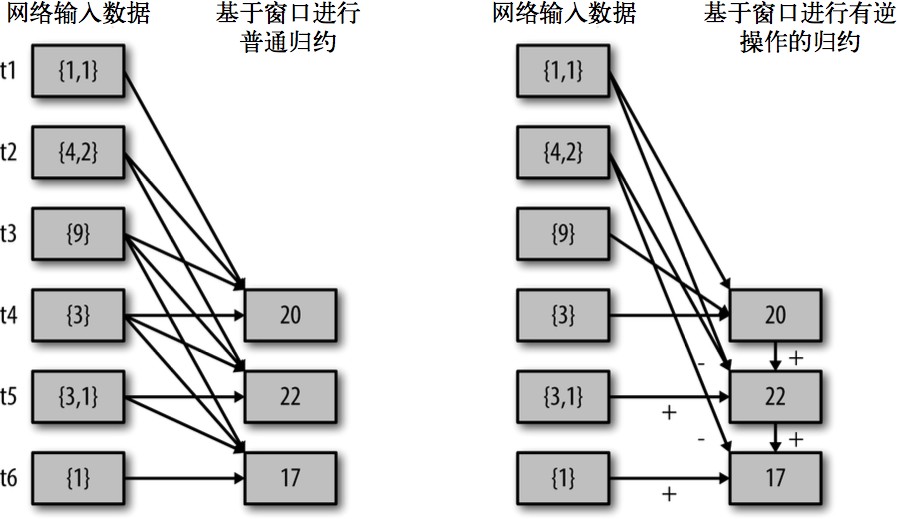
windowedWordCounts = pairs.reduceByKeyAndWindow(lambda x, y: x + y, lambda x, y: x -y, 30, 20)



|  |  |
| --- | --- |
| Transformation | Meaning |
| window**(*windowLength*, *slideInterval*)** | 基于对源 DStream 窗化的批次进行计算返回一个新的DStream |
| countByWindow**(*windowLength*, *slideInterval*)** | 返回一个滑动窗口计数流中的元素。 |
| reduceByWindow**(*func*, *windowLength*, *slideInterval*)** | 通过使用自定义函数整合滑动区间流元素来创建一个新的单元素流。 |
| reduceByKeyAndWindow**(*func*, *windowLength*, *slideI nterval*, [*numTasks*])** | 当在一个(K,V)对的DStream 上调用此函数，会返回一个新(K,V)对的 DStream， 此处通过对滑动窗口中批次数  据使用 reduce 函数来整合每个 |

|  |  |
| --- | --- |
|  | key 的 value 值。**Note:**默认情况下，这个操作使用 Spark 的默认数量并行任务(本地是2)，在集群模式中依据配置属性(spark.default.parallelism)来做 grouping。你可以通过设置可选参数 numTasks 来设置不同数量的 tasks。 |
| reduceByKeyAndWindow**(*func*, *invFunc*, *windowLeng th*, *slideInterval*, [*numTasks*])** | 这个函数是上述函数的更高效版本，每个窗口的 reduce 值都是通过用前一个窗的reduce 值来递增计算。通过reduce 进入到滑动窗口数据并”反向 reduce”离开窗口的旧数据来实现这个操作。一个例子是随着窗口滑动对 keys 的“加”“减”计数。通过前边介绍可以想到，这个函数只适用于”可逆的 reduce 函  数”，也就是这些 reduce 函数有相应的”反 reduce” 函数(以参数 invFunc 形式传入)。如前述函数，reduce 任务的数量通过可选参数来配置。注意： 为了使用这个操作，[检查点](http://spark.apache.org/docs/latest/streaming-programming-guide.html#checkpointing)必须可用。 |
| countByValueAndWindow**(*windowLength*,*slideInterv al*, [*numTasks*])** | 对(K,V)对的 DStream 调用，返回(K,Long)对的新DStream， 其中每个 key 的值是其在滑动窗口中频率。如 上，可配置 reduce 任务数量。 |

reduceByWindow() 和 reduceByKeyAndWindow() 让我们可以对每个窗口更高效地进行归约操作。它们接收一个归约函数，在整个窗口上执行，比如 +。除此以外，它们还有一种特殊形式，通过只考虑新进入窗口的数据和离开窗 口的数据，让 Spark 增量计算归约结果。这种特殊形式需要提供归约函数的一个逆函数，比 如 + 对应的逆函数为 -。对于较大的窗口，提供逆函数可以大大提高执行效率



val ipDStream = accessLogsDStream.map(logEntry => (logEntry.getIpAddress(), 1)) val ipCountDStream = ipDStream.reduceByKeyAndWindow(

{(x, y) => x + y},

{(x, y) => x - y},

Seconds(30),

Seconds(10))

// 加上新进入窗口的批次中的元素 // 移除离开窗口的老批次中的元素 // 窗口时长 // 滑动步长

countByWindow() 和 countByValueAndWindow() 作为对数据进行 计数操作的简写。countByWindow() 返回一个表示每个窗口中元素个数的 DStream， 而 countByValueAndWindow() 返回的 DStream 则包含窗口中每个值的个数，

val ipDStream = accessLogsDStream.map{entry => entry.getIpAddress()}

val ipAddressRequestCount = ipDStream.countByValueAndWindow(Seconds(30), Seconds(10)) val requestCount = accessLogsDStream.countByWindow(Seconds(30), Seconds(10))

WordCount 第三版：3 秒一个批次，窗口 12 秒，滑步 6 秒。

**package** com.atguigu.streaming

**import** org.apache.spark.SparkConf

**import** org.apache.spark.streaming.{Seconds, StreamingContext}

*/\*\**

*\* Created by wuyufei on 06/09/2017.*

*\*/*

**object** WorldCount{

**def** main(args: Array[String]) {

*//* 定义更新状态方法，参数*values* 为当前批次单词频度，*state* 为以往批次单词频度

**val** updateFunc = (values: Seq[Int], state: Option[Int]) => { **val** currentCount = values.foldLeft(0)(\_ + \_)

**val** previousCount = state.getOrElse(0)

*Some*(currentCount + previousCount)

}

**val** conf = **new** SparkConf().setMaster(**"local[2]"**).setAppName(**"NetworkWordCount"**)

**val** ssc = **new** StreamingContext(conf, *Seconds*(3))

ssc.checkpoint(**"."**)

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

**val** lines = ssc.socketTextStream(**"master01"**, 9999)

*// Split each line into words*

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

*//import org.apache.spark.streaming.StreamingContext.\_ // not necessary since Spark 1.3*

*// Count each word in each batch*

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

**val** wordCounts = pairs.reduceByKeyAndWindow((a:Int,b:Int) => (a + b),*Seconds*(12), *Seconds*(6))

*// 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*

*//ssc.stop()*

}

}

### 重要操作

* + - 1. Transform Operation

Transform 原语允许 DStream 上执行任意的 RDD-to-RDD 函数。即使这些函数并没有在 DStream 的 API 中暴露出来，通过该函数可以方便的扩展 Spark API。

该函数每一批次调度一次。

比如下面的例子，在进行单词统计的时候，想要过滤掉 spam 的信息。其实也就是对 DStream 中的 RDD 应用转换。

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

...

}

* + - 1. Join 操作

连接操作（ leftOuterJoin, rightOuterJoin, fullOuterJoin 也可以），可以连接

Stream-Stream，windows-stream to windows-stream、stream-dataset Stream-Stream Joins

val stream1: DStream[String, String] =

val stream2: DStream[String, String] =

val joinedStream = stream1.join(stream2)

val windowedStream1 = stream1.window(Seconds(20)) val windowedStream2 = stream2.window(Minutes(1))

val joinedStream = windowedStream1.join(windowedStream2)

Stream-dataset joins

val dataset: RDD[String, String] =

val windowedStream = stream.window(Seconds(20))

val joinedStream = windowedStream.transform { rdd => rdd.join(dataset) }

## DStreams 输出

输出操作指定了对流数据经转化操作得到的数据所要执行的操作(例如把结果推入外部数据库或输出到屏幕上)。与 RDD 中的惰性求值类似，如果一个 DStream 及其派生出的 DStream 都没有被执行输出操作， 那么这些DStream 就都不会被求值。如果 StreamingContext 中没有设定输出操作，整个context 就都不会启动。

|  |  |
| --- | --- |
| **Output Operation** | **Meaning** |
| **print**() | 在运行流程序的驱动结点上打印DStream 中每一批次数据的最开始 10 个元素。这用于开发和调试。在 Python API 中，同样的操作叫 pprint()。 |
| **saveAsTextFiles**(*prefix*, [*suffix*]) | 以 text 文 件 形 式 存 储 这 个DStream 的内容。每一批次的存储文件名基于参数中的 prefix 和 suffix。” prefix-Time\_IN\_MS[.suffix]”. |
| **saveAsObjectFiles**(*prefix*, [*suffix*]) | 以 Java 对 象 序 列 化 的 方 式 将Stream 中 的 数 据 保 存 为SequenceFiles . 每一批次的存储文件名 基 于 参 数 中 的 为 "prefix- TIME\_IN\_MS[.suffix]". Python 中目  前不可用。 |
| **saveAsHadoopFiles**(*prefix*, [*suffix*]) | 将 Stream 中 的 数 据 保 存 为Hadoop files. 每一批次的存储文件名基 于 参 数 中 的 为 "prefix- TIME\_IN\_MS[.suffix]".  Python API Python 中目前不可用。 |

|  |  |
| --- | --- |
| **foreachRDD**(*func*) | 这是最通用的输出操作，即将函数 func 用于产生于 stream 的每一个RDD。其中参数传入的函数 func 应该实现将每一个 RDD 中数据推送到外部系统，如将 RDD 存入文件或者通过网络将其写入数据库。注意：函数 func 在运行流应用的驱动中被执行，同时其中一般函数 RDD 操作从而强制其对于流 RDD 的运算。 |

