# RDD

Spark的RDD体现了装饰者设计模式。和java的IO流类似。

## 1.1什么是RDD

RDD（Resilient Distributed Dataset）叫做弹性分布式数据集，是Spark中最基本的数据抽象（计算抽象、逻辑抽象）。在代码中表现为一个抽象类，代表一个不可变、可分区、所包含元素可并行计算的集合。

## 1.2 RDD的属性

1）一组分区（partition），即数据集的基本组成单位

2）一个计算每个分区的函数（当前RDD封装的计算操作）

3）RDD之间的依赖关系

4）一个partitioner，即RDD分片函数

5）一个列表，存储存取每个partition的优先位置（preferred location）。因为移动数据不如移动计算，RDD将计算移动到数据上面，然后执行。

## 1.3 RDD特点

RDD表示只读的分区数据集，如果要对RDD进行改动，只能通过RDD的转换操作，由一个RDD得到一个新的RDD，新的RDD包含了从其他RDD衍生所必需的信息。RDDs之间存在依赖关系，RDD的执行是按照血缘关系延时计算的。如果血缘关系过长，可以通过持久化RDD来切断血缘关系。

### 1.3.1 分区

RDD称为弹性分布式数据集，但是它并不存储数据，可理解为一个数据集的代理。通过RDD，我们可以很方便的对这个数据集进行操作。

虽然RDD没有存储数据，但是它记录了应该读取哪些数据，然后对这些数据进行操作。

现在我们对RDD进行分区，把需要读取的数据进行划分，每个分区记录一部分需要读取的数据。那么，就可以对**每个分区的数据开启一个计算任务**，实现**并行计算**。

### 1.3.2 只读

RDD是只读的，要想改变RDD中的数据，只能在现有RDD基础上创建新的RDD。

由一个RDD转换到另一个RDD，可以通过丰富的操作算子实现，不像MapReduce那样只能写map和reduce。

Spark中所有的方法都称为算子，算子分为两类：转换算子和行动算子

转换算子：将RDD进行转换，构建RDD的血缘关系。

行动算子：用来出发RDD的计算，得到RDD的相关计算结果（向应用程序返回结果：**count**、**collect**；向存储系统保存数据：**saveAsTextFile**）

### 1.3.3 依赖

RDDs之间的血缘关系。包括窄依赖和宽依赖。

窄依赖：RDDs之间的分区是一一对应的

宽依赖：下游RDD的每个分区与上游RDD的所有分区都有关系

### 1.3.4 缓存

应用程序中多次使用同一个RDD时，可以将该RDD缓存起来。该RDD只在第一次计算的时候会根据血缘关系得到分区的数据，在后续其他地方用到该RDD时，会直接使用它的缓存数据。

# RDD编程模型

## 2.1 编程模型

要使用Spark，开发者需要编写一个Driver程序，它被提交到集群以调度Worker。Driver中定义了一个或多个RDD，并调用RDD上的action，Worker则执行RDD分区计算任务。

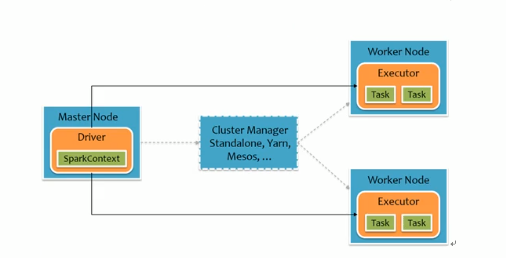
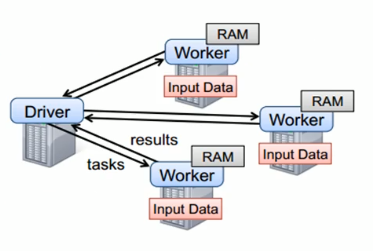
Driver：定义并调用RDD的action。所有创建Spark上下文对象的应用程序称之为Driver

Worker：执行RDD的分区计算任务。所有RDD算子的计算功能全部由Worker（Executor执行）



如果Executor中要使用到Driver中的变量，那么就会涉及到网络IO，这是要保证Driver中的变量能被序列化。





## 2.2 RDD创建

从集合中创建

从外部存储中创建

从其他RDD创建

### 2.2.1 从集合中创建

parallelize和makeRDD 函数

1. parallelize

val rdd = sc.parallelize(Array(1,2,3,4,5,6))

