大数据技术之 Spark Core学习内容

|  |  |  |
| --- | --- | --- |
|  | 文件标识： |  |
| 当前版本： | 1.0 |
| 作 者： | 马卫花 |
| 完成日期： | 2022-10-01 |

1. Spark 概述

**1.1** **Spark** 是什么

Apache Spark用于大规模数据分析的统一引擎，是一种多语言的引擎，用于在单节点或集群上执行数据工程、数据科学和机器学习。Spark是Apache软件基金会旗下的一个顶级项目，也是Apache软件基金会旗下最活跃的开源项目之一。

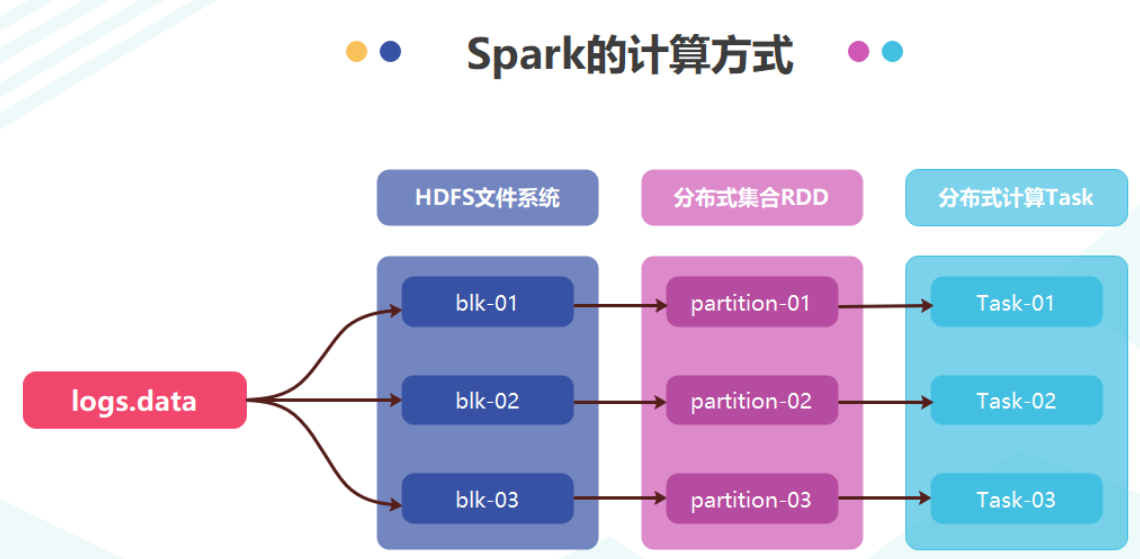




Spark 是一种基于内存的快速、通用、可扩展的大数据分析计算引擎。

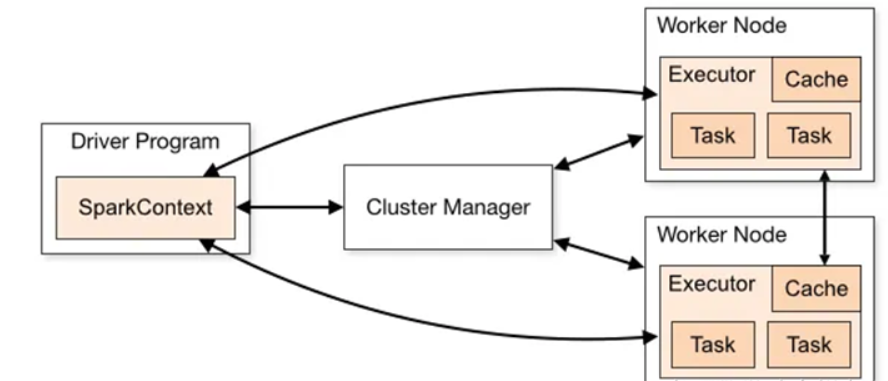
Spark诞生于加州大学伯克利分校AMP实验室，是一个基于**内存的分布式计算框架**。由于Spark是基于**内存**的，相对于Hadoop的MapReduce等计算框架大大提高了大数据处理的实时性，同时Spark也提供了**高容错性和可扩展性**。

加州大学伯克利分校的Matei Zaharia等任发表了一篇关于**Resilient Distributed DataSets**.论文地址：<https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf> 。论文中提到了分布式数据集的概念，它是一种分布式内存抽象，其使得程序员能够在扩大规模集群中做内存运算，并且有一定的容错方式.随虽然简单的一句话,但是Spark的核心数据结构，Spark整个平台都围绕**RDD**进行。



Spark最全的学习文档https://spark.apache.org/docs/3.2.1/index.html。

Spark的运行架构包括集群管理器（Cluster Manager）、运行作业任务的工作节点（Worker Node）、每个应用的任务控制节点（Driver）和每个工作节点上负责具体任务的执行进程（Executor）。其中，集群管理器可以是Spark自带的资源管理器，也可以是YARN或Mesos等资源管理框架。与Hadoop MapReduce计算框架相比，Spark所采用的Executor有两个优点。一是利用多线程来执行具体的任务（HadoopMapReduce采用的是进程模型），减少任务的启动开销。二是 Executor中有一个BlockManager存储模块，会将内存和磁盘共同作为存储设备，当需要多轮迭代计算时，可以将中间结果存储到这个存储模块里，下次需要时就可以直接读该存储模块里的数据，而不需要读写到 HDFS 等文件系统里，因而有效减少了 IO 开销；或者在交互式查询场景下，Executor预先将表缓存到该存储系统上，从而可以提高读写IO的性能。



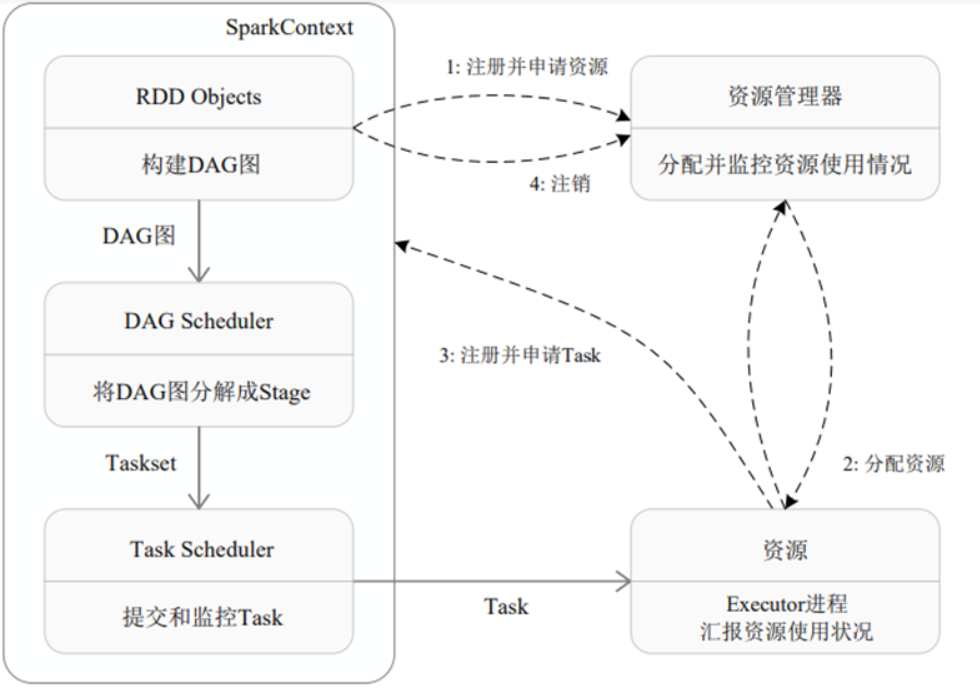
Spark运行基本流程如下图所示，流程如下。

（1）当一个Spark应用被提交时，首先需要为这个应用构建起基本的运行环境，即由任务控制节点（Driver）创建一个SparkContext，由SparkContext负责和资源管理器—Cluster Manager的通信，以及进行资源的申请、任务的分配和监控等。SparkContext 会向资源管理器注册并申请运行Executor的资源。

