



下载APP



03 | RDD常用算子（一）：RDD内部的数据转换

2021-09-15 吴磊

《零基础入门Spark》

课程介绍 >



讲述：吴磊

时长 22:04 大小 20.22M



你好，我是吴磊。

在上一讲的最后，我们用一张表格整理了 Spark [官网](#)给出的 RDD 算子。想要在 Spark 之上快速实现业务逻辑，理解并掌握这些算子无疑是至关重要的。

因此，在接下来的几讲，我将带你一起梳理这些常见算子的用法与用途。不同的算子，就像是厨房里的炒勺、铲子、刀具和各式各样的锅碗瓢盆，只有熟悉了这些“厨具”的操作方法，才能在客人点餐的时候迅速地做出一桌好菜。



今天这一讲，我们先来学习同一个 RDD 内部的数据转换。掌握 RDD 常用算子是做好 Spark 应用开发的基础，而数据转换类算子则是基础中的基础，因此我们优先来学习这类 RDD 算子。

在这些算子中，我们重点讲解的就是 map、mapPartitions、flatMap、filter。这 4 个算子几乎囊括了日常开发中 99% 的数据转换场景，剩下的 mapPartitionsWithIndex，我把它留给你作为课后作业去探索。

算子类型	适用范围	算子用途	算子集合
Transformations	任意RDD	RDD内数据转换	map mapPartitions mapPartitionsWithIndex flatMap filter
	Paired RDD	RDD内数据聚合	groupByKey sortByKey reduceByKey aggregateByKey
	任意RDD	RDD间数据整合	union intersection join cogroup cartesian
	任意RDD	数据整理	sample distinct
	任意RDD	数据分布	coalesce repartition repartitionAndSortWithinPartitions
Actions	任意RDD	数据收集	collect first take takeSample takeOrdered count
	任意RDD	数据持久化	saveAsTextFile saveAsSequenceFile saveAsObjectFile
	任意RDD	数据遍历	foreach

RDD算子分类表

俗话说，巧妇难为无米之炊，要想玩转厨房里的厨具，我们得先准备好米、面、油这些食材。学习 RDD 算子也是一样，要想动手操作这些算子，咱们得先有 RDD 才行。

所以，接下来我们就一起来看看 RDD 是怎么创建的。

创建 RDD


在 Spark 中，创建 RDD 的典型方式有两种：

通过 `SparkContext.parallelize` 在**内部数据**之上创建 RDD；

通过 `SparkContext.textFile` 等 API 从**外部数据**创建 RDD。

这里的内部、外部是相对应用程序来说的。开发者在 Spark 应用中自定义的各类数据结构，如数组、列表、映射等，都属于“内部数据”；而“外部数据”指代的，是 Spark 系统之外的所有数据形式，如本地文件系统或是分布式文件系统中的数据，再比如来自其他大数据组件（Hive、Hbase、RDBMS 等）的数据。

第一种创建方式的用法非常简单，只需要用 `parallelize` 函数来封装内部数据即可，比如下面的例子：


 复制代码

```
1 import org.apache.spark.rdd.RDD
2 val words: Array[String] = Array("Spark", "is", "cool")
3 val rdd: RDD[String] = sc.parallelize(words)
```

你可以在 `spark-shell` 中敲入上述代码，来直观地感受 `parallelize` 创建 RDD 的过程。通常来说，在 Spark 应用内定义体量超大的数据集，其实都是不太合适的，因为数据集完全由 Driver 端创建，且创建完成后，还要在全网范围内跨节点、跨进程地分发到其他 Executors，所以往往会带来性能问题。因此，`parallelize` API 的典型用法，是在“小数据”之上创建 RDD。

要想在真正的“大数据”之上创建 RDD，我们还得依赖第二种创建方式，也就是通过 `SparkContext.textFile` 等 API 从**外部数据**创建 RDD。由于 `textFile` API 比较简单，而且它在日常的开发中出现频率比较高，因此我们使用 `textFile` API 来创建 RDD。在后续对各类 RDD 算子讲解的过程中，我们都会使用 `textFile` API 从文件系统创建 RDD。

为了保持讲解的连贯性，我们还是使用第一讲中的源文件 `wikiOfSpark.txt` 来创建 RDD，代码实现如下所示：

 复制代码

```
1 import org.apache.spark.rdd.RDD
2 val rootPath: String = _
3 val file: String = s"${rootPath}/wikiOfSpark.txt"
4 // 读取文件内容
5 val lineRDD: RDD[String] = spark.sparkContext.textFile(file)
```

