Spark 学习之算子 Transformation 和 Action（四）

作者：Calon
原文链接：https://ld246.com/article/1545389106809
来源网站：链滴

下面记录一些常用的Transformation和Action的用法例子。

初始化

下面的例子都是依赖这个初始化的代码进行演示

```
static JavaSparkContext context;

static {
  SparkSession session = SparkSession.builder().master("local[*]").appName("TestApp").getOr
reate();
  context = JavaSparkContext.fromSparkContext(session.sparkContext());
}
```

Transformation

Map

map是将源JavaRDD的一个一个元素的传入call方法，并经过算法后一个一个的返回从而生成一个新的avaRDD。

```
public static void map() {
  String[] names = {"张无忌", "赵敏", "周芷若"};
  List list = Arrays.asList(names);
  JavaRDD listRDD = context.parallelize(list);
  JavaRDD nameRDD = listRDD.map((Function, String>) v1 -> "Hello," + v1);
  nameRDD.foreach(name -> System.out.println(name));
}
```

打印如下：

```
Hello,张无忌
Hello,周芷若
Hello,赵敏
```

可以看出，对于map算子，源JavaRDD的每个元素都会进行计算，由于是依次进行传参，所以他是有的，新RDD的元素顺序与源RDD是相同的

flatMap

flatMap与map一样，是将RDD中的元素依次的传入call方法，他比map多的功能是能在任何一个传入all方法的元素后面添加任意多元素，而能达到这一点，正是因为其进行传参是依次进行的。

```
public static void flatMap() {
  List list = Arrays.asList("张无忌 赵敏", "宋青书 周芷若");
  JavaRDD listRDD = context.parallelize(list);
  JavaRDD nameRDD = listRDD.flatMap((FlatMapFunction, String>) s -> Arrays.asList(s.split(" "
).iterator());
  nameRDD.foreach(name -> System.out.println(name));
}
```

打印如下：

```
张无忌
赵敏
宋青书
周芷若
```

flatMap的特性决定了这个算子在对需要随时增加元素的时候十分好用，比如在对源RDD查漏补缺时。

mapPartitions

与map方法类似，map是对rdd中的每一个元素进行操作，而mapPartitions则是对rdd中的每个分区迭代器进行操作。如果在map过程中需要频繁创建额外的对象,map需要为每个元素创建一个链接而mpPartition为每个partition创建一个链接),则mapPartitions效率比map高的多。mapPartitions比较合需要分批处理数据的情况。

```
public static void mapParations() {
    List list = Arrays.asList(1, 2, 3, 4, 5, 6);
  JavaRDD listRDD = context.parallelize(list, 2);

  JavaRDD nRDD = listRDD.mapPartitions(iterator -> {
        System.out.println("遍历");
  ArrayList array = new ArrayList<>();
 while (iterator.hasNext()) {
            array.add("hello " + iterator.next());
  }
        return array.iterator();
  });
  nRDD.foreach((VoidFunction) s -> System.out.println(s));
}
```

打印如下：

```
遍历
遍历
hello 4
hello 1
hello 5
hello 6
hello 2
hello 3
```

mapPartitionsWithIndex

每次获取和处理的就是一个分区的数据,并且知道处理的分区的分区号是啥

```
public static void mapPartitionsWithIndex() {
    List list = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);
  JavaRDD listRDD = context.parallelize(list, 2);
  JavaRDD stringJavaRDD = listRDD.mapPartitionsWithIndex((Function2, Iterator, Iterator>) (v1
 v2) -> {
        ArrayList list1 = new ArrayList<>();
 while (v2.hasNext()) {
            list1.add(v1 + "_" + v2.next());
  }
        return list1.iterator();
  }, true);
  stringJavaRDD.foreach(new VoidFunction() {
        @Override
  public void call(String s) throws Exception {
            System.out.println(s);
  }
    });
}
```

打印如下:

```
0_1
0_2
0_3
0_4
1_5
1_6
1_7
1_8
```

union

当要将两个RDD合并时，便要用到union和join，其中union只是简单的将两个RDD累加起来，可以做List的addAll方法。就像List中一样，当使用union及join时，必须保证两个RDD的泛型是一致的。

```
public static void union() {
  final List list1 = Arrays.asList(1, 2, 3, 4);
  final List list2 = Arrays.asList(3, 4, 5, 6);
  final JavaRDD rdd1 = context.parallelize(list1);
  final JavaRDD rdd2 = context.parallelize(list2);
```

```
  JavaRDD unionRDD = rdd1.union(rdd2);
  unionRDD.foreach((VoidFunction) integer -> System.out.println(integer));
}
```

