## SPARK CORE

### Spark 框架两个核心组件

#### Driver

Spark 驱动器节点，用于执行 Spark 任务中的 main 方法，负责实际代码的执行工作。Driver 在 Spark 作业执行时主要负责:

1.将用户程序转化为作业 (job)

2.在 Executor 之间调度任务(task)

3.跟踪 Executor 的执行情况

4.通过 UI 展示查询运行情况

实际上，我们无法准确地描述 Driver 的定义，因为在整个的编程过程中没有看到任何有关Driver 的字眼。所以简单理解，所谓的 Driver 就是驱使整个应用运行起来的程序，也称之为Driver 类。

#### Master Worker

Spark 集群的独立部署环境中，不需要依赖其他的资源调度框架，自身就实现了资源调度的功能，所以环境中还有其他两个核心组件: Master 和 Worker，这里的Master 是一个进程，主要负责资源的调度和分配，并进行集群的监控等职责，类似于 Yarn 环境中的 RM、而Worker 呢，也是进程，一个 Worker 运行在集群中的一台服务器上，由 Master 分配资源对数据进行并行的处理和计算，类似于 Yam 环境中 NM

#### ApplicationMatser

Hadoop 用户向 YARN 集群提交应用程序时提交程序中应该包含 ApplicationMaster，用于向资源调度器申请执行任务的资源容器 Container，运行用户自己的程序任务 job，监控整个任务的执行，跟踪整个任务的状态，处理任务失败等异常情况。

### Spark核心概念

#### Executor与core

Spark Executor 是集群中运行在工作节点(Worker)中的一个JVM 进程，是整个集群中的专门用于计算的节点。在提交应用中，可以提供参数指定计算节点的个数，以及对应的资源。这里的资源一般指的是工作节点 Executor 的内存大小和使用的虚拟 CPU 核(Core) 数量。



#### 并行度

在分布式计算框架中一般都是多个任务同时执行，由于任务分布在不同的计算节点进行计算，所以能够真正地实现多任务并行执行，记住，这里是并行，而不是并发。这里我们将整集群并行执行任务的数量称之为并行度。那么一个作业到底并行度是多少呢?这个取决于框架的默认配置。应用程序也可以在运行过程中动态修改。

### Rdd

#### RDD与Scala集合对象方法区别

RDD的方法和Scala集合对象的方法不一样

集合对象的方法都是在同一个节点的内存中完成的

RDD的方法可以将计算逻辑发送到execute段(分布式节点)执行

为了区分不同的处理效果,所及将RDD的方法称之为算子

RDD的方法外部的操作都是在Driver端执行的,而方法内部的逻辑代码是在Executor端执行的

#### Rdd 的基本原理与io区别

Rdd的数据处理方式类似于IO流,也有装饰者设计模式

Rdd的数据只有在调用collect方法时,才会真正执行业务逻辑操作,之前的封装全部都是功能的扩展

Rdd是不保存数据的,但是IO可以临时保存一部分数据

#### 概念

RDD叫做弹性分布式数据集,是Spark中最基本的数据处理模型.代码中是一个抽象类,它代表一个弹性的,不可变,可分区,里面的元素可并行计算的集合

##### 弹性

存储的弹性:内存与磁盘的自动切换

容错的弹性:数据丢失可以自动回复

计算的弹性:计算出错重试机制

分片的弹性:可根据需要重新分片

##### 分布式:

数据存储在大数据集群不同节点上

##### 数据集

RDD封装了计算逻辑,并不保存数据

##### 数据抽象

RDD是一个抽象类,需要子类具体实现

##### 不可变

RDD封装了计算逻辑,是不可以改变的,想要改变,只能产生新的RDD,在新的RDD里面封装计算逻辑

##### 可分区,并行计算

#### 执行原理

从计算的角度来讲,数据处理过程中需要计算资源和计算模型.执行时需要将计算资源和计算模型进行整合

Spark框架在执行时,先申请资源,然后将应用程序的数据处理逻辑分解成一个个的计算任务,然后将任务发送到已经分配的资源的计算节点上,按照指定的计算模型进行数据计算,最后得到计算结果

#### RDD创建方式

1. 从集合(内存)中创建RDD(makeRdd / parallelize)
2. 从外部文件中创建RDD,将文件中的数据作为处理的数据源(textFile)
3. 从其他RDD创建
4. 直接创建RDD(new)

#### RDD序列化

1. 闭包检查

从计算的角度,算子以外的代码都是在Driver端执行,算子里面的代码都是在Executor端执行.那么在scala的函数式编程中,就会导致算子内经常会用到算子外的数据,这样就形成了闭包的效果,如果使用的算子外的数据无法序列化,就意外着无法传值给Executor端执行,就会发生错误,所以需要在执行任务计算前,检测闭包内的对象时候可以进行序列化,这个操作我们称之为闭包检查.Scala2.12版本后闭包编译方式发生了改变.

