**Spark教程**

**主讲:王一萍，王浩**

# 课程介绍

* 1. Spark数据分析导论
* 2. Spark下载与入门
* 3. RDD编程
* 4. 键值对操作
* 5. 数据读取与保存
* 6. Spark编程与进阶
* 7. Spark SQL
* 8. Spark Streaming
* 9. 集群上运行Spark
* 10. Spark调优与调试
* 11. 基于MLlib的机器学习

# Spark数据分析与导论

1. 什么是Spark

Spark是一个用来快速实现、通用的（？）集群计算平台。

* 在易用性和速度上：

Spark扩展了MapReduce的计算模型，能够高效的支持更多计算模式，包括交互式查询和流处理。Spark的主要特点就是能在内存中进行计算，所以更快，不过即使需要在磁盘上进行的复杂计算，Spark仍然比Mapreduce更高效。

* Spark提供的接口：

提供了Python、Java、Scala和sql的Api以外，可以以大多数的大数据存储组件进行交互，比如Spark读取和写入HDFS、HBase、Hive、Cassandra等。

* Spark项目包含多个组件：

1. Spark Core

Spark Core实现了Spark的基本功能，包括任务调度、内存管理、错误恢复、与存储系统交互等模块，Spark Core 中还包括了对弹性分布式数据集（resilient distributed dataset，简称RDD）的API定义，RDD表示分布在多个计算节点上可以并行操作的元素集合，是Spark主要的编程抽象，Spark Core提供了创建和操作这些集合的多个API。

1. Spark SQL

Spark SQL 是Spark 用来操作结构化数据的程序包。通过Spark SQL，我们可以使用SQL 或者Apache Hive 版本的SQL 方言（HQL）来查询数据。Spark SQL 支持多种数据源，比 如Hive 表、Parquet 以及JSON 等。除了为Spark 提供了一个SQL 接口，Spark SQL 还支 持开发者将SQL 和传统的RDD 编程的数据操作方式相结合，不论是使用Python、Java 还 是Scala，开发者都可以在单个的应用中同时使用SQL 和复杂的数据分析。通过与Spark 所提供的丰富的计算环境进行如此紧密的结合，Spark SQL 得以从其他开源数据仓库工具 中脱颖而出。Spark SQL 是在Spark 1.0 中被引入的。 在Spark SQL 之前，加州大学伯克利分校曾经尝试修改Apache Hive 以使其运行在Spark 上，当时的项目叫作Shark。现在，由于Spark SQL 与Spark 引擎和API 的结合更紧密， Shark 已经被Spark SQL 所取代。

1. Spark Streaming

Spark Streaming 是Spark 提供的对实时数据进行流式计算的组件。比如生产环境中的网页 服务器日志，或是网络服务中用户提交的状态更新组成的消息队列，都是数据流。Spark Streaming 提供了用来操作数据流的API，并且与Spark Core 中的RDD API 高度对应。这 样一来，程序员编写应用时的学习门槛就得以降低，不论是操作内存或硬盘中的数据，还 是操作实时数据流，程序员都更能应对自如。从底层设计来看，Spark Streaming 支持与 Spark Core 同级别的容错性、吞吐量以及可伸缩性。

1. Spark MLlib

Spark 中还包含一个提供常见的机器学习（ML）功能的程序库，叫作MLlib。MLlib 提供了很多种机器学习算法，包括分类、回归、聚类、协同过滤等，还提供了模型评估、数据 导入等额外的支持功能。MLlib 还提供了一些更底层的机器学习原语，包括一个通用的梯度下降优化算法。所有这些方法都被设计为可以在集群上轻松伸缩的架构。

1. 集群管理

就底层而言，Spark 设计为可以高效地在一个计算节点到数千个计算节点之间伸缩计 算。为了实现这样的要求，同时获得最大灵活性，Spark 支持在各种集群管理器（cluster manager）上运行，包括Hadoop YARN、Apache Mesos，以及Spark 自带的一个简易调度器，叫作独立调度器。如果要在没有预装任何集群管理器的机器上安装Spark，那么Spark 自带的独立调度器可以让你轻松入门；而如果已经有了一个装有Hadoop YARN 或Mesos 的集群，通过Spark 对这些集群管理器的支持，你的应用也同样能运行在这些集群上。第 7 章会详细探讨这些不同的选项以及如何选择合适的集群管理器。

* Spark的用户和用途

Spark 是一个用于集群计算的通用计算框架，因此被用于各种各样的应用程序。在前言中 我们提到了本书的两大目标读者人群：数据科学家和工程师。仔细分析这两个群体以及他 们使用Spark 的方式，我们不难发现这两个群体使用Spark 的典型用例并不一致，不过我 们可以把这些用例大致分为两类——数据科学应用和数据处理应用。 当然，这种领域和使用模式的划分是比较模糊的。很多人也兼有数据科学家和工程师的能 力，有的时候扮演数据科学家的角色进行研究，然后摇身一变成为工程师，熟练地编写复 杂的数据处理程序。不管怎样，分开看这两大群体和相应的用例是很有意义的。

* Spark的存储层次

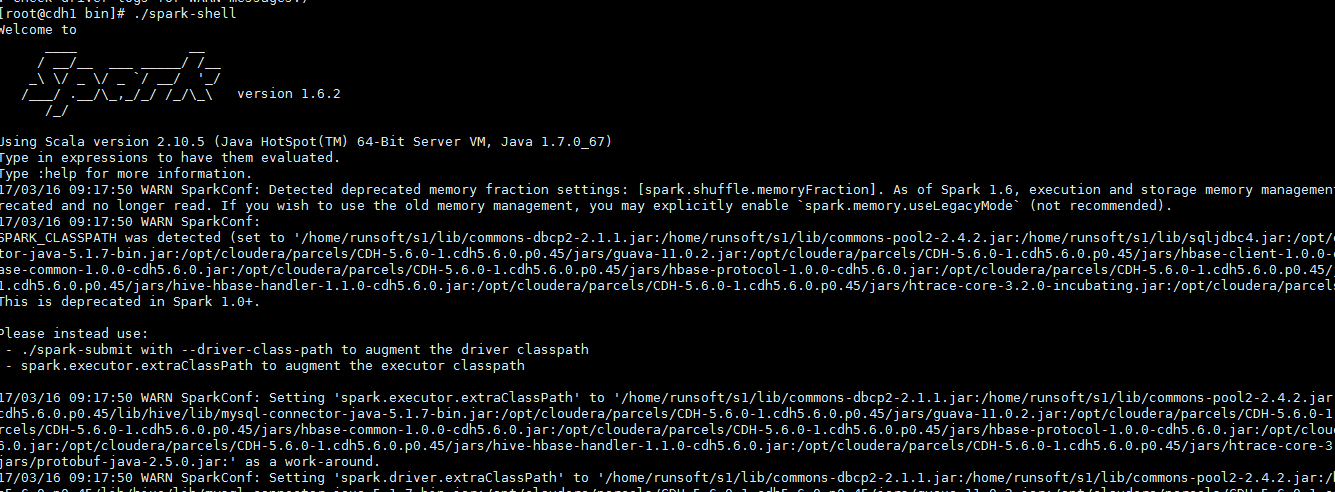
Spark 不仅可以将任何Hadoop分布式文件系统（HDFS）上的文件读取为分布式数据集，也可以支持其他支持Hadoop 接口的系统，比如本地文件、亚马逊S3、Cassandra、Hive、 HBase 等。我们需要弄清楚的是，Hadoop并非Spark的必要条件，Spark支持任何实现了Hadoop 接口的存储系统。Spark支持的Hadoop 输入格式包括文本文件、SequenceFile、 Avro、Parquet等。

# Spark下载与入门

* 下载Spark (TODO)
* Spark中Python和Scala的shell

Spark 带有交互式的shell，可以作即时数据分析。如果你使用过类似R、Python、Scala 所 提供的shell，或操作系统的shell（例如Bash 或者Windows 中的命令提示符），你也会对 Spark shell 感到很熟悉。然而和其他shell 工具不一样的是，在其他shell 工具中你只能使 用单机的硬盘和内存来操作数据，而Spark shell 可用来与分布式存储在许多机器的内存或 者硬盘上的数据进行交互，并且处理过程的分发由Spark 自动控制完成。

Scala 版本的shell，输入： bin/spark-shell



例子：Scala行数统计

scala> val lines = sc.textFile("README.md") // 创建一个名为lines的RDD

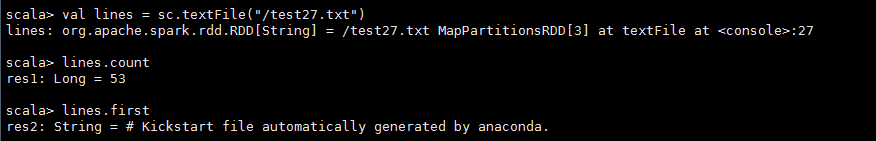
lines: spark.RDD[String] = MappedRDD[...]

scala> lines.count() // 统计RDD中的元素个数

res0: Long = 127

scala> lines.first() // 这个RDD中的第一个元素，也就是README.md的第一行

res1: String = # Apache Spark

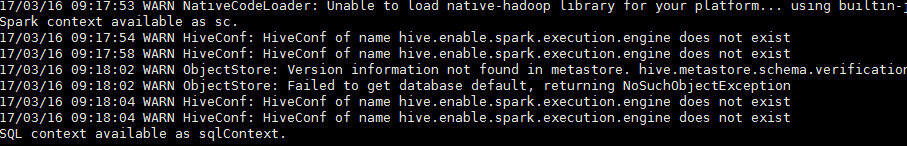


变量lines 是一个RDD，是从你电脑上的一个本地或HDFS的文本文件创建出来的。我们可以在这个RDD 上运行各种并行操作，比如统计这个数据集中的元素个数 （在这里就是文本的行数），或者是输出第一个元素。我们会在后续章节中深入探讨RDD。 在此之前，让我们先花些时间来了解Spark 的基本概念。