好啦，创建好了 RDD，我们就有了可以下锅的食材。接下来，咱们就要正式地走进厨房，把铲子和炒勺挥起来啦。

RDD 内的数据转换


首先，我们先来认识一下 map 算子。毫不夸张地说，在所有的 RDD 算子中，map “出场”的概率是最高的。因此，我们必须掌握 map 的用法与注意事项。

map：以元素为粒度的数据转换

我们先来说说 map 算子的用法：**给定映射函数 f，map(f) 以元素为粒度对 RDD 做数据转换**。其中 f 可以是带有明确签名的带名函数，也可以是匿名函数，它的形参类型必须与 RDD 的元素类型保持一致，而输出类型则任由开发者自行决定。

这种照本宣科的介绍听上去难免会让你有点懵，别着急，接下来我们用些小例子来更加直观地展示 map 的用法。


在 [🔗 第一讲](#) 的 Word Count 示例中，我们使用如下代码，把包含单词的 RDD 转换成元素为 (Key , Value) 对的 RDD，后者统称为 Paired RDD。

 复制代码

```
1 // 把普通RDD转换为Paired RDD
2 val cleanWordRDD: RDD[String] = _ // 请参考第一讲获取完整代码
3 val kvRDD: RDD[(String, Int)] = cleanWordRDD.map(word => (word, 1))
```

在上面的代码实现中，传递给 map 算子的形参，即：word => (word , 1)，就是我们上面说的映射函数 f。只不过，这里 f 是以匿名函数的方式进行定义的，其中左侧的 word 表示匿名函数 f 的输入形参，而右侧的 (word , 1) 则代表函数 f 的输出结果。

如果我们把匿名函数变成带名函数的话，可能你会看的更清楚一些。这里我用一段代码重新定义了带名函数 f。


 复制代码

```
1 // 把RDD元素转换为 ( Key , Value ) 的形式
2
3 // 定义映射函数f
4 def f(word: String): (String, Int) = {
5   return (word, 1)
6 }
7
8 val cleanWordRDD: RDD[String] = _ // 请参考第一讲获取完整代码
9 val kvRDD: RDD[(String, Int)] = cleanWordRDD.map(f)
```

可以看到，我们使用 Scala 的 def 语法，明确定义了带名映射函数 f，它的计算逻辑与刚刚的匿名函数是一致的。在做 RDD 数据转换的时候，我们只需把函数 f 传递给 map 算子即可。不管 f 是匿名函数，还是带名函数，map 算子的转换逻辑都是一样的，你不妨把以上两种实现方式分别敲入到 spark-shell，去验证执行结果的一致性。

到这里为止，我们就掌握了 map 算子的基本用法。现在你就可以定义任意复杂的映射函数 f，然后在 RDD 之上通过调用 map(f) 去翻着花样地做各种各样的数据转换。

比如，通过定义如下的映射函数 f，我们就可以改写 Word Count 的计数逻辑，也就是把 “Spark” 这个单词的统计计数权重提高一倍：

 复制代码

```
1 // 把RDD元素转换为 ( Key , Value ) 的形式
2
3 // 定义映射函数f
4 def f(word: String): (String, Int) = {
5   if (word.equals("Spark")) { return (word, 2) }
6   return (word, 1)
7 }
8
9 val cleanWordRDD: RDD[String] = _ // 请参考第一讲获取完整代码
10 val kvRDD: RDD[(String, Int)] = cleanWordRDD.map(f)
```

尽管 map 算子足够灵活，允许开发者自由定义转换逻辑。不过，就像我们刚刚说的，map(f) 是以元素为粒度对 RDD 做数据转换的，在某些计算场景下，这个特点会严重影响

执行效率。为什么这么说呢？我们来看一个具体的例子。

比方说，我们把 Word Count 的计数需求，从原来的对单词计数，改为对单词的哈希值计数，在这种情况下，我们的代码实现需要做哪些改动呢？我来示范一下：

[复制代码](#)

```
1 // 把普通RDD转换为Paired RDD
2
3 import java.security.MessageDigest
4
5 val cleanWordRDD: RDD[String] = _ // 请参考第一讲获取完整代码
6
7 val kvRDD: RDD[(String, Int)] = cleanWordRDD.map{ word =>
8     // 获取MD5对象实例
9     val md5 = MessageDigest.getInstance("MD5")
10    // 使用MD5计算哈希值
11    val hash = md5.digest(word.getBytes).mkString
12    // 返回哈希值与数字1的Pair
13    (hash, 1)
14 }
```