通用的输出操作 foreachRDD()， 它用来对 DStream 中的 RDD 运行任意计算。这和 transform() 有些类似， 都 可以让我们访 问任意 RDD 。在foreachRDD() 中，可以重用我们在 Spark 中实现的所有行动操作。比如，常见的用例之一是把数据写到诸如 MySQL 的外部数据库中。

需要注意的：

* + 1. 连接不能写在 driver 层面
    2. 如果写在 foreach 则每个 RDD 都创建，得不偿失
    3. 增加 foreachPartition， 在分区创建
    4. 可以考虑使用连接池优化

dstream.foreachRDD { rdd =>

// error val connection = createNewConnection() // executed at the driver 序列化错误

rdd.foreachPartition { partitionOfRecords =>

// ConnectionPool is a static, lazily initialized pool of connections

val connection = ConnectionPool.getConnection()

partitionOfRecords.foreach(record => connection.send(record) // executed at the worker

)

ConnectionPool.returnConnection(connection) // return to the pool for future reuse

}

}

## 累加器和广播变量

累 加 器 (Accumulators) 和 广 播 变 量 (Broadcast variables) 不能从 Spark Streaming 的检查点中恢复。如果你启用检查并也使用了累加器和广播变量，那

么你必须创建累加器和广播变量的延迟单实例从而在驱动因失效重启后他们可以被重新实例化。如下例述：

**object** WordBlacklist {

@volatile **private var** *instance*: Broadcast[Seq[String]] = **null**

**def** getInstance(sc: SparkContext): Broadcast[Seq[String]] = { **if** (*instance* == **null**) {

synchronized {

**if** (*instance* == **null**) {

**val** wordBlacklist = *Seq*(**"a"**, **"b"**, **"c"**)

*instance* = sc.broadcast(wordBlacklist)

}

}

}

*instance*

}

}

**object** DroppedWordsCounter {

@volatile **private var** *instance*: LongAccumulator = **null**

**def** getInstance(sc: SparkContext): LongAccumulator = { **if** (*instance* == **null**) {

synchronized {

**if** (*instance* == **null**) {

*instance* = sc.longAccumulator(**"WordsInBlacklistCounter"**)

}

}

}

*instance*

}

}

wordCounts.foreachRDD { (rdd: RDD[(String, Int)], time: Time) =>

*// Get or register the blacklist Broadcast*

**val** blacklist = WordBlacklist.*getInstance*(rdd.sparkContext)

*// Get or register the droppedWordsCounter Accumulator*

**val** droppedWordsCounter = DroppedWordsCounter.*getInstance*(rdd.sparkContext)

*// Use blacklist to drop words and use droppedWordsCounter to count them*

**val** counts = rdd.filter { **case** (word, count) => **if** (blacklist.value.contains(word)) {

droppedWordsCounter.add(count) **false**

} **else** {

**true**

}

}.collect().mkString(**"["**, **", "**, **"]"**)

**val** output = **"Counts at time "** + time + **" "** + counts

})

## DataFrame ans SQL Operations

你可以很容易地在流数据上使用 DataFrames 和 SQL 。 你必须使用SparkContext 来创建 StreamingContext 要用的 SQLContext。此外，这一过程可以在驱动失效后重启。我们通过创建一个实例化的 SQLContext 单实例来实现这个工作。如下例所示。我们对前例 word count 进行修改从而使用 DataFrames 和 SQL 来产生 word counts。每个 RDD 被转换为 DataFrame，以临时表格配置并用 SQL 进行查询。

**val** words: DStream[String] = ...

words.foreachRDD { rdd =>

*// Get the singleton instance of SparkSession*

**val** spark = SparkSession.builder.config(rdd.sparkContext.getConf).getOrCreate()

**import** spark.implicits.\_

*// Convert RDD[String] to DataFrame*

**val** wordsDataFrame = rdd.toDF(**"word"**)

*// Create a temporary view*

wordsDataFrame.createOrReplaceTempView(**"words"**)

*// Do word count on DataFrame using SQL and print it*

**val** wordCountsDataFrame =

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

}

你也可以从不同的线程在定义于流数据的表上运行 SQL 查询（也就是说， 异步运行 StreamingContext）。仅确定你设置 StreamingContext 记住了足够数量的流数据以使得查询操作可以运行。否则， StreamingContext 不会意识到任何异步的 SQL 查询操作，那么其就会在查询完成之后删除旧的数据。例如，如果你要查询最后一批次， 但是你的查询会运行 5 分钟， 那么你需要调用streamingContext.remember(Minutes(5))(in Scala, 或者其他语言的等价操作)。

## Caching / Persistence

和 RDDs 类似，DStreams 同样允许开发者将流数据保存在内存中。也就是说，在 DStream 上使用 persist()方法将会自动把 DStreams 中的每个 RDD 保存在内存中。当 DStream 中的数据要被多次计算时，这个非常有用（如在同样数据上的多次操作）。对于像 reduceByWindow 和 reduceByKeyAndWindow 以及基于状态的(updateStateByKey) 这种操作，保存是隐含默认的。因此，即使开发者没有调用 persist()，由基于窗操作产生的 DStreams 会自动保存在内存中。

## 7x24 不间断运行

### 检查点机制

检查点机制是我们在 Spark Streaming 中用来保障容错性的主要机制。与应用程序逻辑无关的错误（即系统错位， JVM 崩溃等）有迅速恢复的能力.

它可以使 Spark Streaming 阶段性地把应用数据存储到诸如 HDFS 或Amazon S3 这样的可靠存储系统中， 以供恢复时使用。具体来说，检查点机制主要为以下两个目的服务。

* + - 1. 控制发生失败时需要重算的状态数。SparkStreaming 可以通 过转化图的谱系图来重算状态，检查点机制则可以控制需要在转化图中回溯多远。
      2. 提供驱动器程序容错。如果流计算应用中的驱动器程序崩溃了，你可以重启驱动器程序 并让驱动器程序从检查点恢复，这样 Spark Streaming 就可以读取之前运行的程序处理 数据的进度，并从那里继续。

了实现这个，Spark Streaming 需要为容错存储系统 *checkpoint* 足够的信息从而使得其可以从失败中恢复过来。有两种类型的数据设置检查点。

**Metadata checkpointing**：将定义流计算的信息存入容错的系统如 HDFS。元数据包括：

配置 – 用于创建流应用的配置。

*DStreams* 操作 – 定义流应用的 DStreams 操作集合。

不完整批次 – 批次的工作已进行排队但是并未完成。

**Data checkpointing**： 将产生的 RDDs 存入可靠的存储空间。对于在多批次间合并数据的状态转换，这个很有必要。在这样的转换中， RDDs 的产生基于之前批次的 RDDs，这样依赖链长度随着时间递增。为了避免在恢复期这种无限的时间增长（和链长度成比例），状态转换中间的 RDDs 周期性写入可靠地存储空间（如 HDFS）从而切短依赖链。

总而言之，元数据检查点在由驱动失效中恢复是首要需要的。而数据或者

RDD 检查点甚至在使用了状态转换的基础函数中也是必要的。

出于这些原因，检查点机制对于任何生产环境中的流计算应用都至关重要。你可以通过向 ssc.checkpoint() 方法传递一个路径参数(HDFS、S3 或者本地路径均可)来配置检查点机制,同时你的应用应该能够使用检查点的数据

* + - * 1. 当程序首次启动，其将创建一个新的 StreamingContext，设置所有的流并调用 start()。
        2. 当程序在失效后重启，其将依据检查点目录的检查点数据重新创建一个 StreamingContext。 通过使用 StraemingContext.getOrCreate 很容易获得这个性能。

ssc.checkpoint("hdfs://...")

# 创建和设置一个新的 StreamingContext def functionToCreateContext():

sc = SparkContext(...) # new context ssc = new StreamingContext(...)

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

...

ssc.checkpoint(checkpointDirectory) # 设置检查点目录

return ssc

# 从检查点数据中获取 StreamingContext 或者重新创建一个

context = StreamingContext.getOrCreate(checkpointDirectory, functionToCreateContext)

# 在需要完成的 context 上做额外的配置

# 无论其有没有启动

context ...