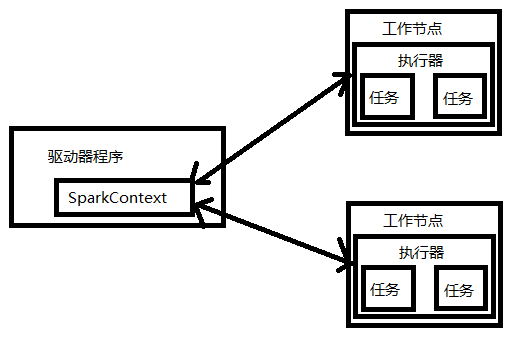
* Spark核心概念简介

每个Spark 应用都由一个驱动器程序（driver program）来发起集群上的各种并行操作。驱动器程序包含应用的main 函数，并且定义了集群上的分布式数据集，还对这些分布式数据集应用了相关操作。在前面的例子里，实际的驱动器程序就是Spark shell本身，你只需要输入想要运行的操作就可以了。

驱动器程序通过一个SparkContext对象来访问Spark。这个对象代表对计算集群的一个连 接。shell启动时已经自动创建了一个SparkContext 对象，是一个叫作sc的变量。我们可以通过例子的方法尝试输出sc来查看它的类型。



一旦有了SparkContext，你就可以用它来创建RDD。在例子，我们调用了 sc.textFile() 来创建一个代表文件中各行文本的RDD。我们可以在这些行上进行各种操作，比如count()。 要执行这些操作，驱动器程序一般要管理多个执行器（executor）节点。比如，如果我们在集群上运行count() 操作，那么不同的节点会统计文件的不同部分的行数。由于我们刚才是 在本地模式下运行Spark shell，因此所有的工作会在单个节点上执行，但你可以将这个shell连接到集群上来进行并行的数据分析。下图展示了Spark 如何在一个集群上运行。



我们有很多用来传递函数的API，可以将对应操作运行在集群上。

scala> val lines = sc.textFile("README.md") // 创建一个叫lines的RDD

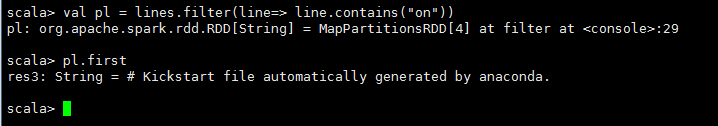
lines: spark.RDD[String] = MappedRDD[...]

scala> val pythonLines = lines.filter(line => line.contains("Python"))

pythonLines: spark.RDD[String] = FilteredRDD[...]

scala> pythonLines.first()

res0: String = ## Interactive Python Shell



注意，向Spark的API传递函数作为参数。

* 如何初始化SparkContext

**Scala：**

import org.apache.spark.SparkConf

import org.apache.spark.SparkContext

import org.apache.spark.SparkContext.\_

val conf = new SparkConf().setMaster("local").setAppName("My App")

val sc = new SparkContext(conf)

**Java：**

import org.apache.spark.SparkConf;

import org.apache.spark.api.java.JavaSparkContext;

SparkConf conf = new SparkConf().setMaster("local").setAppName("My App");

JavaSparkContext sc = new JavaSparkContext(conf);

传递两个参数：

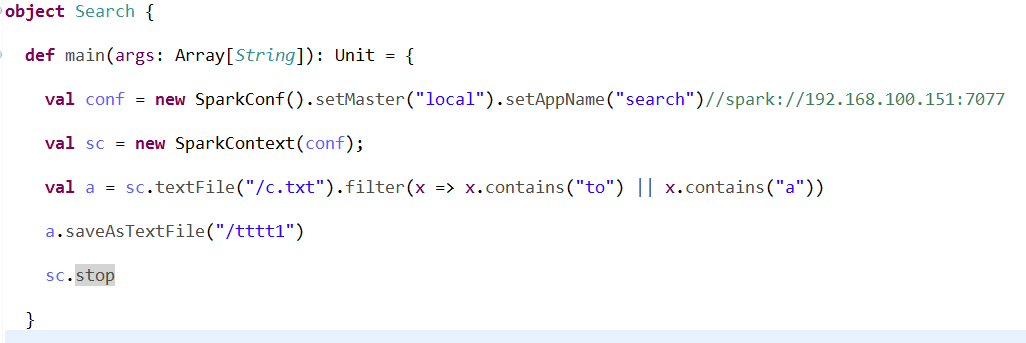
集群URL：告诉Spark 如何连接到集群上。例子中我们使用的是local，这个特殊值可以让Spark 运行在单机单线程上而无需连接到集群。

应用名：在例子中我们使用的是My App。当连接到一个集群时，这个值可以帮助你在集群管理器的用户界面中找到你的应用。

在初始化SparkContext 之后，你可以使用我们前面展示的所有方法（比如利用文本文件）来创建RDD并操控它们。

关闭Spark可以调用SparkContext 的stop()方法，或者直接退出应用（比如通过 System.exit(0) 或者sys.exit()）。

例子：



# RDD编程

Spark数据的核心抽象——弹性分布式数据集（Resilient Distributed Dataset，简 称RDD）。RDD 其实就是分布式的元素集合。在Spark 中，对数据的所有操作不外乎创 建RDD、转化已有RDD 以及调用RDD 操作进行求值。而在这一切背后，Spark 会自动将 RDD 中的数据分发到集群上，并将操作并行化执行。

1. RDD基础

Spark 中的RDD 就是一个不可变的分布式对象集合。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。

用户可以使用两种方法创建RDD：读取一个外部数据集，或在驱动器程序里分发驱动器程 序中的对象集合（比如list 和set）。前面的章节中已经见过使用SparkContext. textFile() 来读取文本文件作为一个字符串RDD 的示例，如下例所示。

lines = sc.textFile("README.md")

创建出来后，RDD 支持两种类型的操作：转化操作（transformation）和行动操作（action）。转化操作会由一个RDD 生成一个新的RDD。例如，根据谓词匹配情况筛选数据就是一个常见的转化操作。在我们的文本文件示例中，我们可以用筛选来生成一个只存储包含单词Python 的字符串的新的RDD，如下例子所示。

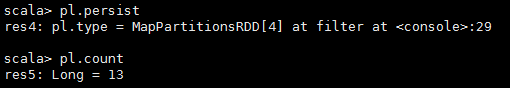
>>> pythonLines = lines.filter(lambda line: "Python" in line)

行动操作会对RDD 计算出一个结果，并把结果返回到驱动器程序中，或把结果存储到外部存储系统（如HDFS）中。first() 就是我们之前调用的一个行动操作，它会返回RDD 的第一个元素，如下例子所示。



转化操作和行动操作的区别：

* 可以在任何时候定义新的RDD，但Spark 只会惰性计算这些RDD。它们只有第一次在一个行动操作中用到时，才会真正计算。
* 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（内存或磁盘）下来。



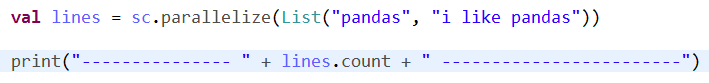
1. 创建RDD

Spark 提供了两种创建RDD 的方式：读取外部数据集，以及在驱动器程序中对一个集合进行并行化。

* 创建RDD最简单的方式就是把程序中一个已有的集合传给SparkContext的parallelize() 方法，

Scala：

val lines = sc.parallelize(List("pandas", "i like pandas"))



Java：

JavaRDD<String> lines = sc.parallelize(Arrays.asList("pandas", "i like pandas"));

* 更常用的方式是从外部存储中读取数据来创建RDD。

Scala:

val lines = sc.textFile("/path/to/README.md")

Java:

JavaRDD<String> lines = sc.textFile("/path/to/README.md");

1. RDD操作：

RDD支持两种操作：转化操作和行动操作。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，会触发实际的计算，比如count()和first()。Spark对待转化操作和行动操作的方式很不一样，因此理解你正在进行的操作的类型是很重要的。如果对于一个特定的函数是属于转化操作还是行动操作感到困惑，你可以看看它的返回值类型：转化操作返回的是RDD，而行动操作返回的是其他的数据类型。

1. RDD转换操作：

RDD 的转化操作是返回新RDD 的操作。转化出来的RDD 是惰性求值的，只有在行动操作中用到这些RDD 时才会被计算。

**Scala：**

val inputRDD = sc.textFile("log.txt")

val errorsRDD = inputRDD.filter(line => line.contains("error"))

**Java：**

JavaRDD<String> inputRDD = sc.textFile("log.txt");

JavaRDD<String> errorsRDD = inputRDD.filter(

new Function<String, Boolean>() {

public Boolean call(String x) {

return x.contains("error");

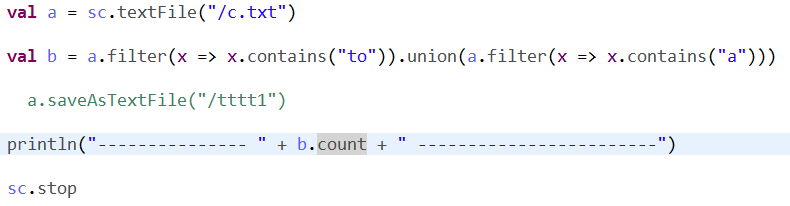
}

}

});

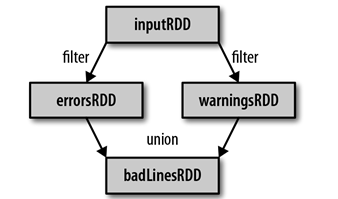
注意，filter() 操作不会改变已有的inputRDD 中的数据。实际上，该操作会返回一个全新的RDD。inputRDD在后面的程序中还可以继续使用，比如我们还可以从中搜索别的单词。 事实上，要再从inputRDD 中找出所有包含单词warning 的行。接下来，我们使用另一个转 化操作union() 来打印出包含error 或warning 的行数。

实现以下？



union() 与filter() 的不同点在于它操作两个RDD 而不是一个。转化操作可以操作任意 数量的输入RDD。

最后，通过转化操作，你从已有的RDD中派生出新的RDD，Spark 会使用谱系图（lineage graph）来记录这些不同RDD之间的依赖关系。Spark 需要用这些信息来按需计算每个RDD，也可以依靠谱系图在持久化的RDD丢失部分数据时恢复所丢失的数据。下图展示了上例中的谱系图。



