链滴

Learning Spark 中文版 -- 第六章 --Spark 高级编程（2）

作者：pattyq

原文链接：https://ld246.com/article/1532571550584

来源网站：链滴

Working on a Per-Partition Basis
（基于分区的作）

以每个分区为基础处理数据使我们可以避免为每个数据项重做配置工作。如打开数据库连接或者创建机数生成器这样的操作，我们希望避免为每个元素重做配置工作。Spark有分区版本的map和foreach通过让RDD的每个分区只运行一次代码，可帮助降低这些操作的成本。

回到我们的呼号例子中，有一个无线电台呼号的在线数据库，我们可以查询联系日志的公共列表。通使用基于分区的操作，我们可以分享数据库的连接池来避免为多个连接配置连接参数，并且可以复用JON解析器。如Example6-10到6-12所示，我们使用mapPartitions()函数，这个函数会给我们一个输RDD每个分区中元素的迭代器，并且期望我们的返回是一个结果的迭代器。

Example 6-10. Shared connection pool in Python

```
def processCallSigns(signs):
    """Lookup call signs using a connection pool"""
    # Create a connection pool
    http = urllib3.PoolManager()
    # the URL associated with each call sign record
    urls = map(lambda x: "http://73s.com/qsos/%s.json" % x, signs)
    # create the requests (non-blocking)
    requests = map(lambda x: (x, http.request('GET', x)), urls)
    # fetch the results
    result = map(lambda x: (x[0], json.loads(x[1].data)), requests)
    # remove any empty results and return
    return filter(lambda x: x[1] is not None, result)

def fetchCallSigns(input):
    """Fetch call signs"""
    return input.mapPartitions(lambda callSigns : processCallSigns(callSigns))

contactsContactList = fetchCallSigns(validSigns)
```

Example 6-11. Shared connection pool and JSON parser in Scala

```
val contactsContactLists = validSigns.distinct().mapPartitions{
    signs =>
    val mapper = createMapper()
    val client = new HttpClient()
    client.start()
    // create http request
    signs.map {sign =>
        createExchangeForSign(sign)
    // fetch responses
    }.map{ case (sign, exchange) =>
        (sign, readExchangeCallLog(mapper, exchange))
    }.filter(x => x._2 != null) // Remove empty CallLogs
}
```

Example 6-12. Shared connection pool and JSON parser in Java

```
// Use mapPartitions to reuse setup work.
```

```
JavaPairRDD<String, CallLog[]> contactsContactLists =
    validCallSigns.mapPartitionsToPair(
    new PairFlatMapFunction<Iterator<String>, String, CallLog[]>() {
        public Iterable<Tuple2<String, CallLog[]>> call(Iterator<String> input) {
        // List for our results.
            ArrayList<Tuple2<String, CallLog[]>> callsignLogs = new ArrayList<>();
            ArrayList<Tuple2<String, ContentExchange>> requests = new ArrayList<>();
            ObjectMapper mapper = createMapper();
            HttpClient client = new HttpClient();
            try {
                client.start();
                while (input.hasNext()) {
                    requests.add(createRequestForSign(input.next(), client));
                }
                for (Tuple2<String, ContentExchange> signExchange : requests) {
                    callsignLogs.add(fetchResultFromRequest(mapper, signExchange));
                }
            } catch (Exception e) {
            }
            return callsignLogs;
        }});

System.out.println(StringUtils.join(contactsContactLists.collect(), ","));
```

当使用基于分区的操作时，Spark会给函数一个分区中元素的迭代器。我们的返回结果也是一个迭代。除了mapPartitions()之外，Spark还有很多其他分区操作，如表6-1中所示:

函数名	所调用内容	返回类型
DD[T]函数签名		
mapPartitions()	分区元素的迭代器	返回元素的迭代器
f: (Iterator[T]) ->Iterator[U]		
mapPartitionsWithIndex()	分区数量和分区元素的迭代器	回元素的迭代器
f: (Int, Iterator[T]) -> Iterator[U]		
foreachPartition()	元素迭代器	无
: (Iterator[T]) -> Unit		