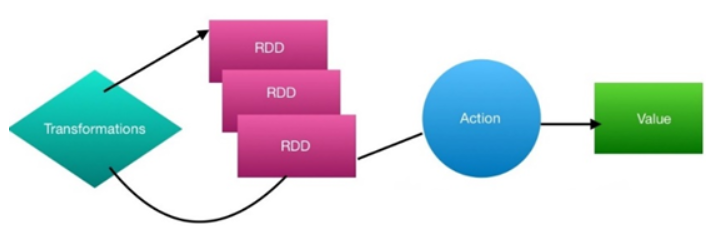
（2）资源管理器为Executor分配资源，并启动Executor进程，Executor运行情况将随着“心跳”发送到资源管理器上。

（3）SparkContext 根据 RDD 的依赖关系构建 DAG，并将 DAG 提交给 DAG 调度器（DAGScheduler）进行解析，将DAG分解成多个“阶段”（每个阶段都是一个任务集），并且计算出各个阶段之间的依赖关系，然后把一个个“任务集”提交给底层的任务调度器（TaskScheduler）进行处理；Executor 向 SparkContext 申请任务，任务调度器将任务分发给 Executor 运行，同时SparkContext将应用程序代码发放给Executor。

（4）任务在Executor上运行，把执行结果反馈给任务调度器，然后反馈给DAG调度器，运行完毕后写入数据并释放所有资源。



Spark的核心建立在统一的抽象RDD之上，这使得Spark的各个组件可以无缝地进行集成，在同一个应用程序中完成大数据计算任务。一个RDD就是一个分布式对象集合，本质上是一个只读的分区记录集合。每个RDD可以分成多个分区，每个分区就是一个数据集片段，并且一个 RDD 的不同分区可以被保存到集群中不同的节点上，从而可以在集群中的不同节点上进行并行计算。RDD提供了一组丰富的操作以支持常见的数据运算，分为“行动”（Action）和“转换”（Transformation）两种类型，前者用于执行计算并指定输出的形式，后者指定RDD之间的相互依赖关系。两类操作的主要区别是，转换操作（如map、filter、groupBy、join等）接受RDD并返回RDD，而行动操作（如count、collect等）接受RDD但是返回非RDD（即输出一个值或结果）。RDD采用了惰性调用，即在RDD的执行过程中，真正的计算发生在RDD的“行动”操作，对于“行动”之前的所有“转换”操作，Spark只是记录下“转换”操作应用的一些基础数据集以及RDD生成的轨迹，即相互之间的依赖关系，而不会触发真正的计算。



**1.2** **Spark** **or** **Hadoop**

Hadoop 的 MR 框架和 Spark 框架都是数据处理框架，那么我们在使用时如何选择呢？（1）Hadoop MapReduce 由于其设计初衷并不是为了满足循环迭代式数据流处理， 因此在多并行运行的数据可复用场景（如：机器学习、图挖掘算法、交互式数据挖掘算法）中存在诸多计算效率等问题。所以 Spark 应运而生，Spark 就是在传统的MapReduce计算框 架的基础上，利用其计算过程的优化，从而大大加快了数据分析、挖掘的运行和读写速度，并将计算单元缩小到更适合并行计算和重复使用的 RDD 计算模型。（2）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。而Spark 所基于的scala语言恰恰擅长函数的处理。（3）Spark 是一个分布式数据快速分析项目。它的核心技术是弹性分布式数据集（Resilient Distributed Datasets）提供了比 MapReduce 丰富的模型，可以快速在内存中对数据集 进行多次迭代，来支持复杂的数据挖掘算法和图形计算算法。（4）Spark 和Hadoop 的根本差异是多个作业之间的数据通信问题 : Spark多个作业之间数据通信是基于内存，而 Hadoop 是基于磁盘。

经过上面的比较， 我们可以看出在绝大多数的数据计算场景中， Spark 确实会比MapReduce 更有优势。但是 Spark 是基于内存的， 所以在实际的生产环境中， 由于内存的限制， 可能会 由于内存资源不够导致 Job 执行失败， 此时， MapReduce 其实是一个更好的选择， 所以 Spark 并不能完全替代 MR。

提示：尽管Spark相对于Hadoop而言具有较大的又是，但是Spark并不能完全替代Hadoop.

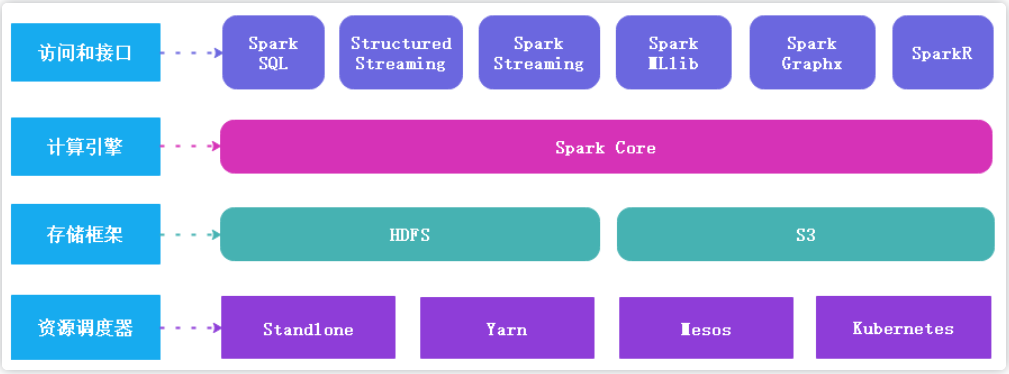
（1）在计算层面，Spark相对于MR有巨大的性能优势，但至今仍有许多工具基于MR构架，例如非常熟悉的Hive ,虽然可以变更为Spark引擎，但是运行依然离不开MR。

（2）Spark仅做计算，而Hadoop生态圈不仅有计算也有存储和资源管理调度。HDFS和YARN仍是许多大数据体系的核心架构。

Spark的四大特征如下图所示：（速度快、使用简单、统一、可扩展）



**2.3** **Spark** **模块**



* Spark Core

Spark Core是Spark的核心，是Spark的基本功能，包含任务调度、内存管理、错误恢复、与存储系统交互等模块。Spark Core中还包含了对弹性分布式数据集(Resilient distributed dataset,简称RDD)的API定义。

* Spark SQL

以Spark Core为基础，Spark SQL 是Spark用来操作结构化数据的程序包。通过Spark SQL,我们可使用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。

