Spark介绍

2018年2月4日 17:21

Lightning-fast cluster computing

Apache Spark™ is a fast and general engine for large-scale data processing.

Spark Introduce

Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk.

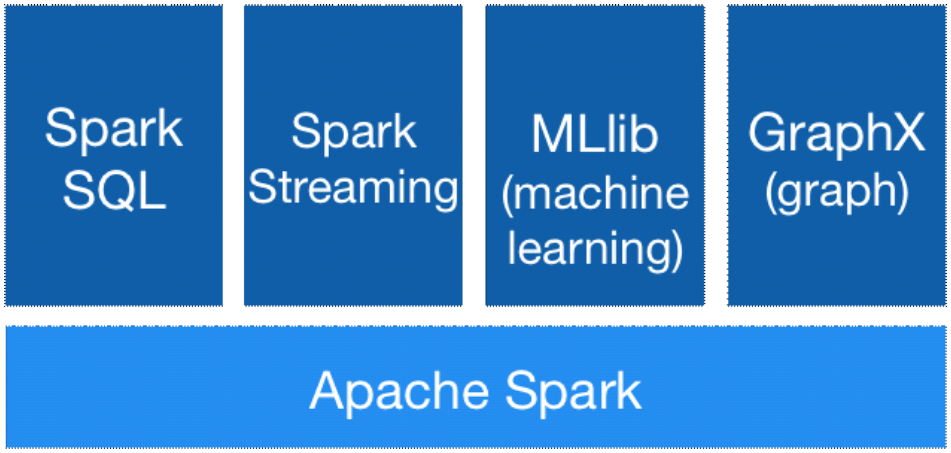
Apache Spark has an advanced DAG execution engine that supports acyclic data flow and in-memory computing.

Write applications quickly in Java, Scala, Python, R.

Spark offers over 80 high-level operators that make it easy to build parallel apps. And you can use it *interactively* from the Scala, Python and R shells.

Combine SQL, streaming, and complex analytics.

Spark powers a stack of libraries including SQL and DataFrames, MLlib for machine learning, GraphX, and Spark Streaming. You can combine these libraries seamlessly in the same application.



Spark是UC Berkeley AMP lab (加州大学伯克利分校的AMP实验室)所开源的，后贡献给Apache。是一种快速、通用、可扩展的大数据分析引擎。它是不断壮大的大数据分析解决方案家族中备受关注的明星成员，为分布式数据集的处理提供了一个有效框架，并以高效的方式处理分布式数据集。**Spark集批处理、实时流处理、交互式查询、机器学习与图计算于一体**，避免了多种运算场景下需要部署不同集群带来的资源浪费。目前，Spark社区也成为大数据领域和Apache软件基金会最活跃的项目之一，其活跃度甚至远超曾经只能望其项背的Hadoop。

Spark的技术背景

无论是工业界还是学术界，都已经广泛使用高级集群编程模型来处理日益增长的数据，如MapReduce和Dryad。这些系统将分布式编程简化为自动提供位置感知性调度、容错以及负载均衡，使得大量用户能够在商用集群上分析超大数据集。

大多数现有的集群计算系统都是基于**非循环的数据流模型**。即从稳定的物理存储（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（Directed Acyclic Graph，有向无环图），然后写回稳定存储。DAG数据流图能够在运行时自动实现任务调度和故障恢复。

尽管非循环数据流是一种很强大的抽象方法，但仍然有些应用无法使用这种方式描述。这类应用包括：
①机器学习和图应用中常用的迭代算法（每一步对数据执行相似的函数）
②交互式数据挖掘工具（用户反复查询一个数据子集）

基于数据流的框架并不明确支持工作集，所以需要将数据输出到磁盘，然后在每次查询时重新加载，这会带来较大的开销。针对上述问题，**Spark实现了一种分布式的内存抽象**，称为**弹性分布式数据集**（Resilient Distributed Dataset，**RDD**）。它支持基于工作集的应用，同时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。

Spark VS MapReduce

2018年2月4日 17:52

MapReduce存在的问题

一个 Hadoop job 通常都是这样的：

1）从 HDFS 读取输入数据；

2）在 Map 阶段使用用户定义的 mapper function, 然后把结果Spill到磁盘；

3）在 Reduce 阶段，从各个处于 Map 阶段的机器中读取 Map 计算的中间结果，使用用户定义的 reduce function, 通常最后把结果写回 HDFS;

Hadoop的问题在于，一个 Hadoop job 会进行多次磁盘读写，比如写入机器本地磁盘，或是写入分布式文件系统中（这个过程包含磁盘的读写以及网络传输）。考虑到磁盘读取比内存读取慢了几个数量级，所以像**Hadoop 这样高度依赖磁盘读写的架构就一定会有性能瓶颈。**

此外，在实际应用中我们通常需要设计复杂算法处理海量数据, 而且不是一个 Hadoop job 可以完成的。比如机器学习领域，需要大量使用迭代的方法训练机器学习模型。而像 Hadoop 的基本模型就只包括了一个 Map 和 一个 Reduce 阶段，想要完成复杂运算就需要切分出无数单独的 Hadoop jobs, 而且每个 Hadoop job 都是磁盘读写大户，这就让 Hadoop 显得力不从心。

随着业界对大数据使用越来越深入，大家都呼唤一个更强大的处理框架，能够真正解决更多复杂的大数据问题。

Spark的优势

2009年，美国加州大学伯克利分校的 AMPLab 设计并开发了名叫 Spark 的大数据处理框架。真如其名，Spark 像燎原之火，迅猛占领大数据处理框架市场。

Spark 没有像 Hadoop 一样使用磁盘读写，而转用性能高得多的内存存储输入数据、处理中间结果、和存储最终结果。在大数据的场景中，很多计算都有循环往复的特点，像 Spark 这样允许在内存中缓存输入输出，上一个 job 的结果马上可以被下一个使用，性能自然要比 Hadoop MapReduce 好得多。

同样重要的是，Spark 提供了更多灵活可用的数据操作，比如 filter, join, 以及各种对 key value pair 的方便操作，甚至提供了一个通用接口，让用户根据需要开发定制的数据操作。

此外，Spark 本身作为平台也开发了 streaming 处理框架 spark streaming, SQL 处理框架 Dataframe, 机器学习库 MLlib, 和图处理库 GraphX. 如此强大，如此开放，基于 Spark 的操作，应有尽有。

Hadoop 的 MapReduce 为什么不使用内存存储？

是历史原因。当初 MapReduce 选择磁盘，除了要保证数据存储安全以外，更重要的是当时企业级数据中心购买大容量内存的成本非常高，选择基于内存的架构并不现实；现在 Spark 真的赶上了好时候，企业可以轻松部署多台大内存机器，内存大到可以装载所有要处理的数据。