1. 行动操作

行动操作是第二种类型的RDD操作，它们会把最终求得的结果返回到驱动器程序，或者写入外部存储系统中。由于行动操作需要生成实际的输出，它们会强制执行那些求值必须用到的RDD的转化操作。

需要使用两个行动操作来实现：用count() 来返回计数结果，用take() 来收集 RDD 中的一些元素：

**Scala：**

println("Input had " + badLinesRDD.count() + " concerning lines")

println("Here are 10 examples:")

badLinesRDD.take(10).foreach(println)

**Java：**

System.out.println("Input had " + badLinesRDD.count() + " concerning lines")

System.out.println("Here are 10 examples:")

for (String line: badLinesRDD.take(10)) {

System.out.println(line);

}

在驱动器程序中使用take()获取了RDD中的少量元素。然后在本地遍历这些元素，并在驱动器端打印出来。RDD还有一个collect() 函数，可以用来获取整个RDD中的数据。如果你的程序把RDD筛选到一个很小的规模，并且你想在本地处理这些数据时，就可以使用它。记住，只有当你的整个数据集能在单台机器的内存中放得下时，才能使用collect()，因此，collect()不能用在大规模数据集上。**想想后果？**

通常要把数据写到诸如HDFS或Amazon S3这样的分布式的存储系统中。你可以使用saveAsTextFile()、saveAsSequenceFile()，或者任意的其他行动操作来把RDD的数据内容以各种自带的格式保存起来。

1. 惰性求值

RDD的转化操作都是惰性求值的。

这意味着在被调用行动操作之前Spark不会开始计算。

惰性求值意味着当我们对RDD 调用转化操作（例如调用map()）时，操作不会立即执行。

相反，Spark 会在内部记录下所要求执行的操作的相关信息。我们不应该把RDD看作存 放着特定数据的数据集，而最好把每个RDD当作我们通过转化操作构建出来的、记录如何计算数据的指令列表。把数据读取到RDD的操作也同样是惰性的。因此，当我们调用 sc.textFile()时，数据并没有读取进来，而是在必要时才会读取。和转化操作一样的是，读取数据的操作也有可能会多次执行。

Spark 使用惰性求值，这样就可以把一些操作合并到一起来减少计算数据的步骤。在类似Hadoop MapReduce的系统中，开发者常常花费大量时间考虑如何把操作组合到一起，以 减少MapReduce 的周期数。而在Spark中，写出一个非常复杂的映射并不见得能比使用很 多简单的连续操作获得好很多的性能。因此，用户可以用更小的操作来组织他们的程序， 这样也使这些操作更容易管理。

1. 向Spark传递函数

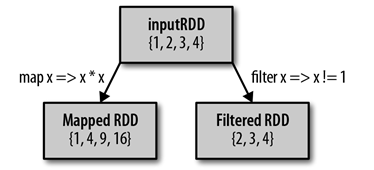
TODO

1. 常见转化操作和行动操作

首先来讲讲哪些转化操作和行动操作被任意数据类型的RDD支持。

* 1. 基本RDD

可能会用到的两个最常用的转化操作是map() 和filter()。转化操作map() 接收一个函数，把这个函数用于RDD 中的每个元素，将函数的返回结果作为结果RDD 中对应元素的值。而转化操作filter()则接收一个函数，并将RDD中满足该函数的 元素放入新的RDD中返回。



可以使用map() 来做各种各样的事情：可以把我们的URL集合中的每个URL对应的主机名提取出来，也可以简单到只对各个数字求平方值。map()的返回值类型不需要和输入类型一样。这样如果有一个字符串RDD，并且我们的map() 函数是用来把字符串解析并返回一个Double 值的，那么此时我们的输入RDD类型就是RDD[String]，而输出类型是RDD[Double]。

通过map（）对RDD中的所有数求平方：

**Scala：**

val input = sc.parallelize(List(1, 2, 3, 4))

val result = input.map(x => x \* x)

println(result.collect().mkString(","))

Java:

JavaRDD<Integer> rdd = sc.parallelize(Arrays.asList(1, 2, 3, 4));

JavaRDD<Integer> result = rdd.map(new Function<Integer, Integer>() {

public Integer call(Integer x) {

return x\*x;

}

});

System.out.println(StringUtils.join(result.collect(), ","));

希望对每个输入元素生成多个输出元素。实现该功能的操作叫作flatMap()。 和map()类似，我们提供给flatMap() 的函数被分别应用到了输入RDD的每个元素上。不过返回的不是一个元素，而是一个返回值序列的迭代器。输出的RDD 倒不是由迭代器组成的。我们得到的是一个包含各个迭代器可访问的所有元素的RDD。flatMap() 的一个简单用途是把输入的字符串切分为单词，如下：

**Scala：**

val lines = sc.parallelize(List("hello world", "hi"))

val words = lines.flatMap(line => line.split(" "))

words.first() // 返回"hello"

**Java：**

JavaRDD<String> lines = sc.parallelize(Arrays.asList("hello world", "hi"));

JavaRDD<String> words = lines.flatMap(new FlatMapFunction<String, String>() {

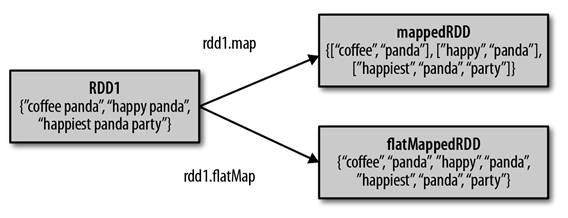
public Iterable<String> call(String line) {

return Arrays.asList(line.split(" "));

}

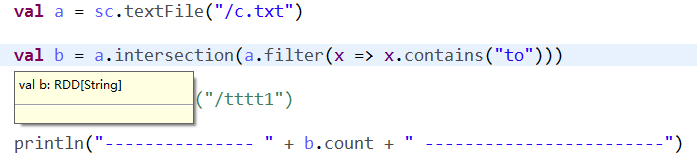
});

words.first(); // 返回"hello"

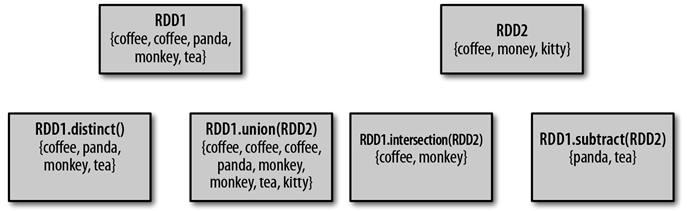


* 1. RDD互操作：

返回两个RDD中都有的元素：



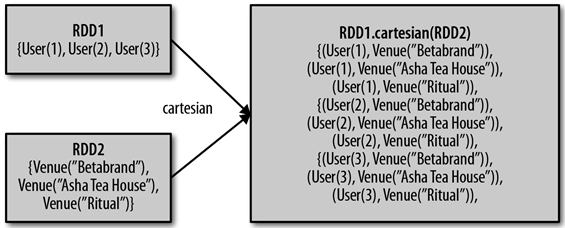
下图展示了四种互操作方法：

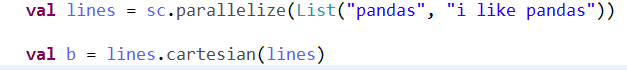


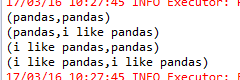
Rdd.distinct去掉重复数据，开销很大，因为需要将所有数据通过网络进行混洗（shuffle），以确保每个元素都只有一份。

笛卡尔积：

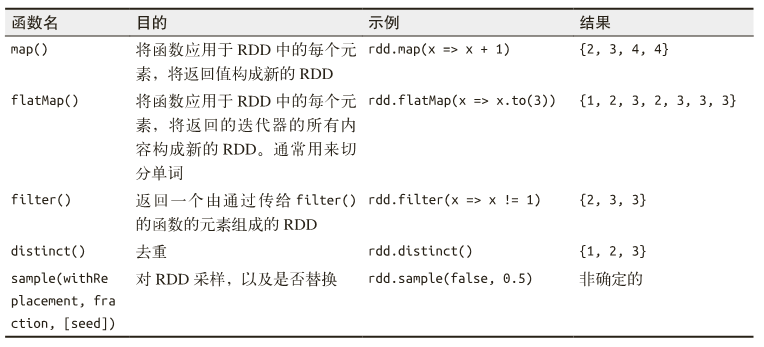
Cartesian(Other)转化操作会返回所有可能的(a,b)，其中a是源RDD中的元素，而b则来自另一个RDD。笛卡尔积在希望考虑所有可能的组合的相似度时比较有用。



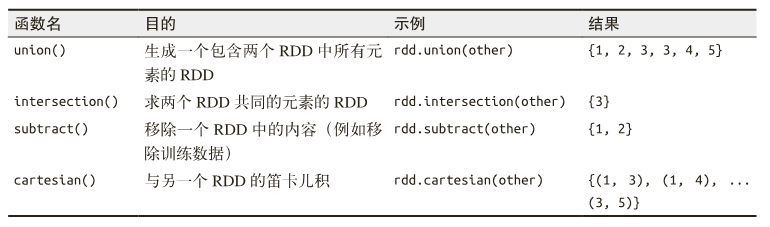




对一个数据为{1, 2, 3, 3}的RDD进行基本的RDD转化操作：



对数据分别为{1, 2, 3}和{3, 4, 5}的RDD进行针对两个RDD的转化操作



* 1. 行动操作

很有可能会用到基本RDD上最常见的行动操作reduce()。它接收一个函数作为参数，这个 函数要操作两个RDD的元素类型的数据并返回一个同样类型的新元素。一个简单的例子就是函数+，可以用它来对我们的RDD进行累加。使用reduce()，可以很方便地计算出RDD中所有元素的总和、元素的个数，以及其他类型的聚合操作。

**Scala：**

val sum = rdd.reduce((x, y) => x + y)

**Java：**

Integer sum = rdd.reduce(new Function2<Integer, Integer, Integer>() {

public Integer call(Integer x, Integer y) {

return x + y;

}

});

Aggregate()

**Scala:**

val result = input.aggregate((0, 0))(

(acc, value) => (acc.\_1 + value, acc.\_2 + 1),

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

)

val avg = result.\_1 / result.\_2.toDouble

**Java:**

class AvgCount implements Serializable {

public AvgCount(int total, int num) {

this.total = total;

this.num = num;