* Structured Streaming

Strutured Streaming的关键思想是将持续不断的数据当作一个不断追加的表。将输入的流数据当作一张“输入表”。把每一条到达的数据作为输入表的新的一行来追加。以Spark SQL为基础，进行数据的流式计算。

* Spark Streaming

Spark Streaming采用微批的处理方法。每一个批处理间隔的为一个批，也就是一个RDD，我们对RDD进行操作就可以源源不断的接收、处理数据。

* GraphX

GraphX是Spark中用于图形和图形并行计算的新组件。

* Spark MLIb

以Spark Core为基础，提供常见的机器学习功能的程序库。包括分类、回归、聚类、协同过滤等，还提供模型评估、数据导入等功能。

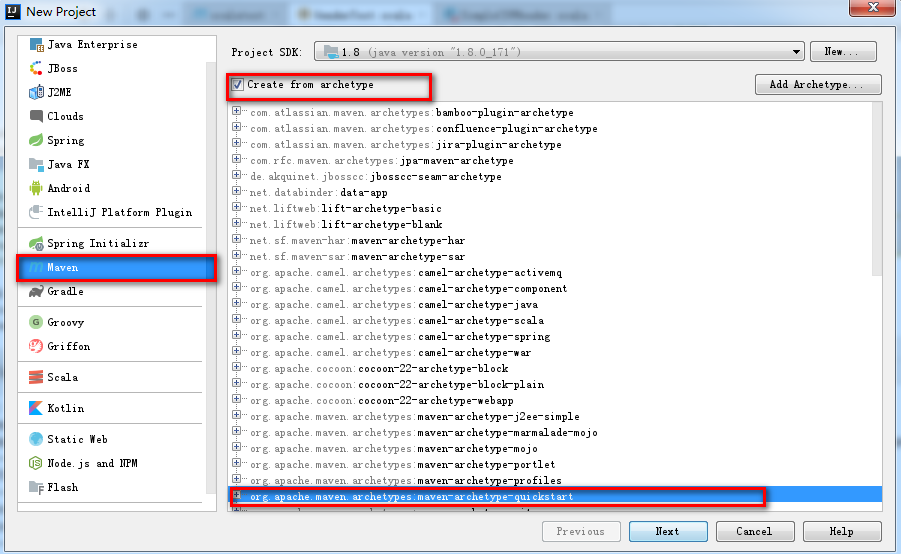
2 **Spark** 快速上手

在大数据早期的课程中我们已经学习了 MapReduce 框架的原理及基本使用， 并了解了 其底层数据处理的实现方式。

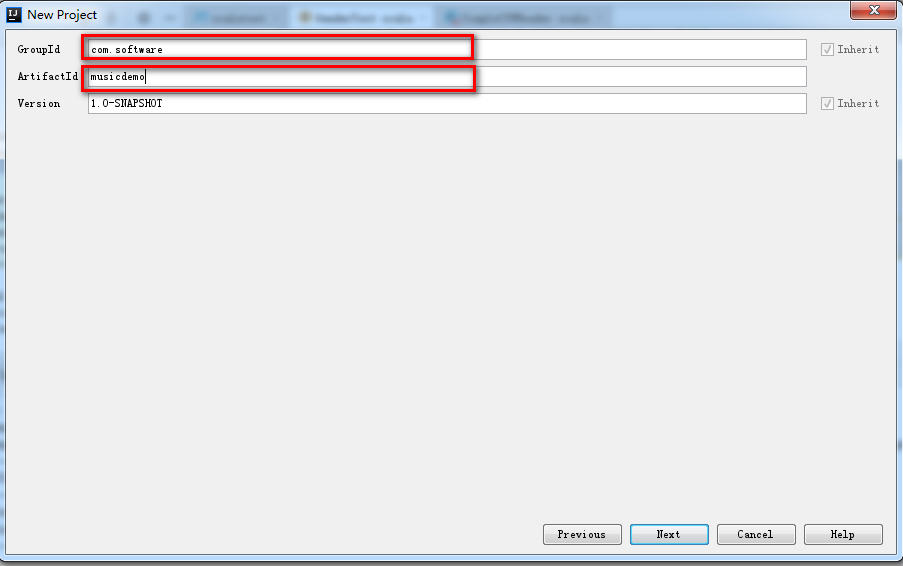
## 2.1 项目配置

1）新建maven项目

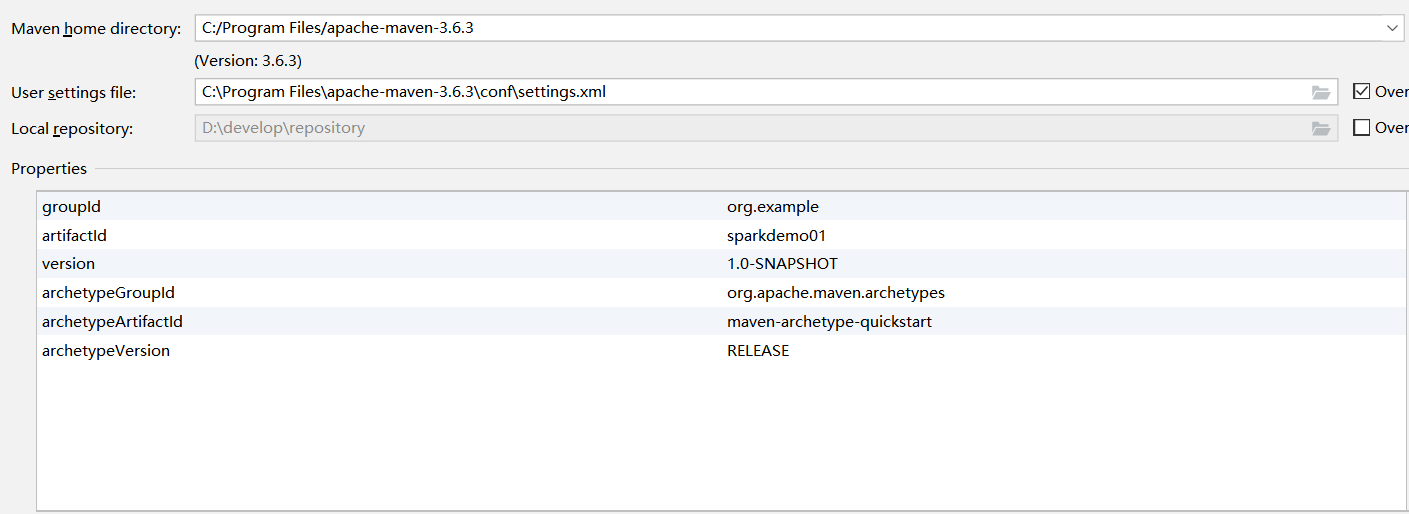
打开IDEA，选择file->new project新建maven项目，如下图所示：