1. 序列化方法和属性

从计算的角度,算子以外的代码都是在Driver端执行,算子里面的代码都是在Executor端执行

1. Kryo序列化框架:

Java的序列化能够序列化任何的类,但是比较重(字节多),序列化后,对象的提交也比较大.spark出于性能的考虑,Spark2.0开始支持另外一种Kryo序列化机制,Kryo速度是serializable的10倍.当Rdd在Shuffle数据的时候,简单数据类型,数组和字符串类型已经在Spark内部使用Kryo来序列化.

注意:使用Kryo序列化,也要继承Serializable接口

#### RDD算子

1. mapPartitions:可以以分区为单位进行数据转换操作,但是会将整个分区的数据加载到内存中进行引用,如果处理完的数据是不会被释放掉的,存在对象的引用,在内存较小,数据量较大的场合下容易出现内存溢出.

注:MAP与mapPartitions的区别:

数据处理角度:Map算子是分区内一个数据一个数据的执行,类似于串行操作.而mapPartitions算子是以分区为单位进行批处理操作

功能的角度:Map算子主要目的将数据源中的数据进行转换和改变.但是不会减少或增多数据. mapPartitions算子需要传递一个迭代器,返回一个迭代器,没有要求的元素的个数保持不变,所以可以增加或减少数据

性能的角度:Map算子因为类似于串行操作,所以性能比较低,而mapPartitions算子类似于批处理,所以性能比较高.但是mapPartitions算子会长时间占用内存,那么这样会导致内存可能不够用,出现内存溢出的错误.所以在内存有限的情况下,不推荐使用,使用map操作.

1. mapPartitionsWithIndex:将待处理的数据以分区为单位发送到计算节点进行处理,这里的处理是指可以进行任意的处理,哪怕是过滤数据,在处理同时可以获取当前分区索引
2. flatMap(扁平映射):将处理的数据进行扁平化后在进行映射处理,所以算子也称之为扁平映射
3. glom:将同一个分区的数据转换为相同类型的内存数组进行处理,分区不变
4. groupBy:将数据根据指定的规则集进行分组,分区默认不变,但是数据会被打乱重新组合,我们将这样的操作称之为shuffle.极限情况下,数据可能被分在同一个分区中.

一个组的数据在一个分区中,但是并不是说一个分区中只有一个组

1. filter:将数据根据指定的规则进行筛选过滤,符合规则的数据保留,不符合规则的数据丢弃.当数据进行筛选过滤后,分区不变,但是分区内的数据可能不均衡,生产环境下可能会出现数据倾斜
2. distinct:将数据集中重复的数据去重
3. coalesce:根据数据量缩减分区,用于大数据集过滤后,提高小数据集的执行效率.当Spark程序中,存在过多的小任务的时候,可以通过coalesce方法,收缩合并分区,减少分区的个数,减少任务调度成本.缩减分区时不会重新排列分区,如果想要数据均衡,可以采用shuffle

该算子也可以扩大分区,但是必须shuffle即重新排列组合,默认是不shuffle的(false)