}

public int total;

public int num;

public double avg() {

return total / (double) num;

} }

Function2<AvgCount, Integer, AvgCount> addAndCount =

new Function2<AvgCount, Integer, AvgCount>() {

public AvgCount call(AvgCount a, Integer x) {

1. total += x;
2. num += 1;

return a;

} };

Function2<AvgCount, AvgCount, AvgCount> combine =

new Function2<AvgCount, AvgCount, AvgCount>() {

public AvgCount call(AvgCount a, AvgCount b) {

1. total += b.total;

a.num += b.num;

return a;

} };

AvgCount initial = new AvgCount(0, 0);

AvgCount result = rdd.aggregate(initial, addAndCount, combine);

System.out.println(result.avg());

对一个数据位{1,2,3,3}的Rdd进行基本的RDD行动操作





* 1. 在不同的RDD类型间转换：

有些函数只能用于特定类型的RDD，比如mean()和variance()只能用在数值RDD上， 而join()只能用在键值对RDD上。

Scala

在Scala中，将RDD转为有特定函数的RDD（比如在RDD[Double] 上进行数值操作）是由隐式转换来自动处理的。

* 1. 持久化（缓存）

Spark RDD 是惰性求值的，而有时我们希望能多次使用同一个RDD。如果简单地对RDD调用行动操作，Spark每次都会重算RDD以及它的所有依赖。这在迭代算法中消耗格外大，因为迭代算法常常会多次使用同一组数据。

Scala 中的两次执行：

val result = input.map(x => x\*x)

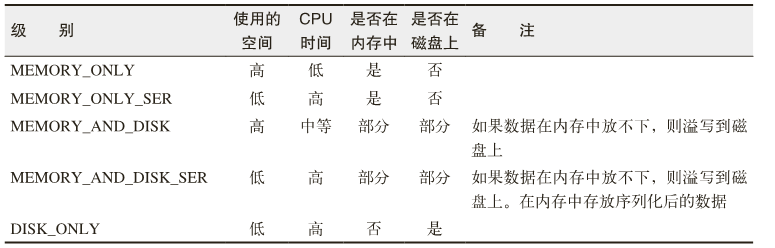
println(result.count())

println(result.collect().mkString(","))

为了避免多次计算同一个RDD，可以让Spark对数据进行持久化。当我们让Spark 持久化存储一个RDD时，计算出RDD的节点会分别保存它们所求出的分区数据。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。

出于不同的目的，我们可以为RDD选择不同的持久化级别。在Scala和Java中，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。在Python中，我们会始终序列化要持久化存储的数据，所以持久化级别默认值就是以序列化后的对象存储在JVM 堆空间中。当我们把数据写到磁盘或者堆外存储上时，也总是使用序列化后的数据。

org.apache.spark.storage.StorageLevel和pyspark.StorageLevel中的持久化级别；如有必要，可以通过在存储级别的末尾加上“\_2”来把持久化数据存为两份



在Scala 中使用persist()：

val result = input.map(x => x \* x)

result.persist(StorageLevel.DISK\_ONLY)

println(result.count())

println(result.collect().mkString(","))

注意：我们在第一次对这个RDD调用行动操作前就调用了persist()方法。persist()调 用本身不会触发强制求值。

如果要缓存的数据太多，内存中放不下，Spark 会自动利用最近最少使用（LRU）的缓存 策略把最老的分区从内存中移除。对于仅把数据存放在内存中的缓存级别，下一次要用到已经被移除的分区时，这些分区就需要重新计算。但是对于使用内存与磁盘的缓存级别的分区来说，被移除的分区都会写入磁盘。不论哪一种情况，都不必担心你的作业因为缓存了太多数据而被打断。不过，缓存不必要的数据会导致有用的数据被移出内存，带来更多重算的时间开销。

RDD还有一个方法叫作unpersist()，调用该方法可以手动把持久化的RDD从缓存中移除。

# 键值对操作

键值对RDD 是Spark 中许多操作所需要的常见数据类型。本章就来介绍如何操作键值对 RDD。键值对RDD 通常用来进行聚合计算。我们一般要先通过一些初始ETL（抽取、转 化、装载）操作来将数据转化为键值对形式。键值对RDD 提供了一些新的操作接口（比如 统计每个产品的评论，将数据中键相同的分为一组，将两个不同的RDD 进行分组合并等）。 本章也会讨论用来让用户控制键值对RDD在各节点上分布情况的高级特性：分区。有时，使用可控的分区方式把常被一起访问的数据放到同一个节点上，可以大大减少应用的通信开销。这会带来明显的性能提升。

1. 目标

Spark 为包含键值对类型的RDD提供了一些专有的操作。这些RDD被称为pairRDD。PairRDD是很多程序的构成要素，因为它们提供了并行操作各个键或跨节点重新进行数据分组的操作接口。例如，pair RDD提供reduceByKey()方法，可以分别归约每个键对应的数据， 还有join()方法，可以把两个RDD中键相同的元素组合到一起，合并为一个RDD。我们通常从一个RDD中提取某些字段（例如代表事件时间、用户ID或者其他标识符的字段），并使用这些字段作为pair RDD操作中的键。

1. 创建Pair RDD

在Spark 中有很多种创建pair RDD的方式。很多存储键值对的数据格式会在读取时直接返回由其键值对数据组成的pair RDD。此外，当需要把一个普通的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。后面会展示如何将由文本行组成的RDD转换为以每行的第一个单词为键的pair RDD。

* 在Scala 中使用第一个单词作为键创建出一个pair RDD：

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

Java没有自带的二元组类型，因此Spark的Java API 让用户使用scala.Tuple2 类来创建二元组。这个类很简单：Java用户可以通过new Tuple2(elem1, elem2) 来创建一个新的二元 组，并且可以通过.\_1() 和.\_2()方法访问其中的元素。

* Java用户还需要调用专门的Spark 函数来创建pair RDD。例如，要使用mapToPair() 函数来代替基础版的map() 函数：

PairFunction<String, String, String> keyData = new PairFunction<String, String, String>() {

public Tuple2<String, String> call(String x) {

return new Tuple2(x.split(" ")[0], x);

}

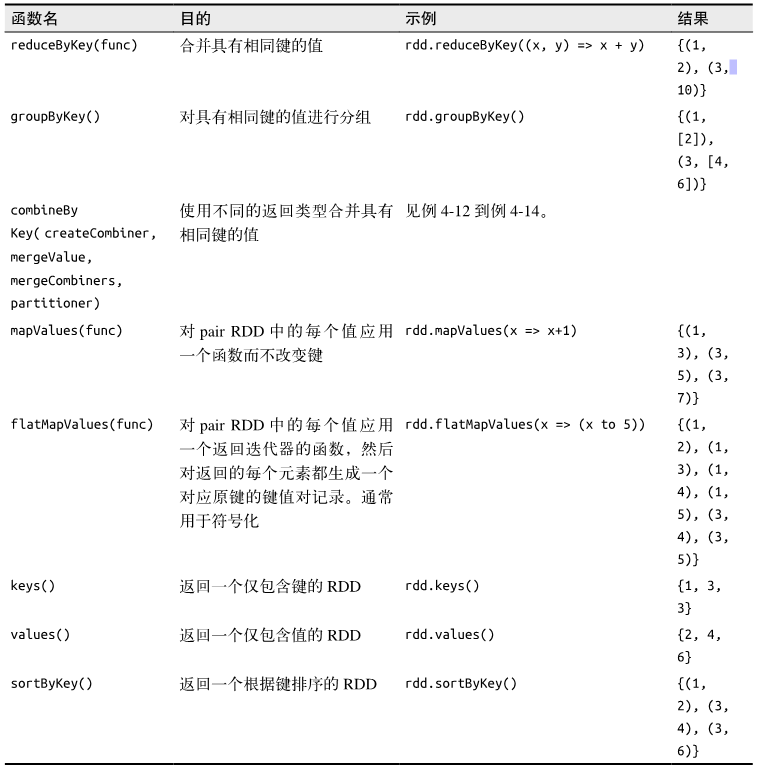
};

JavaPairRDD<String, String> pairs = lines.mapToPair(keyData);

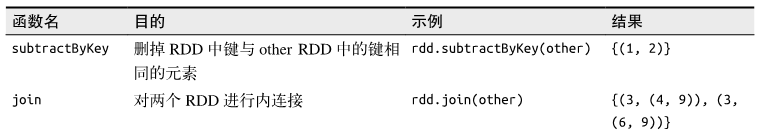
1. Pair RDD转换操作

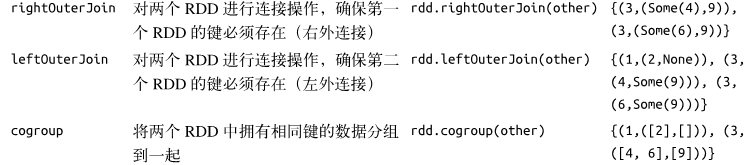
Pair RDD可以使用所有标准RDD上的可用的转化操作。由于pair RDD中包含二元组，所以需要传递的函数应当操作二元组而不是独立的元素。

Pair RDD的转化操作（以键值对集合{(1, 2), (3, 4), (3, 6)}为例）



针对两个pair RDD的转化操作（rdd = {(1, 2), (3, 4), (3, 6)}other = {(3, 9)}）：





1. Pair RDD 也还是RDD（元素为Java 或Scala 中的Tuple2 对象或Python 中的元组），因此 同样支持RDD 所支持的函数。

用**Scala** 对第二个元素进行筛选：

pairs.filter{case (key, value) => value.length < 20}

**Java：**

Function<Tuple2<String, String>, Boolean> longWordFilter =

new Function<Tuple2<String, String>, Boolean>() {

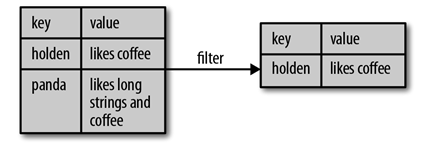
public Boolean call(Tuple2<String, String> keyValue) {

return (keyValue.\_2().length() < 20);

}

};

JavaPairRDD<String, String> result = pairs.filter(longWordFilter);



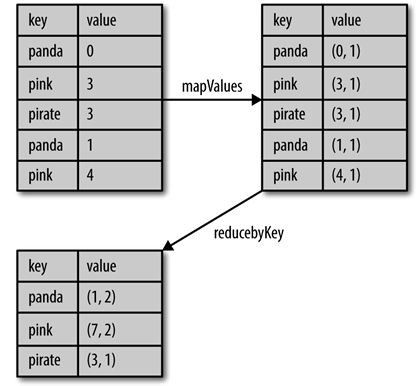
1. 聚合操作

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。

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。

* 在Scala 中使用reduceByKey() 和mapValues() 计算每个键对应的平均值

rdd.mapValues(x => (x, 1)).reduceByKey((x, y) => (x.\_1 + y.\_1, x.\_2 + y.\_2))



* 用Scala 实现单词计数：

val input = sc.textFile("hdfs://...")

val words = input.flatMap(x => x.split(" "))

val result = words.map(x => (x, 1)).reduceByKey((x, y) => x + y)

* 用Java 实现单词计数

JavaRDD<String> input = sc.textFile("hdfs://...")