Spark单机模式安装

2018年2月4日 17:33

实现步骤：

1）安装和配置好JDK

2）上传和解压Spark安装包

3）进入Spark安装目录下的conf目录

复制conf spark-env.sh.template 文件为 spark-env.sh

在其中修改，增加如下内容：

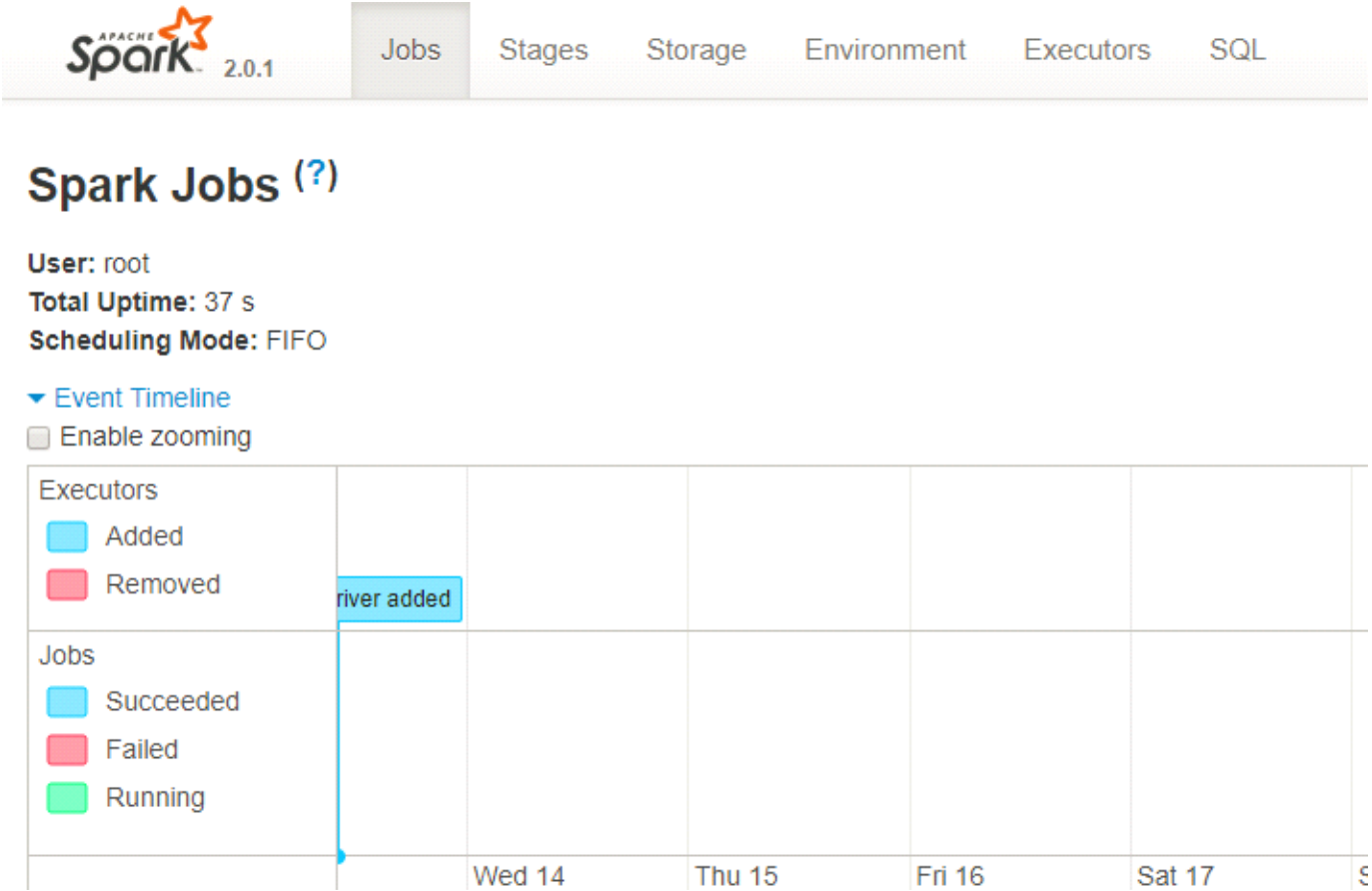
SPARK_LOCAL_IP=服务器IP地址

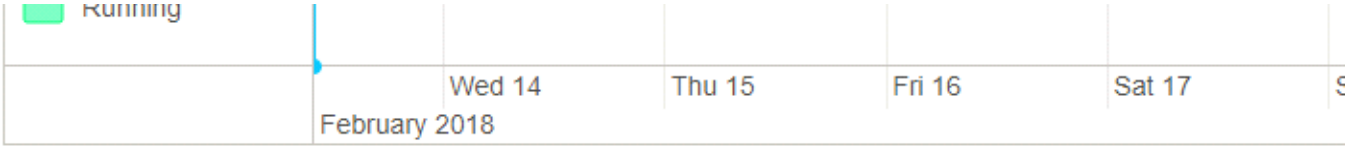
Spark单机模式启动

在bin目录下执行：sh spark-shell --master=local

启动后 发现打印消息

Spark context Web UI available at http://192.168.242.101:4040//Spark的浏览器界面





Spark context available as 'sc' (master = local, app id = local-1490336686508).//Spark提供了环境对象 sc

Spark session available as 'spark'.//Spark提供了会话独享spark

RDD介绍

2018年2月4日 17:34

概述

Resilient Distributed Datasets (RDDs)

Spark revolves around the concept of a *resilient distributed dataset* (RDD), which is a **fault-tolerant collection** of elements that can be operated on **in parallel**. There are two ways to create RDDs: *parallelizing* an existing collection in your driver program, or referencing a dataset in an external storage system, such as a shared filesystem, HDFS, HBase, or any data source offering a Hadoop InputFormat.

RDD就是带有分区的集合类型

弹性分布式数据集（RDD），特点是可以并行操作，并且是容错的。有两种方法可以创建RDD：

1）执行Transform操作（变换操作），

2）读取外部存储系统的数据集，如HDFS，HBase，或任何与Hadoop有关的数据源。

RDD入门示例

案例一：

Parallelized collections are created by calling SparkContext's parallelize method on an existing collection in your driver program (a Scala Seq). The elements of the collection are copied to form a distributed dataset that can be operated on in parallel. For example, here is how to create a parallelized collection holding the numbers 1 to 5:

```
val data = Array(1, 2, 3, 4, 5)
val r1 = sc.parallelize(data)
val r2 = sc.parallelize(data,2)
```

你可以这样理解RDD：它是spark提供的一个特殊集合类。诸如普通的集合类型，如传统的Array：（1,2,3,4,5）是一个整体，但转换成RDD后，我们可以对数据进行Partition（分区）处理，这样做的目的就是为了分布式。

你可以让这个RDD有两个分区，那么有可能是这个形式：RDD(1,2) (3,4)。

这样设计的目的在于：可以进行分布式运算。

注：创建RDD的方式有多种，比如案例一中是基于一个基本的集合类型（Array）转换而来，像parallelize这样的方法还有

很多，之后就会学到。此外，我们也可以在读取数据集时就创建RDD。

案例二：

Spark can create distributed datasets from any storage source supported by Hadoop, including your local file system, HDFS, Cassandra, HBase, Amazon S3, etc. Spark supports text files, SequenceFiles, and any other Hadoop InputFormat.