# 启动 context

context.start() contaxt.awaitTermination()

如果检查点目录(checkpointDirectory) 存在，那么 context 将会由检查点数

据重新创建。如果目录不存在（首次运行），那么函数functionToCreateContext 将会被调用来创建一个新的 context 并设置DStreams。

注意 RDDs 的检查点引起存入可靠内存的开销。在 RDDs 需要检查点的批次里，处理的时间会因此而延长。所以，检查点的间隔需要很仔细地设 置。在小尺寸批次（1 秒钟）。每一批次检查点会显著减少操作吞吐量。反之，检查点设置的过于频繁导致“ 血统” 和任务尺寸增长，这会有很不好的

影响对于需要 RDD 检查点设置的状态转换，默认间隔是批次间隔的乘数一般至少为 10 秒钟。可以通过 dstream.checkpoint(checkpointInterval)。通常， 检查点设置间隔是 5-10 个 DStream 的滑动间隔。

### WAL 预写日志

WAL 即 write ahead log（预写日志），是在 1.2 版本中就添加的特性。作用就是，将数据通过日志的方式写到可靠的存储，比如 HDFS、s3，在 driver 或worker failure 时可以从在可靠存储上的日志文件恢复数据。WAL 在 driver 端和executor 端都有应用。

**WAL** 在 **driver** 端的应用

用于写日志的对象 writeAheadLogOption: WriteAheadLog。在StreamingContext 中的 JobScheduler 中的 ReceiverTracker 的 ReceivedBlockTracker 构造函数中被创建，ReceivedBlockTracker 用于管理已接收到的 blocks 信息。需要注意的是，这里只需要启用 checkpoint 就可以创建该 driver 端的 WAL 管理实

例，而不需要将 spark.streaming.receiver.writeAheadLog.enable 设置为 true。

写什么、何时写、写多少**,**

首选需要明确的是，ReceivedBlockTracker 通过 WAL 写入 log 文件的内容是

3 种事件（当然，会进行序列化）：

kafka 直连方式，Receiver 方式。

* case class BlockAdditionEvent(receivedBlockInfo: ReceivedBlockInfo)；即新增了一个 block 及该 block 的具体信息，包括streamId、blockId、数据条数等
* case class BatchAllocationEvent(time: Time, allocatedBlocks: AllocatedBlocks)；即为某个 batchTime 分配了哪些 blocks 作为该 batch RDD 的数据源
* case class BatchCleanupEvent(times: Seq[Time])；即清理了哪些batchTime 对应的 block
* 知道了写了什么内容，结合源码，也不难找出是什么时候写了这些内容。需要再次注意的是，写上面这三种事件，也不需要将spark.streaming.receiver.writeAheadLog.enable 设置为 true。

**WAL** 在 **executor** 端的应用

Receiver 接收到的数据会源源不断的传递给 ReceiverSupervisor，是否启用WAL 机制（即是否将 spark.streaming.receiver.writeAheadLog.enable 设置为true）会影响 ReceiverSupervisor 在存储 block 时的行为：

* 不启用 WAL：你设置的 StorageLevel 是什么，就怎么存储。比如MEMORY\_ONLY 只会在内存中存一份，MEMORY\_AND\_DISK 会在内存和磁盘上各存一份等
* 启用 WAL：在 StorageLevel 指定的存储的基础上，写一份到 WAL 中。存储一份在 WAL 上，更不容易丢数据但性能损失也比较大

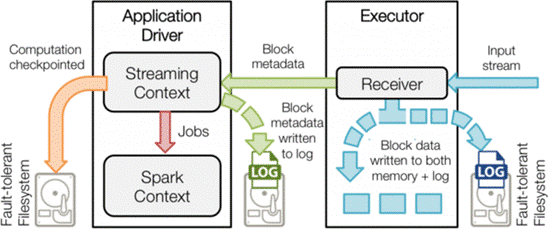
关于是否要启用 WAL，要视具体的业务而定：

* 若可以接受一定的数据丢失，则不需要启用 WAL，因为对性能影响较大
* 若完全不能接受数据丢失，那就需要同时启用 checkpoint 和 WAL，
* 重点：面试中问：sparkStreaming 读取 kafka 数据如何保证数据不丢失：
* 1： kafka 使用低消费者模式，自己维护偏移量，将 offset 保存在 redis 中，
* 2： 使用Receiver 连接模式进行数据读取。
* 3：sparkStraming 需要同时启用 checkpoint 和 WAL
* checkpoint 保存着执行进度（比如已生成但未完成的 jobs），WAL 中保存着 blocks 及 blocks 元数据（比如保存着未完成的 jobs 对应的 blocks 信息及 block 文件）。同时，这种情况可能要在数据源和 Streaming Application 中联合来保证 exactly once 语义

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。

1. 接收器通知 driver。
2. 接收块中的元数据（ metadata）被发送到 driver 的 StreamingContext。这个元数据包括：
3. 定位其在 executor 内存中数据的块 referenceid，
4. 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。

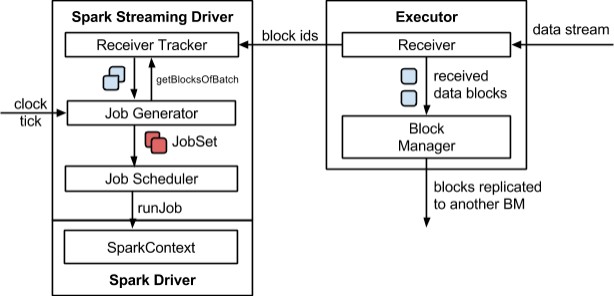
### 背压机制

默认情况下，Spark Streaming 通过 Receiver 以生产者生产数据的速率接收数据，计算过程中会出现 batch processing time > batch interval 的情况， 其中batch processing time 为实际计算一个批次花费时间， batch interval 为Streaming 应用设置的批处理间隔。这意味着 Spark Streaming 的数据接收速率高于 Spark 从队列中移除数据的速率，也就是数据处理能力低，在设置间隔内不能完全处理当前接收速率接收的数据。如果这种情况持续过长的时间，会造成数据在内存中堆积，导致 Receiver 所在 Executor 内存溢出等问题（如果设置StorageLevel 包含 disk, 则内存存放不下的数据会溢写至 disk, 加大延迟）。Spark 1.5 以前版本，用户如果要限制 Receiver 的数据接收速率，可以通过设置静态配制参数“spark.streaming.receiver.maxRate”的值来实现，此举虽然可以通过限制接收速率，来适配当前的处理能力，防止内存溢出，但也会引入其它问题。比如：producer 数据生产高于 maxRate，当前集群处理能力也高于 maxRate，

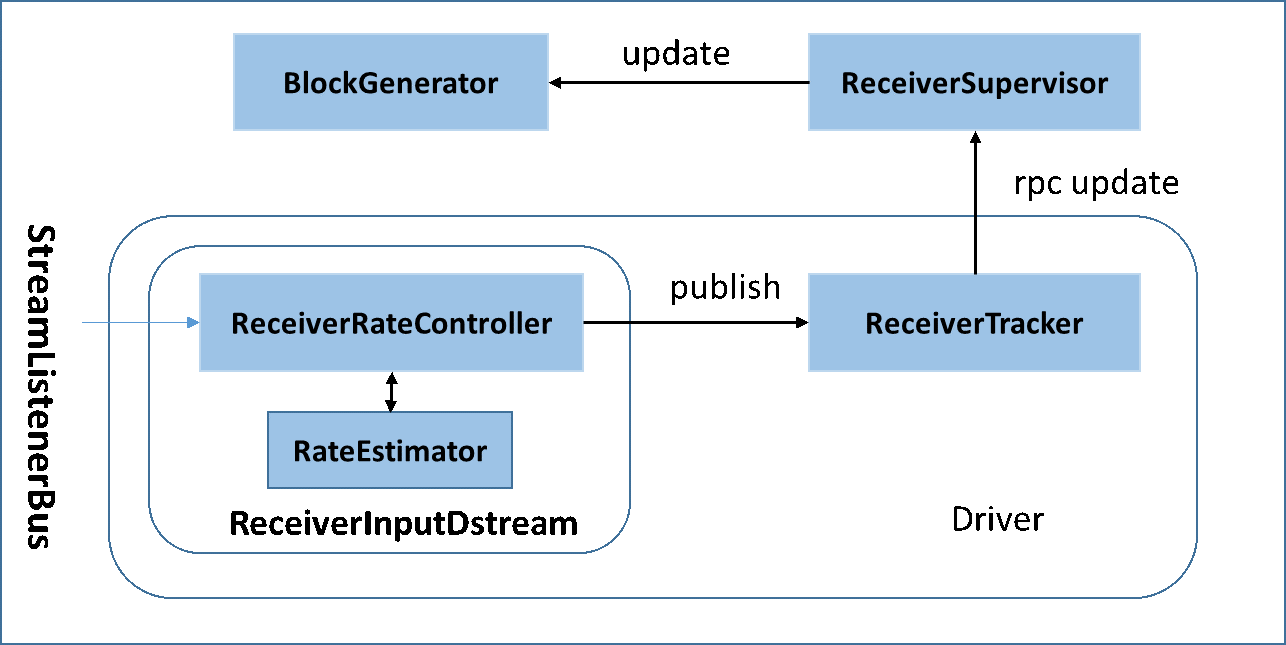
这就会造成资源利用率下降等问题。为了更好的协调数据接收速率与资源处理能力，Spark Streaming 从 v1.5 开始引入反压机制（ back-pressure）,通过动态控制数据接收速率来适配集群数据处理能力。

Spark Streaming Backpressure: 根据 JobScheduler 反馈作业的执行信息来动态调整 Receiver 数据接收率。通过属性“spark.streaming.backpressure.enabled” 来控制是否启用 backpressure 机制，默认值 false，即不启用。

Streaming 架构如下图所示



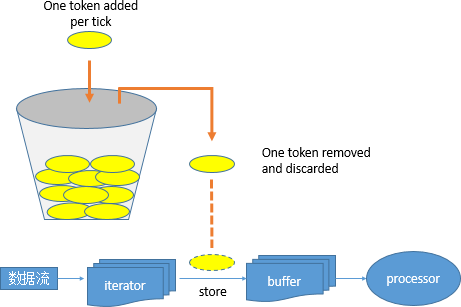
在原架构的基础上加上一个新的组件 RateController, 这个组件负责监听“OnBatchCompleted”事件， 然后从中抽取 processingDelay 及 schedulingDelay 信息. Estimator 依据这些信息估算出最大处理速度（ rate） ， 最后由基于Receiver 的 Input Stream 将 rate 通过 ReceiverTracker 与 ReceiverSupervisorImpl 转发给 BlockGenerator（继承自 RateLimiter）.