JavaRDD<String> words = rdd.flatMap(new FlatMapFunction<String, String>() {

public Iterable<String> call(String x) {

return Arrays.asList(x.split(" "));

}

});

JavaPairRDD<String, Integer> result = words.mapToPair(

new PairFunction<String, String, Integer>() {

public Tuple2<String, Integer> call(String x) {

return new Tuple2(x, 1);

} }).reduceByKey(

new Function2<Integer, Integer, Integer>() {

public Integer call(Integer a, Integer b) {

return a + b;

}

});

combineByKey()有多个参数分别对应聚合操作的各个阶段，因而非常适合用来解释聚合操 作各个阶段的功能划分。

1. 并行调优：

到目前为止，我们已经讨论了所有的转化操作的分发方式，但是还没有探讨Spark是怎样确定如何分割工作的。每个RDD 都有固定数目的分区，分区数决定了在RDD 上执行操作 时的并行度。在执行聚合或分组操作时，可以要求Spark使用给定的分区数。Spark 始终尝试根据集群的大小推断出一个有意义的默认值，但是有时候你可能要对并行度进行调优来获取更好的性能表现。本章讨论的大多数操作符都能接收第二个参数，这个参数用来指定分组结果或聚合结果的RDD 的分区数。

* 在Scala 中自定义reduceByKey() 的并行度

val data = Seq(("a", 3), ("b", 4), ("a", 1))

sc.parallelize(data).reduceByKey((x, y) => x + y) // 默认并行度

sc.parallelize(data).reduceByKey((x, y) => x + y) // 自定义并行度

Spark 提供了repartition()函数。它会把数据通过网络进行混洗，并创建出新的分区集合。切记，对数据进行重新分区是代价相对比较大的操作。Spark 中也有一个优化版的repartition()，叫作coalesce()。你可以使用Java或Scala中的rdd. partitions.size()以及Python 中的rdd.getNumPartitions查看RDD的分区数，并确保调用coalesce()时将RDD合并到比现在的分区数更少的分区中。

1. 数据分组

对于有键的数据，一个常见的用例是将数据根据键进行分组——比如查看一个顾客的所有订单。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]]。

除了对单个RDD的数据进行分组，还可以使用一个叫作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。

1. 链接

将有键的数据与另一组有键的数据一起使用是对键值对数据执行的最有用的操作之一。连接数据可能是pair RDD最常用的操作之一。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。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。

* 在Scala shell 中进行内连接：

storeAddress = {

(Store("Ritual"), "1026 Valencia St"), (Store("Philz"), "748 Van Ness Ave"),

(Store("Philz"), "3101 24th St"), (Store("Starbucks"), "Seattle")}

storeRating = { (Store("Ritual"), 4.9), (Store("Philz"), 4.8))}

storeAddress.join(storeRating) == {

(Store("Ritual"), ("1026 Valencia St", 4.9)),

(Store("Philz"), ("748 Van Ness Ave", 4.8)),

(Store("Philz"), ("3101 24th St", 4.8))}

* leftOuterJoin() 与rightOuterJoin()

storeAddress.leftOuterJoin(storeRating) ==

{(Store("Ritual"),("1026 Valencia St",Some(4.9))),

(Store("Starbucks"),("Seattle",None)), (Store("Philz"),("748 Van Ness Ave",Some(4.8))),

(Store("Philz"),("3101 24th St",Some(4.8)))}

storeAddress.rightOuterJoin(storeRating) ==

{(Store("Ritual"),(Some("1026 Valencia St"),4.9)),

(Store("Philz"),(Some("748 Van Ness Ave"),4.8)),

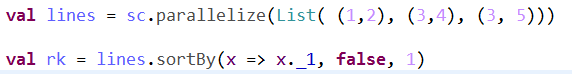
(Store("Philz"), (Some("3101 24th St"),4.8))}

1. 数据排序：

让数据排好序是很有用的，尤其是在生成下游输出时。如果键有已定义的顺序，就可以对这种键值对RDD进行排序。当把数据排好序后，后续对数据进行collect()或save() 等操作都会得到有序的数据。

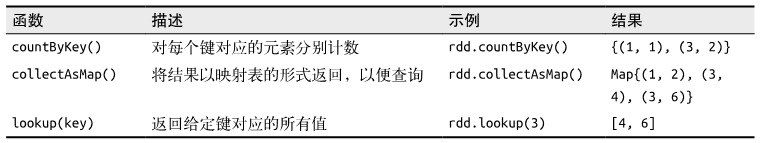
倒序排列，因此sortByKey()函数接收一个叫作ascending的参数，表示我们是否想要让结果按升序排序（默认值为true）。

* 在Scala 中以字符串顺序对整数进行自定义排序



1. Pari RDD的行动操作

Pair RDD的行动操作（以键值对集合{(1, 2), (3, 4), (3, 6)}为例）



1. 数据分区

讨论最后一个Spark 特性是对数据集在节点间的分区进行控制。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。和单节点的程序需要为记录集合选择合适的数据结构一样，Spark程序可以通过控制RDD分区方式来减少通信开销。分区并不是对所有应用都有好处的——比如，如果给定RDD只需要被扫描一次，我们完全没有必要对其预先进行分区处理。只有当数据集多次在诸如连接这种基于键的操作中使用时，分区才会有帮助。

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。尽管Spark 没有给出显示控制每个键具体落在哪一个工作节点上的方法（部分原因是 Spark 即使在某些节点失败时依然可以工作），但Spark可以确保同一组的键出现在同一个 节点上。比如，你可能使用哈希分区将一个RDD分成了100个分区，此时键的哈希值对100 取模的结果相同的记录会被放在一个节点上。你也可以使用范围分区法，将键在同一个范围区间内的记录都放在同一个节点上。

* Scala 自定义分区方式

val userData =

sc.sequenceFile[UserID, UserInfo]("hdfs://...")

.partitionBy(new HashPartitioner(100)) // 构造100个分区

.persist()

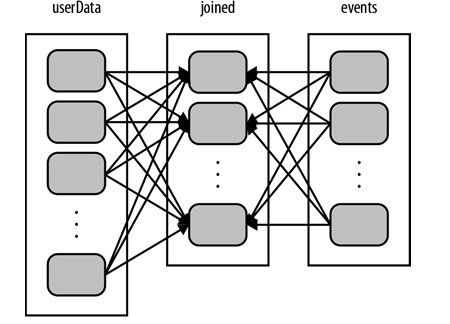
由于在构建userData 时调用了partitionBy()，Spark 就知道了该RDD 是根据键的哈希值来分区的，这样在调用join() 时，Spark 就会利用到这一点。具体来说，当调用userData. join(events) 时，Spark只会对events进行数据混洗操作，将events 中特定UserID的记录发送到userData的对应分区所在的那台机器上。

注意：partitionBy()是一个转化操作，因此它的返回值总是一个新的RDD，但它不会改变 原来的RDD。

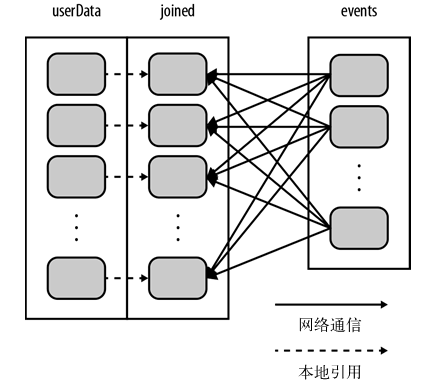
此外，传给partitionBy()的100表示分区数目，它会控制之后对这个RDD 进行进一步操作（比如连接操作）时有多少任务会并行执行。

总的来说，这个值至少应该和集群中的总核心数一样。

* 使用partitionBy前：



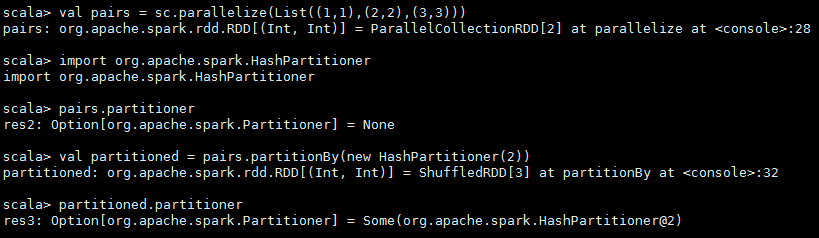
* 使用partitionBy后：



许多其他Spark操作会自动为结果RD设定已知的分区方式信息，而且除join()外还有很多操作也会利用到已有的分区信息。比如，sortByKey()和groupByKey() 会分别生成范围分区的RDD和哈希分区的RDD。而另一方面，诸如map()这样的操作会导致新的RDD失去父RDD的分区信息，因为这样的操作理论上可能会修改每条记录的键。接下来的几节中，我们会讨论如何获取RDD的分区信息，以及数据分区是如何影响各种Spark操作的。

