1.弹性分布式数据集 RDD

1.1. RDD 概述

RDD 论文，中文版 ： http://spark.apachecn.org/paper/zh/spark-rdd.html

1.1.1. 产生背景

为了解决开发人员能在大规模的集群中以一种容错的方式进行内存计算，提出了 RDD 的概念，而当前的很多框架对迭代式算法场景与交互性数据挖掘场景的处理性能非常差，这个是 RDDs 的提出的动机。

1.1.2. 什么是 RDD

RDD 是 Spark 的计算模型。RDD（Resilient Distributed Dataset）叫做**弹性的分布式数据集合**，是 Spark 中最基本的数据抽象，它代表一个不可变、只读的，被分区的数据集。操作 RDD 就像操作本地集合一样，有很多的方法可以调用，使用方便，而无需关心底层的调度细节。

1.2. 创建 RDD

1）集合并行化创建（通过 scala 集合创建） scala 中的本地集合—> spark RDD

```
val arr = Array(1,2,3,4,5)
val rdd = sc.parallelize(arr)
```

val rdd =sc.**makeRDD**(arr)

```
scala> val arr = Array(1,3,5,6,7)
arr: Array[Int] = Array(1, 3, 5, 6, 7)

scala> val rdd = sc.parallelize(arr)
rdd: org.apache.spark.rdd.RDD[Int] = Par
```

通过集合并行化方式创建 RDD，适用于本地测试，做实验

2）外部文件系统，比如 HDFS 等

val rdd2 = sc.textFile("hdfs://hdp-01:9000/words.txt")

// 读取本地文件

val rdd2 = sc.textFile(“file:///root/words.txt”)

3）从父 RDD 转换成新的子 RDD

调用 Transformation 类的方法，生成新的 RDD

```
scala> val rdd = sc.parallelize(arr)
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[3]

scala> val rdd2 = rdd.map(_ * 100)
rdd2: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[4] at
```

spark 上的所有的方法，有一个专有的名词，叫做算子。

1.3. RDD 的分区

说对 rdd 进行操作，实际上是操作的 rdd 中的每一个分区，分区的数量决定了并行的数量。

使用 rdd.partitions.size 查看分区数量。

```
scala> val rdd = sc.parallelize(arr)
rdd: org.apache.spark.rdd.RDD[Int] =

scala> rdd.partitions.size
res1: Int = 6
```

如果从外部创建 RDD，比如从 hdfs 中读取数据， 正常情况下，分区的数量是和我们读取的文件的 block 块数量是一致的，但是如果只有一个 block 块，那么分区数量是 2。 也就是说最低的分区数量是 2。

如果是集合并行化创建得到的 rdd，分区的数量，默认的和最大可用的 cores 数量相等。

（--total-executor-cores > 可用的 cores？可用的 cores:--total-executor-cores）

通过集合并行化创建的 rdd 是可以任意修改分区的数量的。

1.4. RDD 编程 API

1.4.1. RDD 算子

算子是 RDD 中定义的方法，分为转换(transformantion)和动作(action)。Tranformation 算子并不会触发 Spark 提交作业，直至 Action 算子才提交任务执行，这是一个延迟计算的设计技巧，可以避免内存过快被中间计算占满，从而提高内存的利用率。

RDD 拥有的操作比 MR 丰富的多，不仅仅包括 Map、Reduce 操作，还包括 filter、sort、join、save、count 等操作，并且中间结果不需要保存，所以 Spark 比 MR 更容易方便完成更复杂的任务。

RDD 支持两种类型的操作：

转换（Transformation） 现有的 RDD 通过转换生成一个新的 RDD。lazy 模式，延迟执行。

转换函数包括：map，filter，flatMap，groupByKey，reduceByKey，aggregateByKey，union,join, coalesce 等等。

动作（Action） 在 RDD 上运行计算，并返回结果给驱动程序(Driver)或写入文件系统。

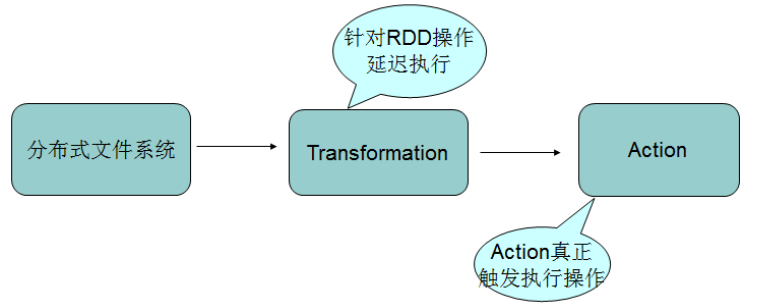
动作操作包括：reduce，collect，count，first，take，countByKey 以及 foreach 等等。