1. makeRDD

val rdd = sc.makeRDD(Array(1,2,3,4,5,6))

makeRDD底层也是调用parallelize方法

### 2.2.2 从外部存储中创建

testFile函数

可以从本地文件系统，也可以从HDFS等分布式文件系统中读

*读取文件时，传递的分区参数为最小分区数，但是不一定是这个分区数，取决于hadoop读取文件时的分片规则（？？？）*

### 2.2.3 从其他RDD创建

## 2.3 RDD的转化

RDD整体上分为Value类型和Key-Value类型。

### 2.3.1Value类型

#### 2.3.1.1 map(func)案例

1、作用：返回一个新的RDD，该RDD由每一个元素经过**func函数**转换后组成

2、需求：创建一个1-10数组的RDD，将所有元素\*2形成新的RDD

|  |
| --- |
| 1、创建  val listRDD: RDD[Int] = sc.makeRDD(1 to 10)   1. map   val mapRDD:RDD[Int] = **listRDD.map( \_\*2)**   1. 打印   mapRDD.collect().foreach(println) |

#### 2.3.1.2 mapPartitions(func)案例

1、作用：类似于map，但独立地在RDD的每个分区（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]，即输入是一个分区，输出还是一个分区。假设有**N个元素**，有**M个分区**，那么**map的函数将被调用N次**，而**mapPartitions被调用M次**，一个函数一次处理所有分区。

|  |
| --- |
| 1、创建  val listRDD: RDD[Int] = sc.makeRDD(1 to 10)   1. mapPartitions   val mapPartitionsRDD: RDD[Int] = **listRDD.mapPartitions(datas => {datas.map(\_\*2)}) #datas即为分到一个分区的数据**   1. 打印   mapPartitionsRDD.collect().foreach(println) |

2、需求：创建一个RDD，使每个元素\*2组成新的RDD

\*\*\*map和mapPartitions的区别\*\*\*

map(\_\*2) 算是Spark中的一个计算

.mapPartitions(datas => {datas.map(\_\*2)})整体算是Spark中的一个计算【datas.map(\_\*2)是scala中的计算，不是Spark的！！！】

所以：

1、mapPartitions效率优于map，因为减少了发送到执行器执行的交互次数（IO次数）

如果有2个分区，每个分区有2个数据，map需要发送4次，mapPartitions只需要发2次（分区数）

1. mapPartitions每次处理一个分区的数据，这个分区的数据处理完后，原RDD分区中数据才能释放，所以可能会出现内存溢出（OOM）
2. 所以当内存空间较大时，可以使用mapPartitions提高效率

#### 2.3.1.3 mapPartitionsWithIndex(func)案例

1、作用：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T的RDD上运行时，func的函数类型必须是(Int, Interator[T])=>Interator[U]

2、需求：创建一个RDD，使**每个元素跟所在区间形成一个元组**，组成一个新的RDD

|  |
| --- |
| 1、创建  val listRDD: RDD[Int] = sc.makeRDD(1 to 10, 2) #10个数放到2个分区中  2、mapPartitionsWithIndex  val tupleRDD: RDD[Int, String] = **listRDD.mapPartitionsWithIndex{**  **case (num, datas) => {**  **datas.map((\_, “分区号：” + num)) #num是数据所在分区的索引**  **}**  **}**  3、打印  tupleRDD.collect().foreach(println)  结果是下面，表示1到5被分在了0分区中，6到10被分在了1分区中  (1,分区号：0)  (2,分区号：0)  (3,分区号：0)  (4,分区号：0)  (5,分区号：0)  (6,分区号：1)  (7,分区号：1)  (8,分区号：1)  (9,分区号：1)  (10,分区号：1) |

#### 2.3.1.4 flatMap(func) 案例

1、作用：类似于map，但是每一个输入元素可以被映射为0或多个输出元素（所以func应该**返回一个序列**，而不是单一元素）

|  |
| --- |
| 1、创建  val listRDD: RDD[List[Int]] = sc.makeRDD(Array(List(1,2), List(3,4))) #这里的每一个元素是list  2、flatMap  val flatMapRDD:RDD[Nothing] = **listRDD.flastMap(datas => datas) #把每个list直接输出**  3、打印  flatMapRDD.collect().foreach(println)  结果是：  1  2  3  4 |