Text file RDDs can be created using SparkContext’s textFile method. This method takes an URI for the file (either a local path on the machine, or a hdfs://, s3n://, etc URI) and reads it as a collection of lines. Here is an example invocation:

```
val distFile = sc.textFile("data.txt")
```

查看RDD

scala>rdd.collect

收集rdd中的数据组成Array返回，此方法将会把分布式存储的rdd中的数据**集中**到一台机器中组建Array。

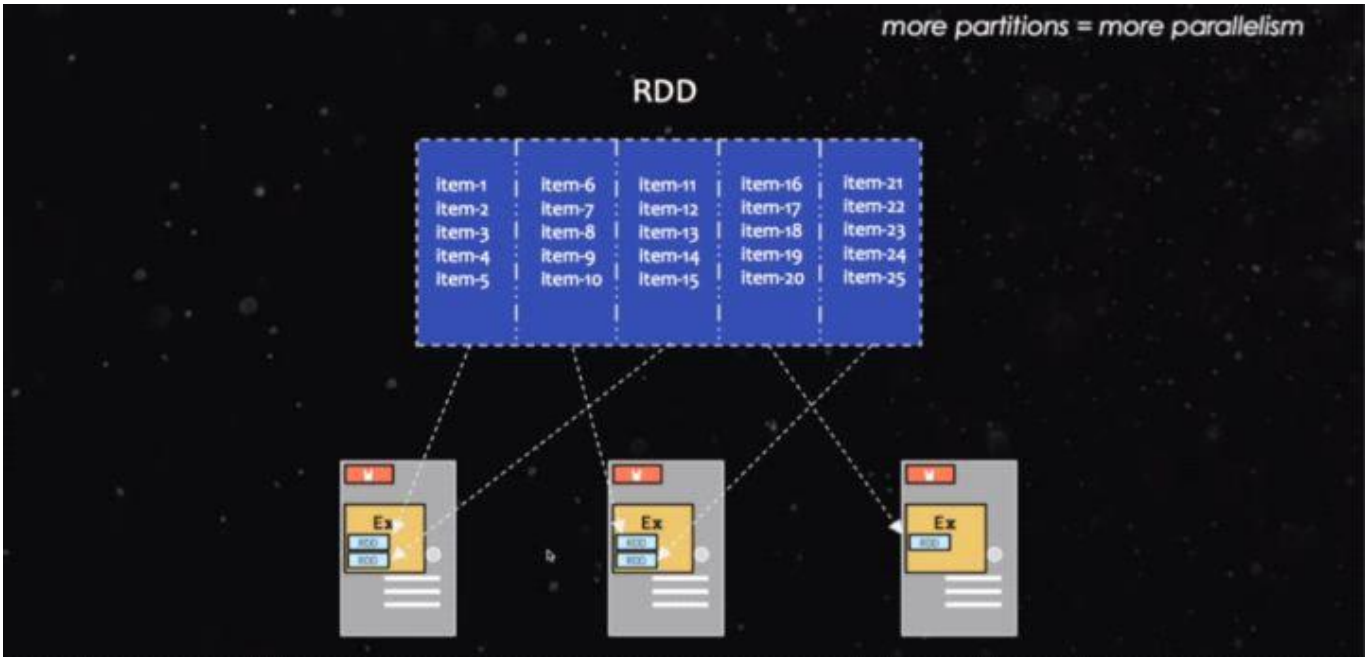
在生产环境下一定要**慎用这个方法**，容易内存溢出。

查看RDD的分区数量：scala>rdd.partitions.size

查看RDD每个分区的元素：scala>rdd.glom.collect

此方法会将每个分区的元素以Array形式返回

分区概念



在上图中， 一个RDD有item1~item25个数据，共5个分区，分别在3台机器上进行处理。

此外，spark并没有原生的提供rdd的分区查看工具 我们可以自己来写一个

示例代码：

```
import org.apache.spark.rdd.RDD

import scala.reflect.ClassTag

object su {

def debug[T: ClassTag](rdd: RDD[T]) = {

rdd.mapPartitionsWithIndex((i: Int, iter: Iterator[T]) => {

val m = scala.collection.mutable.Map[Int, List[T]]()

var list = List[T]()

while (iter.hasNext) {

list = list :+ iter.next

}

m(i) = list

m.iterator

}).collect().foreach((x: Tuple2[Int, List[T]]) => {
```

```
val i = x._1

println(s"partition:[$i]")

x._2.foreach { println }

})

}

}
```

RDD操作

2018年2月4日 20:09

概述

针对RDD的操作，分两种，一种是Transformation（变换），一种是Actions（执行）。

Transformation（变换）操作属于懒操作，不会真正触发RDD的处理计算。

Actions（执行）操作才会真正触发。

Transformations

Transformation	Meaning
map(*func*)	Return a new distributed dataset formed by passing each element of the source through a function *func*. 参数是函数，函数应用于RDD每一个元素，返回值是新的RDD **案例展示：** map 将函数应用到rdd的每个元素中 val rdd = sc.makeRDD(List(1,3,5,7,9)) rdd.map(_*10)

flatMap(*func*)	Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item).
	扁平化map，对RDD每个元素转换，然后再扁平化处理
	案例展示：
	flatMap 扁平map处理
	val rdd = sc.makeRDD(List("hello world","hello count","world spark"),2)
	rdd.map(_.split(" "))//Array(Array(hello, world), Array(hello, count), Array(world, spark))
	rdd.flatMap(_.split(" "))//Array[String] = Array(hello, world, hello, count, world, spark)
	//Array[String] = Array(hello, world, hello, count, world, spark)
	注：map和flatMap有何不同？
	map: 对RDD每个元素转换
	flatMap: 对RDD每个元素转换，然后再扁平化（即去除集合）
	所以，一般我们在读取数据源后，第一步执行的操作是flatMap

filter(*func*)	Return a new dataset formed by selecting those elements of the source on which *func* returns true.参数是函数，函数会过滤掉不符合条件的元素，返回值是新的RDD **案例展示：** filter 用来从rdd中过滤掉不符合条件的数据

	val rdd = sc.makeRDD(List(1,3,5,7,9)); rdd.filter(_<5);

mapPartitions(*func*)	Similar to map, but runs separately on each partition (block) of the RDD, so *func* must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.
	该函数和map函数类似，只不过映射函数的参数由RDD中的每一个元素变成了RDD中每一个分区的迭代器。
	案例展示：：
	val rdd3 = rdd1.mapPartitions{ x => {
	val result = List[Int]()
	var i = 0
	while(x.hasNext){
	i += x.next()
	}
	result.::(i).iterator
	}}
	
	scala>rdd3.collect
	scala> rdd3.collect
	res10: Array[Int] = Array(3, 12)

mapPartitionsWithIndex(*func*)	Similar to mapPartitions, but also provides *func* with an integer value representing the index of the partition, so *func* must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T.
	