除了可以避免重复的配置工作，我们有时还可以使用mapPartitions()来避免创建对象的开销。有时们需要创建一个对象来聚合不同类型的结果。回想一下第三章我们计算平均数的时候，当时把数值RD转换成了元组RDD，这样便于跟踪在归约步骤中元素处理的数量从而计算平均值。除了这种对每个元都处理的方式，我们可以为每个分区创建一次元组，示例如下:

Example 6-13. Average without mapPartitions() in Python

```
def combineCtrs(c1, c2):
    return (c1[0] + c2[0], c1[1] + c2[1])

def basicAvg(nums):
    """Compute the average"""
    nums.map(lambda num: (num, 1)).reduce(combineCtrs)
```

Example 6-14. Average with mapPartitions() in Python

```
def partitionCtr(nums):
    """Compute sumCounter for partition"""
```

```
    sumCount = [0, 0]
    for num in nums:
        sumCount[0] += num
        sumCount[1] += 1
    return [sumCount]

def fastAvg(nums):
    """Compute the avg"""
    sumCount = nums.mapPartitions(partitionCtr).reduce(combineCtrs)
    return sumCount[0] / float(sumCount[1])
```

Piping to External Programs(连接外部程序)

Spark提供三种开箱即用的语言支持，你似乎在编写Spark程序时不会遇到语言上的障碍。但是，如果cala，Java和Python都不是你想要的，那也没关系，SPark提供了一个使用其他语言(如R脚本)传输数给程序的机制，。

Spark在RDD上提供了一个pipe()方法。这个方法可以让我们使用其他语言编写job，但前提是这种语支持Unix标准流的读入和写出。使用pipe（），可以编写一个RDD转换，它从标准输入读取每个RD元素作为String，接着按照需要操作该String，然后将结果作为String写入标准输出。提供的接口和程模型是受到了一些限制，但是有时后你需要做的就是像写在map()或filter()中使用的本地代码函数样。

你可能会想把RDD的内容传输给外部程序或者脚本，因为你可能已经构建并测试了一个复杂的软件，且你想在Spark中重用他。有很多数据科学家有R语言写的代码，我们可以通过pipe()函数与R程序进交互。

在Example6-15中，我们使用R语言的库来计算所有联系人的距离。RDD中的每个元素通过换行符作分割写出，并且程序输出的每一行都是结果RDD中的字符串元素。为了使R程序转换输入数据简便一，我们把数据转换成mylat, mylon, theirlat, theirlon格式。我们以逗号作为分隔符。

Example 6-15. R distance program

```
#!/usr/bin/env Rscript
library("Imap")
f <- file("stdin")
open(f)
while(length(line <- readLines(f,n=1)) > 0) {
    # process line
    contents <- Map(as.numeric, strsplit(line, ","))
    mydist <- gdist(contents[[1]][1], contents[[1]][2],
                    contents[[1]][3], contents[[1]][4],
                    units="m", a=6378137.0, b=6356752.3142, verbose = FALSE)
    write(mydist, stdout())
}
```

如果把上述代码写入一个名为./src/R/finddistance.R的可执行文件，那么使用方法大体如下:

```
$ ./src/R/finddistance.R
37.75889318222431,-122.42683635321838,37.7614213,-122.4240097
349.2602
coffee
NA
ctrl-d
```

目前为止很不错，我们学会了从stdin读取的每一行转换成stdout输出的方法了。现在我们需要让每工作节点都能使用finddistance.R并且能让shell脚本转换我们的RDD。完成这两点其实在Spark中很单，示例如下:

Example 6-16. Driver program using pipe() to call finddistance.R in Python

```
# Compute the distance of each call using an external R program
distScript = "./src/R/finddistance.R"
distScriptName = "finddistance.R"
sc.addFile(distScript)
def hasDistInfo(call):
    """Verify that a call has the fields required to compute the distance"""
    #验证计算距离的必要字段
    requiredFields = ["mylat", "mylong", "contactlat", "contactlong"]
    return all(map(lambda f: call[f], requiredFields))
def formatCall(call):
    """Format a call so that it can be parsed by our R program"""
    #格式话数据便于R程序解析
    return "{0},{1},{2},{3}".format(
        call["mylat"], call["mylong"],
        call["contactlat"], call["contactlong"])

pipeInputs = contactsContactList.values().flatMap(
    lambda calls: map(formatCall, filter(hasDistInfo, calls)))
distances = pipeInputs.pipe(SparkFiles.get(distScriptName))
print distances.collect()
```

Example 6-17. Driver program using pipe() to call finddistance.R in Scala

```
// Compute the distance of each call using an external R program
// adds our script to a list of files for each node to download with this job
//通过外部的R程序计算每个呼叫的距离
//把脚本添加到这个job中每个节点都要下载的文件列表
val distScript = "./src/R/finddistance.R"
val distScriptName = "finddistance.R"
sc.addFile(distScript)
val distances = contactsContactLists.values.flatMap(x => x.map(y =>
    s"$y.contactlay,$y.contactlong,$y.mylat,$y.mylong")).pipe(Seq(
        SparkFiles.get(distScriptName)))
println(distances.collect().toList)
```

Example 6-18. Driver program using pipe() to call finddistance.R in Java

```
// Compute the distance of each call using an external R program
// adds our script to a list of files for each node to download with this job
String distScript = "./src/R/finddistance.R";
String distScriptName = "finddistance.R";
sc.addFile(distScript);
JavaRDD<String> pipeInputs = contactsContactLists.values()
    .map(new VerifyCallLogs()).flatMap(
    new FlatMapFunction<CallLog[], String>() {
        public Iterable<String> call(CallLog[] calls) {
            ArrayList<String> latLons = new ArrayList<String>();
            for (CallLog call: calls) {
```

```
            latLons.add(call.mylat + "," + call.mylong +
                    "," + call.contactlat + "," + call.contactlong);
        }
        return latLons;
      }
    });
JavaRDD<String> distances = pipeInputs.pipe(SparkFiles.get(distScriptName));
System.out.println(StringUtils.join(distances.collect(), ","));
```

使用SparkContext.addFile(path)方法，我们可以构建一个在单个job中每个工作节点都要下载的文列表。这些文件可以来自驱动程序的本地文件系统（如例子中那样），或者来自HDFS或其他Hadoo支持的文件系统，也可以是一个HTTP,HTTPS或FTP的URI。当一个action在一个job中运行，每个节都会开始下载这些文件。这些文件可以在工作节点的SparkFiles.getRootDirectory路径或者SparkFiles get(filename)中找到。当然，这只是确定pipe()方法能够在每个工作节点上找到脚本文件的方法之一你可以使用其他的远程复制工具来将脚本文件放在每个节点可知位置上。

使用SparkContext.addFile（path）添加的所有文件都存放在同一个目录中，因此使用唯一名称非常要。

一旦确定了脚本是可以使用的，RDD上的pipe()方法可以很简单地把RDD中的元素传输给脚本。也许智能的findDistance版本会接受SEPARATOR作为命令行参数。在那种情况下，以下任何一个都可以成这项工作，尽管第一个是首选：

- rdd.pipe(Seq(SparkFiles.get("finddistance.R"), ","))
- rdd.pipe(SparkFiles.get("finddistance.R") + " ,")

第一种方式，我们把命令的调用当做具有位置的元素序列来传递（命令本身在零偏移的位置上）；第种方式，我们把它作为一个单独的命令字符串传递，然后Spark会分解成位置参数 。