由于 map(f) 是以元素为单元做转换的，那么对于 RDD 中的每一条数据记录，我们都需要实例化一个 MessageDigest 对象来计算这个元素的哈希值。

在工业级生产系统中，一个 RDD 动辄包含上百万甚至是上亿级别的数据记录，如果处理每条记录都需要事先创建 MessageDigest，那么实例化对象的开销就会聚沙成塔，不知不觉地成为影响执行效率的罪魁祸首。

那么问题来了，有没有什么办法，能够让 Spark 在更粗的数据粒度上去处理数据呢？还真有，mapPartitions 和 mapPartitionsWithIndex 这对“孪生兄弟”就是用来解决类似的问题。相比 mapPartitions，mapPartitionsWithIndex 仅仅多出了一个数据分区索引，因此接下来我们把重点放在 mapPartitions 上面。

mapPartitions：以数据分区为粒度的数据转换

按照介绍算子的惯例，我们还是先来说说 mapPartitions 的用法。mapPartitions，顾名思义，就是**以数据分区为粒度，使用映射函数 f 对 RDD 进行数据转换**。对于上述单词哈希值计数的例子，我们结合后面的代码，来看看如何使用 mapPartitions 来改善执行性能：

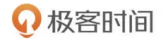
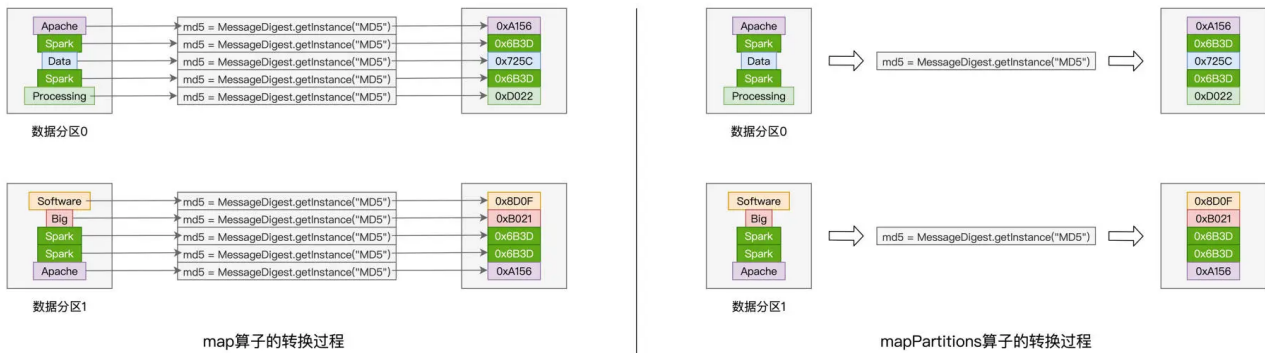
 复制代码

```
1 // 把普通RDD转换为Paired RDD
2
3 import java.security.MessageDigest
4
5 val cleanWordRDD: RDD[String] = _ // 请参考第一讲获取完整代码
6
7 val kvRDD: RDD[(String, Int)] = cleanWordRDD.mapPartitions( partition => {
8     // 注意！这里是以数据分区为粒度，获取MD5对象实例
9     val md5 = MessageDigest.getInstance("MD5")
10    val newPartition = partition.map( word => {
11        // 在处理每一条数据记录的时候，可以复用同一个Partition内的MD5对象
12        md5.digest(word.getBytes).mkString
13    })
14    newPartition
15 })
```

可以看到，在上面的改进代码中，mapPartitions 以数据分区（匿名函数的形参 partition）为粒度，对 RDD 进行数据转换。具体的数据处理逻辑，则由代表数据分区的形参 partition 进一步调用 map(f) 来完成。你可能会说：“partition.map(f) 仍然是以元素为粒度做映射呀！这和前一个版本的实现，有什么本质上的区别呢？”