	函数作用同mapPartitions，不过提供了两个参数，第一个参数为分区的索引。
	案例展示：
	var rdd1 = sc.makeRDD(1 to 5,2)
	
	var rdd2 = rdd1.mapPartitionsWithIndex{
	(index,iter) => {
	var result = List[String]()
	var i = 0
	while(iter.hasNext){
	i += iter.next()
	}
	result.::(index + "|" + i).iterator
	}
	}
	scala> rdd2.collect
	res11: Array[String] = Array(0|3, 1|12)
	
	案例展示：
	val rdd = sc.makeRDD(List(1,2,3,4,5),2);
	rdd.mapPartitionsWithIndex((index,iter)=>{
	var list = List[String]()
	while(iter.hasNext){
	if(index==0)
	list = list :+ (iter.next + "a")
	else {
	list = list :+ (iter.next + "b")
	}
	}

	list.iterator }); scala> res12.collect res13: Array[String] = Array(1a, 2a, 3b, 4b, 5b)

union(*otherDataset*)	Return a new dataset that contains the union of the elements in the source dataset and the argument.
	案例展示：
	union 并集 -- 也可以用++实现
	val rdd1 = sc.makeRDD(List(1,3,5));
	val rdd2 = sc.makeRDD(List(2,4,6,8));
	val rdd = rdd1.union(rdd2);
	val rdd = rdd1 ++ rdd2;
	scala> rdd.collect
	res17: Array[Int] = Array(1, 3, 5, 2, 4, 6, 8)

intersection(*otherDataset*)	Return a new RDD that contains the intersection of elements in the source dataset and the argument.
	案例展示：
	intersection 交集
	val rdd1 = sc.makeRDD(List(1,3,5,7));
	val rdd2 = sc.makeRDD(List(5,7,9,11));
	val rdd = rdd1.intersection(rdd2);
	res18: Array[Int] = Array(7, 5)

subtract	**案例展示：**
	subtract 差集
	val rdd1 = sc.makeRDD(List(1,3,5,7,9));
	val rdd2 = sc.makeRDD(List(5,7,9,11,13));
	val rdd = rdd1.subtract(rdd2);

distinct([*numTasks*]))	Return a new dataset that contains the distinct elements of the source dataset. 没有参数，将RDD里的元素进行去重操作 **案例展示：** val rdd = sc.makeRDD(List(1,3,5,7,9,3,7,10,23,7)); rdd.distinct
groupByKey([*numTasks*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. **Note:** If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using reduceByKey or aggregateByKey will yield much better performance. **Note:** By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional numTasks argument to set a different number of tasks.

	案例展示：
	scala>val rdd = sc.parallelize(List(("cat",2), ("dog",5),("cat",4),("dog",3),("cat",6),("dog",3),("cat",9),("dog",1)),2);
	scala>rdd.groupByKey()
	
	注：groupByKey对于数据格式是有要求的，即操作的元素**必须是一个二元tuple**，
	tuple._1 是key， tuple._2是value
	比如下面这种数据格式：
	sc.parallelize(List("dog", "tiger", "lion", "cat", "spider", "eagle"), 2)就不符合要求
	以及这种：
	sc.parallelize(List(("cat",2,1), ("dog",5,1),("cat",4,1),("dog",3,2),("cat",6,2),("dog",3,4),("cat",9,4),("dog",1,4)),2);

reduceByKey(*func*, [*numTasks*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*, **which must be of type (V,V)** => V. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument. **案例展示：** scala>var rdd = sc.makeRDD(List(("hello",1),("spark",1),("hello",1),("world",1))) rdd.reduceByKey(_+_); **注：**reduceByKey操作的数据格式必须是一个二元tuple
aggregateByKey(*zeroValue*) (*seqOp*, *combOp*, [*numTasks*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument. **使用方法及案例展示：** aggregateByKey(zeroValue)(func1,func2) • zeroValue表示初始值，初始值会参与func1的计算 • 在分区内，按key分组，把每组的值进行fun1的计算 • 再将每个分区每组的计算结果按fun2进行计算 **scala>** val rdd = sc.parallelize(List(("cat",2),("dog",5),("cat",4),("dog",3),("cat",6),("dog",3),("cat",9),("dog",1)),2); **查看分区结果：** partition:[0] (cat,2) (dog,5) (cat,4) (dog,3) partition:[1] (cat,6) (dog,3) (cat,9)

	(dog, 1)
	
	scala> rdd.aggregateByKey(0)(_+_,_+_);
	(dog,12), (cat,21)
	scala> rdd.aggregateByKey(0)(_+_,_*_);
	(dog,32), (cat,90)

sortByKey([*ascending*], [*numTasks*])	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean `ascending` argument. **案例展示：** val d2 = sc.parallelize(Array(("cc",32),("bb",32),("cc",22),("aa",18),("bb",6),("dd",16),("ee",104),("cc",1),("ff",13),("gg",68),("bb",44))) d2.sortByKey(true).collect res31: Array[(String, Int)] = Array((aa,18), (bb,6), (bb,44), (bb,32), (cc,32), (cc,22), (cc,1), (dd,16), (ee,104), (ff,13), (gg,68))
join(*otherDataset*, [*numTasks*])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through `leftOuterJoin`, `rightOuterJoin`, and `fullOuterJoin`. **案例展示：** val rdd1 = sc.makeRDD(List(("cat",1),("dog",2))) val rdd2 = sc.makeRDD(List(("cat",3),("dog",4),("tiger",9))) rdd1.join(rdd2);
cartesian(*otherDataset*)	When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements). 参数是RDD，求两个RDD的笛卡儿积 **案例展示：** cartesian 笛卡尔积 val rdd1 = sc.makeRDD(List(1,2,3)) val rdd2 = sc.makeRDD(List("a","b")) rdd1.cartesian(rdd2);
coalesce(*numPartitions*)	Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset. coalesce(n,true/false)扩大或缩小分区 **案例展示：** val rdd = sc.makeRDD(List(1,2,3,4,5),2)