1. 获取RDD分区方式

Scala获取RDD 的分区方式：



创建出了第二个RDD。如果确实要在后续操作中使用partitioned，那就应当在定义 partitioned时，在第三行输入的最后加上persist()。这和之前的例子中需要对userData调用persist()的原因是一样的：如果不调用persist()的话，后续的RDD操作会对partitioned的整个谱系重新求值，这会导致对pairs一遍又一遍地进行哈希分区操作。

1. 从分区中获益的操作

Spark 的许多操作都引入了将数据根据键跨节点进行混洗的过程。

对于像reduceByKey() 这样只作用于单个RDD的操作，运行在未分区的RDD上的时候会 导致每个键的所有对应值都在每台机器上进行本地计算，只需要把本地最终归约出的结 果值从各工作节点传回主节点，所以原本的网络开销就不算大。而对于诸如cogroup()和join() 这样的二元操作，预先进行数据分区会导致其中至少一个RDD（使用已知分区器的那个RDD）不发生数据混洗。如果两个RDD使用同样的分区方式，并且它们还缓存在 同样的机器上（比如一个RDD是通过mapValues()从另一个RDD中创建出来的，这两个RDD就会拥有相同的键和分区方式），或者其中一个RDD还没有被计算出来，那么跨节点的数据混洗就不会发生了。

1. 自定义分区方式

Spark 还允许你通过提供一个自定义的Partitioner 对象来控制RDD的分区方式。

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。

Scala 自定义分区方式：

class DomainNamePartitioner(numParts: Int) extends Partitioner {

override def numPartitions: Int = numParts

override def getPartition(key: Any): Int = {

val domain = new Java.net.URL(key.toString).getHost()

val code = (domain.hashCode % numPartitions)

if(code < 0) {

code + numPartitions // 使其非负

}else{

code

}

}

// 用来让Spark区分分区函数对象的Java equals方法

override def equals(other: Any): Boolean = other match {

case dnp: DomainNamePartitioner =>

dnp.numPartitions == numPartitions

case \_ =>

false

}

}

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。

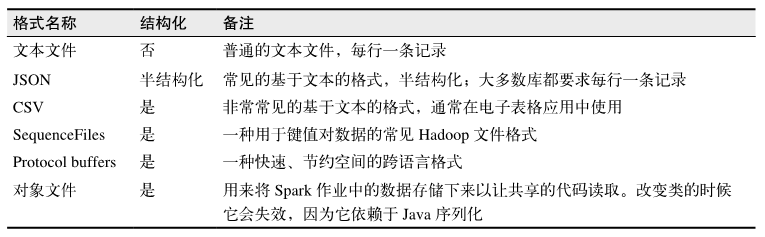
# 数据读取与保存

1. 目标

Spark 支持很多种输入输出源。一部分原因是Spark 本身是基于Hadoop 生态圈而构建，特 别是Spark 可以通过Hadoop MapReduce 所使用的InputFormat 和OutputFormat 接口访问 数据，而大部分常见的文件格式与存储系统（例如S3、HDFS、Cassandra、HBase 等）都支持。

1. 文件格式

Spark 对很多种文件格式的读取和保存方式都很简单。从诸如文本文件的非结构化的文件，到诸如JSON格式的半结构化的文件，再到诸如SequenceFile 这样的结构化的文件，Spark 都可以支持。Spark会根据文件扩展名选择对应的处理方式。这一过程是封装好的，对用户透明。



1. 文本文件

在Spark中读写文本文件很容易。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。

* 读取文本文件

只需要使用文件路径作为参数调用SparkContext 中的textFile() 函数，就可以读取一个文本文件。如果要控制分区数的话，可以指定minPartitions。

在Scala 中读取一个文本文件

val input = sc.textFile("hdfs:///home/holden/repos/spark/README.md")

如果多个输入文件以一个包含数据所有部分的目录的形式出现，可以用两种方式来处 理。可以仍使用textFile 函数，传递目录作为参数，这样它会把各部分都读取到RDD 中。有时候有必要知道数据的各部分分别来自哪个文件（比如将键放在文件名中的时间数据），有时候则希望同时处理整个文件。如果文件足够小，那么可以使用SparkContext. wholeTextFiles()方法，该方法会返回一个pair RDD，其中键是输入文件的文件名。

wholeTextFiles()在每个文件表示一个特定时间段内的数据时非常有用。如果有表示不同阶段销售数据的文件，则可以很容易地求出每个阶段的平均值：

在Scala 中求每个文件的平均值

val input = sc.wholeTextFiles("hdfs://home/holden/salesFiles")

val result = input.mapValues{y =>

val nums = y.split(" ").map(x => x.toDouble)

nums.sum / nums.size.toDouble

}

* 保存文本文件

rk.saveAsTextFile("/ddd")

1. JSON

* 读取JSON

SparkSQL讲 TODO更简单方式

* 保存JSON

同上

1. SequenceFile

* 读取SequenceFile

val data = sc.sequenceFile(inFile, classOf[Text], classOf[IntWritable])

. map{case (x, y) => (x.toString, y.get())}

* 保存SequenceFile

val data = sc.parallelize(List(("Panda", 3), ("Kay", 6), ("Snail", 2)))

data.saveAsSequenceFile(outputFile)

1. 本地/“常规”文件系统

val input = sc.textFile("file:/d:/demodata.txt")

1. Apache Hive

* 用Scala 创建HiveContext 并查询数据

import org.apache.spark.sql.hive.HiveContext

val hiveCtx = new org.apache.spark.sql.hive.HiveContext(sc)

val rows = hiveCtx.sql("SELECT name, age FROM users")

val firstRow = rows.first()

println(firstRow.getString(0)) // 字段0是name字段

1. Json

* 在Scala 中使用Spark SQL 读取JSON 数据

val tweets = hiveCtx.jsonFile("tweets.json")

tweets.registerTempTable("tweets")

val results = hiveCtx.sql("SELECT user.name, text FROM tweets")

1. 数据库

Spark可以从任何支持Java数据库连接（JDBC）的关系型数据库中读取数据，包括MySQL、Postgre等系统。要访问这些数据，需要构建一个org.apache.spark.rdd.JdbcRDD，将SparkContext和其他参数一起传给它。

* Scala 中的JdbcRDD

def createConnection() = {

Class.forName("com.mysql.jdbc.Driver").newInstance();

DriverManager.getConnection("jdbc:mysql://localhost/test?user=holden");

}

def extractValues(r: ResultSet) = {

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

}

val data = new JdbcRDD(sc, createConnection, "SELECT \* FROM panda WHERE ? <= id AND id <= ?",

lowerBound = 1, upperBound = 3, numPartitions = 2,

mapRow = extractValues

)

println(data.collect().toList)

1. HBase

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。Result 类包含多种根据列获取值的方法，在其API 文档（https://hbase. apache.org/apidocs/org/apache/hadoop/hbase/client/Result.html）中有所描述。 要将Spark 用于HBase，你需要使用正确的输入格式调用SparkContext.newAPIHadoopRDD。

import org.apache.hadoop.hbase.HBaseConfiguration

Import org.apache.hadoop.hbase.client.Result

import org.apache.hadoop.hbase.io.ImmutableBytesWritable

import org.apache.hadoop.hbase.mapreduce.TableInputFormat

val conf = HBaseConfiguration.create()

conf.set(TableInputFormat.INPUT\_TABLE, "tablename") // 扫描哪张表

val rdd = sc.newAPIHadoopRDD(

conf, classOf[TableInputFormat], classOf[ImmutableBytesWritable],classOf[Result])

# Spark编程进阶

1. 累加器

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。Spark 的两个共享变量，累加器与广 播变量，分别为结果聚合与广播这两种常见的通信模式突破了这一限制。

第一种共享变量，即累加器，提供了将工作节点中的值聚合到驱动器程序中的简单语法。 累加器的一个常见用途是在调试时对作业执行过程中的事件进行计数。例如，假设我们在 从文件中读取呼号列表对应的日志，同时也想知道输入文件中有多少空行（也许不希望在 有效输入中看到很多这样的行）。

* 在Scala 中累加空行

val sc = new SparkContext(...)

val file = sc.textFile("file.txt")

val blankLines = sc.accumulator(0) // 创建Accumulator[Int]并初始化为0

val callSigns = file.flatMap(line => {

if (line == "") {

blankLines += 1 // 累加器加1

}

line.split(" ")

})

callSigns.saveAsTextFile("output.txt")

println("Blank lines: " + blankLines.value)

* 累加器与容错性

Spark 会自动重新执行失败的或较慢的任务来应对有错误的或者比较慢的机器。例如，如果对某分区执行map() 操作的节点失败了，Spark 会在另一个节点上重新运行该任务。即使该节点没有崩溃，而只是处理速度比别的节点慢很多，Spark 也可以抢占式地在另一个节点上启动一个“投机”（speculative）型的任务副本，如果该任务更早结束就可以直接获取结果。即使没有节点失败，Spark 有时也需要重新运行任务来获取缓存中被移除出内存的数据。因此最终结果就是同一个函数可能对同一个数据运行了多次，这取决于集群发生了什么。

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。思考为什么？

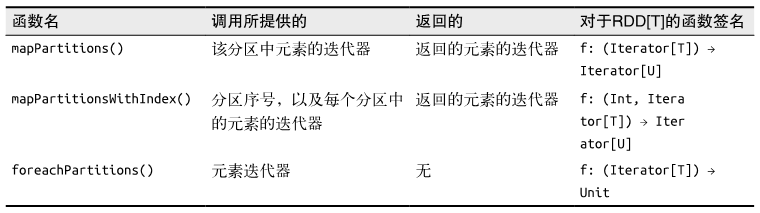
1. 广播变量

Spark的第二种共享变量类型是广播变量，它可以让程序高效地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。

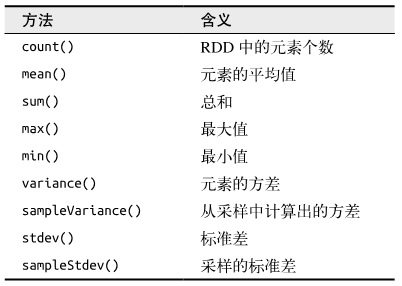
val broadcastVar = sc.broadcast(Array(1, 2, 3))

broadcastVar.value

1. 基于分区进行操作



1. 数值RDD的操作



# Spark SQL

* Spark SQL 可以从各种结构化数据源（例如JSON、Hive、Parquet 等）中读取数据。
* Spark SQL 不仅支持在Spark 程序内使用SQL 语句进行数据查询，也支持从类似商业 智能软件Tableau 这样的外部工具中通过标准数据库连接器（JDBC/ODBC）连接Spark SQL 进行查询。
* 当在Spark 程序内使用Spark SQL 时，Spark SQL 支持SQL 与常规的Python/Java/Scala 代码高度整合，包括连接RDD 与SQL 表、公开的自定义SQL 函数接口等。这样一来， 许多工作都更容易实现了。