流量控制点

当 Receiver 开始接收数据时，会通过 supervisor.pushSingle()方法将接收的数据存入 currentBuffer 等待 BlockGenerator 定时将数据取走，包装成 block. 在将数据存放入 currentBuffer 之时，要获取许可（令牌）。如果获取到许可就可以将数据存入 buffer, 否则将被阻塞，进而阻塞 Receiver 从数据源拉取数据。

其令牌投放采用令牌桶机制进行， 原理如下图所示:



令牌桶机制： 大小固定的令牌桶可自行以恒定的速率源源不断地产生令牌。如果令牌不被消耗，或者被消耗的速度小于产生的速度，令牌就会不断地增多，直到把桶填满。后面再产生的令牌就会从桶中溢出。最后桶中可以保存的最大令牌数永远不会超过桶的大小。当进行某操作时需要令牌时会从令牌桶中取出相应的令牌数，如果获取到则继续操作，否则阻塞。用完之后不用放回。

### 驱动器程序容错

驱动器程序的容错要求我们以特殊的方式创建 StreamingContext。我们需要把检查点目录提供给 StreamingContext。与直接调用 new StreamingContext 不同，应该使用 StreamingContext.getOrCreate() 函数。

配置过程如下：

1、 启动 Driver 自动重启功能

* standalone: 提交任务时添加 --supervise 参数
* yarn:设置 yarn.resourcemanager.am.max-attempts 或者spark.yarn.maxAppAttempts
* mesos: 提交任务时添加 --supervise 参数

2、 设置 checkpoint StreamingContext.setCheckpoint(hdfsDirectory)

3、支持从 checkpoint 中重启配置

def createContext(checkpointDirectory: String): StreamingContext = { val ssc = new StreamingContext ssc.checkpoint(checkpointDirectory)

ssc

}

val ssc = StreamingContext.getOrCreate(checkpointDirectory, createContext(checkpointDirectory))

### 工作节点容错

为了应对工作节点失败的问题，Spark Streaming 使用与 Spark 的容错机制相同的方法。所 有从外部数据源中收到的数据都在多个工作节点上备份。所有从备份数据转化操作的过程 中创建出来的 RDD 都能容忍一个工作节点的失败，因为根据 RDD 谱系图， 系统可以把丢 失的数据从幸存的输入数据备份中重算出来。对于 reduceByKey 等 Stateful 操作重做的 lineage 较长的，强制启动 checkpoint，减少重做几率

### 接收器容错

运行接收器的工作节点的容错也是很重要的。如果这样的节点发生错误， Spark Streaming 会在集群中别的节点上重启失败的接收器。然而，这种情况会不会导致数据的丢失取决于 数据源的行为(数据源是否会重发数据)以及接收器的实现(接收器是否会向数据源确认 收到数据)。举个例子，使用 Flume 作为数据源时，两种接收器的主要区别在于数据丢失 时的保障。在“接收器从数据池中拉取数据”的模型中，Spark 只会在数据已经在集群中 备份时才会从数据池中移除元素。而在“向接收器推数据”的模型中，如果接收器在数据 备份之前失败，一些数据可能就会丢失。总的来说，对于任意一个接收器，你必须同时考 虑上游数据源的容错性(是否支持事务)来确保零数据丢失。

一般主要是通过将接收到数据后先写日志（ WAL）到可靠文件系统中，后才写入实际的 RDD。如果后续处理失败则成功写入 WAL 的数据通过 WAL 进行恢复，未成功写入 WAL 的数据通过可回溯的 Source 进行重放

总的来说，接收器提供以下保证。

* 所有从可靠文件系统中读取的数据(比如通过 StreamingContext.hadoopFiles 读取的) 都是可靠的，因为底层的文件系统是有备份的。Spark Streaming 会记住哪些数据存放到 了检查点中，并在应用崩溃后从检查点处继续执行。
* 对于像 Kafka、推式 Flume、Twitter 这样的不可靠数据源，Spark 会把输入数据复制到其 他节点上，但是如果接收器任务崩溃， Spark 还是会丢失数据。在 Spark 1.1 以及更早的版 本中，收到的数据只被备份到执行器进程的内存中，所以一旦驱动器程序崩溃(此时所 有的执行器进程都会丢失连接)， 数据也会丢失。在 Spark 1.2 中，收到的数据被记录到诸 如 HDFS 这样的可靠的文件系统中， 这样即使驱动器程序重启也不会导致数据丢失。

综上所述， 确保所有数据都被处理的最佳方式是使用可靠的数据源( 例如HDFS、拉式 Flume 等)。如果你还要在批处理作业中处理这些数据，使用可靠数据源是最佳方式， 因为 这种方式确保了你的批处理作业和流计算作业能读取到相同的数据，因而可以得到相同的结果。

##### 操作过程如下：

* 启用 checkpoint ssc.setCheckpoint(checkpointDir)
* 启用 WAL sparkConf.set("spark.streaming.receiver.writeAheadLog.enable", "true")
* 对 Receiver 使用可靠性存储 StoreageLevel.MEMORY\_AND\_DISK\_SER or StoreageLevel.MEMORY\_AND\_DISK\_SER2

### 处理保证

由于 Spark Streaming 工作节点的容错保障， Spark Streaming 可以为所有的转化操作提供 “精确一次”执行的语义， 即使一个工作节点在处理部分数据时发生失败，最终的转化结

果(即转化操作得到的 RDD)仍然与数据只被处理一次得到的结果一样。然而， 当把转化操作得到的结果使用输出操作推入外部系统中时， 写结果

的任务可能因故 障而执行多次，一些数据可能也就被写了多次。由于这引入

了外部系统， 因此我们需要专 门针对各系统的代码来处理这样的情况。我们 可以使用事务操作来写入外部系统(即原子 化地将一个 RDD 分区一次写入)，

或者设计幂等的更新操作(即多次运行同一个更新操作 仍生成相同的结果)。比如 Spark Streaming 的 saveAs...File 操作会在一个文件写完时自动 将其原子化地移动到最终位置上，以此确保每个输出文件只存在一份。

## 性能考量

最常见的问题是 Spark Streaming 可以使用的最小批次间隔是多少。总的来说，500 毫秒已经被证实为对许多应用而言是比较好的最小批次大小。寻找最小批次大小的最佳实践是从一个比较大的批次大小(10 秒左右)开始，不断使用更小的批次大小。如果 Streaming 用 户界面中显示的处理时间保持不变，你就可以进一步减小批次大小。如果处理时间开始增 加， 你可能已经达到了应用的极限。

相似地，对于窗口操作，计算结果的间隔(也就是滑动步长)对于性能也有巨大的影响。 当计算代价巨大并成为系统瓶颈时，就应该考虑提高滑动步长了。2

减少批处理所消耗时间的常见方式还有提高并行度。有以下三种方式可以提高并行度：

* 增加接收器数目 有时如果记录太多导致单台机器来不及读入并分发的话，接收器会成为系统瓶颈。这时 你就需要通过创建多个输入 DStream(这样会创建多个接收器)来增加接收器数目，然 后使用 union 来把数据合并为一个数据源。
* 将收到的数据显式地重新分区如果接收器数目无法再增加， 你可以通过使用 DStream.repartition 来显式重新分区输 入流( 或者合并多个流得到的数据流)来重新分配收到的数据。
* 提高聚合计算的并行度 对于像 reduceByKey() 这样的操作，你可以在第二个参数中指定并行度，我们在介绍 RDD 时提到过类似的手段。

##### reduceByKey((x,y)=>{x+y},50):50 值得是并行度。

# 第5章 高级解析

## DStreamGraph 对象解析

在 Spark Streaming 中，DStreamGraph 是一个非常重要的组件，主要用来：

* + 1. 通过成员 inputStreams 持有 Spark Streaming 输入源及接收数据的方式
    2. 通过成员 outputStreams 持有 Streaming app 的 output 操作，并记录DStream 依赖关系
    3. 生成每个 batch 对应的 jobs

下面，通过分析一个简单的例子，结合源码分析来说明 DStreamGraph 是如何发挥作用的。例子如下：

val sparkConf = new SparkConf().setAppName("HdfsWordCount")

val ssc = new StreamingContext(sparkConf, Seconds(2))

val lines = ssc.textFileStream(args(0)) val words = lines.flatMap(\_.split(" "))

val wordCounts = words.map(x => (x, 1)).reduceByKey(\_ + \_) wordCounts.print()

ssc.start() ssc.awaitTermination()