仔细观察，你就会发现，相比前一个版本，我们把实例化 MD5 对象的语句挪到了 map 算子之外。如此一来，以数据分区为单位，实例化对象的操作只需要执行一次，而同一个数据分区中所有的数据记录，都可以共享该 MD5 对象，从而完成单词到哈希值的转换。

通过下图的直观对比，你会发现，以数据分区为单位，mapPartitions 只需实例化一次 MD5 对象，而 map 算子却需要实例化多次，具体的次数则由分区内数据记录的数量来决定。



map与mapPartitions的区别

对于一个有着上百万条记录的 RDD 来说，其数据分区的划分往往是在百这个量级，因此，相比 map 算子，mapPartitions 可以显著降低对象实例化的计算开销，这对于 Spark 作业端到端的执行性能来说，无疑是非常友好的。

实际上。除了计算哈希值以外，对于数据记录来说，凡是可以共享的操作，都可以用 mapPartitions 算子进行优化。这样的共享操作还有很多，比如创建用于连接远端数据库的 Connections 对象，或是用于连接 Amazon S3 的文件系统句柄，再比如用于在线推理的机器学习模型，等等，不一而足。你不妨结合实际工作场景，把你遇到的共享操作整理到留言区，期待你的分享。

相比 mapPartitions，mapPartitionsWithIndex 仅仅多出了一个数据分区索引，这个数据分区索引可以为我们获取分区编号，当你的业务逻辑中需要使用到分区编号的时候，不妨考虑使用这个算子来实现代码。除了这个额外的分区索引以外，mapPartitionsWithIndex 在其他方面与 mapPartitions 是完全一样的。

介绍完 map 与 mapPartitions 算子之后，接下来，我们趁热打铁，再来看一个与这两者功能类似的算子：flatMap。

flatMap：从元素到集合、再从集合到元素

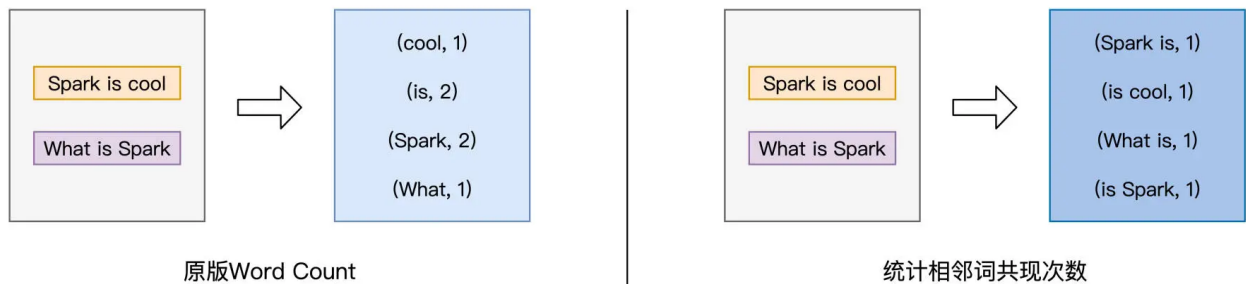
flatMap 其实和 map 与 mapPartitions 算子类似，在功能上，与 map 和 mapPartitions 一样，flatMap 也是用来做数据映射的，在实现上，对于给定映射函数 f，flatMap(f) 以元素为粒度，对 RDD 进行数据转换。

不过，与前两者相比，flatMap 的映射函数 f 有着显著的不同。对于 map 和 mapPartitions 来说，其映射函数 f 的类型，都是（元素）=>（元素），即元素到元素。而 flatMap 映射函数 f 的类型，是（元素）=>（集合），即元素到集合（如数组、列表等）。因此，flatMap 的映射过程在逻辑上分为两步：