1. [起始点 :](http://www.apache.wiki/pages/viewpage.action?pageId=2886696) SparkContext

Spark 中所有功能的入口点是 SparkContext类。去创建一个HiveContext：

val hc = new org.apache.spark.sql.hive.HiveContext(sc)

1. 创建DataFrame

与一个 HiveContext一起，应用程序可以从一个 [已存在的 RDD](http://spark.apache.org/docs/latest/sql-programming-guide.html" \l "interoperating-with-rdds)，或者一个 Hive 表中，或者从 [Spark 数据源](http://spark.apache.org/docs/latest/sql-programming-guide.html" \l "data-sources) 中创建 DataFrame。

举个例子，下面基于一个 JSON 文件的内容创建一个 DataFrame :

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

// Displays the content of the DataFrame to stdout

df.show()

df.printSchema()

1. 这里包括一些使用 **DF**进行结构化数据处理的示例：

import sql.implicits.\_ 导入隐式转换

df.select("age").show

df.select($"name", $"age" + 1).show

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

df.groupBy("age").count().show

1. 以编程方式运行SQL查询

df.createOrReplaceTempView("people")

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

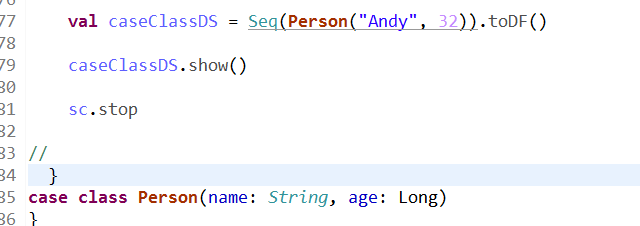
sqlDF.show()

1. 创建Dataset

case class Person(name: String, age: Long)

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

caseClassDS.show()





对df进行操作：

val csc = caseClassDS.map(x => new Person(x.getAs[String]("name") , x.getAs[Long]("age") + 1))

csc.foreach(print)

1. [以编程的方式指定 Schema](http://www.apache.wiki/pages/viewpage.action?pageId=2886721)

当 case class 不能够在执行之前被定义（例如，records 记录的结构在一个 string 字符串中被编码了，或者一个 text 文本 dataset 将被解析并且不同的用户投影的字段是不一样的）。一个 DataFrame 可以使用下面的三步以编程的方式来创建。

* 从原始的 RDD 创建 RDD 的 RowS（行）。
* Step 1 被创建后，创建 Schema 表示一个 StructType 匹配 RDD 中的Rows（行）的结构。
* 通过 SparkSession 提供的 createDataFrame 方法应用 Schema 到 RDD 的RowS（行）。

val peopleRDD = sql.sparkContext.textFile("/people.txt")

// The schema is encoded in a string

val schemaString = "name age"

// Generate the schema based on the string of schema

val fields = schemaString.split(" ")

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

val schema = StructType(fields)

// Convert records of the RDD (people) to Rows

val rowRDD = peopleRDD

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

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

// Apply the schema to the RDD

val peopleDF = sql.createDataFrame(rowRDD, schema).toDF()

// Creates a temporary table using the DataFrame

peopleDF.registerTempTable("people")

// SQL can be run over a temporary view created using DataFrames

val results = sql.sql("SELECT name FROM people")

results.show

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

c.foreach(print)

1. [通用的 Load/Save 函数](http://www.apache.wiki/pages/viewpage.action?pageId=2886098)

* 直接在文件上运行 **SQL**查询来替代使用 **API**将文件加载到 **DataFrame** 再进行查询。

val people = sql.sql("select \* from json.`/people.json`")

people.foreach(print)

* 保存模式

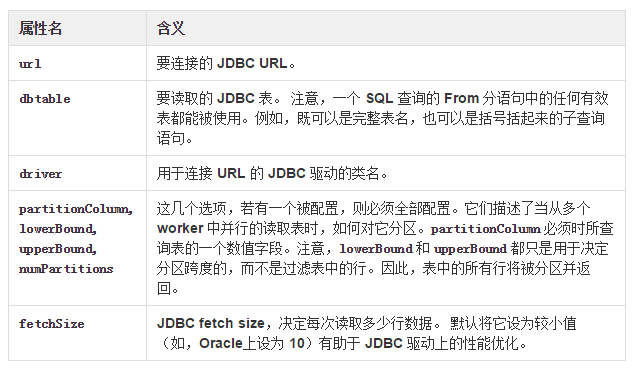
val people = sql.sql("select \* from json.`/people.json`")

people.foreach(print)

people.write.mode("error").json("/people17.json")



1. [JDBC 连接其它数据库](http://www.apache.wiki/pages/viewpage.action?pageId=2886198)



val jdbcDF = sql.read

.format("jdbc")

.option("url", "jdbc:postgresql:dbserver")

.option("dbtable", "schema.tablename")

.option("user", "username")

.option("password", "password")

.load()

1. 数据类型：



# Spark Streaming

1. 概述

Spark Streaming 是 Spark Core API 的扩展，它支持弹性的，高吞吐的，容错的实时数据流的处理。数据可以通过多种数据源获取，例如 Kafka，Flume，Kinesis 以及 TCP sockets，也可以通过例如 map，reduce，join，window 等的高阶函数组成的复杂算法处理。最终，处理后的数据可以输出到文件系统，数据库以及实时仪表盘中。事实上，你还可以在数据流上使用 [Spark机器学习](http://www.apache.wiki/pages/viewpage.action?pageId=2883740) 以及 [图形处理算法](http://www.apache.wiki/pages/viewpage.action?pageId=2883742) 。



在内部，它工作原理如下，Spark Streaming 接收实时输入数据流并将数据切分成多个批数据，然后交由 Spark 引擎处理并分批的生成结果数据流。

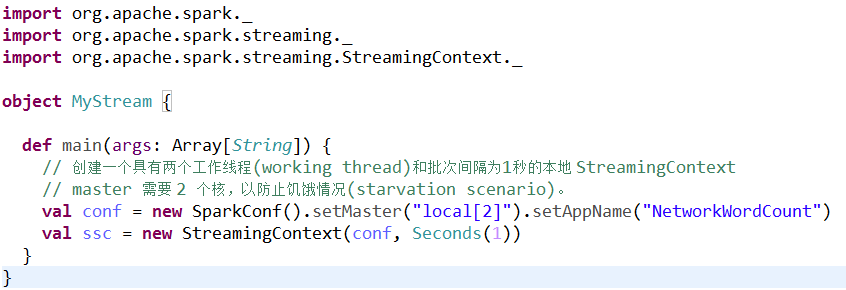


Spark Streaming 提供了一个高层次的抽象叫做离散流（discretized stream）或者 DStream，它代表一个连续的数据流。DStream 可以通过来自数据源的输入数据流创建，例如 Kafka，Flume 以及 Kinesis，或者在其他 DStream 上进行高层次的操作创建。在内部，一个 DStream 是通过一个 RDD 的序列来表示。

1. 一个简单的示例

在我们详细介绍如何你自己的 Spark Streaming 程序的细节之前，让我们先来看一看一个简单的 Spark Streaming 程序的样子。比方说，我们想要计算从一个监听 TCP socket 的数据服务器接收到的文本数据（text data）中的字数。所有你需要做的就是照着下面的步骤做。

首先，我们导入了 Spark Streaming 类和部分从 StreamingContext 隐式转换到我们的环境的名称，目的是添加有用的方法到我们需要的其他类（如 DStream）。 StreamingContext 是所有流功能的主要入口点。我们创建了一个带有 2 个执行线程和间歇时间为 1 秒的本地 StreamingContext 。



在使用这种背景下，我们可以创建一个代表从 TCP 源流数据的离散流（DStream），指定主机名（hostname）（例如 localhost）和端口（例如 9999）。

// 创建一个将要连接到 hostname:port 的离散流，如 localhost:9999

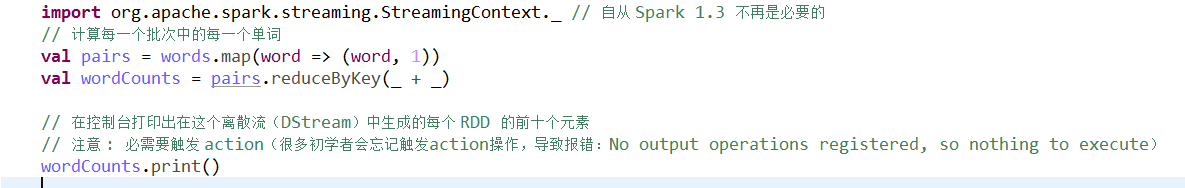
val lines = ssc.socketTextStream("localhost", 9999)

上一步的这个 lines 离散流（DStream）表示将要从数据服务器接收到的数据流。在这个 离散流（DStream）中的每一条记录都是一行文本（text）。接下来，我们想要通过空格字符（space characters）拆分这些数据行（lines）成单词（words）。

// 将每一行拆分成单词

val words = lines.flatMap(\_.split(" "))

flatMap 是一种一对多的离散流（DStream）操作，它会通过在源离散流（source DStream）中根据每个记录（record）生成多个新纪录的形式创建一个新的离散流（DStream）。在这种情况下，在这种情况下，每一行（each line）都将被拆分成多个单词（words）和代表单词离散流（words DStream）的单词流。接下来，我们想要计算这些单词。



上一步的 words 离散流进行了进一步的映射（一对一的转变）为一个 (word, 1)对 的离散流（DStream），这个离散流然后被规约（reduce）来获得数据中每个批次（batch）的单词频率。最后，wordCounts.print() 将会打印一些每秒生成的计数。