1. repartition:根据数量扩大分区(底层就是coalesce,默认shuffle)
2. sortBy:根据指定的规则对数据进行排序

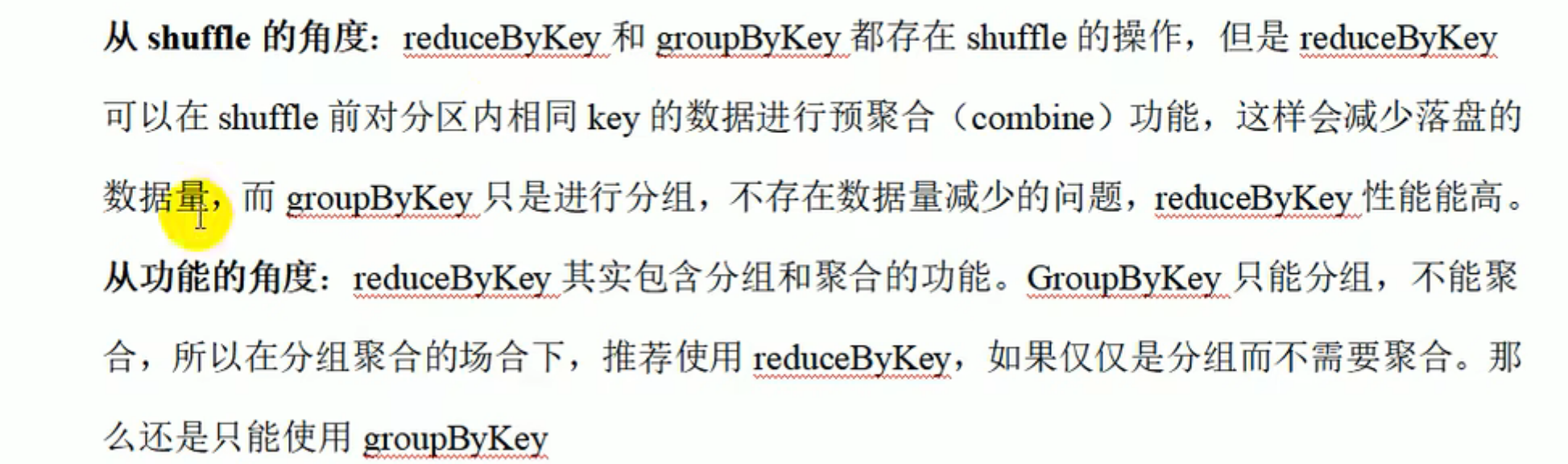
双value类型

1. intersection 对源RDD和参数RDD求交集后返回一个新的RDD
2. union 对源RDD和参数RDD求并集后返回一个新的RDD
3. subtract 对源RDD和参数RDD求差集后返回一个新的RDD
4. zip 拉链

Key-Value类型

1. partitionBy 将数据按照指定Partitioner重新进行分区.Spark默认的分区器是HashPartitioner
2. reduceByKey 可以将数据按照相同的Key对Value进行聚合
3. groupByKey 将分区的数据直接转换为相同类型的内存数组进行后续处理

reduceByKey与groupByKey的区别:groupByKey会导致数据的打乱重组,存在shuffle操作.spark中,shuffle操作必须落盘处理,不能再内存中数据等待,会导致内存溢出.shuffle操作性能比较低.reduceByKey在shuffle之前会对数据进行预处理(combine预聚合功能),可以有效减少shuffle时落盘的数据量,提升shuffle的性能.



1. aggregateByKey 将数据根据不同规则进行分区内计算和分区间计算
2. foldByKey 当aggregateByKey中分区间和分区内的计算方法相同时,可简化为foldByKey
3. combineByKey::需要三个参数:1.将相同key的第一个数据进行结构的转换,实现操作2.分区内的计算规则3.分区间的计算规则

reduceByKey,foldByKey,aggregateByKeY,combineByKey的区别

reduceByKey:相同key的第一个数据不进行任何计算,分区内和分区间计算规则相同

foldByKey:相同key的第一个数据和初始值进行分区内计算,分区内和分区间计算规则相同

1. join:咱类型为(K,V)和(K,W)的RDD上调用,返回一个相同的key对应的所有元素连接在一起的(K,(V,W))的RDD
2. leftOuterJoin:类似于sql中的左外连接
3. cogroup(分组连接))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

行动算子

1. reduce:聚集RDD中的所有元素,先聚合分区内数据,再聚合分区间数据
2. collect:在驱动程序中,以数组Array的形式返回数据集的所有元素
3. count:返回RDD中元素的个数
4. first:获取数据源中第一个元素
5. take:获取RDD中前n个数据
6. takeOrdered:数据排序后,取n个数据
7. aggregate:分区的数据通过初始值和分区内的数据进行聚合,然后在和初始值进行分区间的数据聚合
8. fold:折叠操作,aggregate的简化版操作
9. countByKey:通过key统计个数
10. sava:将数据保存到不同格式的文件中
11. foreach:分布式遍历RDD中的每一个元素,调用指定函数