以元素为单位，创建集合；

去掉集合“外包装”，提取集合元素。

这么说比较抽象，我们还是来举例说明。假设，我们再次改变 Word Count 的计算逻辑，由原来统计单词的计数，改为统计相邻单词共现的次数，如下图所示：



变更 Word Count 计算逻辑

对于这样的计算逻辑，我们该如何使用 flatMap 进行实现呢？这里我们先给出代码实现，然后再分阶段地分析 flatMap 的映射过程：

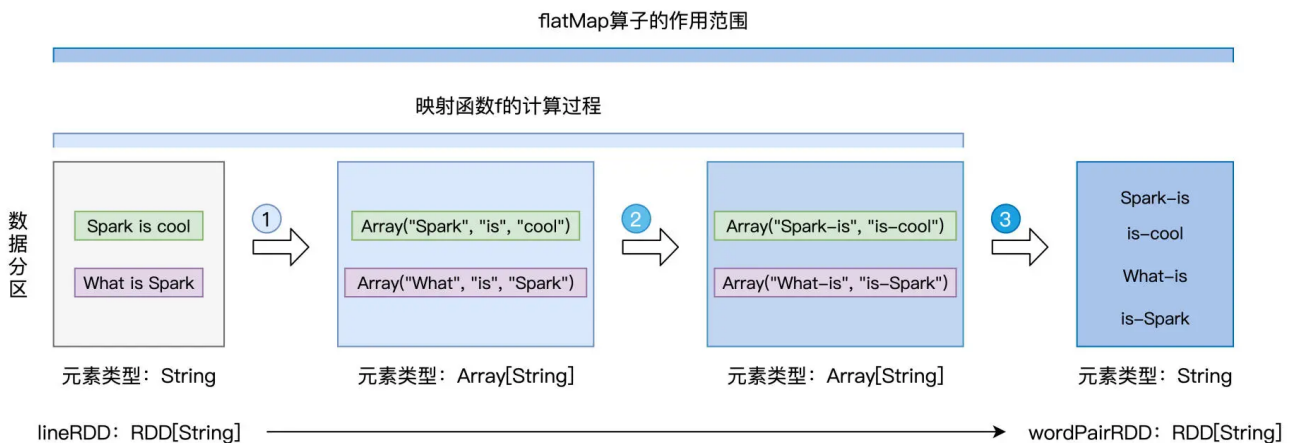
复制代码

```
1 // 读取文件内容
2 val lineRDD: RDD[String] = _ // 请参考第一讲获取完整代码
3 // 以行为单位提取相邻单词
4 val wordPairRDD: RDD[String] = lineRDD.flatMap( line => {
5     // 将行转换为单词数组
6     val words: Array[String] = line.split(" ")
7     // 将单个单词数组，转换为相邻单词数组
8     for (i <- 0 until words.length - 1) yield words(i) + "-" + words(i+1)
9 })
```

在上面的代码中，我们采用匿名函数的形式，来提供映射函数 f 。这里 f 的形参是 `String` 类型的 `line`，也就是源文件中的一行文本，而 f 的返回类型是 `Array[String]`，也就是 `String` 类型的数组。在映射函数 f 的函数体中，我们先用 `split` 语句把 `line` 转化为单词数组，然后再用 `for` 循环结合 `yield` 语句，依次把单个的单词，转化为相邻单词词对。

注意，`for` 循环返回的依然是数组，也即类型为 `Array[String]` 的词对数组。由此可见，函数 f 的类型是 $(String) \Rightarrow (Array[String])$ ，也就是刚刚说的第一步，**从元素到集合**。但如果我们去观察转换前后的两个 RDD，也就是 `lineRDD` 和 `wordPairRDD`，会发现它们的类型都是 `RDD[String]`，换句话说，它们的元素类型都是 `String`。

回顾 `map` 与 `mapPartitions` 这两个算子，我们会发现，转换前后 RDD 的元素类型，与映射函数 f 的类型是一致的。但在 `flatMap` 这里，却出现了 RDD 元素类型与函数类型不一致的情况。这是怎么回事呢？其实呢，这正是 `flatMap` 的“奥妙”所在，为了让你直观地理解 `flatMap` 的映射过程，我画了一张示意图，如下所示：



不难发现，映射函数 f 的计算过程，对应着图中的步骤 1 与步骤 2，每行文本都被转化为包含相邻词对的数组。紧接着，`flatMap` 去掉每个数组的“外包装”，提取出数组中类型为 `String` 的词对元素，然后以词对为单位，构建新的数据分区，如图中步骤 3 所示。这就是 `flatMap` 映射过程的第二步：**去掉集合“外包装”，提取集合元素**。

得到包含词对元素的 `wordPairRDD` 之后，我们就可以沿用 Word Count 的后续逻辑，去计算相邻词汇的共现次数。你不妨结合文稿中的代码与第一讲中 Word Count 的代码，去实现完整版的“相邻词汇计数统计”。

filter : 过滤 RDD


在今天的最后，我们再来学习一下，与 map 一样常用的算子：filter。filter，顾名思义，这个算子的作用，是对 RDD 进行过滤。就像是 map 算子依赖其映射函数一样，filter 算子也需要借助一个判定函数 f，才能实现对 RDD 的过滤转换。