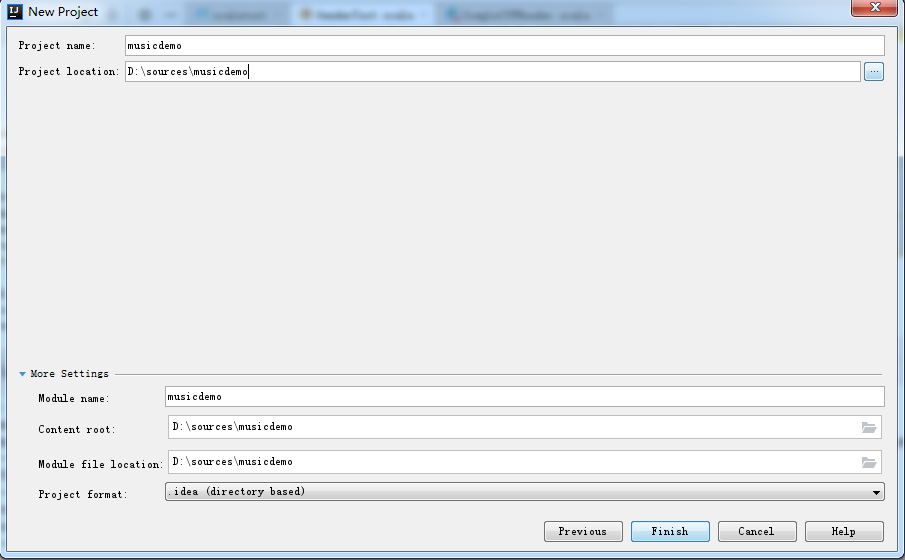
点击next按钮，配置maven相关属性



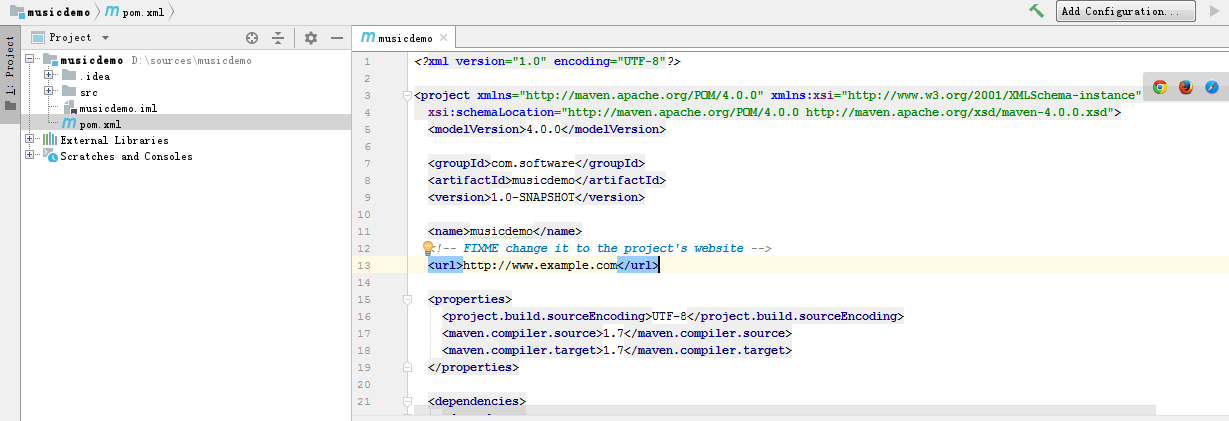
点击Next，设置maven的安装目录以及本地数据仓库



单击next进入下一页,选择项目的路径



设置完成后点击Finish按钮，项目的目录结构如下图所示：



1. 配置pom.xml文件

打开项目中的pom.xml文件，设置如下的配置信息，然后实现自动导包的操作。

|  |
| --- |
| <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>  <groupId>org.example</groupId>  <artifactId>scala\_demo</artifactId>  <version>1.0-SNAPSHOT</version>  <name>scala\_demo</name>  <!-- FIXME change it to the project's website -->  <url>http://www.example.com</url>  <properties>  <spark.version>3.0.0</spark.version>  <scala.version>2.12</scala.version>  </properties>  <dependencies>  <dependency>  <groupId>org.apache.spark</groupId>  <artifactId>spark-core\_${scala.version}</artifactId>  <version>${spark.version}</version>  </dependency>    <dependency>  <groupId>org.apache.spark</groupId>  <artifactId>spark-sql\_${scala.version}</artifactId>  <version>${spark.version}</version>  </dependency>  </dependencies>  </project> |

## 2.2 统计单词个数案例

[示例代码]:

def main(args: Array[String]): Unit = {

//1.创建Spark运行配置对象

val sparkConf = new SparkConf().setMaster("local").setAppName("WordCountExample")

//2.创建Spark上下文环境对象

var sc: SparkContext = new SparkContext(sparkConf)

//val inputFile = args(0)

val inputFile = "src\\words.txt"

//3.读取文件数据（外部源文件）

var fileRdd: RDD[String] = sc.textFile(inputFile)

//4.按照空格切分文件中的单词

var wordSplitRDD: RDD[String] = fileRdd.flatMap(data => data.split(" "))

//var wordsSplitResult =fileRdd.flatMap(data => data.split(" "))

//wordsSplitResult.foreach(println)

//5.转换结构(单词，初始值：1)

var wordRddInit: RDD[(String, Int)] = wordSplitRDD.map(data => (data, 1))

var wordTotal: RDD[(String, Int)] = wordRddInit.reduceByKey(\_ + \_)

var tuples: Array[(String, Int)] = wordTotal.collect()

//输出结果

tuples.foreach(println)

//关闭SparkContext对象

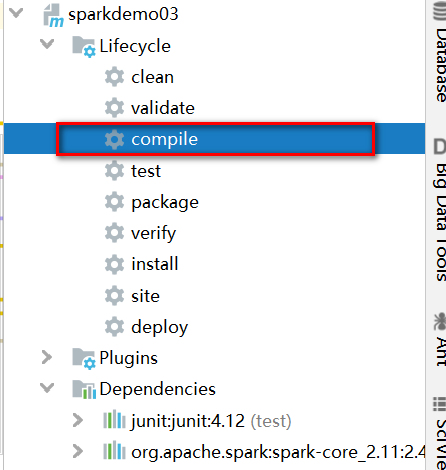
sc.stop()

}

1. 重新编译项目

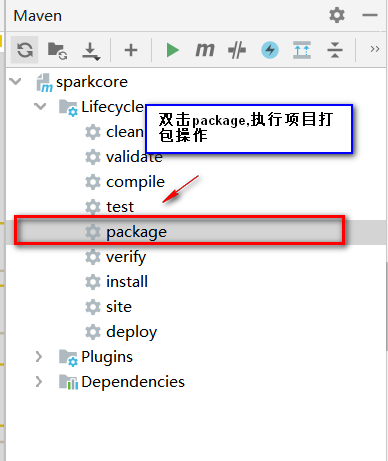
Idea环境中 选择build菜单-->Rebuild project,重新编译项目

也可以选择compile命令

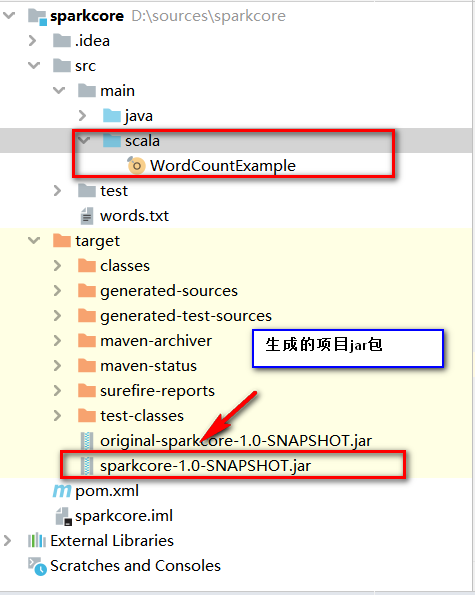


1. 项目打包

选择Idea环境，Maven窗口中的Lifecycle-->package,双击package执行项目打包操作，



执行后，在项目的target目录下生成项目的jar包,项目的结构如下图所示：



1. 在虚拟机上运行

（1） 将解析的文件words.txt上传至HDFS

（2）在Spark的安装目录下执行如下的命令：

./bin/spark-submit --class com.software.client.WordCountExample --master local sparkdemo-1.0-SNAPSHOT.jar /wordcount

3. **Spark** 核心编程

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。

**3.1.1** 什么是 **RDD**

RDD（Resilient Distributed Dataset）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。在Spark的操作中，对数据的所有操作不外乎创建RDD、转化已有的RDD以及输出RDD操作进行求值.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示地将工作集缓存在内存中，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：转化操作和行动操作。RDD的转化操作时返回一个新的RDD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者把结果写入外部系统的操作。比如count()和first().