创建 **DStreamGraph** 实例

代码 val ssc = new StreamingContext(sparkConf, Seconds(2)) 创 建 了StreamingContext 实例，StreamingContext 包含了 DStreamGraph 类型的成员graph，graph 在 StreamingContext 主构造函数中被创建，如下

private[streaming] val graph: DStreamGraph = { if (isCheckpointPresent) {

cp\_.graph.setContext(this) cp\_.graph.restoreCheckpointData() cp\_.graph

} else {

require(batchDur\_ != null, "Batch duration for StreamingContext cannot be null")

val newGraph = new DStreamGraph() newGraph.setBatchDuration(batchDur\_) newGraph

}

}

可以看到，若当前 checkpoint 可用，会优先从 checkpoint 恢复 graph，否

则新建一个。还可以从这里知道的一点是： graph 是运行在 driver 上的

**DStreamGraph** 记录输入源及如何接收数据

DStreamGraph 有和 application 输入数据相关的成员和方法，如下：

private val inputStreams = new ArrayBuffer[InputDStream[\_]]() def addInputStream(inputStream: InputDStream[\_]) {

this.synchronized { inputStream.setGraph(this) inputStreams += inputStream

}

}

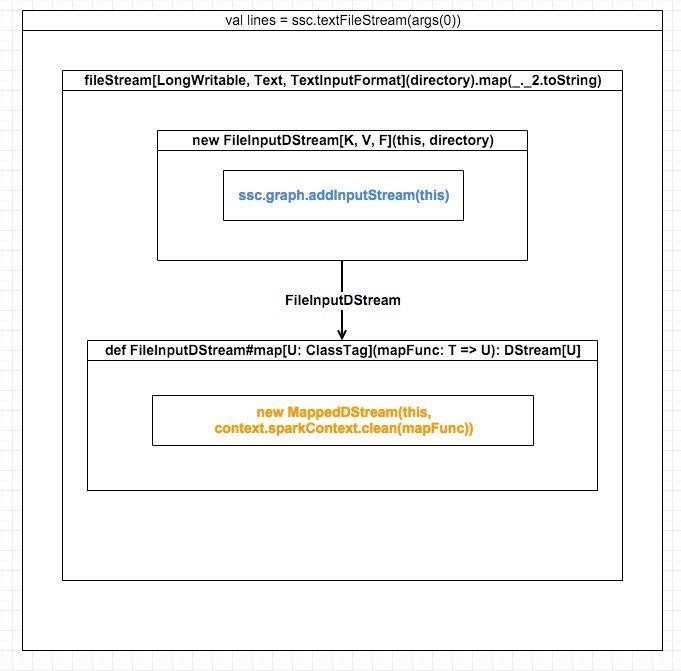
成员 inputStreams 为 InputDStream 类型的数组， InputDStream 是所有

input streams( 数据输入流) 的虚基类。该类提供了 start() 和 stop() 方法供streaming 系统来开始和停止接收数据。那些只需要在 driver 端接收数据并转成 RDD 的 input streams 可 以 直 接 继 承 InputDStream ， 例 如FileInputDStream 是 InputDStream 的子类，它监控一个 HDFS 目录并将新文件转成 RDDs。而那些需要在 workers 上运行 receiver 来接收数据的 Input DStream， 需要继承 ReceiverInputDStream，比如 KafkaReceiver。

我们来看看 val lines = ssc.textFileStream(args(0)) 调用。

为了更容易理解，我画出了 val lines = ssc.textFileStream(args(0)) 的调用流

程



从上面的调用流程图我们可以知道：

1. ssc.textFileStream 会触发新建一个 FileInputDStream。FileInputDStream 继承于 InputDStream，其 start()方法定义了数据源及如何接收数据
2. 在 FileInputDStream 构造函数中，会调用ssc.graph.addInputStream(this)，将自身添加到 DStreamGraph 的inputStreams: ArrayBuffer[InputDStream[\_]] 中，这样 DStreamGraph 就知道了这个 Streaming App 的输入源及如何接收数据。可能你会奇怪为什么inputStreams 是数组类型，举个例子，这里再来一个 val lines1 = ssc.textFileStream(args(0))，那么又将生成一个 FileInputStream 实例添加到 inputStreams，所以这里需要集合类型
3. 生成 FileInputDStream 调用其 map 方法，将以 FileInputDStream 本身作为 partent 来构造新的 MappedDStream。对于 DStream 的 transform 操作，都将生成一个新的 DStream，和 RDD transform 生成新的 RDD 类似

与 MappedDStream 不同，所有继承了 InputDStream 的定义了输入源及接收数据方式的 sreams 都没有 parent，因为它们就是最初的 streams。

**DStream** 的依赖链

每个 DStream 的子类都会继承 def dependencies: List[DStream[\_]] = List() 方法，该方法用来返回自己的依赖的父 DStream 列表。比如，没有父 DStream 的 InputDStream 的 dependencies 方法返回 List()。

MappedDStream 的实现如下：

class MappedDStream[T: ClassTag, U: ClassTag] ( parent: DStream[T],

mapFunc: T => U

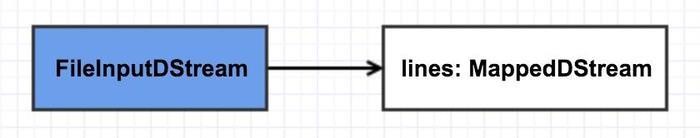
) extends DStream[U](parent.ssc) {

override def dependencies: List[DStream[\_]] = List(parent)

...

}

在上例中，构造函数参数列表中的 parent 即在 ssc.textFileStream 中 new

的定义了输入源及数据接收方式的最初的 FileInputDStream 实例， 这里的dependencies 方法将返回该 FileInputDStream 实例，这就构成了第一条依赖。可用如下图表示，这里特地将 input streams 用蓝色表示， 以强调其与普通由transform 产生的 DStream 的不同：

继续来看 val words = lines.flatMap(\_.split(" "))， flatMap 如下：

def flatMap[U: ClassTag](flatMapFunc: T => Traversable[U]): DStream[U]

= ssc.withScope {

new FlatMappedDStream(this, context.sparkContext.clean(flatMapFunc))

}

每一个 transform 操作都将创建一个新的 DStream， flatMap 操作也不例

外，它会创建一个 FlatMappedDStream， FlatMappedDStream 的实现如下：

class FlatMappedDStream[T: ClassTag, U: ClassTag]( parent: DStream[T],

flatMapFunc: T => Traversable[U]

) extends DStream[U](parent.ssc) {

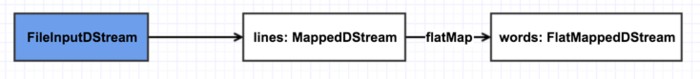
override def dependencies: List[DStream[\_]] = List(parent)

...

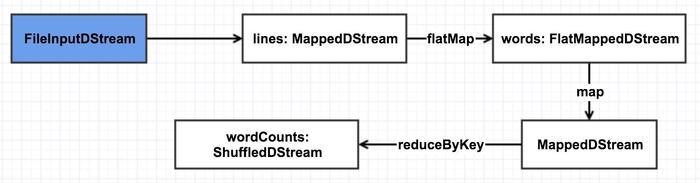
}

与 MappedDStream 相同，FlatMappedDStream#dependencies 也返回其依赖

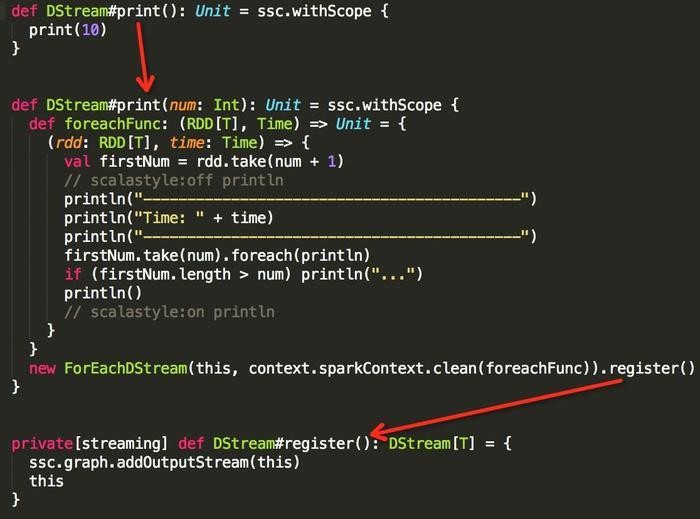
的父 DStream，及 lines，到这里，依赖链就变成了下图：



之后的几步操作不再这样具体分析，到生成 wordCounts 时，依赖图将变成下面这样：



在 DStream 中，与 transofrm 相对应的是 output 操作，包括 print, saveAsTextFiles, saveAsObjectFiles, saveAsHadoopFiles, foreachRDD。output 操作中，会创建 ForEachDStream 实例并 调用 register 方法将自身添加到DStreamGraph.outputStreams 成员中，该 ForEachDStream 实例也会持有是调用的哪个 output 操作。本例的代码调用如下，只需看箭头所指几行代码

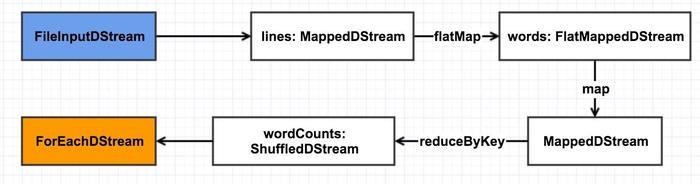


与 DStream transform 操作返回一个新的 DStream 不同，output 操作不会

返回任何东西，只会创建一个 ForEachDStream 作为依赖链的终结。

至此，生成了完成的依赖链，也就是 DAG，如下图（这里将 ForEachDStream

标为黄色以显示其与众不同）：



## ReceiverTracker 与数据导入

Spark Streaming 在数据接收与导入方面需要满足有以下三个特点：

* + 1. 兼容众多输入源，包括 HDFS, Flume, Kafka, Twitter and ZeroMQ。还可以自定义数据源
    2. 要能为每个 batch 的 RDD 提供相应的输入数据
    3. 为适应 7\*24h 不间断运行，要有接收数据挂掉的容错机制

有容乃大，兼容众多数据源

InputDStream 是所有 input streams( 数据输入流) 的虚基类。该类提供了start() 和 stop()方法供 streaming 系统来开始和停止接收数据。那些只需要在driver 端 接 收 数 据 并 转 成 RDD 的 input streams 可 以 直 接 继 承InputDStream，例如 FileInputDStream 是 InputDStream 的子类， 它监控一个HDFS 目录并将新文件转成 RDDs。而那些需要在 workers 上运行 receiver 来接收数据的 Input DStream ， 需 要 继 承 ReceiverInputDStream ， 比 如KafkaReceiver