打印如下：

```
1
2
4
3
4
3
6
5
```

groupByKey

groupByKey是将RDD中的元素进行分组，组名是call方法中的返回值，groupByKey是将PairRDD中有相同key值的元素归为一组。

```
public static void groupByKey() {
    List, String>> list = Arrays.asList(
        new Tuple2("美国", "特朗普"),
        new Tuple2("中国", "大大"),
        new Tuple2("美国", "希拉里"),
        new Tuple2("中国", "小小")
    );
  JavaPairRDD, String> listRDD = context.parallelizePairs(list);

  JavaPairRDD, Iterable> groupByKeyRDD = listRDD.groupByKey();
  groupByKeyRDD.foreach(tuple -> {
    String country = tuple._1;
    Iterator iterator = tuple._2.iterator();
    String people = "";
    while (iterator.hasNext()) {
      people = people + iterator.next() + " ";
    }
    System.out.println(country + "人员:" + people);
  });
}
```

打印如下：

```
美国人员:特朗普 希拉里
中国人员:大大 小小
```

join

join是将两个PairRDD合并，并将有相同key的元素分为一组，可以理解为groupByKey和Union的结。

```
public static void join() {
    final List, String>> names = Arrays.asList(
```

```
            new Tuple2, String>(1, "东方不败"),
 new Tuple2, String>(2, "令狐冲"),
 new Tuple2, String>(3, "林平之")
    );
 final List, Integer>> scores = Arrays.asList(
            new Tuple2, Integer>(1, 99),
 new Tuple2, Integer>(2, 98),
 new Tuple2, Integer>(3, 97)
    );

 final JavaPairRDD, String> nemesrdd = context.parallelizePairs(names);
 final JavaPairRDD, Integer> scoresrdd = context.parallelizePairs(scores);

 final JavaPairRDD, Tuple2, Integer>> joinRDD = nemesrdd.join(scoresrdd);
  joinRDD.foreach(tuple -> System.out.println("学号:" + tuple._1 + " 姓名:" + tuple._2._1 + " 成
:" + tuple._2._2));
}
```

打印如下：

```
学号:1 姓名:东方不败 成绩:99
学号:3 姓名:林平之 成绩:97
学号:2 姓名:令狐冲 成绩:98
```

sample

对一个数据集进行随机抽样，withReplacement表示是否有放回抽样，fraction表示返回数据集大小百分比的数量，比如数据集有20条数据，fraction=10,则返回50条，参数seed指定生成随机数的种。

```
public static void sample() {
    ArrayList list = new ArrayList<>();
 for (int i = 1; i <= 20; i++) {
        list.add(i);
  }
    JavaRDD listRDD = context.parallelize(list);
  JavaRDD sampleRDD = listRDD.sample(false, 0.5, 0);
  sampleRDD.foreach(num -> System.out.print(num + " "));
}
```

打印如下：

```
16 18 13 6 7 8 9 1 2 4
```

cartesian

用于求笛卡尔积

```
public static void cartesian() {
    List list1 = Arrays.asList("A", "B");
  List list2 = Arrays.asList(1, 2, 3);
  JavaRDD list1RDD = context.parallelize(list1);
  JavaRDD list2RDD = context.parallelize(list2);
```

```
  list1RDD.cartesian(list2RDD).foreach(tuple -> System.out.println(tuple._1 + "->" + tuple._2));
}
```

打印如下：

```
A->2
A->1
A->3
B->1
B->2
B->3
```

filter

过滤操作，满足filter内function函数为true的RDD内所有元素组成一个新的数据集

```
public static void filter() {
    List list = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
  JavaRDD listRDD = context.parallelize(list);
  JavaRDD filterRDD = listRDD.filter(num -> num % 2 == 0);
  filterRDD.foreach(num -> System.out.print(num + " "));
}
```

打印如下：

```
8 10 6 4 2
```

distinct

返回一个在源数据集去重之后的新数据集，即去重。

```
public static void distinct() {
  List list = Arrays.asList(1, 1, 2, 2, 3, 3, 4, 5);
  JavaRDD listRDD = (JavaRDD) context.parallelize(list);
  listRDD.distinct().foreach(num -> System.out.println(num+",  "));
}
```