**3.1.2 基础编程**

* **RDD 创建**

在 Spark 中创建 RDD（resilient distributed dataset） 的创建方式可以分为四种：

**1)** 从集合（内存）中创建 **RDD**

RDD是一个容错的、只读的、可进行并行操作的数据结构，是一个分布在集群中各个节点的存放元素的集合。RDD有3种不同的创建方法。一种是对程序种存在的基本数据结构种的集合进行并行化（Set、List、Array），另一种是通过已有RDD转化得到新的RDD,这两种都是通过内存已有集合创建RDD.还有一种是直接读取外部存储的数据集。

SparkContext类中有两个方法：parallelize和makeRDD.通过parallelize或makeRDD可将单机数据创建为分布式RDD.这两种方法都是利用内存中已存在的集合，复制集合中的元素去创建一个可用于并行计算的分布式数据集RDD.

1. parallelize可以输入：要转化的集合，必须是是Seq集合,Seq集合是一种序列。所谓的序列值得是一类具有一定长度的可迭代访问的对象，其中每个元素均带有一个从0开始计数的固定索引位置。

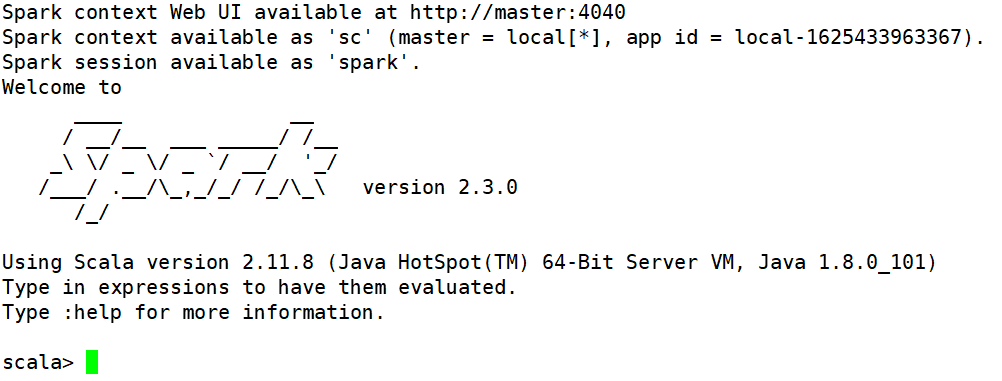
第二个参数表示分区数，不设分区数，默认为该Application分配到的资源的CPU数。

创建RDD的操作步骤：

启动Spark Shell交互式命令终端，如下所示：

|  |
| --- |
| [root@master bin]# ./spark-shell |

显示如下图所示的效果:



设置日志级别，命令如下所示:

|  |
| --- |
| scala> sc  res0: org.apache.spark.SparkContext = org.apache.spark.SparkContext@7f2c995b  scala> org.apache.log4j.Logger.getLogger("org").setLevel(org.apache.log4j.Level.WARN)  scala> org.apache.log4j.Logger.getLogger("akka").setLevel(org.apache.log4j.Level.WARN)  scala> sc.setLogLevel("WARN")  scala> sc.setLogLevel("INFO") |

从集合中创建RDD

( 1) parallelize():通过parallelize函数把一般数据结构加载为RDD

def parallelize[T:ClassTag](seq:Seq[T],numSlices:Int = defaultParallelise):RDD[T]

通过parallelize创建一个RDD,默认分区个数为2.设置分区个数为3后创建RDD,查询结果显示分区个数为3,命令如下所示:

|  |
| --- |
| scala> val list = List("a","b","c","d")  list: List[String] = List(a, b, c, d)  scala> val rdd1 = sc.parallelize(list)  rdd1: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[0] at parallelize at <console>:26  scala> rdd1.partitions.size  res5: Int = 2  scala> val list = List("a","b","c","d")  list: List[String] = List(a, b, c, d) |

( 2) makeRDD():通过parallelize函数把一般数据结构加载为RDD

makeRDD有两种方法实现：一种是跟parallelize完全一致；另一种是接收的参数类型是Seq[(T,Seq[String])].第二种方法生成的RDD中保存的是T的值，但是Seq[String]部分的数据会按照Seq[(T,Seq[String])]的顺序存放到各个分区中，一个Seq[String]对应存到一个分区，为数据提供位置信息。通过presferredLocations可以根据位置信息查看每个分区的值。这种makeRDD的实现不可以自己制定RDD的分区个数，而是固定为Seq[String]参数的个数。

|  |
| --- |
| scala> val list=sc.makeRDD(List("a","b","c","d"))  scala> list.partitions.size  scala> val list = sc.makeRDD(List("a","b","c","d"),3)  scala> list1.partitions.size  scala> val list=List((1,List("a","b")),(2,List("c","d")))  scala> val list1 = sc.makeRDD(list)  scala> list1.partitions.size  res2: Int = 2  scala> list1.preferredLocations(list1.partitions(0))  res4: Seq[String] = List(a, b)  scala> list1.preferredLocations(list1.partitions(1))  res5: Seq[String] = List(c,d) |

**(3)** 从外部存储（文件） 创建 **RDD**

由外部存储系统的数据集创建 RDD 包括：本地的文件系统，所有 Hadoop 支持的数据集，比如 HDFS 、HBase 等。从外部存储创建RDD是指直接读取一个存放在文件系统的数据文件创建RDD,第一种创建RDD的方式常用于测试。从外部读取数据创建RDD可以有很多种数据来源，通过SparkContext对象的textFile方法读取数据集，支持多种类型数据集，如目录、文本文件、压缩文件和通配符匹配的文件等，并且允许设定分区个数。

* 从HDFS文件创建RDD

直接通过testFile命令读取HDFS文件的位置即可，如下所示的命令:

|  |
| --- |
| scala> val test =sc.textFile("/input/input.txt")  test: org.apache.spark.rdd.RDD[String] = /input/input.txt MapPartitionsRDD[4] at textFile at <console>:24 |

* 从Linux本地文件创建RDD

通过testFile命令，在文件的路径前面加上<file://>表示从本地Linux文件系统读取。命令如下所示：

|  |
| --- |
| scala> val test=sc.textFile("file:///home/data/word.txt")  test: org.apache.spark.rdd.RDD[String] = file:///home/data/word.txt MapPartitionsRDD[6] at textFile at <console>:24  scala> test.count  res6: Long = 10 |