只需在 driver 端接收数据的 input stream 一般比较简单且在生产环境中使用的比较少，本文不作分析，只分析继承了 ReceiverInputDStream 的 input stream 是如何导入数据的。

ReceiverInputDStream 有一个 def getReceiver(): Receiver[T]方法，每个继承了 ReceiverInputDStream 的 input stream 都必须实现这个方法。该方法用来获取将要分发到各个 worker 节点上用来接收数据的 receiver（ 接收器）。不同的 ReceiverInputDStream 子 类 都 有 它 们 对 应 的 不 同 的 receiver ，如KafkaInputDStream 对应 KafkaReceiver，FlumeInputDStream 对应 FlumeReceiver， TwitterInputDStream 对应 TwitterReceiver， 如果你要实现自己的数据源，也需要定义相应的 receiver。

继承 ReceiverInputDStream 并定义相应的 receiver，就是 Spark Streaming

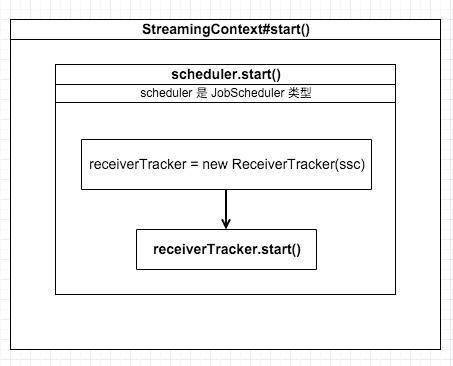
能兼容众多数据源的原因。

为每个 **batch** 的 **RDD** 提供输入数据

在 StreamingContext 中，有一个重要的组件叫做 ReceiverTracker，它是Spark Streaming 作业调度器 JobScheduler 的成员，负责启动、管理各个receiver 及管理各个 receiver 接收到的数据。

确定 **receiver** 要分发到哪些 **executors** 上执行创建 **ReceiverTracker** 实例

我们来看 StreamingContext#start() 方法部分调用实现，如下：

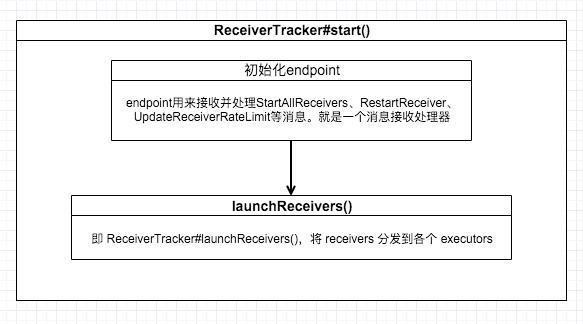


可以看到，StreamingContext#start() 会调用 JobScheduler#start() 方法，在JobScheduler#start() 中 ， 会 创 建 一 个 新 的 ReceiverTracker 实 例receiverTracker，并调用其 start() 方法。

ReceiverTracker#start()

继续跟进 ReceiverTracker#start()，如下图，它主要做了两件事：

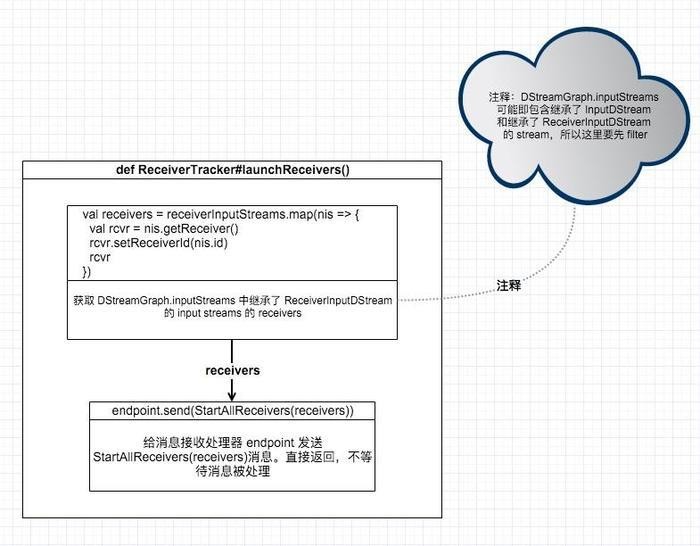
1. 初始化一个 endpoint: ReceiverTrackerEndpoint，用来接收和处理来自ReceiverTracker 和 receivers 发送的消息
2. 调用 launchReceivers 来自将各个 receivers 分发到 executors 上



ReceiverTracker#launchReceivers()

继续跟进 launchReceivers，它也主要干了两件事：

1. 获取 DStreamGraph.inputStreams 中继承了 ReceiverInputDStream 的input streams 的 receivers。也就是数据接收器
2. 给消息接收处理器 endpoint 发送 StartAllReceivers(receivers)消息。直接返回，不等待消息被处理



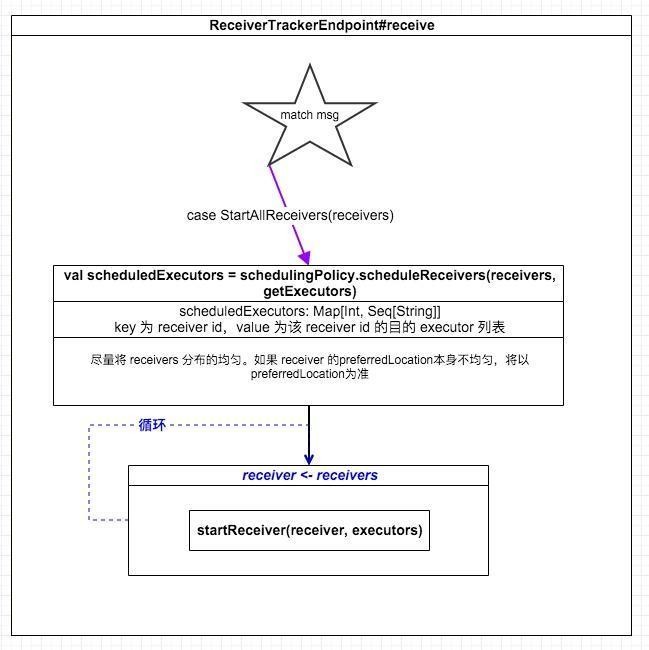
处理 **StartAllReceivers** 消息

endpoint 在接收到消息后，会先判断消息类型，对不同的消息做不同处理。对于 StartAllReceivers 消息，处理流程如下：

计算每个 receiver 要分发的目的 executors。遵循两条原则：

* 将 receiver 分布的尽量均匀
* 如果 receiver 的 preferredLocation 本身不均匀，以preferredLocation 为准

遍历每个 receiver，根据第 1 步中得到的目的 executors 调用startReceiver 方法



到这里，已经确定了每个 receiver 要分发到哪些 executors 上

启动 **receivers**

接上，通过 ReceiverTracker#startReceiver(receiver: Receiver[\_], scheduledExecutors: Seq[String]) 来启动 receivers，我们来看具体流程：



如上流程图所述，分发和启动 receiver 的方式不可谓不精彩。其中，

startReceiverFunc 函数主要实现如下：

val supervisor = new ReceiverSupervisorImpl(

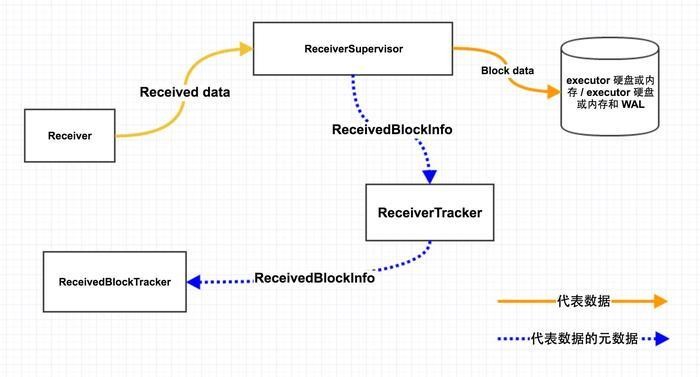
receiver, SparkEnv.get, serializableHadoopConf.value, checkpointDirOption)

supervisor.start() supervisor.awaitTermination()

supervisor.start() 中会调用 receiver#onStart 后立即返回。receiver#onStart 一般自行新建线程或线程池来接收数据， 比如在 KafkaReceiver 中，就新建了线程池，在线程池中接收 topics 的数据。

supervisor.start() 返回后，由 supervisor.awaitTermination() 阻塞住线程， 以让这个 task 一直不退出，从而可以源源不断接收数据。

数据流转



上图为 receiver 接收到的数据的流转过程，让我们来逐一分析

Step1: Receiver -> ReceiverSupervisor

这一步中，Receiver 将接收到的数据源源不断地传给

ReceiverSupervisor。Receiver 调用其 store(...) 方法，store 方法中继续调用supervisor.pushSingle 或 supervisor.pushArrayBuffer 等方法来传递数据。Receiver#store 有多重形式， ReceiverSupervisor 也有 pushSingle、pushArrayBuffer、pushIterator、pushBytes 方法与不同的 store 对应。

* pushSingle: 对应单条小数据
* pushArrayBuffer: 对应数组形式的数据
* pushIterator: 对应 iterator 形式数据
* pushBytes: 对应 ByteBuffer 形式的块数据

对于细小的数据，存储时需要 BlockGenerator 聚集多条数据成一块，然后再成块存储；反之就不用聚集， 直接成块存储。当然，存储操作并不在Step1 中执行，只为说明之后不同的操作逻辑。