所谓判定函数，它指的是类型为 (RDD 元素类型) => (Boolean) 的函数。可以看到，判定函数 f 的形参类型，必须与 RDD 的元素类型保持一致，而 f 的返回结果，只能是 True 或者 False。在任何一个 RDD 之上调用 filter(f)，其作用是保留 RDD 中满足 f (也就是 f 返回 True) 的数据元素，而过滤掉不满足 f (也就是 f 返回 False) 的数据元素。

老规矩，我们还是结合示例来讲解 filter 算子与判定函数 f。

在上面 flatMap 例子的最后，我们得到了元素为相邻词汇对的 wordPairRDD，它包含的是像 "Spark-is"、"is-cool" 这样的字符串。为了仅保留有意义的词对元素，我们希望结合标点符号列表，对 wordPairRDD 进行过滤。例如，我们希望过滤掉像 "Spark-&"、"|-data" 这样的词对。

掌握了 filter 算子的用法之后，要实现这样的过滤逻辑，我相信你很快就能写出如下的代码实现：

 复制代码

```
1 // 定义特殊字符列表
2 val list: List[String] = List("&", "|", "#", "^", "@")
3
4 // 定义判定函数f
5 def f(s: String): Boolean = {
6   val words: Array[String] = s.split("-")
7   val b1: Boolean = list.contains(words(0))
8   val b2: Boolean = list.contains(words(1))
9   return !b1 && !b2 // 返回不在特殊字符列表中的词汇对
10 }
11
12 // 使用filter(f)对RDD进行过滤
13 val cleanedPairRDD: RDD[String] = wordPairRDD.filter(f)
```

掌握了 filter 算子的用法之后，你就可以定义任意复杂的判定函数 f，然后在 RDD 之上通过调用 filter(f) 去变着花样地做数据过滤，从而满足不同的业务需求。

重点回顾

好啦，到此为止，关于 RDD 内数据转换的几个算子，我们就讲完了，我们一起来做个总结。今天这一讲，你需要掌握 map、mapPartitions、flatMap 和 filter 这 4 个算子的作用和具体用法。

首先，我们讲了 map 算子的用法，它允许开发者自由地对 RDD 做各式各样的数据转换，给定映射函数 f，map(f) 以元素为粒度对 RDD 做数据转换。其中 f 可以是带名函数，也可以是匿名函数，它的形参类型必须与 RDD 的元素类型保持一致，而输出类型则任由开发者自行决定。

为了提升数据转换的效率，Spark 提供了以数据分区为粒度的 mapPartitions 算子。mapPartitions 的形参是代表数据分区的 partition，它通过在 partition 之上再次调用 map(f) 来完成数据的转换。相比 map，mapPartitions 的优势是以数据分区为粒度初始化共享对象，这些共享对象在我们日常的开发中很常见，比如数据库连接对象、S3 文件句柄、机器学习模型等等。

紧接着，我们介绍了 flatMap 算子。flatMap 的映射函数 f 比较特殊，它的函数类型是（元素）=>（集合），这里集合指的是像数组、列表这样的数据结构。因此，flatMap 的映射过程在逻辑上分为两步，这一点需要你特别注意：

以元素为单位，创建集合；

去掉集合“外包装”，提取集合元素。

最后，我们学习了 filter 算子，filter 算子的用法与 map 很像，它需要借助判定函数 f 来完成对 RDD 的数据过滤。判定函数的类型必须是（RDD 元素类型）=>（Boolean），也就是形参类型必须与 RDD 的元素类型保持一致，返回结果类型则必须是布尔值。RDD 中的元素是否能够得以保留，取决于判定函数 f 的返回值是 True 还是 False。

虽然今天我们只学了 4 个算子，但这 4 个算子在日常开发中的出现频率非常之高。掌握了这几个简单的 RDD 算子，你几乎可以应对 RDD 中 90% 的数据转换场景。希望你对这几个算子多多加以练习，从而在日常的开发工作中学以致用。