**3)** 从其他 **RDD** 创建

主要是通过一个 RDD 运算完后， 再产生新的 RDD 。

**4)** 直接创建 **RDD**（**new**）

使用new 的方式直接构造 RDD，一般由 Spark 框架自身使用。

* **RDD 转换算子**

RDD 根据数据处理方式的不同将算子整体上分为 Value 类型、双 Value 类型和 Key-Value 类型

⚫ **Value** 类型

**1)** **map**

➢ 函数签名

def map[U: ClassTag](f: T => U): RDD[U]

➢ 函数说明

将处理的数据逐条进行映射转换，这里的转换可以是类型的转换， 也可以是值的转换。

|  |
| --- |
| val distData = sc.parallelize(List(1,3,45,3,76))  val sq\_dist = distData.map(x=>x\*x) |

示例：使用map函数产生键值对RDD

|  |
| --- |
| scala> val data = sc.parallelize(List("hello","world","my","love"))  data: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[9] at parallelize at <console>:24  scala> data.map(x=>(x,1))  res8: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[10] at map at <console>:26  scala> res8.collect  res12: Array[(String, Int)] = Array((hello,1), (world,1), (my,1), (love,1)) |

提示：map将原来RDD的每个数据项通过map中的用户自定义的函数f转换成一个新的RDD,map操作不会改变RDD的分区个数。

**2)** **flatMap**

➢ 函数签名

def flatMap[U: ClassTag](f: T => TraversableOnce[U]): RDD[U]

➢ 函数说明

将处理的数据进行扁平化后再进行映射处理， 所以算子也称之为扁平映射

|  |
| --- |
| val dataRDD = sc.makeRDD(List(  List(1,2),List(3,4)  ),1)  val dataRDD1 = dataRDD.flatMap (  list => list  ) |

❖ 小功能： 将 List(List(1,2),3,List(4,5))进行扁平化操作

**3)** **groupBy**

➢ 函数签名

def groupBy[K](f: T => K)(implicit kt: ClassTag[K]): RDD[(K, Iterable[T])]

➢ 函数说明

将数据根据指定的规则进行分组, 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。

|  |
| --- |
| val dataRDD = sc.makeRDD(List(1,2,3,4),1)  val dataRDD1 = dataRDD.groupBy(\_%2) |

**4)** **filter**

➢ 函数签名

def filter(f: T => Boolean): RDD[T]

➢ 函数说明

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。

|  |
| --- |
| val dataRDD = sc.makeRDD(List(1,2,3,4),1)  val dataRDD1 = dataRDD.filter(\_%2 == 0) |

**5)** **distinct**

➢ 函数签名

def distinct()(implicit ord: Ordering[T] = null): RDD[T]

def distinct(numPartitions: Int)(implicit ord: Ordering[T] = null): RDD[T]

➢ 函数说明

将数据集中重复的数据去重

val dataRDD = sc.makeRDD(List(

1,2,3,4,1,2

),1)

val dataRDD1 = dataRDD.distinct()

**6)** **sortBy**

➢ 函数签名

def sortBy[K](

f: (T) => K,

ascending: Boolean = true,

numPartitions: Int = this.partitions.length)

(implicit ord: Ordering[K], ctag: ClassTag[K]): RDD[T]

➢ 函数说明

该操作用于排序数据。在排序之前可以将数据通过f函数进行处理,之后按照f函数处理的结果进行排序，默认为升序排列。排序后新产生的 RDD 的分区数与原 RDD 的分区数一致。中间存在 shuffle 的过程。

val dataRDD = sc.makeRDD(List(

1,2,3,4,1,2

),2)

val dataRDD1 = dataRDD.sortBy(num=>num, false)

⚫ 双 **Value** 类型

**7)** **intersection**

➢ 函数签名

def intersection(other: RDD[T]): RDD[T]

➢ 函数说明

对源 RDD 和参数 RDD 求交集后返回一个新的 RDD

val dataRDD1 = sc.makeRDD(List(1,2,3,4))

val dataRDD2 = sc.makeRDD(List(3,4,5,6))

val dataRDD = dataRDD1.intersection (dataRDD2)

**8)** **union**

➢ 函数签名

def union(other: RDD[T]): RDD[T]

➢ 函数说明

对源 RDD 和参数 RDD 求并集后返回一个新的 RDD

val dataRDD1 = sc.makeRDD(List(1,2,3,4))

val dataRDD2 = sc.makeRDD(List(3,4,5,6))

val dataRDD = dataRDD1.union (dataRDD2)

**9)** **subtract**

➢ 函数签名

def subtract(other: RDD[T]): RDD[T]

➢ 函数说明

以一个 RDD 元素为主， 去除两个 RDD 中重复元素， 将其他元素保留下来。求差集

val dataRDD1 = sc.makeRDD(List(1,2,3,4))

val dataRDD2 = sc.makeRDD(List(3,4,5,6))

val dataRDD = dataRDD1.subtract (dataRDD2)

**10)** **zip**

➢ 函数签名

def zip[U: ClassTag](other: RDD[U]): RDD[(T, U)]

➢ 函数说明

将两个 RDD 中的元素， 以键值对的形式进行合并。其中， 键值对中的 Key 为第 1 个 RDD中的元素， Value 为第 2 个 RDD 中的相同位置的元素。

val dataRDD1 = sc.makeRDD(List(1,2,3,4))

val dataRDD2 = sc.makeRDD(List(3,4,5,6))

val dataRDD = dataRDD1.zip (dataRDD2)

⚫ **Key** **-** **Value** 类型

**11)** **reduceByKey（重点）**

➢ 函数签名

def reduceByKey(func: (V, V) => V): RDD[(K, V)]

def reduceByKey(func: (V, V) => V, numPartitions: Int): RDD[(K, V)]

➢ 函数说明

可以将数据按照**相同的 Key 对 Value 进行聚合**

val dataRDD1 = sc.makeRDD(List(("a",1),("b",2),("c",3),(“a”,4),(“a”,5)))

val dataRDD2 = dataRDD1.reduceByKey((x,y) => x + y)

val dataRDD3 = dataRDD1.reduceByKey(\_+\_, 2)

**12)** **groupByKey**

➢ 函数签名

def groupByKey(): RDD[(K, Iterable[V])]

def groupByKey(numPartitions: Int): RDD[(K, Iterable[V])]

def groupByKey(partitioner: Partitioner): RDD[(K, Iterable[V])]

➢ 函数说明

将数据源的数据根据 key 对 value 进行分组

val dataRDD1 = sc.makeRDD(List(("a",1),("b",2),("c",3)))

val dataRDD2 = dataRDD1.groupByKey ()

val dataRDD3 = dataRDD1.groupByKey (2)

思考一个问题： reduceByKey 和 groupByKey 的区别？

从 **shuffle** 的角度：reduceByKey 和 groupByKey 都存在 shuffle 的操作， 但是 reduceByKey 可以在 shuffle 前对分区内相同 key 的数据进行预聚合（combine）功能， 这样会减少落盘的 数据量， 而 groupByKey 只是进行分组，不存在数据量减少的问题， reduceByKey 性能比较 高。