Step2.1: ReceiverSupervisor -> BlockManager -> disk/memory

在这一步中，主要将从 receiver 收到的数据以 block（数据块）的形式存

储

存储 block 的是 receivedBlockHandler: ReceivedBlockHandler，根据参数

spark.streaming.receiver.writeAheadLog.enable 配置的不同，默认为 false，

receivedBlockHandler 对象对应的类也不同，如下：

private val receivedBlockHandler: ReceivedBlockHandler = { if (WriteAheadLogUtils.enableReceiverLog(env.conf)) {

//< 先写 WAL，再存储到 executor 的内存或硬盘

new WriteAheadLogBasedBlockHandler(env.blockManager, receiver.streamId,

receiver.storageLevel, env.conf, hadoopConf, checkpointDirOption.get)

} else {

//< 直接存到 executor 的内存或硬盘

new BlockManagerBasedBlockHandler(env.blockManager, receiver.storageLevel)

}

}

启动 WAL 的好处就是在 application 挂掉之后，可以恢复数据。

//< 调用 receivedBlockHandler.storeBlock 方法存储 block，并得到一个blockStoreResult

val blockStoreResult = receivedBlockHandler.storeBlock(blockId, receivedBlock)

//< 使用 blockStoreResult 初始化一个 ReceivedBlockInfo 实例

val blockInfo = ReceivedBlockInfo(streamId, numRecords, metadataOption, blockStoreResult)

//< 发送消息通知 ReceiverTracker 新增并存储了 block trackerEndpoint.askWithRetry[Boolean](AddBlock(blockInfo))

不管是 WriteAheadLogBasedBlockHandler 还是BlockManagerBasedBlockHandler 最终都是通过 BlockManager 将 block 数据存储 execuor 内存或磁盘或还有 WAL 方式存入。

这里需要说明的是 streamId，每个 InputDStream 都有它自己唯一的 id， 即 streamId， blockInfo 包含 streamId 是为了区分 block 是哪个 InputDStream 的数据。之后为 batch 分配 blocks 时，需要知道每个InputDStream 都有哪些未分配的 blocks。

Step2.2: ReceiverSupervisor -> ReceiverTracker

将 block 存储之后，获得 block 描述信息 blockInfo: ReceivedBlockInfo，这里面包含： streamId、数据位置、数据条数、数据 size 等信息。

之后，封装以 block 作为参数的 AddBlock(blockInfo) 消息并发送给

ReceiverTracker 以通知其有新增 block 数据块。

Step3: ReceiverTracker -> ReceivedBlockTracker

ReceiverTracker 收到 ReceiverSupervisor 发来的 AddBlock(blockInfo) 消息后，直接调用以下代码将 block 信息传给 ReceivedBlockTracker：

private def addBlock(receivedBlockInfo: ReceivedBlockInfo): Boolean = { receivedBlockTracker.addBlock(receivedBlockInfo)

}

receivedBlockTracker.addBlock 中，如果启用了 WAL，会将新增的 block

信息以 WAL 方式保存。

无论 WAL 是否启用，都会将新增的 block 信息保存到streamIdToUnallocatedBlockQueues: mutable.HashMap[Int, ReceivedBlockQueue]中， 该变量 key 为 InputDStream 的唯一 id， value 为已存储未分配的 block 信息。之后为 batch 分配 blocks，会访问该结构来获取每个 InputDStream 对应的未消费的 blocks。

## 动态生成 JOB

JobScheduler 有两个重要成员，一是 ReceiverTracker，负责分发 receivers 及源源不断地接收数据；二是 JobGenerator，负责定时的生成 jobs 并checkpoint。

定时逻辑

在 JobScheduler 的主构造函数中，会创建 JobGenerator 对象。在

JobGenerator 的主构造函数中，会创建一个定时器：

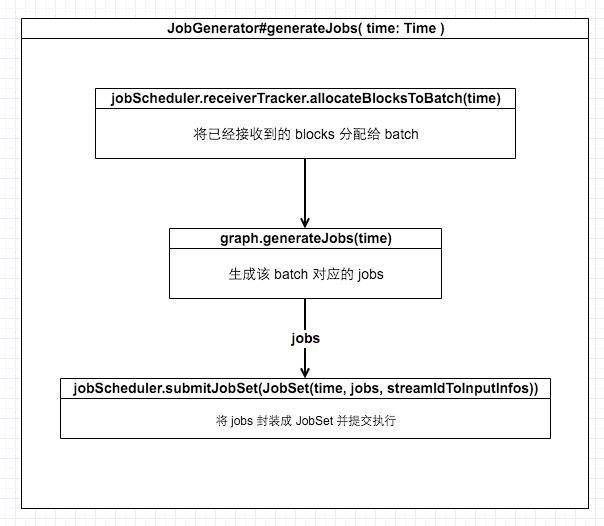
private val timer = new RecurringTimer(clock, ssc.graph.batchDuration.milliseconds,

longTime => eventLoop.post(GenerateJobs(new Time(longTime))), "JobGenerator")

该定时器每隔 ssc.graph.batchDuration.milliseconds 会执行一次eventLoop.post(GenerateJobs(new Time(longTime))) 向 eventLoop 发送GenerateJobs(new Time(longTime))消息， **eventLoop** 收到消息后会进行这个**batch** 对应的 **jobs** 的生成及提交执行，eventLoop 是一个消息接收处理器。

需要注意的是，timer 在创建之后并不会马上启动，将在

StreamingContext#start() 启动 Streaming Application 时间接调用到

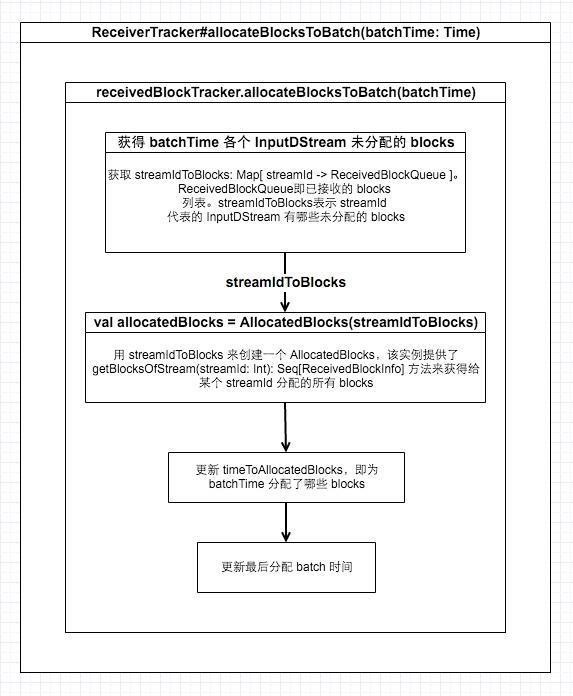
timer.start(restartTime.milliseconds) 才启动。为 **batch** 生成 **jobs**

eventLoop 在接收到 GenerateJobs(new Time(longTime))消息后的主要处理流程有以上图中三步：

* + 1. 将已接收到的 blocks 分配给 batch
    2. 生成该 batch 对应的 jobs
    3. 将 jobs 封装成 JobSet 并提交执行

接下来我们就将逐一展开这三步进行分析

将已接受到的 **blocks** 分配给 **batch**



上图是根据源码画出的为 batch 分配 blocks 的流程图，这里对 『获得

batchTime 各个 InputDStream 未分配的 blocks』作进一步说明：

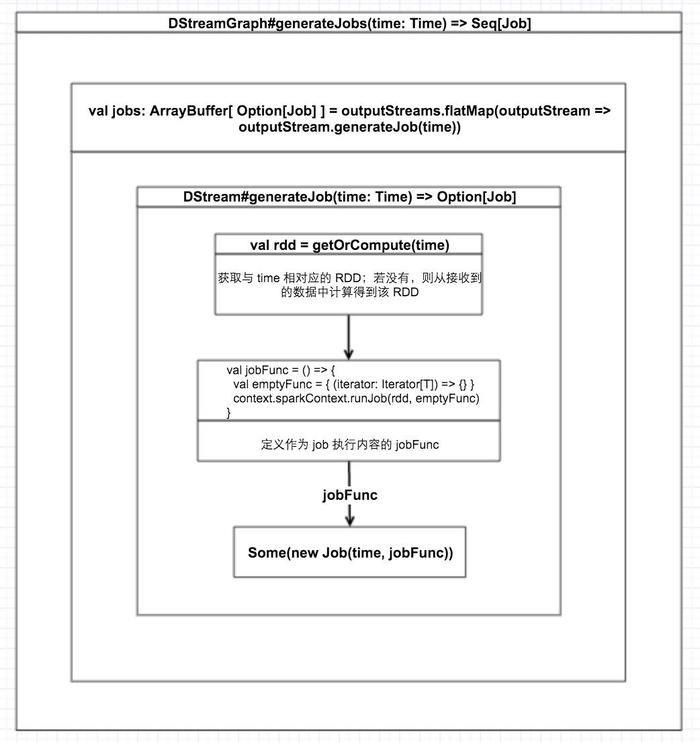
我们知道了各个 ReceiverInputDStream 对应的 receivers 接收并保存的blocks 信息会保存在ReceivedBlockTracker#streamIdToUnallocatedBlockQueues， 该成员 key 为streamId， value 为该 streamId 对应的 InputDStream 已接收保存但尚未分配的 blocks 信息。

所以获取某 InputDStream 未分配的 blocks 只要以该 InputDStream 的streamId 来从 streamIdToUnallocatedBlockQueues 来 get 就好。获取之后， 会清楚该 streamId 对应的 value， 以保证 block 不会被重复分配。