	rdd9.coalesce(3,true);//如果是扩大分区 需要传入一个true 表示要重新shuffle
	rdd9.coalesce(2);//如果是缩小分区 默认就是false 不需要明确的传入

repartition(*numPartitions*)	Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network. repartition(n) 等价于上面的coalesce

partitionBy	通常我们在创建RDD时指定分区规则 将会导致 数据自动分区
	我们也可以通过partitionBy方法人为指定分区方式来进行分区
	常见的分区器有：
	• HashPartitioner
	• RangePartitioner
	案例展示：
	import org.apache.spark._
	val r1 = sc.makeRDD(List((2,"aaa"),(9,"bbb"),(7,"ccc"),(9,"ddd"),(3,"eee"),(2,"fff")),2);
	val r2=r1.partitionBy(new HashPartitioner(2))//按照键的 hash%分区数 得到的编号去往指定的分区 这种方式可以实现将相同键的数据 分发给同一个分区的效果
	val r3=r1.partitionBy(new RangePartitioner(2,r1))//将数据按照值的字典顺序进行排序 再分区

Actions

Action	Meaning
reduce(*func*)	Aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel. 并行整合所有RDD数据，例如求和操作
collect()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data. 返回RDD所有元素，将rdd分布式存储在集群中不同分区的数据 获取到一起组成一个数组返回 要注意 这个方法将会把所有数据收集到一个机器内，容易造成内存的溢出 在生产环境下千万慎用
count()	Return the number of elements in the dataset. 统计RDD里元素个数 **案例展示：** val rdd = sc.makeRDD(List(1,2,3,4,5),2) rdd.count
first()	Return the first element of the dataset (similar to take(1)).
take(*n*)	Return an array with the first *n* elements of the dataset. **案例展示：** take 获取前几个数据 val rdd = sc.makeRDD(List(52,31,22,43,14,35)) rdd.take(2)

takeOrdered(*n*, *[ordering]*)	Return the first *n* elements of the RDD using either their natural order or a custom comparator. **案例展示：** takeOrdered(n) 先将rdd中的数据进行升序排序 然后取前n个 val rdd = sc.makeRDD(List(52,31,22,43,14,35)) rdd.takeOrdered(3)
top(n)	top(n) 先将rdd中的数据进行降序排序 然后取前n个 val rdd = sc.makeRDD(List(52,31,22,43,14,35)) rdd.top(3)
saveAsTextFile(*path*)	Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file. **案例示例：** saveAsTextFile 按照文本方式保存分区数据 val rdd = sc.makeRDD(List(1,2,3,4,5),2); rdd.saveAsTextFile("/root/work/aaa")
countByKey()	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
foreach(*func*)	Run a function *func* on each element of the dataset. This is usually done for side effects such as updating an Accumulator or interacting with external storage systems. Note: modifying variables other than Accumulators outside of the foreach() may result in undefined behavior. See Understanding closures for more details.

案例：通过rdd实现统计文件中的单词数量

sc.textFile("/root/work/words.txt").flatMap(_.split(" ")).map((_,1)).reduceByKey(_+_).saveAsTextFile("/root/work/wcresult")

DAG概念

2018年2月22日 13:56

概述

Spark会根据用户提交的计算逻辑中的**RDD的转换和动作来生成RDD之间的依赖关系**，同时这个计算链也就生成了逻辑上的DAG。接下来以“Word Count”为例，详细描述这个DAG生成的实现过程。

Spark Scala版本的Word Count程序如下：

```
1： val file=sc.textFile("hdfs://hadoop01:9000/hello1.txt")
2： val counts = file.flatMap(line => line.split(" "))
3：             .map(word => (word, 1))
4：             .reduceByKey(_ + _)
5： counts.saveAsTextFile("hdfs://...")
```

file和counts都是RDD，其中file是从HDFS上读取文件并创建了RDD，而counts是在file的基础上通过flatMap、map和reduceByKey这三个RDD转换生成的。最后，counts调用了动作saveAsTextFile，用户的计算逻辑就从这里开始提交的集群进行计算。那么上面这5行代码的具体实现是什么呢？

1）行1：sc是org.apache.spark.SparkContext的实例，它是用户程序和Spark的交互接口，会负责连接到集群管理者，并根据用户设置或者系统默认设置来申请计算资源，完成RDD的创建等。

sc.textFile（"hdfs：//..."）就完成了一个org.apache.spark.rdd.HadoopRDD的创建，并且完成了一次RDD的转换：通过map转换到一个org.apache.spark.rdd.MapPartitions-RDD。也就是说，file实际上是一个MapPartitionsRDD，它保存了文件的所有行的数据内容。

2）行2：将file中的所有行的内容，以空格分隔为单词的列表，然后将这个按照行构成的单词列表合并为一个列表。最后，以每个单词为元素的列表被保存到MapPartitionsRDD。

3）行3：将第2步生成的MapPartittionsRDD再次经过map将每个单词word转为（word，1）的元组。这些元组最终被放到一个MapPartitionsRDD中。

4）行4：首先会生成一个MapPartitionsRDD，起到map端combiner的作用；然后会生成一个ShuffledRDD，它从上一个RDD的输出读取数据，作为reducer的开始；最后，还会生成一个MapPartitionsRDD，起到reducer端reduce的作用。

5）行5：向HDFS输出RDD的数据内容。最后，调用org.apache.spark.SparkContext#runJob向集群提交这个计算任务。

RDD之间的关系可以从两个维度来理解：一个是RDD是从哪些RDD转换而来，也就是RDD的parent RDD（s）是什么；还有就是依赖于parent RDD（s）的哪些Partition（s）。这个关系，就是RDD之间的依赖，org.apache.spark.Dependency。根据依赖于parent RDD（s）的Partitions的不同情况，Spark将这种依赖分为两种，一种是**宽依赖**，一种是**窄依赖**。