我们也可以在需要的情况下明确设置pipe()的环境变量。直接把环境变量的map作为pipe()的第二个数，Spark会去设置这些在值。

你现在应该至少理解了如何使用pipe()来利用外部命令处理RDD的元素，还有如何将命令脚本分发到群中工作节点能找到的位置。

Numeric RDD Operations（数值RDD的操作）

Spark提供了几种包含数值数据RDD的描述性统计操作。除此之外，更复杂的统计和机器学习方法之，我们将在后面的第11章中介绍这些方法。

Spark的数值操作是通过流式算法实现的，它允许一次构建我们模型的一个元素。描述性统计对数据次性计算出结果并且调用stats()会返回一个StatsCounter对象。表6-2介绍了StatsCounter对象的方。

Table 6-2. Summary statistics available fromStatsCounter

方法	作用
count()	元素数量
mean()	元素平均值
sum()	总值
max()	最大值

min() 最小值

variance() 方差

sampleVariance() 样本方差

stdev() 标准差

sampleStdev() 样本标准差

如果您只想计算其中一个统计数据，那么也可以直接在RDD上调用相应的方法，例如rdd.mean（）或dd.sum（）。

在例6-19到6-21中，我们将使用汇总统计来从我们的数据中删除一些异常值。因为我们可能会遍历同的RDD两次（一次是汇总统计，第二次是删除异常值），所以将RDD缓存起来。回到我们的呼叫日的例子上，我们可以从呼叫日志上删除距离太远的联系地点。

Example 6-19. Removing outliers in Python

```
# Convert our RDD of strings to numeric data so we can compute stats and
# remove the outliers.
distanceNumerics = distances.map(lambda string: float(string))
stats = distanceNumerics.stats()
stddev = std.stdev()
mean = stats.mean()
reasonableDistances = distanceNumerics.filter(
    lambda x: math.fabs(x - mean) < 3 * stddev)
print reasonableDistances.collect()
```

Example 6-20. Removing outliers in Scala

```
// Now we can go ahead and remove outliers since those may have misreported locations
// first we need to take our RDD of strings and turn it into doubles.
val distanceDouble = distance.map(string => string.toDouble)
val stats = distanceDoubles.stats()
val stddev = stats.stdev
val mean = stats.mean
val reasonableDistances = distanceDoubles.filter(x => math.abs(x-mean) < 3 * stddev)
println(reasonableDistance.collect().toList)
```

Example 6-21. Removing outliers in Java

```
// First we need to convert our RDD of String to a DoubleRDD so we can
// access the stats function
JavaDoubleRDD distanceDoubles = distances.mapToDouble(new DoubleFunction<String>() {
    public double call(String value) {
       return Double.parseDouble(value);
    }});
final StatCounter stats = distanceDoubles.stats();
final Double stddev = stats.stdev();
final Double mean = stats.mean();
JavaDoubleRDD reasonableDistances =
    distanceDoubles.filter(new Function<Double, Boolean>() {
       public Boolean call(Double x) {
          return (Math.abs(x-mean) < 3 * stddev);}});
System.out.println(StringUtils.join(reasonableDistance.collect(), ","));
```

配合累加器和广播变量，分区处理，外部程序接口和汇总统计，我们最终完成了示例程序。

完整的代码在 src/python/ChapterSixExample.py,src/main/scala/com/oreilly/learningsparkexamples/scala/ChapterSixExample.scala,和src/main/java/com/oreilly/learningsparkexamples/java/hapterSixExample.java这几个文件之中。

Conclusion（总结）

这一章节，我们介绍了一些Spark的高级编程特性，你可以使用这些特性来提高你程序的效率或可读。后续章节将介绍部署和调整Spark应用程序，以及用于SQL，流媒体以及机器学习的内置库。我们将开始看到更复杂和更完整的示例应用程序，它们利用了迄今为止描述的大部分功能，并且也会对你己使用Spark的方式有所指导或启发。