在实际调用中，为 batchTime 分配 blocks 时，会从streamIdToUnallocatedBlockQueues 取出未分配的 blocks 塞进timeToAllocatedBlocks: mutable.HashMap[Time, AllocatedBlocks] 中，以在之后作为该 batchTime 对应的 RDD 的输入数据。

通过以上步骤，就可以为 batch 的所有 InputDStream 分配 blocks。也就是为 batch 分配了 blocks。

生成该 **batch** 对应的 **jobs**



为指定 batchTime 生成 jobs 的逻辑如上图所示。你可能会疑惑，为什么DStreamGraph#generateJobs(time: Time) 为什么返回 Seq[Job]，而不是单个 job。这是因为，在一个 batch 内，可能会有多个 OutputStream 执行了多次output 操作，每次 output 操作都将产生一个 Job，最终就会产生多个

Jobs。

我们结合上图对执行流程进一步分析。

在 DStreamGraph#generateJobs(time: Time) 中，对于 DStreamGraph 成员ArrayBuffer[DStream[\_]]的每一项，调用 DStream#generateJob(time: Time)来生成这个 outputStream 在该 batchTime 的 job。该生成过程主要有三步：

**Step1:** 获取该 **outputStream** 在该 **batchTime** 对应的 **RDD**

每个 DStream 实例都有一个 generatedRDDs: HashMap[Time, RDD[T]] 成

员，用来保存该 DStream 在每个 batchTime 生成的 RDD，当DStream#getOrCompute(time: Time)调用时

* 首先会查看generatedRDDs 中是否已经有该 time 对应的 RDD，若有则直接返回
* 若无，则调用 compute(validTime: Time)来生成 RDD，这一步根据每个InputDStream 继承 compute 的实现不同而不同。例如，对于

FileInputDStream，其 compute 实现逻辑如下：

* + - 1. 先通过一个 findNewFiles() 方法，找到多个新 file
      2. 对每个新 file，都将其作为参数调用 sc.newAPIHadoopFile(file)， 生成一个 RDD 实例
      3. 将 2 中的多个新 file 对应的多个 RDD 实例进行 union，返回一个union 后的 UnionRDD

**Step2:** 根据 **Step1** 中得到的 **RDD** 生成最终 **job** 要执行的函数 **jobFunc**

jobFunc 定义如下：

val jobFunc = () => {

val emptyFunc = { (iterator: Iterator[T]) => {} } context.sparkContext.runJob(rdd, emptyFunc)

}

可以看到，每个 outputStream 的 output 操作生成的 Job 其实与 RDD

action 一样，最终调用 SparkContext#runJob 来提交 RDD DAG 定义的任务

**Step3:** 根据 **Step2** 中得到的 **jobFunc** 生成最终要执行的 **Job** 并返回

Step2 中得到了定义 Job 要干嘛的函数-jobFunc， 这里便以 jobFunc 及

batchTime 生成 Job 实例：

Some(new Job(time, jobFunc))

该 Job 实例将最终封装在 JobHandler 中被执行

至此，我们搞明白了 JobScheduler 是如何通过一步步调用来动态生成每个 batchTime 的 jobs。下文我们将分析这些动态生成的 jobs 如何被分发及如何执行。

## job 的提交与执行

我们分析了 JobScheduler 是如何动态为每个 batch 生成 jobs，那么生成的 jobs 是如何被提交的。

在 JobScheduler 生成某个 batch 对应的 Seq[Job] 之后，会将 batch 及Seq[Job] 封装成一个 JobSet 对象，JobSet 持有某个 batch 内所有的 jobs， 并记录各个 job 的运行状态。

之后，调用 JobScheduler#submitJobSet(jobSet: JobSet) 来提交 jobs，在该函数中，除了一些状态更新，主要任务就是执行

jobSet.jobs.foreach(job => jobExecutor.execute(new JobHandler(job)))

即，对于 jobSet 中的每一个 job， 执行 jobExecutor.execute(new JobHandler(job))，要搞懂这行代码干了什么，就必须了解 JobHandler 及jobExecutor。

JobHandler

JobHandler 继承了 Runnable，为了说明与 job 的关系，其精简后的实现如下：

private class JobHandler(job: Job) extends Runnable with Logging { import JobScheduler.\_

def run() {

\_eventLoop.post(JobStarted(job)) PairRDDFunctions.disableOutputSpecValidation.withValue(true) {

job.run()

}

\_eventLoop = eventLoop

if (\_eventLoop != null) {

\_eventLoop.post(JobCompleted(job))

}

}

}

JobHandler#run 方法主要执行了 job.run()，该方法最终将调用到

『生成该 batch 对应的 jobs 的 Step2 定义的 jobFunc』， jonFunc 将提

交对应 RDD DAG 定义的 job。

JobExecutor

知道了 JobHandler 是用来执行 job 的， 那么 JobHandler 将在哪里执行

job 呢？答案是

jobExecutor，jobExecutor 为 JobScheduler 成员，是一个线程池，在

JobScheduler 主构造函数中创建，如下：

private val numConcurrentJobs = ssc.conf.getInt("spark.streaming.concurrentJobs", 1) private val jobExecutor =

ThreadUtils.newDaemonFixedThreadPool(numConcurrentJobs, "streaming-job- executor")

JobHandler 将最终在 线程池 jobExecutor 的线程中被调用， jobExecutor 的线程数可通过 spark.streaming.concurrentJobs 配置，默认为 1。若配置多个线程，就能让多个 job 同时运行，若只有一个线程，那么同一时刻只能有一个 job 运行。

以上，即 jobs 被执行的逻辑。

## Block 的生成与存储

ReceiverSupervisorImpl 共提供了 4 个将从 receiver 传递过来的数据转换成 block 并存储的方法，分别是：

* pushSingle: 处理单条数据
* pushArrayBuffer: 处理数组形式数据
* pushIterator: 处理 iterator 形式处理
* pushBytes: 处理 ByteBuffer 形式数据

其中，pushArrayBuffer、pushIterator、pushBytes 最终调用pushAndReportBlock；而 pushSingle 将调用defaultBlockGenerator.addData(data)，我们分别就这两种形式做说明

pushAndReportBlock

我们针对存储 block 简化 pushAndReportBlock 后的代码如下：

def pushAndReportBlock( receivedBlock: ReceivedBlock, metadataOption: Option[Any], blockIdOption: Option[StreamBlockId]

) {

...

val blockId = blockIdOption.getOrElse(nextBlockId) receivedBlockHandler.storeBlock(blockId, receivedBlock)

...

}

首先获取一个新的 blockId，之后调用 receivedBlockHandler.storeBlock, receivedBlockHandler 在 ReceiverSupervisorImpl 构造函数中初始化。当启用了 checkpoint 且 spark.streaming.receiver.writeAheadLog.enable 为 true 时， receivedBlockHandler 被初始化为 WriteAheadLogBasedBlockHandler 类型； 否则将初始化为 BlockManagerBasedBlockHandler 类型。

WriteAheadLogBasedBlockHandler#storeBlock 将 ArrayBuffer, iterator, bytes 类型的数据序列化后得到的 serializedBlock

* + 1. 交由 BlockManager 根据设置的 StorageLevel 存入 executor 的内存或磁盘中
    2. 通过 WAL 再存储一份

而 BlockManagerBasedBlockHandler#storeBlock 将 ArrayBuffer, iterator, bytes 类型的数据交由 BlockManager 根据设置的 StorageLevel 存入executor 的内存或磁盘中，并不再通过 WAL 存储一份

pushSingle

pushSingle 将调用 BlockGenerator#addData(data: Any) 通过积攒的方式来存储数据。接下来对 BlockGenerator 是如何积攒一条一条数据最后写入block 的逻辑

上图为 BlockGenerator 的各个成员，首选对各个成员做介绍：

currentBuffer

变长数组，当 receiver 接收的一条一条的数据将会添加到该变长数组的尾部

* 可能会有一个 receiver 的多个线程同时进行添加数据，这里是同步操作
* 添加前，会由 rateLimiter 检查一下速率，是否加入的速度过快。如果过快的话就需要 block 住，等到下一秒再开始添加。最高频率由spark.streaming.receiver.maxRate 控制，默认值为 Long.MaxValue，具体含义是单个 Receiver 每秒钟允许添加的条数。

blockIntervalTimer & blockIntervalMs

分别是定时器和时间间隔。blockIntervalTimer 中有一个线程，每隔

blockIntervalMs 会执行以下操作：

1. 将 currentBuffer 赋值给 newBlockBuffer
2. 将 currentBuffer 指向新的空的 ArrayBuffer 对象
3. 将 newBlockBuffer 封装成 newBlock
4. 将 newBlock 添加到 blocksForPushing 队列中 blockIntervalMs 由spark.streaming.blockInterval 控制，默认是 200ms。

blockPushingThread & blocksForPushing & blockQueueSize

blocksForPushing 是一个定长数组，长度由 blockQueueSize 决定，默认为 10，可通过 spark.streaming.blockQueueSize 改变。上面分析到， blockIntervalTimer 中的线程会定时将 block 塞入该队列。

还有另一条线程不断送该队列中取出 block，然后调用ReceiverSupervisorImpl.pushArrayBuffer(...) 来将 block 存储，这条线程就是blockPushingThread。

PS: blocksForPushing 为 ArrayBlockingQueue 类型。ArrayBlockingQueue 是一个阻塞队列，能够自定义队列大小， 当插入时，如果队列已经没有空闲位置，那么新的插入线程将阻塞到该队列，一旦该队列有空闲位置，那么阻塞的线程将执行插入

以上，通过分析各个成员，也说明了 BlockGenerator 是如何存储单条数据的。