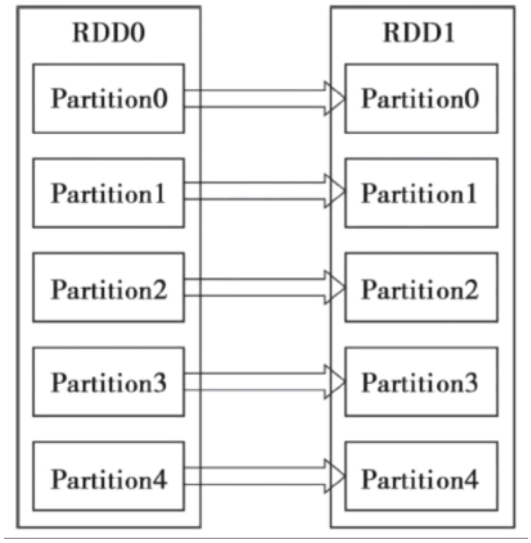
RDD的依赖关系

2018年2月22日 14:20

RDD的依赖关系

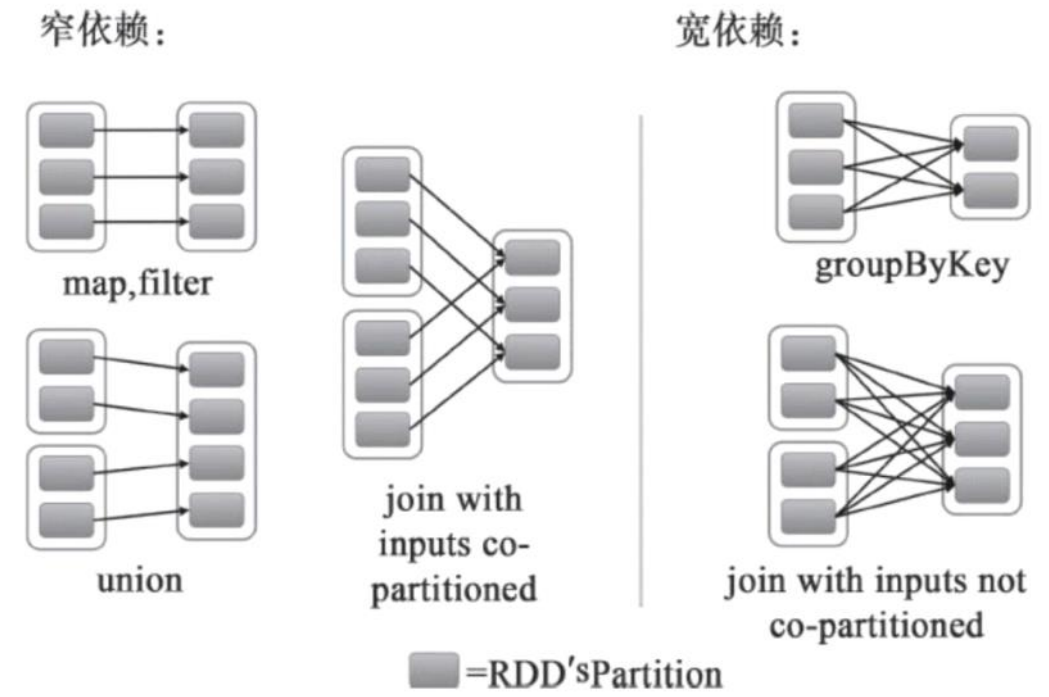
RDD和它依赖的parent RDD（s）的关系有两种不同的类型，即**窄依赖**（narrow dependency）和**宽依赖**（wide dependency）。

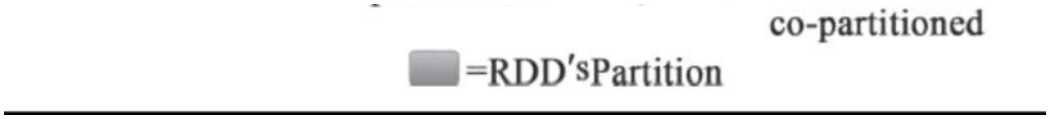
1）窄依赖指的是每一个parent RDD的Partition最多被子RDD的一个Partition使用，如下图所示。



2）宽依赖指的是多个子RDD的Partition会依赖同一个parent RDD的Partition。

我们可以从不同类型的转换来进一步理解RDD的窄依赖和宽依赖的区别，如下图所示。





窄依赖

对于窄依赖操作，它们只是将Partition的数据根据转换的规则进行转化，并不涉及其他的处理，可以简单地认为只是将数据从一个形式转换到另一个形式。

窄依赖底层的源码：

```
abstract class NarrowDependency[T](_rdd: RDD[T]) extends Dependency[T] {
  //返回子RDD的partitionId依赖的所有的parent RDD的Partition(s)
  def getParents(partitionId: Int): Seq[Int]
  override def rdd: RDD[T] = _rdd
}

class OneToOneDependency[T](rdd: RDD[T]) extends NarrowDependency[T](rdd) {
  override def getParents(partitionId: Int) = List(partitionId)
}
```

所以对于窄依赖，并不会引入昂贵的Shuffle。所以执行效率非常高。**如果整个DAG中存在多个连续的窄依赖**，则可以将这些连续的窄依赖整合到一起连续执行，中间不执行shuffle 从而提高效率，**这样的优化方式称之为流水线优化。**

此外，针对窄依赖，如果子RDD某个分区数据丢失，只需要找到父RDD对应依赖的分区，恢复即可。但如果是宽依赖，当分区丢失时，最糟糕的情况是要重算所有父RDD的所有分区。

宽依赖

对于groupByKey这样的操作，子RDD的所有Partition（s）会依赖于parent RDD的所有Partition（s），**子RDD的Partition是parent RDD的所有Partition Shuffle的结果。**

宽依赖的源码：

```
class ShuffleDependency[K, V, C](
  @transient _rdd: RDD[_ <: Product2[K, V]],
  val partitioner: Partitioner,
  val serializer: Option[Serializer] = None,
  val keyOrdering: Option[Ordering[K]] = None,
  val aggregator: Option[Aggregator[K, V, C]] = None,
  val mapSideCombine: Boolean = false)
extends Dependency[Product2[K, V]] {
```

```
override def rdd = _rdd.asInstanceOf[RDD[Product2[K, V]]]
//获取新的shuffleId
val shuffleId: Int = _rdd.context.newShuffleId()
//向ShuffleManager注册Shuffle的信息
val shuffleHandle: ShuffleHandle =
_rdd.context.env.shuffleManager.registerShuffle(
    shuffleId, _rdd.partitions.size, this)

    _rdd.sparkContext.cleaner.foreach(_.registerShuffleForCleanup(this))
}
```

Shuffle概述

spark中一旦遇到宽依赖就需要进行shuffle的操作，所谓的shuffle的操作的本质就是将数据汇总后重新分发的过程。

这个过程数据要汇总到一起，数据量可能很大所以不可避免的需要**进行数据落磁盘的操作**，会降低程序的性能，所以spark并不是完全内存不读写磁盘，只能说它尽力避免这样的过程来提高效率 。

spark中的shuffle，在早期的版本中，会产生多个临时文件，但是这种多临时文件的策略造成大量文件的同时的读写，磁盘的性能被分摊给多个文件，每个文件读写效率都不高，影响spark的执行效率。所以在后续的spark中(1.2.0之后的版本)的shuffle中，只会产生一个文件，并且数据会经过排序再附加索引信息，减少了文件的数量并通过排序索引的方式提升了性能。