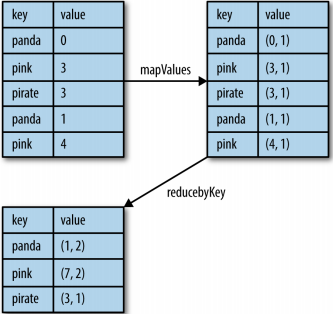
从功能的角度：reduceByKey 其实包含分组和聚合的功能。 GroupByKey 只能分组，不能聚 合， 所以在分组聚合的场合下， 推荐使用 reduceByKey，如果仅仅是分组而不需要聚合。那 么还是只能使用 groupByKey

**13)** mapValues

➢ 函数说明

mapValues是针对键值对(key,value)类型的数据中的value进行Map操作，而不是对key进行处理。针对键值对RDD rdd1的value进行map映射，把原来的value映射成(value,10)

|  |
| --- |
| val data =sc.parallelize(List("first","second"))  val datamap=data.map(x=>(x,1))  datamap.collect  val result=datamap.mapValues(y=>(y,10))  result.collect |



data RDD读取原始数据，经过mapValues后转换为rdd1,rdd1的value包含原始数据的个数以及附带值1；rdd2经过rdd1的reduceByKey,把相同的key的value整合,整合后的value的操作是把对应的个数相加，所以最后结果中的value的第一个字段就是总个数，而第二个字段则是原始数据的行数.

|  |
| --- |
| val data=sc.parallelize(List(("panda",0),("pink",3),("panda",1),("pink",4)))  val rddf1=data.mapValues(y=>(y,1))  rddf1.collect  val rdd2 = rddf1.reduceByKey((y1,y2)=>(y1.\_1 + y2.\_1,y1.\_2+y2.\_2))  rdd2.collect |

**14)** **foldByKey**

➢ 函数签名

def foldByKey(zeroValue: V)(func: (V, V) => V): RDD[(K, V)]

➢ 函数说明

当分区内计算规则和分区间计算规则相同时， aggregateByKey 就可以简化为 foldByKey

val dataRDD1 = sc.makeRDD(List(("a",1),("b",2),("a",3)))

val dataRDD2 = dataRDD1.foldByKey (0)(\_+\_)

**15)**  **combineByKey(难点)**

➢ 函数签名

def combineByKey[C](

createCombiner: V => C,

mergeValue: (C, V) => C,

mergeCombiners: (C, C) => C): RDD[(K, C)]

➢ 函数说明

def combineByKey[C](

**createCombiner: V => C,**      //组合器函数，用于将V类型转换成C类型，输入参数为RDD[K,V]中的V,输出为C

mergeValue: (C, V) => C,       //合并值函数，将一个C类型和一个V类型值合并成一个C类型，输入参数为(C,V)，输出为C

 mergeCombiners: (C, C) => C,        //合并组合器函数，用于将两个C类型值合并成一个C类型，输入参数为(C,C)，输出为C

partitioner: Partitioner,      //结果RDD分区数，默认保持原有的分区数

mapSideCombine: Boolean = true,       //分区函数,默认为HashPartitioner

 serializer: Serializer = null)  //是否需要在Map端进行combine操作，类似于MapReduce中的combine，默认为true

聚合操作会遍历分区中的所有元素，因此每个元素（这里指的是键值对）的键只有两种情况：以前没有出现过或以前出现过。

（ 1）如果以前没有出现过，则执行的是createCombiner方法，createCombiner()会在遇到的键对应的累加器中赋予初始值，否则执行mergeValue方法.

( 2) 对于已经出现过的键( key),调用mergeValue来进行聚合操作，对该键的累加器对应的当前值( C格式) 与新值（ V格式）进行合并。

（ 3）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（全是C格式）进行合并。

小练习： 将数据 List(("a", 88), ("b", 95), ("a", 91), ("b", 93), ("a", 95), ("b", 98))求每个 key 的平

均值

val list: List[(String, Int)] = List(("a", 88), ("b", 95), ("a", 91), ("b", 93),("a", 95), ("b", 98))

val input: RDD[(String, Int)] = sc.makeRDD(list, 2)

val combineRdd: RDD[(String, (Int, Int))] = input.combineByKey (

(\_, 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) )

val avg\_score = combineRDD.map(x=>(x.\_1,(x.\_2.\_1.toDouble/x.\_2.\_2)))

**16)** **sortByKey**

➢ 函数签名

def sortByKey(ascending: Boolean = true, numPartitions: Int = self.partitions.length)

: RDD[(K, V)]

➢ 函数说明

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

的

val dataRDD1 = sc.makeRDD(List(("a",1),("b",2),("c",3)))

val sortRDD1: RDD[(String, Int)] = dataRDD1.sortByKey(true)

val sortRDD1: RDD[(String, Int)] = dataRDD1.sortByKey(false)

**17)** **join**

➢ 函数签名

defjoin[W](other: RDD[(K, W)]): RDD[(K, (V, W))]

➢ 函数说明

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

(K,(V,W))的 RDD

val rdd: RDD[(Int, String)] = sc.makeRDD(Array((1, "a"),(2, "b"), (3, "c")))

val rdd1: RDD[(Int, Int)] = sc.makeRDD(Array((1, 4), (2, 5), (3, 6)))

rdd.join (rdd1).collect().foreach(println)

**18)** **leftOuterJoin**

➢ 函数签名

def leftOuterJoin[W](other: RDD[(K, W)]): RDD[(K, (V, Option[W]))]

➢ 函数说明

类似于 SQL 语句的左外连接

val dataRDD1 = sc.makeRDD(List(("a",1),("b",2),("c",3)))

val dataRDD2 = sc.makeRDD(List(("a",1),("b",2),("c",3)))

val rdd: RDD[(String, (Int, Option[Int]))] = dataRDD1.leftOuterJoin (dataRDD2)

* **RDD 行动算子**

**1)** **reduce**

➢ 函数签名

def reduce(f: (T, T) => T): T

➢ 函数说明

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

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

// 聚合数据

val reduceResult:Int = rdd.reduce(\_+\_)

**2)** **collect**

➢ 函数签名

def collect(): Array[T]

➢ 函数说明

在驱动程序中， 以数组 Array 的形式返回数据集的所有元素

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

// 收集数据到 Driver

rdd.collect().foreach(println)

**3)** **count**

➢ 函数签名

def count(): Long

➢ 函数说明

返回 RDD 中元素的个数

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

// 返回 RDD 中元素的个数

val countResult: Long = rdd.count()

**4)** **first**

➢ 函数签名

def first(): T

➢ 函数说明

返回 RDD 中的第一个元素

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

// 返回 RDD 中第一个元素

val firstResult: Int = rdd.first()

println(firstResult)

**5)** **take**

➢ 函数签名

def take(num: Int): Array[T]

➢ 函数说明

返回一个由 RDD 的前 n 个元素组成的数组

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

// 返回 RDD 中元素的个数

val takeResult: Array[Int] = rdd.take(2)

println(takeResult.mkString(","))

**6)** **takeOrdered**

➢ 函数签名

deftakeOrdered(num: Int)(implicit ord: Ordering[T]): Array[T]

➢ 函数说明

返回该 RDD 排序后的前 n 个元素组成的数组

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

// 返回 RDD 中元素的个数

val result: Array[Int] = rdd.takeOrdered(2)

**7)** **fold**

➢ 函数签名

def fold(zeroValue: T)(op: (T, T) => T): T

➢ 函数说明

折叠操作

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

val foldResult: Int = rdd.fold(0)(\_+\_)