打印如下：

```
4，3，2，1，5
```

intersection

对于源数据集和其他数据集求交集，并去重。

```
public static void intersection() {
    List list1 = Arrays.asList(1, 2, 3, 4);
  List list2 = Arrays.asList(3, 4, 5, 6);
  JavaRDD list1RDD = context.parallelize(list1);
  JavaRDD list2RDD = context.parallelize(list2);
  list1RDD.intersection(list2RDD).foreach(num -> System.out.print(num+","));
}
```

打印如下：

3,4

coalesce

重新分区，减少RDD中分区的数量到numPartitions。

```
public static void coalesce() {
    List list = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);
  JavaRDD listRDD = context.parallelize(list, 3);
  listRDD.coalesce(1).foreach(num -> System.out.println(num));
}
```

replication

进行重分区，解决的问题：本来分区数少->增加分区数

```
public static void replication() {
    List list = Arrays.asList(1, 2, 3, 4);
  JavaRDD listRDD = context.parallelize(list, 1);
  listRDD.repartition(2).foreach(num -> System.out.println(num));
}
```

repartitionAndSortWithinPartitions

repartitionAndSortWithinPartitions函数是repartition函数的变种，与repartition函数不同的是，repartitionAndSortWithinPartitions在给定的partitioner内部进行排序，性能比repartition要高。

```
public static void repartitionAndSortWithinPartitions() {
    List list = Arrays.asList(1, 4, 55, 66, 33, 48, 23);
  JavaRDD listRDD = context.parallelize(list, 1);
  JavaPairRDD, Integer> pairRDD = listRDD.mapToPair(num -> new Tuple2<>(num, num));
  pairRDD.repartitionAndSortWithinPartitions(new HashPartitioner(2))
          .mapPartitionsWithIndex((index, iterator) -> {
              ArrayList list1 = new ArrayList<>();
 while (iterator.hasNext()) {
                  list1.add(index + "_" + iterator.next());
  }
              return list1.iterator();
  }, false)
          .foreach(str -> System.out.println(str));
}
```

cogroup

cogroup对两个RDD中的KV元素，每个RDD中相同key中的元素分别聚合成一个集合。与reduceByKy不同的是针对两个RDD中相同的key的元素进行合并。

```
public static void cogroup() {
    List, String>> list1 = Arrays.asList(
          new Tuple2, String>(1, "www"),
```

```
 new Tuple2, String>(2, "bbs")
   );

  List, String>> list2 = Arrays.asList(
          new Tuple2, String>(1, "cnblog"),
 new Tuple2, String>(2, "cnblog"),
 new Tuple2, String>(3, "very")
   );

  List, String>> list3 = Arrays.asList(
          new Tuple2, String>(1, "com"),
 new Tuple2, String>(2, "com"),
 new Tuple2, String>(3, "good")
   );

  JavaPairRDD, String> list1RDD = context.parallelizePairs(list1);
  JavaPairRDD, String> list2RDD = context.parallelizePairs(list2);
  JavaPairRDD, String> list3RDD = context.parallelizePairs(list3);

  list1RDD.cogroup(list2RDD, list3RDD).foreach(tuple ->
          System.out.println(tuple._1 + " " + tuple._2._1() + " " + tuple._2._2() + " " + tuple._2._3()
);
}
```

打印如下：

```
1 [www] [cnblog] [com]
2 [bbs] [cnblog] [com]
3 [] [very] [good]
```

sortByKey

sortByKey函数作用于Key-Value形式的RDD，并对Key进行排序。从函数的实现可以看出，它主要受两个函数，含义和sortBy一样，这里就不进行解释了。该函数返回的RDD一定是ShuffledRDD类型，因为对源RDD进行排序，必须进行Shuffle操作，而Shuffle操作的结果RDD就是ShuffledRDD。其这个函数的实现很优雅，里面用到了RangePartitioner，它可以使得相应的范围Key数据分到同一个partition中，然后内部用到了mapPartitions对每个partition中的数据进行排序，而每个partition中数的排序用到了标准的sort机制，避免了大量数据的shuffle。

```
public static void sortByKey() {
    List, String>> list = Arrays.asList(
          new Tuple2<>(99, "张三丰"),
 new Tuple2<>(96, "东方不败"),
 new Tuple2<>(66, "林平之"),
 new Tuple2<>(98, "聂风")
   );
  JavaPairRDD, String> listRDD = context.parallelizePairs(list);
  listRDD.sortByKey(true).foreach(tuple -> System.out.println(tuple._2 + "->" + tuple._1));
}
```