#### Rdd依赖关系

相邻的两个RDD的关系称之为依赖关系

多个连续的Rdd的依赖关系,称之为血缘关系

每个RDD会保存血缘关系

RDD不会保存数据的,RDD为了提供容错性,需要将RDD间的关系保存下来

一旦出现错误,可以根据血缘关系将数据源重新读取进行计算

##### OneToOne(窄)依赖

新的RDD的一个分区的数据依赖于旧的RDD一个分区的数据

窄依赖表示每一个父RDD的Partition最多被子RDD的一个Partition使用

##### Shuffle(宽)依赖

新的RDD的一个分区的数据依赖于旧的RDD多个分区的数据

宽依赖表示同一个父RDD的Partition被多个子RDD的Partition依赖,会引起Shuffle

##### 阶段

当RDD中存在shuffle依赖时,阶段会自动增加一个

阶段的数量=shuffle依赖的数量+1

ResultStage只有一个,最后需要执行的阶段

任务的数量 = 当前阶段中最后一个RDD的分区数量

#### RDD任务划分

RDD任务切分中间分为:Application,Job,Stage,和Task

Application(应用):初始化一个SparkContext即生成一个Application

Job(工作):一个Action算子就会生成一个Job

Stage(阶段):Stage等于宽依赖的个数加一

Task(任务):一个Stage阶段中,最后一个RDD的分区个数就是Task的个数.

注意:Application>Job>Stage>Task 每一层都是1对n的关系

#### RDD持久化

RDD中不存储数据

如果一个RDD需要重复使用,那么需要从头再次执行来获取数据

RDD对象可以重用,但是数据无法重用

Cache默认持久化的操作,只能将数据保存到内存中,如果想要保存到硬盘文件,需要调用persist

RDD通过Cache或者Persist方法将前面的计算结果缓存,默认情况下会把数据以缓存在JVM的堆内存中.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算子时,该RDD将会被缓存在计算节点的内存中,并供后面重用..

RDD对象的持久化操作不一定是为了重用,在数据执行较长,获取数据比较重要的场合也可以采用持久化操作.

#### RDD CheckPoint检查点

所谓的检查点其实就是将RDD中间结果写入磁盘

由于血缘依赖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,如果检查点之后有节点出现问题,可以从检查点开始重做血缘,减小了开销

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.

Checkpoint需要落盘,需要指定检查点指定路径

检查点路径保存的文件在作业执行完毕后不会被删除

一般保存路径都是在分布式存储系统:HDFS

##### 持久化与检查点区别

Cache:将数据临时存储在内存中进行数据重用.

会在血缘关系中添加新的依赖,一旦出现问题,可以重头读取数据

Persist:将数据临时存储在磁盘文件中进行数据重用,

设计到磁盘IO,性能较低,但是数据安全.如果作业执行完毕,临时保存的数据文件就会丢失.

Checkpoint:将数据长久的保存在磁盘文件中镜像数据重用,

涉及到磁盘IO,性能较低,但是数据安全,

为了保证数据安全,所以一般情况下,会独立执行作业..

为了能够提高效率,一般情况下是和cache联用.

执行过程中会切断血缘关系,重新建立新的血缘

Checkpoint等用于改变数据源

#### RDD分区器

Spark目前支持hash分区和Range分区,和用户自定义分区.Hash分区为当前的默认分区.分区器直接决定了RDD中分区的个数,RDD中每条数据经过shuffle后进入哪个分区,进而决定了Reduce的个数.

只用Key-Value类型的RDD才有分区器,非Key-Value类型的RDD分区的值是None.

每个RDD的分区ID范围:0~(numPartitions-1),决定这个值是属于哪个分区的.

1. Hash分区:对于给定的Key,计算其hashCode,并除以分区个数取余
2. Range 分区：将一定范围内的数据映射到一个分区中，尽量保证每个分区数据均匀，而且分区间有序

#### RDD文件读取与保存

Spark的数据读取及数据保存可以从两个维度来做区分:文件格式以及文件系统.文件格式分为:text文件,csv文件,sequence文件以及Object文件.

sequence 文件: SequenceFile 文件是 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)。在 SparkContext 中，可以调用 sequenceFile[keyClass, valueClass](path)。

object 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[T: ClassTag](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。

### 累加器

分布式共享只写变量

实现原理:

累加器用来把 Executor 端变量信息聚合到 Driver 端。在 Driver 程序中定义的变量，在 Executor 端的每个 Task 都会得到这个变量的一份新的副本，每个 task 更新这些副本的值后， 传回 Driver 端进行 merge。

少加:转换算子中调用累加器,如果没有行动算子的话,那么不会执行

一般情况下,累加器会放在行动算子中进行操作

### 广播变量

分布式共享只读变量

闭包数据,都是以Task为单位发送的,每个任务中包含闭包数据,这样可能导致一个executor中含有大量重复的数据,并且占用大量的内存.