#### 2.3.1.5 glom案例

1、作用：将每一个分区形成一个数组，形成新的RDD类型是**RDD[Array[T]]**

2、需求：创建一个含4个分区的RDD，并将每个分区的数据放到一个数组中

|  |
| --- |
| 1、创建  val listRDD: RDD[Int] = sc.makeRDD(List(1,2,3,4,5,6,7,8), 2)   1. glom 将一个分区的数据放到一个数组中   val glomRDD: RDD[Array[Int]]= listRDD.glom()  3、打印  glomRDD.collect().foreach(array=>{  println(array.mkString(“,”)) #array变string  })  结果是：  1,2,3,4  5,6,7,8 |

#### 2.3.1.6 groupBy(func) 案例

1、作用：分组，按照传入函数的返回值进行分组。将相同的key对应的值放入一个迭代器

2、需求：创建一个RDD，按照元素模以2的值进行分组

|  |
| --- |
| 1、创建  val listRDD: RDD[Int] = sc.makeRDD(List(1,2,3,4))   1. groupBy 将一个分区的数据放到一个数组中   # 分组后的数据形成了对偶元组(K-V)，K表示分组的key，V表示分组的数据集合  val groupByRDD: RDD[(Int, Iterator[Int])]= listRDD.groupBy(i=>i%2)  3、打印  groupRDD.collect().foreach(println)  结果是：  (0,CompactBuffer(2,4))  (1,CompactBuffer(1,3)) |

#### 2.3.1.7 filter(func) 案例

1、作用：过滤。返回一个新的RDD

2、需求：创建一个RDD，过滤出模以2的值为0的元素

|  |
| --- |
| 1、创建  val listRDD: RDD[Int] = sc.makeRDD(List(1,2,3,4))   1. filter   # 生成数据，根据指定规则进行过滤  val filterRDD: RDD[Int]= listRDD.filter(i=>i%2==0)  3、打印  filterRDD.collect().foreach(println)  结果是：  (0,CompactBuffer(2,4))  (1,CompactBuffer(1,3)) |

#### 2.3.1.8 sample(withReplacement, fraction, seed) 案例

1、作用：以指定的随机种子随机抽样出数量为fraction（0到1之间的小数）的数据，withReplacement表示是抽出的数据是否放回，true为有放回的抽样，false为无放回的抽样，seed用于指定随机数生成器的种子。

2、需求：创建一个RDD（1-10），从中选择放回和不放回抽样

|  |
| --- |
| 1、创建  val listRDD: RDD[Int] = sc.makeRDD(1 to 10)  2、sample 抽样  val sampleRDD: RDD[Int]= listRDD.sample(false, 0.4, 1)  3、打印  sampleRDD.collect().foreach(println)  结果是：  1  2  4  10 |

#### 2.3.1.9 distinct([numTasks]) 案例

1、作用：对源RDD进行去重后返回一个新的RDD。默认情况下，只有8个并行任务来操作，但是可以传入一个可选的numTasks参数来操作它

2、需求：创建一个RDD，使用distinct对其去重

|  |
| --- |
| 1、创建  val listRDD: RDD[Int] = sc.makeRDD(List(1,2,1,4,3,6,7,5,4,3,2))  2、sample 抽样 **是shuffle**  val distinctRDD: RDD[Int]= listRDD.distinct(2) #2表示去重之后的分区数  3、打印  distinctRDD.collect().foreach(println)  结果是：  1  2  4  10 |

#### 2.3.1.10 coalesce(numPartitions) 案例

1、作用：缩减分区数，用于大数据集过滤后，提高小数据集的执行效率

2、需求：创建一个4个分区的RDD，对其缩减分区

|  |
| --- |
| 1、创建  val listRDD: RDD[Int] = sc.makeRDD(1 to16 , 4)   1. coalesce 缩减（合并）分区   println(“缩减分区前：”+ listRDD.partitions.size)  val coalesceRDD: RDD[Int]= listRDD.coalesce(3) #3表示分区数改成3  println(“缩减分区后：”+ coalesceRDD.partitions.size)  3、打印  distinctRDD.collect().foreach(println)  结果是：  缩减分区前：4  缩减分区后：3 |