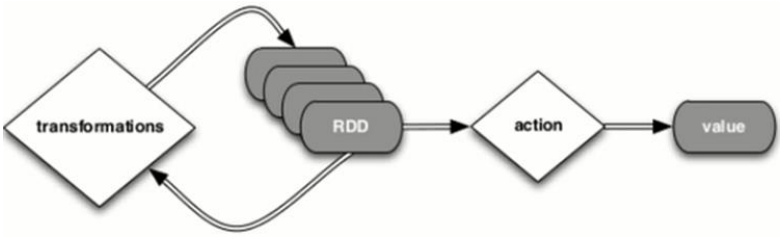
collect 该方法把数据收集到 driver 端 Array 数组类型

所有的 transformation 只有遇到 action 才能被执行。

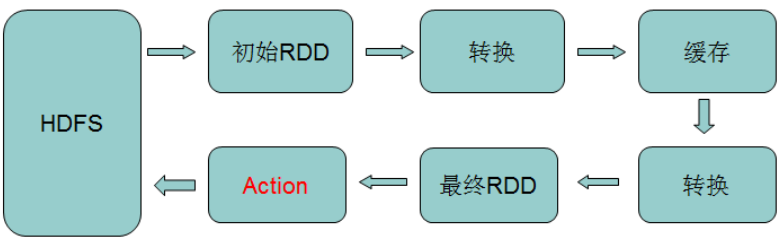
当触发执行 action 之后，数据类型不再是 rdd 了，数据就会存储到指定文件系统中，或者直接打印结果或者收集起来。

RDD 操作流程示意：





RDD 的转换与操作



wordcount 示例，查看 lazy 特性。

只有在执行 action 时，才会真正开始运算，并得到结果或存入文件中。

1.4.2. Transformation

RDD 中的所有转换都是延迟加载的，也就是说，它们并不会直接计算结果。相反的，它们只是记住这些应用到基础数据集（例如一个文件）上的转换动作。只有当发生一个要求返回结果给 Driver 的动作时，这些转换才会真正运行。这种设计让 Spark 更加有效率地运行。

对 RDD 中的元素执行的操作，实际上就是对 RDD 中的每一个分区的数据进行操作，不需要关注数据在哪个分区中。

常用的 Transformation：

转换	含义
map(func)	返回一个新的 RDD，该 RDD 由每一个输入元素经过 func 函数转换后组成
filter(func)	返回一个新的 RDD，该 RDD 由经过 func 函数计算后返回值为 true 的输入元素组成
flatMap(func)	先 map，再 flatten 压平
union(otherDataset)	对源 RDD 和参数 RDD 求并集后返回一个新的 RDD
intersection(otherDataset)	对源 RDD 和参数 RDD 求交集后返回一个新的 RDD
subtract(otherDataset)	求差集后返回新的 RDD，出现在源 rdd 中，不在 otherrdd 中
distinct([numTasks]))	对源 RDD 进行去重后返回一个新的 RDD
mapPartitions(func)	类似于 map，但独立地在 RDD 的每一个分片上运行，因此在类型为 T 的 RDD 上运行时，func 的函数类型必须是 Iterator[T] => Iterator[U]
mapPartitionsWithIndex(func)	类似于 mapPartitions，但 func 带有一个整数参数表示分片的索引值，因此在类型为 T 的 RDD 上运行时，func 的函数类型必须是 (Int, Interator[T]) => Iterator[U]
sortBy(func,[ascending], [numTasks])	与 sortByKey 类似，但是更灵活
sortByKey([ascending], [numTasks])	在一个(K,V)的 RDD 上调用，K 必须实现 Ordered 接口，返回一个按照 key 进行排序的(K,V)的 RDD
join(otherDataset, [numTasks])	在类型为(K,V)和(K,W)的 RDD 上调用，返回一个相同 key 对应的所有元素对在一起的(K,(V,W))的 RDD