Executor其实就是一个JVM,所以在启动时,会自动分配内存,完全可以将任务中的闭包数据放置在executor的内存中,达到共享的目的.

Spark中的广播变量就可以将闭包的数据保存到executor的内存中

Spark中的广播变量不能够更改:分布式共享只读变量

实现原理:

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， 广播变量用起来都很顺手。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。

## SPARK SQL

### SPARK SQL特点:

1. 易整合:无缝的整合了SQL查询和Spark编程
2. 统一的数据访问:使用相同的方式连接不同的数据源
3. 兼容Hive:在已有仓库上直接运行SQL或者HiveQL
4. 标准数据连接:通过JDBC或者ODBC来连接

### DataFrame:

在 Spark 中，DataFrame 是一种以 RDD 为基础的分布式数据集，类似于传统数据库中 的二维表格。DataFrame 与 RDD 的主要区别在于，前者带有 schema 元信息，即 DataFrame 所表示的二维表数据集的每一列都带有名称和类型。这使得 Spark SQL 得以洞察更多的结构 信息，从而对藏于 DataFrame 背后的数据源以及作用于 DataFrame 之上的变换进行了针对性 的优化，最终达到大幅提升运行时效率的目标。反观 RDD，由于无从得知所存数据元素的 具体内部结构，Spark Core 只能在 stage 层面进行简单、通用的流水线优化。 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.

### DataSet:

DataSet 是分布式数据集合。DataSet 是 Spark 1.6 中添加的一个新抽象，是 DataFrame的一个扩展。它提供了 RDD 的优势（强类型，使用强大的 lambda 函数的能力）以及 SparkSQL 优化执行引擎的优点。DataSet 也可以使用功能性的转换（操作 map，flatMap，filter等等）。