**8)** **countByKey**

➢ 函数签名

defcountByKey(): Map[K, Long]

➢ 函数说明

统计每种 key 的个数

val rdd: RDD[(Int, String)] = sc.makeRDD(List((1, "a"), (1, "a"), (1, "a"), (2, "b"), (3, "c"), (3, "c")))

// 统计每种 key 的个数

val result: collection.Map[Int, Long] = rdd.countByKey()

**9)** **save** 相关算子

➢ 函数签名

def saveAsTextFile(path: String): Unit

def saveAsObjectFile(path: String): Unit

def saveAsSequenceFile(

path: String,

codec: Option[Class[\_ <: CompressionCodec]] = None): Unit

➢ 函数说明

将数据保存到不同格式的文件中

// 保存成 Text 文件

rdd.saveAsTextFile ("output")

// 序列化成对象保存到文件

rdd.saveAsObjectFile ("output1")

**11)** **foreach**

➢ 函数签名

def foreach(f: T => Unit): Unit = withScope {

}

val cleanF = sc.clean(f)

sc.runJob(this, (iter: Iterator[T]) => iter.foreach(cleanF))

函数说明

分布式遍历 RDD 中的每一个元素，调用指定函数

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

// 收集后打印

rdd.map (num=>num).collect().foreach(println)

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

// 分布式打印

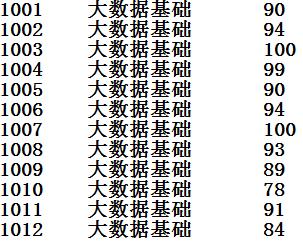
rdd.foreach (println)

4. **Spark** 练习任务

## 1 任务1

创建三个文件保存本次考试学生成绩：学生信息文件(student.txt),存储学生的学号、姓名；Hadoop考试成绩的文件(hadoop.txt),存储学生的学号、考试科目以及考试成绩；Spark考试成绩的文件（spark.txt）,存储学生的学号、考试科目以及考试成绩。Hadoop考试成绩以及Scala考试成绩的数据存储如下图所示。

要求：讲这三个文件上传至HDFS上，然后通过Spark的算子操作加载外部文件转换为RDD对象。



|  |
| --- |
| [root@master student]# hdfs dfs -put result\_bigdata.txt /student  [root@master student]# hdfs dfs -put result\_math.txt /student  scala> val bigdata = sc.textFile("/student/result\_bigdata.txt")  bigdata: org.apache.spark.rdd.RDD[String] = result\_bigdata.txt MapPartitionsRDD[20] at textFile at <console>:24  scala> **val math=sc.textFile("/student/result\_math.txt")**  math: org.apache.spark.rdd.RDD[String] = result\_math.txt MapPartitionsRDD[22] at textFile at <console>:24 |

## 2 任务2

查询学生成绩表中**前5名**的学生成绩信息。

|  |
| --- |
| scala> val m\_bigdata=bigdata.map{x=>val line = x.split(",");(line(0),line(1),line(2).toInt)}  m\_bigdata: org.apache.spark.rdd.RDD[(String, String, Int)] = MapPartitionsRDD[27] at map at <console>:25  scala> val m\_math=math.map{x=>val line = x.split("\t");(line(0),line(1),line(2).toInt)}  m\_math: org.apache.spark.rdd.RDD[(String, String, Int)] = MapPartitionsRDD[28] at map at  <console>:25  scala> **val sort\_bigdata = m\_bigdata.sortBy(x=>x.\_3,false)**  sort\_bigdata: org.apache.spark.rdd.RDD[(String, String, Int)] = MapPartitionsRDD[33] at sortBy at <console>:25  scala> **val sort\_math=m\_math.sortBy(x=>x.\_3,false)**  sort\_math: org.apache.spark.rdd.RDD[(String, String, Int)] = MapPartitionsRDD[38] at sortBy at <console>:25  scala> **sort\_bigdata.take(5)**  res18: Array[(String, String, Int)] = Array((1003,大数据基础,100), (1007,大数据基础,100), (1004,大数据基础,99), (1002,大数据基础,94), (1006,大数据基础,94))  scala> sort\_math.take(5)  res19: Array[(String, String, Int)] = Array((1003,应用数学,100), (1004,应用数学,100), (1001,应用数学,96), (1002,应用数学,94), (1005,应用数学,94)) |

## 3 任务3

编程查询单科成绩为100的学生编号。要求联合hadoop成绩表和scala成绩表两个表实现联合查询

|  |
| --- |
| scala> **val bigdata\_id =**m\_bigdata**.filter(x=>x.\_3 == 100).map(x=>x.\_1)**  bigdata\_id: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[35] at map at <console>:25  scala> **val math\_id =**m\_math**.filter(x=>x.\_3==100).map(x=>x.\_1)**  math\_id: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[37] at map at <console>:25  scala> **val id = bigdata\_id.union(math\_id).distinct()**  id: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[41] at distinct at <console>:27  scala> id.collect  res7: Array[String] = Array(1003, 1007) |

## 4 任务4

编程计算每个学生的总成绩，并打印出来。

|  |
| --- |
| scala> val all\_score=m\_bigdata union m\_math  all\_score: org.apache.spark.rdd.RDD[(String, String, Int)] = UnionRDD[34] at union at <console>:27  scala> val score=all\_score.map(x=>(x.\_1,x.\_3)).reduceByKey((a,b)=>a+b)  score: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[36] at reduceByKey at <console>:25  scala> score.collect  res13: Array[(String, Int)] = Array((1005,184), (1012,175), (1001,186), (1009,173), (1002,188), (1006,174), (1010,164), (1003,200), (1007,190), (1008,187), (1011,170), (1004,199)) |

## 5 任务5

编程计算学生的平均成绩，并打印出来。

|  |
| --- |
| scala> **val bigdata=sc.textFile("result\_bigdata.txt").map{x=>val line=x.split("\t");(line(0),line(2).toInt)}**  bigdata: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[22] at map at <console>:24  scala> **val math=sc.textFile("result\_math.txt").map{x=>val line=x.split("\t");(line(0),line(2).toInt)}**  math: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[25] at map at <console>:24  scala> val scores=bigdata.union(math).map(x=>(x.\_1,x.\_2))  scores: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[27] at map at <console>:27  scala> val cb\_score=scores.combineByKey(  | count=>(count,1),  | (acc:(Int,Int),count)=>(acc.\_1+count,acc.\_2+1),  | (acc1:(Int,Int),acc2:(Int,Int))=>(acc1.\_1+acc2.\_1,acc1.\_2+acc2.\_2))  cb\_score: org.apache.spark.rdd.RDD[(String, (Int, Int))] = ShuffledRDD[28] at combineByKey at <console>:25  scala> val avg\_score=cb\_score.map(x=>(x.\_1,x.\_2.\_1.toDouble/x.\_2.\_2))  avg\_score: org.apache.spark.rdd.RDD[(String, Double)] = MapPartitionsRDD[29] at map at <console>:25  scala> avg\_score.collect  res8: Array[(String, Double)] = Array((1005,92.0), (1012,87.5), (1001,93.0), (1009,86.5), (1002,94.0), (1006,87.0), (1010,82.0), (1003,100.0), (1007,95.0), (1008,93.5), (1011,85.0), (1004,99.5) |