cogroup(otherDataset, [numTasks])	在类型为(K,V)和(K,W)的 RDD 上调用，返回一个(K,(Iterable<V>,Iterable<W>))类型的 RDD
cartesian(otherDataset)	笛卡尔积
mapValues(func)	在一个(K,V)的 RDD 上调用
groupBy (func, [numTasks])	根据自定义条件进行分组
groupByKey([numTasks])	在一个(K,V)的 RDD 上调用，返回一个(K, Iterator[V])的 RDD
reduceByKey(func, [numTasks])	在一个(K,V)的 RDD 上调用，返回一个(K,V)的 RDD，使用指定的 reduce 函数，将相同 key 的值聚合到一起，与 groupByKey 类似，reduce 任务的个数可以通过第二个可选的参数来设置
aggregateByKey(zeroValue)(seqOp, combOp, [numTasks])	针对分区内部使用 seqOp 方法，针对最后的结果使用 combOp 方法。
coalesce(numPartitions**)**	用于对 RDD 进行重新分区，第一个参数是分区的数量，第二个参数是是否进行 shuffle，可不传，默认不 shuffle
repartition(numPartitions)	用于对 RDD 进行重新分区，相当于 shuffle 版的 calesce

groupBy 的返回值类型：

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

T: 元素的类型　K：指定的 key

groupByKey

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

reduceByKey

优先选择 reduceByKey，　语法更简洁

性能优越

reduceByKey 会进行分区内聚合，再经过网络传输，发送到相对应的分区中。

sortBy 既可以作用于 RDD[K] ，还可以作用于 RDD[(k,v)]

sortByKey　只能作用于 RDD[K,V] 类型上。

1.4.3. Action

动作	含义
reduce(*func*)	通过 func 函数聚集 RDD 中的所有元素
collect()	在驱动程序中，以数组的形式返回数据集的所有元素
collectAsMap	类似于 collect。该函数用于 Pair RDD，最终返回 Map 类型的结果。
count()	返回 RDD 的元素个数
first()	返回 RDD 的第一个元素（类似于 take(1)）
take(*n*)	返回一个由数据集的前 n 个元素组成的数组

saveAsTextFile(*path*)	将数据集的元素以 textfile 的形式保存到 HDFS 文件系统或者其他支持的文件系统
top（n)	按照默认排序（降序） 取数据
takeOrdered(*n, [ordering]*)	与 top 类似，顺序相反 默认是升序
countByKey()	针对(K,V)类型的 RDD，返回一个(K,Int)的 map，表示每一个 key 对应的元素个数。
foreach(*func*)	在数据集的每一个元素上，运行函数 func 进行更新。foreach，任务在 executor 中运行，打印信息也会在 executor 中显示
foreachPartition	对分区进行操作

foreach 和 foreachPartition
foreachParititon 每次迭代一个分区，foreach 每次迭代一个元素。
该方法没有返回值，或者 Unit
主要作用于，没有返回值类型的操作（打印结果，写入到 mysql 数据库中）
在写入到 mysql 数据库的时候，优先使用 foreachPartititon

```
* 结果 存入到 mysql 中
* foreachPartition
* 1,map mapPartition   转换类的算子， 返回值
* 2, 写mysql  数据库的连接
* 100 万          100 万次的连接
* 200 个分区      200 次连接  一个分区中的数据，共用一个连接
```

foreach 和 map 的区别：
map 有返回值，foreach 没有返回值（Unit 类型）
map 是 transformation，lazy 执行，foreach 是 action 算子，触发任务运行
处理的都是每一条数据。

```
rdd1.foreach(println)
rdd1.foreachPartition(it=>println(it.mkString("")))
```

coalesce 和 repartition

def coalesce(numPartitions: Int, shuffle: Boolean = false)(implicit ord: Ordering[T] =

null): RDD[T]

coalesce(n) 原来的分区中的数据，不会被分配到多个分区中，

将 RDD 分区的数量修改为 numPartitions，常用于减少分区

第一个参数为重分区的数目，第二个为是否进行 shuffle，默认为 false

当需要调大分区时，必须设置 shuffle 为 true，才能有效，否则分区数不变

```
def repartition(numPartitions: Int)(implicit ord: Ordering[T] = null): RDD[T] = withScope {
  coalesce(numPartitions, shuffle = true)
}
```