DAG的生成与Stage的划分

2018年2月22日 14:37

DAG的生成

原始的RDD（s）通过一系列转换就形成了DAG。RDD之间的依赖关系，包含了RDD由哪些Parent RDD（s）转换而来和它依赖parent RDD（s）的哪些Partitions，是DAG的重要属性。

借助这些依赖关系，DAG可以认为这些RDD之间形成了Lineage（血统，血缘关系）。借助Lineage，能保证一个RDD被计算前，它所依赖的parent RDD都已经完成了计算；**同时也实现了RDD的容错性**，即如果一个RDD的部分或者全部的计算结果丢失了，那么就需要重新计算这部分丢失的数据。

Spark的Stage

Spark在执行任务（job）时，首先会根据依赖关系，将DAG划分为不同的阶段（Stage）。

处理流程是：

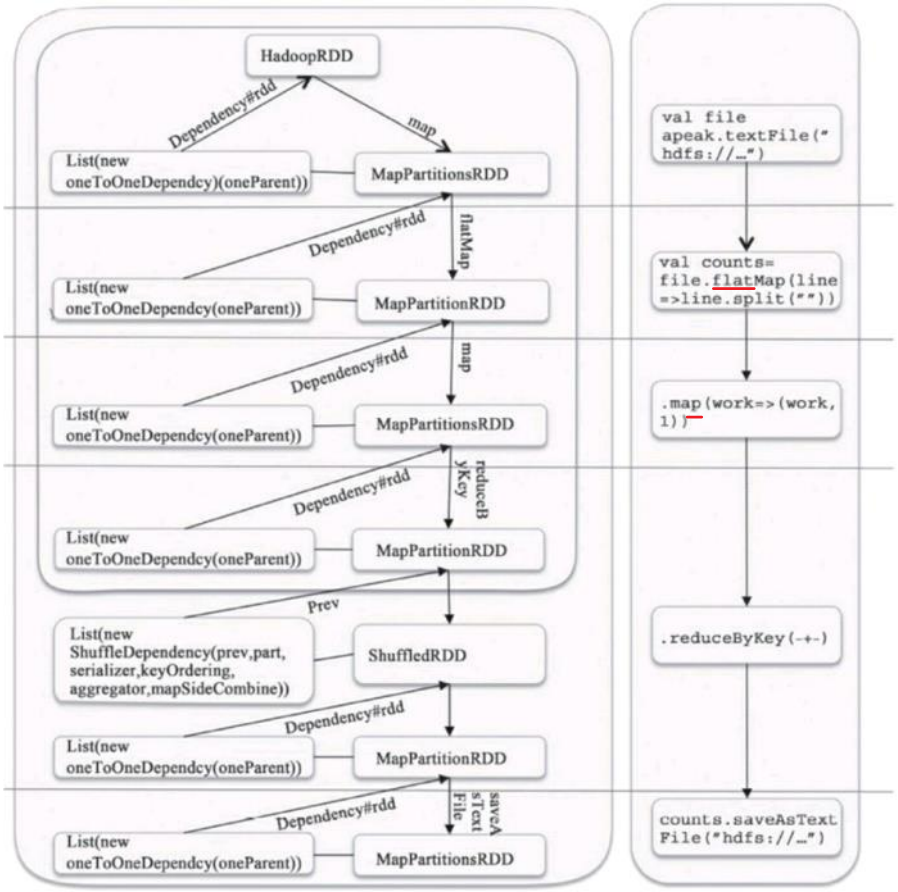
1）Spark在执行Transformation类型操作时都不会立即执行，而是懒执行（计算）

2）执行若干步的Transformation类型的操作后，一旦遇到Action类型操作时，才会真正触发执行（计算）

3）执行时，从当前Action方法向前回溯，如果遇到的是窄依赖则应用流水线优化，继续向前找，直到碰到某一个宽依赖

4）因为宽依赖必须要进行shuffle，无法实现优化，所以将这一次段执行过程组装为一个stage

5）再从当前宽依赖开始继续向前找。重复刚才的步骤，从而将这个DAG还分为若干的stage



在stage内部可以执行流水线优化，而在stage之间没办法执行流水线优化，因为有shuffle。但是这种机制已经尽力的去避免了shuffle。

Spark的Job和Task

原始的RDD经过一系列转换后（一个DAG），会在最后一个RDD上触发一个动作，这个动作会生成一个Job。

所以可以这样理解：一个DAG对应一个Spark的Job。

在Job被划分为一批计算任务（Task）后，这批Task会被提交到集群上的计算节点去计算

Spark的Task分为两种：

1）org.apache.spark.scheduler.ShuffleMapTask

2）org.apache.spark.scheduler.ResultTask

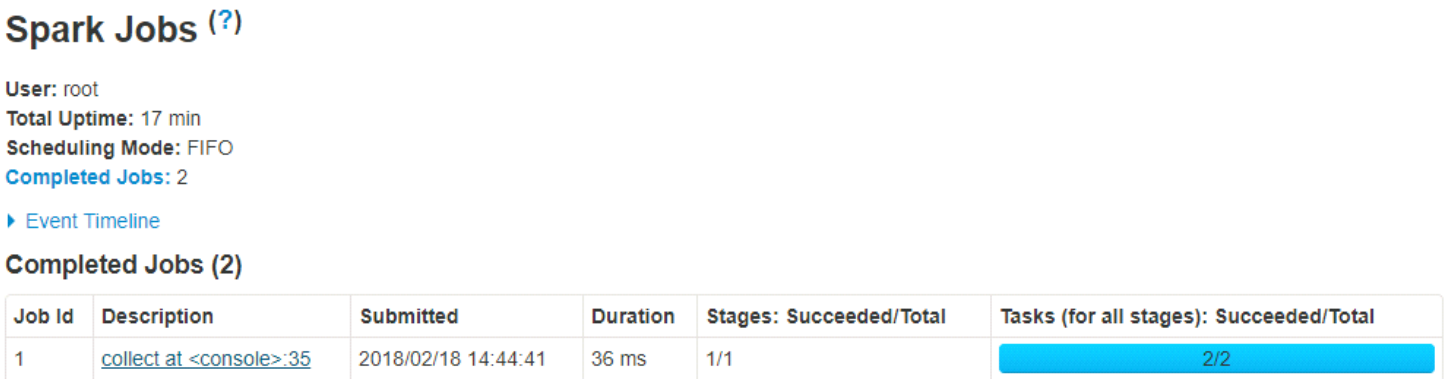
简单来说，DAG的最后一个阶段会为每个结果的Partition生成一个ResultTask，其余所有的阶段都会生成ShuffleMapTask。