➢ DataSet 是 DataFrame API 的一个扩展，是 SparkSQL 最新的数据抽象

➢ 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；

➢ 用样例类来对 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；

➢ DataSet 是强类型的。比如可以有 DataSet[Car]，DataSet[Person]。

➢ 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

表结构信息都用 Row 来表示。获取数据时需要指定顺序

## Spark Streaming

//数据处理的方式角度

流式(Streaming)数据处理

批量(batch)数据处理

//数据处理延迟的长短

实时数据处理: 毫秒级别

离线数据处理: 小时or天 级别

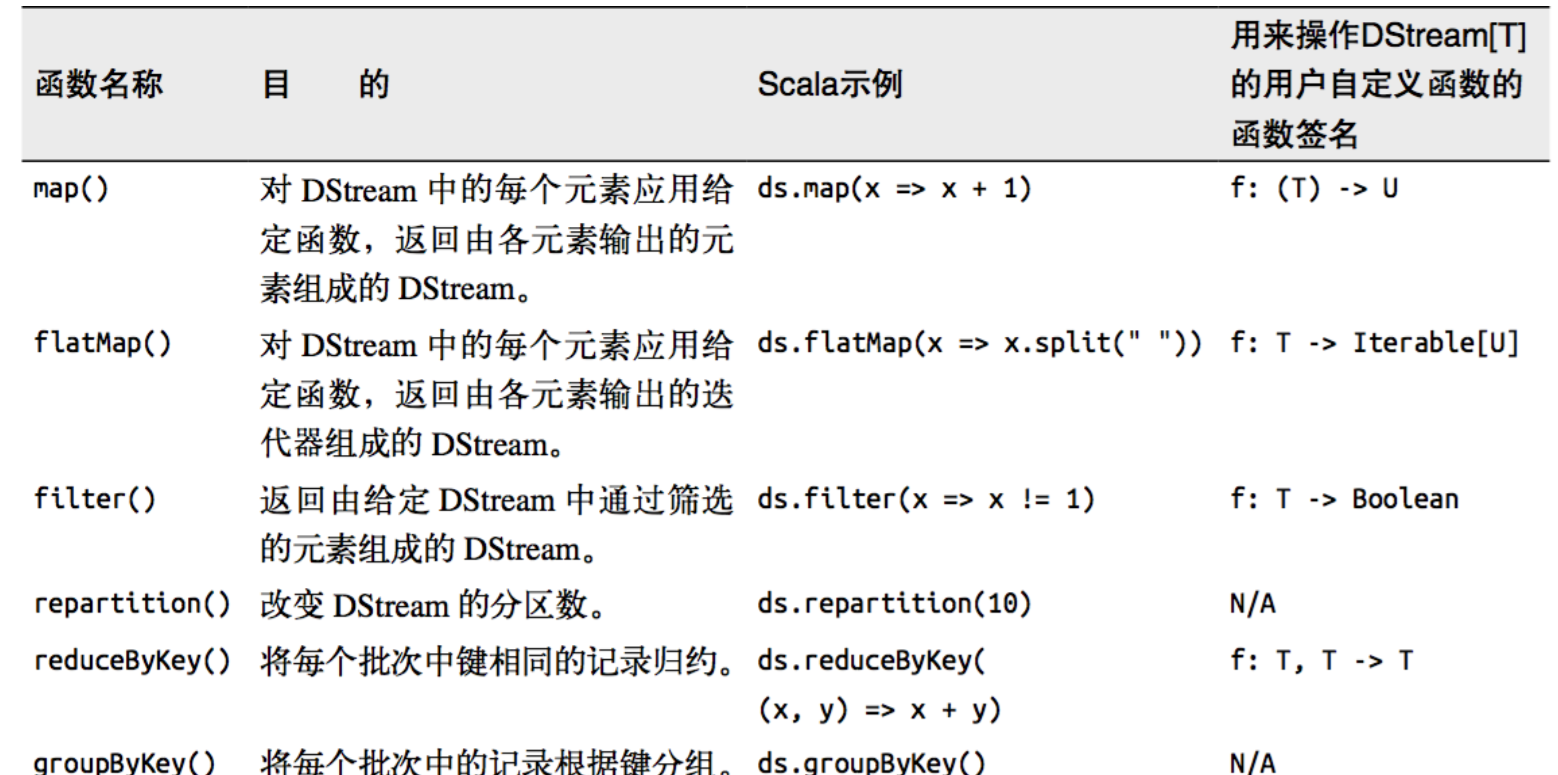
SparkStreaming :准实时(秒,分钟),微批次(时间)的数据处理框架

### DStream转换

DStream 上的操作与 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。

#### 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。

##### Transform

Transform 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。其实也 就是对 DStream 中的 RDD 应用转换。

##### Join

两个流之间的 join 需要两个流的批次大小一致，这样才能做到同时触发计算。计算过程就是 对当前批次的两个流中各自的 RDD 进行 join，与两个 RDD 的 join 效果相同。

#### 有状态转化操作

##### 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，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。

##### WindowOperations

Window Operations 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。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。

窗口时长：计算内容的时间范围；

滑动步长：隔多久触发一次计算。

注意：这两者都必须为采集周期大小的整数倍

关于 Window 的操作还有如下方法：

（1）window(windowLength, slideInterval): 基于对源 DStream 窗化的批次进行计算返回一个新的 Dstream；

（2）countByWindow(windowLength, slideInterval): 返回一个滑动窗口计数流中的元素个数；

（3）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；

（4）reduceByKeyAndWindow(func, windowLength, slideInterval, [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。

（5）reduceByKeyAndWindow(func, invFunc, windowLength, slideInterval, [numTasks]): 这个函数是上述函数的变化版本，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。

### DStream 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。

输出操作如下：

➢ 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 print()。

➢ 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 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。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。

注意：

1) 连接不能写在 driver 层面（序列化）

2) 如果写在 foreach 则每个 RDD 中的每一条数据都创建，得不偿失；

3) 增加 foreachPartition，在分区创建（获取）

### 优雅关闭

流式任务需要 7\*24 小时执行，但是有时涉及到升级代码需要主动停止程序，但是分 布式程序，没办法做到一个个进程去杀死，所有配置优雅的关闭就显得至关重要了。 使用外部文件系统来控制内部程序关闭。