随机重新 shuffle RDD 中的数据，并创建 numPartitions 个分区。这个操作总会通过网络来 shuffle 全部数据。常用于扩大分区

分区数调大调小，都会 shuffle 全部数据，是重量级算子

常用用法?

coalesce(10,true) = repariti ton(10)

如果不需要数据的 shuffle，减少或者合并分区，就使用 coalesce(num)

如果需要数据的 shuffle，或者需要扩大分区数量，优先使用 repartition（num）

扩大分区，作用：提升并行度（业务逻辑比较复杂，需要提升并行度）

```
    // 重分区的api
    rdd1.coalesce(2)     // coalesce(numPartitions: Int, shuffle: Boolean = false
    rdd1.repartition(2)  // coalesce(numPartitions, shuffle = true)
//    repartition 就是coalesce,第二个参数为true
    /**  repartition 要进行数据的shuffle   不管是扩大分区，还是减少分区，都进行shuffle
      *  coalesce 默认没有进行数据的shuffle   减少分区，直接是分区合并。
      *  coalesce   扩大分区,
       */

    val f = (i:Int,it:Iterator[Int]) =>
        it.map(t=> s"part:$i,values:$t")
```

另外还有一类可以修改分区的方式：

在调用 shuffle 类的算子时，可以在参数中设置分区的数量：

```
def reduceByKey(func: (V, V) => V, numPartitions: Int):
```

mapPartitions 和 mapPartitionsWithIndex

def mapPartitions[U](f: (Iterator[T]) => Iterator[U], preservesPartitioning: Boolean = false)(implicit arg0: ClassTag[U]): RDD[U]

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。

如果在映射的过程中需要频繁创建额外的对象，使用 mapPartitions 要比 map 高效。

该方法，看上去是操作的每一条数据，实际上是对 RDD 中的每一个分区进行 iterator，

```
mapPartitions( it: Iterator => {it.map(x => x * 10)})
```

mapPartitionsWithIndex

def mapPartitionsWithIndex[U](f: (Int, Iterator[T]) => Iterator[U], preservesPartitioning: Boolean = false)(implicit arg0: ClassTag[U]): RDD[U]

类似于 mapPartitions，不过提供了两个参数，第一个参数为分区的索引。

mapPartitionsWithIndex 的 func 接受两个参数，第一个参数是分区的索引，第二个是一个数据集分区的迭代器。而输出的是一个包含经过该函数转换的迭代器。

```
val func = (index: Int, iter: Iterator[Int]) => {
  iter.map(x => "[partID:" +  index + ", val: " + x + "]")
}
```

val rdd1 = sc.parallelize(List(1,2,3,4,5,6,7,8,9), 2)

rdd1.mapPartitionsWithIndex(func).collect

```
/**
      * map
      * mapValues
      * mapPartition      操作的是每一个分区     函数的输入参数类型是
Iterator[元素类型]
      * mapPartitionWithIndex
      *
      */

    // 创建rdd  同时指定分区的数量为2 个  数据会被打散，然后平均分配
    val rdd = sc.makeRDD(List(1, 3, 5, 7, 9), 2) // part 0 1

    rdd.map({
      i =>
        // 具体的元素
        i * 10
    })
    // 该方法每次操作的对象是一个迭代器，对应的就是一个分区的数据
    rdd.mapPartitions({
      it =>
        // 分区
        it.map(_ * 10)
    })

    // 函数
    val f=(index:Int,it:Iterator[Int])=>{
      it.map({
        t=> s"part:$index,value=$t"
      })
    }

    val index1: RDD[String] = rdd.mapPartitionsWithIndex({
      // 第一个参数，是分区的索引  分区编号
      // 第二个参数:  分区的数据 Iterator[Int]
      case (index, it) =>
        it.map({
          t =>
            s"part:$index,value=$t"
        })
    })
```

```
//    ArrayBuffer(part:0,value=1, part:0,value=3, part:1,value=5,
part:1,value=7, part:1,value=9)
    // 收集数据并打印
    println(index1.collect().toBuffer)
```

collect 方法：

```
* @note This method should only be used if the resulting array is expected to be small, as
* all the data is loaded into the driver's memory.
*/
```