每课一练

讲完了正课，我来给你留 3 个思考题：

1. 请你结合 [官网](#) 的介绍，自学 `mapPartitionsWithIndex` 算子。请你说一说，在哪些场景下可能会用到这个算子？
2. 对于我们今天学过的 4 个算子，再加上没有详细解释的 `mapPartitionsWithIndex`，你能说说，它们之间有哪些共性或是共同点吗？
3. 你能说一说，在日常的工作中，还遇到过哪些可以在 `mapPartitions` 中初始化的共享对象呢？

欢迎你在评论区回答这些练习题。你也可以把这一讲分享给更多的朋友或者同事，和他们一起讨论讨论，交流是学习的催化剂。我在评论区等你。

分享给需要的人，Ta 订阅后你可得 **20 元现金奖励**

 赞 4  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 02 | RDD与编程模型：延迟计算是怎么回事？

下一篇 04 | 进程模型与分布式部署：分布式计算是怎么回事？

更多学习推荐

175 道 Go 工程师 大厂常考面试题

限量免费领取 

精选留言 (7)

 写留言

qinsi

2021-09-16

不是很懂spark。mapPartitions()那里，光从代码上来看的话，在map()的闭包里可以访问到外面mapPartitions()闭包里的同一个md5实例，从而达到共享实例的效果。那么有没有可能在最外层创建一个全局的md5实例，这样就算只用map()，在闭包里访问的也是这同一个实例？这样会有什么问题吗？或者说在这种情况下mapPartitions()相比map()还有什么优势？

展开 ∨

作者回复: 非常好的问题~ 先赞一个 

先说结论，你说的没错，通过在Driver端创建全局变量，然后在map中进行引用，也可以达到mapPartitions同样的效果。原理就是你说的函数闭包，其实本质上，它就是个对象，包含了外部数据结构的函数对象。

但是，需要注意的是，通过这种方式创建的闭包，一定要保证Driver端创建的全局变量是可以序列化的，也就是实现了Serializable接口。只有满足这个前提，Spark才能在这个闭包之上创建分布式任务，否则会报“Task not serializable”错误。

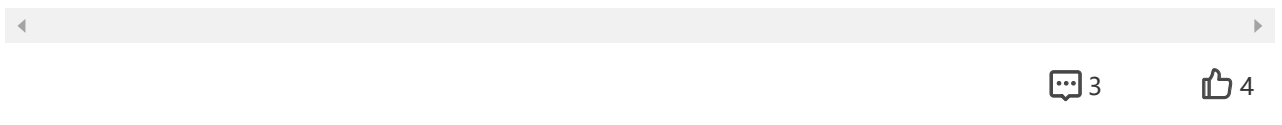
这个错误其实很好理解，要把Driver创建的对象带到Executors，这个对象一定是要可序列化的才行。不过遗憾的是，很多共享对象，比如咱们本讲的MD5，都没有实现Serializable接口，所以D

river端创建的MD5实例，Spark “带不动”（没法从Driver带到Executors），这一点你不妨动手试一试~

不过这个点提得非常好，对于那些自定义的Class，我们可以让它实现Serializable接口，从而可以轻松地在Driver创建Spark可以“带的动”的全局变量。

说到这里，我追问老弟一句：在Driver创建MD5实例，map中直接引用，Spark会报“Task not Serializable”；那为什么在mapPartitions里面创建MD5实例，map引用这个对象，Spark却没有报“Task not Serializable”错误呢？

老弟不妨再想一想，咱们评论区继续讨论~



Geek_390836

2021-09-16

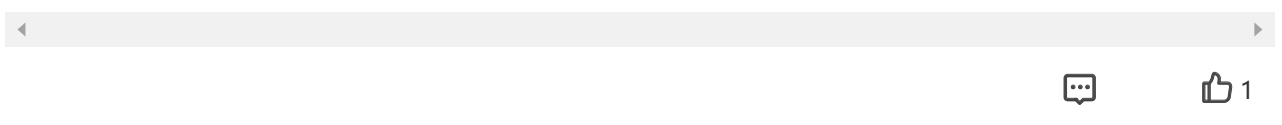
参考map和mapPartitions，为什么mapPartitions中的map，是对record进行getBytes而不是word.getBytes操作，刚学spark，求老师解答