打印如下：

```
张三丰->99
林平之->66
```

```
聂风->98
东方不败->96
```

aggregateByKey

aggregateByKey函数对PairRDD中相同Key的值进行聚合操作，在聚合过程中同样使用了一个中立初始值。和aggregate函数类似，aggregateByKey返回值的类型不需要和RDD中value的类型一致因为aggregateByKey是对相同Key中的值进行聚合操作，所以aggregateByKey函数最终返回的类型是Pair RDD，对应的结果是Key和聚合好的值；而aggregate函数直接是返回非RDD的结果，这点需注意。在实现过程中，定义了三个aggregateByKey函数原型，但最终调用的aggregateByKey函数一致。

```
public static void aggregateByKey() {
    List list = Arrays.asList("you,jump", "i,jump");
  JavaRDD listRDD = context.parallelize(list);
  listRDD.flatMap(line -> Arrays.asList(line.split(",")).iterator())
          .mapToPair(word -> new Tuple2<>(word, 1))
          .aggregateByKey(0, (x, y) -> x + y, (m, n) -> m + n)
          .foreach(tuple -> System.out.println(tuple._1 + "->" + tuple._2));
}
```

打印如下：

```
you->1
i->1
jump->2
```

Action

reduce

reduce是将RDD中的所有元素进行合并，当运行call方法时，会传入两个参数，在call方法中将两个数合并后返回，而这个返回值会返回一个新的RDD中的元素再次传入call方法中，继续合并，直到合到只剩下一个元素时。

```
public static void reduce() {
    List list = Arrays.asList(1, 2, 3, 4, 5, 6);
  JavaRDD listRDD = context.parallelize(list);

  Integer result = listRDD.reduce((x, y) -> x + y);
  System.out.println(result);
}
```

打印如下：

```
21
```

reduceByKey

和reduce类似，但是reduceByKey仅将RDD中所有K,V对中K值相同的V进行合并。

```
public static void reduceByKey() {
    List, Integer>> list = Arrays.asList(
            new Tuple2, Integer>("中国", 99),
 new Tuple2, Integer>("美国", 97),
 new Tuple2, Integer>("泰国", 89),
 new Tuple2, Integer>("中国", 77)
    );
  JavaPairRDD, Integer> listRDD = context.parallelizePairs(list);

  JavaPairRDD, Integer> resultRDD = listRDD.reduceByKey((x, y) -> x + y);
  resultRDD.foreach(tuple -> System.out.println("国家: " + tuple._1 + "->" + tuple._2));
}
```

打印如下：

```
门派: 美国->97
门派: 中国->176
门派: 泰国->89
```

collect

将一个RDD以一个Array数组形式返回其中的所有元素。

```
public static void collect(){
    List list = Arrays.asList("you,jump", "i,jump");
  JavaRDD listRDD = context.parallelize(list);
  List collect = listRDD.collect();
}
```

count

返回数据集中元素个数，默认Long类型。

first

返回数据集的第一个元素

countByKey

用于统计RDD[K,V]中每个K的数量，返回具有每个key的计数的（k，int）pairs的hashMap

foreach

对数据集中每一个元素运行函数function

saveAsObjectFile

将数据集中元素以ObjectFile形式写入本地文件系统或者HDFS等

saveAsSequenceFile

将dataSet中元素以Hadoop SequenceFile的形式写入本地文件系统或者HDFS等。

saveAsTextFile

将dataSet中元素以文本文件的形式写入本地文件系统或者HDFS等。Spark将对每个元素调用toStrin方法，将数据元素转换为文本文件中的一行记录。

takeOrdered

返回RDD中前n个元素，并按默认顺序排序（升序）或者按自定义比较器顺序排序。

take

返回一个包含数据集前n个元素的数组（从0下标到n-1下标的元素），不排序。

扫一扫有惊喜：
[![imagepng](http://itechor.top/solo/upload/bb791a58c3a84193b7f643b6849482c5_image.pg)](http://ym0214.com)