不能直接把数据收集到 driver 段，然后再执行入库操作。

效率太低，容易引起 dirver 端崩溃了。OOM

1.5. RDD 的属性

操作 RDD 就像操作本地集合一样，数据会以分区为单位，分散到多台机器中运行。

```
* Internally, each RDD is characterized by five main properties:
*
*  - A list of partitions
*  - A function for computing each split
*  - A list of dependencies on other RDDs
*  - Optionally, a Partitioner for key-value RDDs (e.g. to say that the RDD is hash-partitioned)
*  - Optionally, a list of preferred locations to compute each split on (e.g. block locations for
*    an HDFS file)
```

1）一组分片列表（Partition），即数据集的基本组成单位。对于 RDD 来说，每个分片都会被一个计算任务处理，并决定并行计算的粒度。

2）一个计算每个分区的函数。Spark 中 RDD 的计算是以分片为单位的，每个 RDD 都会实现 compute 函数以达到这个目的。compute 函数会对迭代器进行复合，不需要保存每次计算的结果。

对一个分片进行计算，得出一个可遍历的结果

HadoopRDD –》MappartitionsRDD，数据以及逻辑上的转换

compute 函数负责的是父 RDD 分区数据到子 RDD 分区数据的逻辑变换。

3）RDD 之间的依赖关系列表。RDD 的每次转换都会生成一个新的 RDD，所以 RDD 之间就会形成类似于流水线一样的前后依赖关系。在部分分区数据丢失时，Spark 可以通过这个依赖关系重新计算丢失的分区数据，而不是对 RDD 的所有分区进行重新计算。

依赖的分类：宽依赖和窄依赖

4）一个 Partitioner，即 RDD 的分区函数。当前 Spark 中实现了两种类型的分区函数，一个是基于哈希的 HashPartitioner，另外一个是基于范围的 RangePartitioner。只有对于 key-value 的 RDD，才会有 Partitioner，非 key-value 的 RDD 的 Parititioner 的值是 None。Partitioner 函数不但决定了 RDD 本身的分区数量，也决定了 parent RDD Shuffle 输出时的分区数量。还可以自定义分区器

groupByKey,reduceByKey　　hashPartititoner

sortByKey　　RangePartitioner

RDD[Int]　　None

5）一个数据存储列表，存储每个 Partition 的优先位置（preferred location）。对于一个 HDFS 文件来说，这个列表保存的就是每个 Partition 所在的块的位置。按照“移动数据不如移动计算”的理念，Spark 在进行任务调度的时候，会尽可能地将计算任务分配到其所要处理数据块的存储位置。

comupte ,数据的优先位置，rdd 的依赖关系，都不需要我们控制。

分区，分区函数。可以主动修改的

3 个 executor， hdp-02 hdp-03 hdp-04

path = hdfs://hdp-01:9000/wordcount/input/wc.log

该文件有两个 block 块，3 个副本，分别在 hdp-01 hdp-02 hdp-03 上

sc.textFile(path).map(_*10).collect

hdp-02 和 hdp-03 上有数据，也有运算资源，优先把任务在该机器上启动（不需要通过网络传输数据，而是直接读取本地文件）

1.6.分区器：

HashPartitioner 和 RangePartitioner

```
HashPartitioner
val  rdd = sc.makeRDD(List(1,3,4,6,5,9,12,15),3)
val rdd1 = rdd.zipWithIndex.groupByKey()
val f=(i:Int,it:Iterator[(Int,Iterable[Long])]) => it.map(t=> s"part:$i,value:${t._1}")
// 查看结果：
rdd1.mapPartitionsWithIndex(f).collect
//结果数据： Array(part:0,value:15, part:0,value:6, part:0,value:3, part:0,value:12, part:0,value:9, part:1,value:4, part:1,value:1, part:2,value:5)
```

RangePartitioner:

```
val rdd2 = sc.makeRDD(List((1,"xx"),(2,"xx1"),(2,"xx3"),(3,"xx4"),(1000,"xx111"),(6,"xxxxx"),(9,"xx9")),3)
val rdd3  = rdd2.sortByKey()
val f2 = (i:Int,it:Iterator[(Int,String)]) =>it.map(t=> s"part:$i,value:${t._1}")
```

查看结果：

```
rdd3.mapPartitionsWithIndex(f2).collect
// Array(part:0,value:1, part:0,value:2, part:0,value:2, part:1,value:3, part:1,value:6, part:2,value:9, part:2,value:1000)
```