可视化理解窄依赖和宽依赖

案例 单词统计

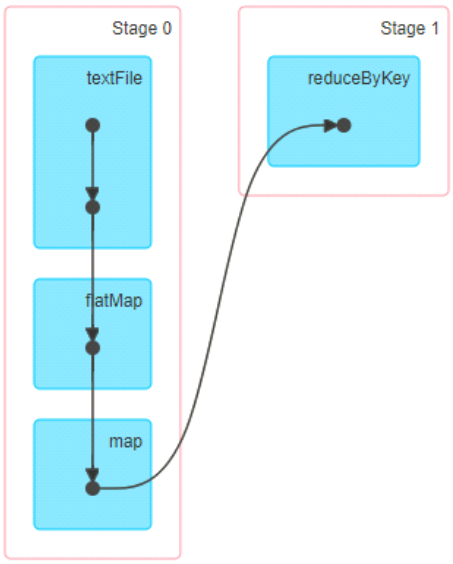
scala>val data=sc.textFile("/home/software/hello.txt",2)
scala> data.flatMap(_.split(" ")).map((_,1)).reduceByKey(_+_).collect

1）打开web页面控制台（ip:4040端口地址），刷新，会发现刚才的操作会出现在页面上



2）点击 Description下的 collect at..... 进入job的详细页面

3）点击 DAG Visualization 会出现如下图形



Spark框架核心概念

2018年4月2日 16:11

1.RDD。弹性分布式数据集，是Spark最核心的数据结构。有分区机制，所以可以分布式进行处理。有容错机制，通过RDD之间的依赖关系来恢复数据。
2.依赖关系。RDD的依赖关系是通过各种**Transformation（变换）**来得到的。父RDD和子RDD之间的依赖关系分两种：①窄依赖 ②宽依赖
①针对窄依赖：父RDD的分区和子RDD的分区关系是：一对一
窄依赖不会发生Shuffle，执行效率高，spark框架底层会针对多个连续的窄依赖执行流水线优化，从而提高性能。例如 map flatMap等方法都是窄依赖方法

②针对宽依赖：父RDD的分区和子RDD的分区关系是：一对多
宽依赖会产生shuffle，会产生磁盘读写，无法优化。

3.DAG。有向无环图，当一个RDD的依赖关系形成之后，就形成了一个DAG。一般来说，一个DAG，最后都至少会触发一个Action操作，触发执行。一个Action对应一个Job任务。

4.Stage。一个DAG会根据RDD之间的依赖关系进行Stage划分，流程是：以Action为基准，向前回溯，遇到宽依赖，就形成一个Stage。遇到窄依赖，则执行流水线优化（将多个连续的窄依赖放到一起执行）

5.task。任务。一个分区对应一个task。可以这样理解：一个Stage是一组Task的集合

6.RDD的Transformation（变换）操作：懒执行，并不会立即执行

7.RDD的Action(执行）操作：触发真正的执行

Spark集群模式安装

2018年2月4日 22:10

实现步骤：

1）上传解压spark安装包

2）进入spark安装目录的conf目录

3）配置spark-env.sh文件

配置示例：

#本机ip地址

SPARK_LOCAL_IP=hadoop01

#spark的shuffle中间过程会产生一些临时文件，此项指定的是其存放目录，不配置默认是在 /tmp目录下

SPARK_LOCAL_DIRS=/home/software/spark/tmp

export JAVA_HOME=/home/software/jdk1.8

4）在conf目录下，编辑slaves文件

配置示例：

hadoop01

hadoop02

hadoop03

5）配置完后，将spark目录发送至其他节点，并更改对应的 **SPARK_LOCAL_IP** 配置

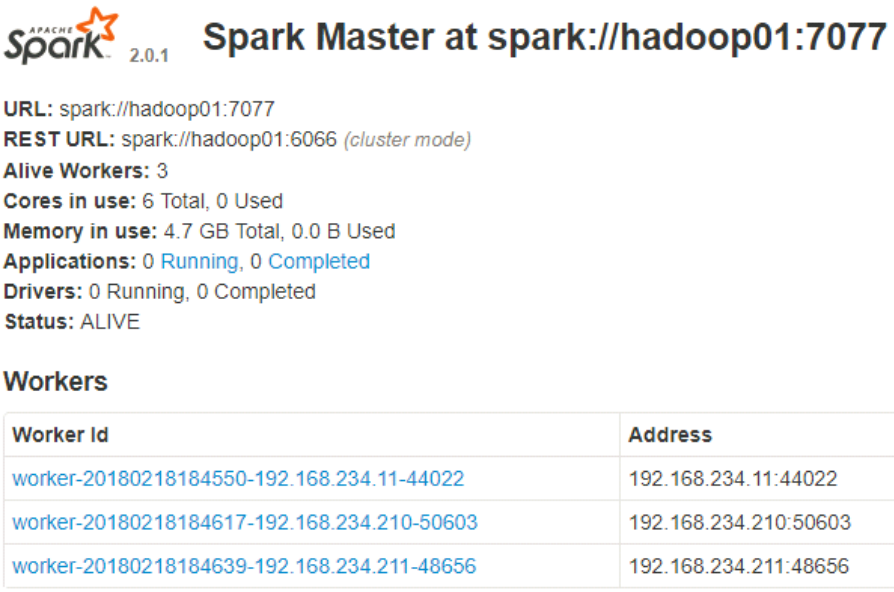
启动集群

1）如果你想让 01 虚拟机变为master节点，则进入01 的spark安装目录的sbin目录
执行： sh start-all.sh

2）通过jps查看各机器进程，
01：Master +Worker
02：Worker
03：Worker

3）通过浏览器访问管理界面

http://192.168.234.11:8080



4）通过spark shell 连接spark集群

进入spark的bin目录

执行：sh spark-shell.sh --master spark://192.168.234.11:7077

6）在集群中读取文件：

sc.textFile("/root/work/words.txt")

默认读取本机数据 这种方式需要在集群的每台机器上的对应位置上都一份该文件 浪费磁盘

7）所以应该通过hdfs存储数据

sc.textFile("hdfs://hadoop01:9000/mydata/words.txt");

注：可以在spark-env.sh 中配置选项 HADOOP_CONF_DIR 配置为hadoop的etc/hadoop的地址 使默认访问的是hdfs的路径

注：如果修改默认地址是hdfs地址 则如果想要访问文件系统中的文件 需要指明协议为file 例如 sc.text("file:///xxx/xx")