请注意，当这些行（lines）被执行的时候， Spark Streaming 只有建立在启动时才会执行计算，在它已经开始之后，并没有真正地处理。为了在所有的转换都已经完成之后开始处理，我们在最后运行 :

ssc.start() // 启动计算

ssc.awaitTermination() // 等待计算的终止

首先需要运行 Netcat（一个在大多数类 Unix 系统中的小工具）作为我们使用的数据服务器。

$ nc -lk 9999

然后，在另一个不同的终端，你可以运行这个例子通过执行 :

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

1. 基本概念

* 初始化 StreamingContext

为了初始化一个 Spark Streaming 程序，一个 StreamingContext 对象必须要被创建出来，它是所有的 Spark Streaming 功能的主入口点。

一个 [StreamingContext](http://spark.apache.org/docs/latest/api/scala/index.html" \l "org.apache.spark.streaming.StreamingContext)对象可以从一个 [SparkConf](http://spark.apache.org/docs/latest/api/scala/index.html" \l "org.apache.spark.SparkConf)对象中创建出来。

**import** org.apache.spark.**\_**

**import** org.apache.spark.streaming.**\_**

**val** conf **=** **new** SparkConf().setAppName(appName).setMaster(master)

**val** ssc **=** **new** StreamingContext(conf, Seconds(1))

一个 StreamingContext 对象也可以从一个现有的 SparkContext 对象中创建出。

**val** sc **=** ...                // existing SparkContext

**val** ssc **=** **new** StreamingContext(sc, Seconds(1))

* 一个 context 定义之后，你必须做以下几个方面。

通过创建输入 DStreams 定义输入源。

通过应用转换和输出操作 DStreams 定义流计算（streaming computations）。

开始接收数据，并用 streamingContext.start() 处理它。

等待处理被停止（手动停止或者因为任何错误停止）使用 StreamingContext.awaitTermination() 。

该处理可以使用 streamingContext.stop() 手动停止。

* 要记住的要点 :

一旦一个 context 已经启动，将不会有新的数据流的计算可以被创建或者添加到它。

一旦一个 context 已经停止，它不会被重新启动。

同一时间内在 JVM 中只有一个 StreamingContext 可以被激活。

在 StreamingContext 上的 stop() 同样也停止了 SparkContext 。为了只停止 StreamingContext ，设置 stop() 的可选参数，名叫 stopSparkContext 为 false 。

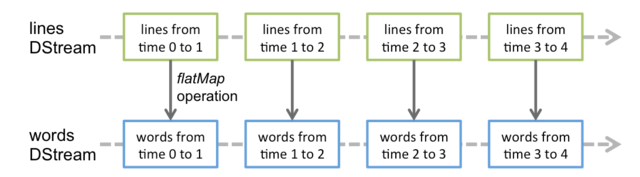
一个 SparkContext 可以被重新用于创建多个 StreamingContexts，只要是当前的 StreamingContext 被停止（不停止 SparkContext）之前创建下一个StreamingContext 就可以。

1. Discretized Streams（DStreams）（离散化流）

Discretized Stream（离散化流）或者 DStream（离散流）是 Spark Streaming 提供的基本抽象。它代表了一个连续的数据流，无论是从源接收到的输入数据流，还是通过变换输入流所产生的处理过的数据流。在内部，一个离散流（DStream）被表示为一系列连续的 RDDs，RDD 是 Spark 的一个不可改变的，分布式的数据集的抽象。在一个 DStream 中的每个 RDD 包含来自一定的时间间隔的数据，如下图所示。



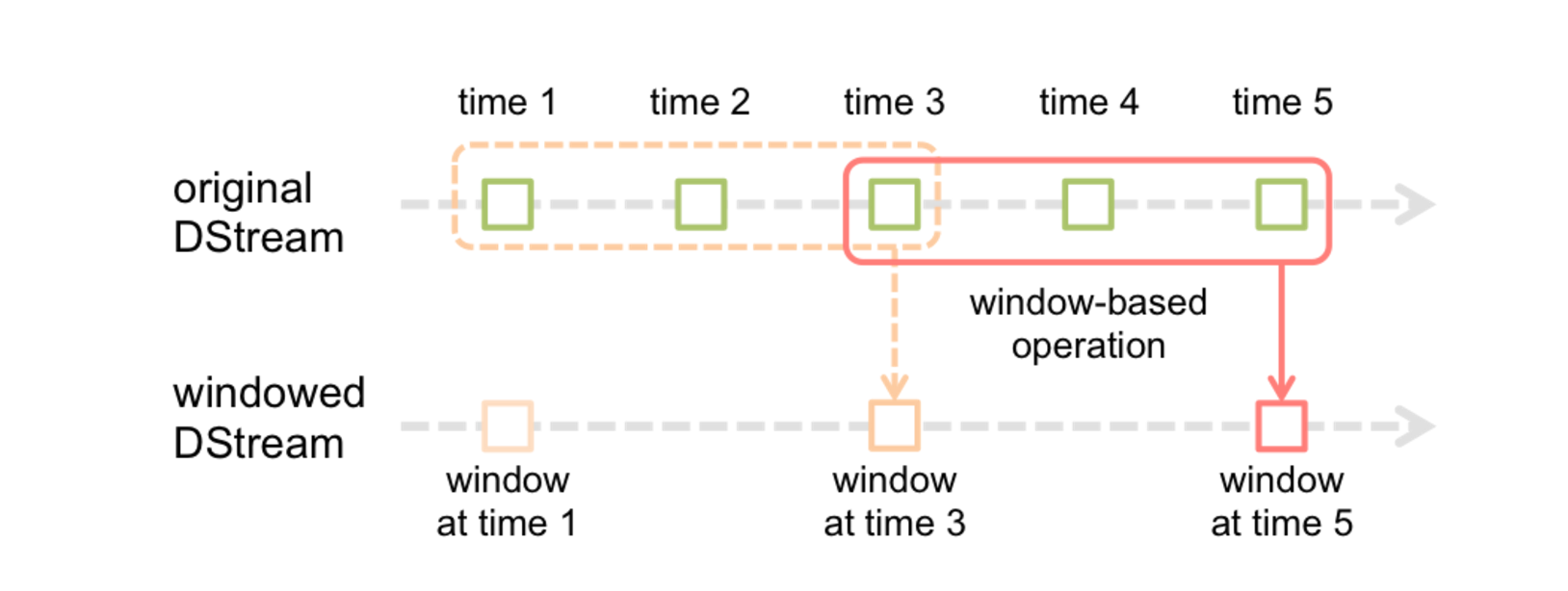
应用于 DStream 的任何操作转化为对于底层的 RDDs 的操作。例如，在先前的例子，转换一个行（lines）流成为单词（words）中，flatMap 操作被应用于在行离散流（lines DStream）中的每个 RDD 来生成单词离散流（words DStream）的 RDDs 。如下图所示。



这些底层的 RDD 变换由 Spark 引擎（engine）计算。 DStream 操作隐藏了大多数这些细节并为了方便起见，提供给了开发者一个更高级别的 API 。这些操作细节会在后边的章节中讨论。

1. Window （窗口）操作

Spark Streaming 也支持窗口计算，它允许你在一个滑动窗口数据上应用 transformation 算子。下图阐明了这个滑动窗口。



如上图显示，窗口在源 DStream 上滑动，合并和操作落入窗内的源 RDDs，产生窗口化的 DStream 的 RDDs。在这个具体的例子中，程序在三个时间单元的数据上进行窗口操作，并且每两个时间单元滑动一次。 这说明，任何一个窗口操作都需要指定两个参数 :

window length（窗口长度）: 窗口的持续时间。

sliding interval（滑动的时间间隔）: 窗口操作执行的时间间隔。

这两个参数必须是源 DStream 上  batch interval（批时间间隔）的倍数。

下面举例说明窗口操作。例如，你想扩展前面的[例子](http://www.apache.wiki/pages/viewpage.action?pageId=2885892)用来计算过去 30 秒的词频，间隔时间是 10 秒。为了达到这个目的，我们必须在过去 30 秒的 pairs DStream 上应用 reduceByKey 操作。用方法 reduceByKeyAndWindow 实现。

// Reduce last 30 seconds of data, every 10 seconds

**val** windowedWordCounts **=** pairs.reduceByKeyAndWindow((a**:**Int,b**:**Int) **=**> (a + b), Seconds(30), Seconds(10))

一些常用的窗口操作如下所示，这些操作都需要用到上文提到的两个参数 - 窗口长度和滑动的时间间隔。



1. DStreams 上的输出操作

输出操作允许 DStream 的操作推到如数据库、文件系统等外部系统中。因为输出操作实际上是允许外部系统消费转换后的数据，它们触发的实际操作是 DStream 转换。目前，定义了下面几种输出操作 :



1. DataFrame 和 SQL 操作

你可以很容易地使用 [DataFrames 和 SQL](http://spark.apache.org/docs/1.6.1/sql-programming-guide.html)Streaming 操作数据。 需要使用 SparkContext 或者正在使用的 StreamingContext 创建一个 SparkSession。这样做的目的就是为了使得驱动程序可以在失败之后进行重启。 使用懒加载模式创建单例的 SparkSession 对象。下面的示例所示。在原先的 [单词统计](http://spark.apache.org/docs/1.6.1/streaming-programming-guide.html" \l "a-quick-example)程序的基础上进行修改，使用 DataFrames 和 SQL 生成单词统计。 每个 RDD 转换为 DataFrame，注册为临时表,然后使用 SQL 查询。

/\*\* 流程序中的DataFrame操作 \*/

**val** words**:** DStream[String] **=** ...

words.foreachRDD { rdd **=**>

  // 获取单例的SQLContext

**val** sqlContext **=** SQLContext.getOrCreate(rdd.sparkContext)

**import** sqlContext.implicits.**\_**

  // 将RDD [String]转换为DataFrame

**val** wordsDataFrame **=** rdd.toDF("word")

  // 注册临时表

  wordsDataFrame.registerTempTable("words")

  // 在DataFrame上使用SQL进行字计数并打印它

**val** wordCountsDataFrame **=**

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

  wordCountsDataFrame.show()

}

1. 保证数据完整性

见代码TODO