展开 ∨

作者回复: 好问题~

你说的对，这里笔误了，已经让编辑帮忙改成了“word.getBytes”哈~

感谢纠正~



钱鹏 Allen

2021-09-18

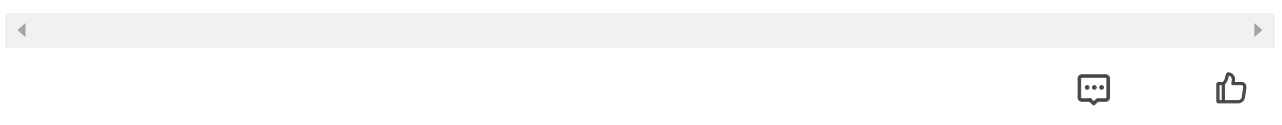
mapPartitionsWithIndex 需要结合分区索引使用

map filter flatMap mapPartitions mapPartitionsWithIndex 通过函数，传递参数，返回新的RDD

mapPartitions 采集系统中，只需实例化一次电表号，可读其他各类读数

展开 ∨

作者回复: 正解~



**大叮当**

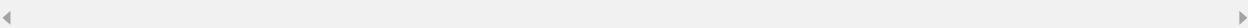
2021-09-16

同问老师,AIK问的问题，什么时候用小括号什么时候用花括号啊，感觉scala实在有点过于灵活

作者回复: Scala语法确实比较灵活，一般来说，简单表达式用()，复杂表达式用{}。

比如，简单函数体：`(x => x + 1)`、`(_ + 1)`，等等；

复杂函数体：`{case x: String => “一大堆关于x的转换” }`

**求索**

2021-09-16

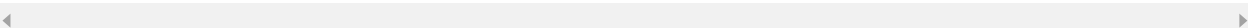
请教个问题，spark sql执行缓慢，然而资源却闲置状态，这是何解

作者回复: 这里的资源，具体指的是哪类资源呢？CPU、内存、磁盘还是网络？

我理解，听上去是CPU资源闲置，也就是利用率不高。如果是这样的话，那么大概率是因为任务时间大部分都花在I/O wait上了。也就是CPU一直在“等活儿干”，CPU闲置期间，任务的主要操作很可能是Shuffle期间的磁盘I/O和网络I/O，这个时候，CPU确实没什么活儿可干，只能等，所以看上去是CPU闲置，实则是它在“等活儿”~

解决办法的话，可以围绕着提升Shuffle的效率去展开，这部分我们在后面的Shuffle和Broadcast Join部分会讲，敬请期待~

当然，这些分析，都是基于对资源是CPU的假设，如果是其他资源，我们再讨论~

**AIK**

2021-09-15

1、mapPartitionsWithIndex的应用场景：获取数据所在的分区并对特定分区数据进行特定处理。

2、这些算子首先都是惰性算子，又都使用函数作为参数传入，达到一对一的效果。

3、在mapPartitions中只创建过连接数据库的共享对象，其他的没有遇到过，还请老师能不能多给几个实际案例，提高一下对mapPartitions的认知。...

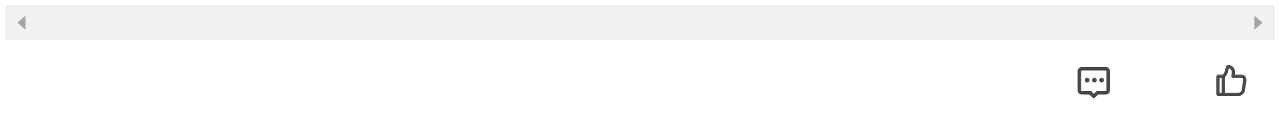
展开 ∨

作者回复: 1)、2)、3) 都没问题~

关于Scala的语法，确实比较灵活，一般来说，简单表达式用()，复杂表达式用{}。

比如，简单函数体：`(x => x + 1)`、`(_ + 1)`，等等；

复杂函数体：`{case x: String => "一大堆关于x的转换" }`



崔小豪

2021-09-15

遇到一个关于filter 函数的 bug on reduceByKey。

我运行第一个例子代码如下可以得到正确的结果：

...

```
val cleanedPairRDD: RDD[String] = wordPairRDD.filter(word => !word.equals(""))
```

.....

展开 ✓

作者回复: 这个列表`List("&", "|", "#", "^", "@", "\", "-")`，里面多了一个“-”，在后面分割字符串的时候：`val words: Array[String] = s.split("-")`，同样用到了“-”。所以这里分割出来的字符串数组，它的大小可能会出现不一致。

把分隔符换成其他字符就好了~

