Spark笔记（1）：RDD编程

2019-06-19 | 大数据，Spark | 2

Spark笔记系列我们准备以《Spark大数据分析》这本书的总体框架为主线，从RDD编程的核心概念说起，到基本的RDD操作、数据IO、Spark Job，以及Spark SQL、Spark Streaming、Spark MLlib这些Spark组件，结合实例系统的进行讲解，之后会将其延伸开来，争取照顾到Spark的方方面面。

“Apache Spark is a unified analytics engine for large-scale data processing.”这是来自官网的介绍，Spark是一个用于大数据处理的统一分析引擎，上亿的大数据集在单机上跑一个分析几乎不可能，而在Spark上可以以分钟级别的速度就可以完成，这要归功于其先进的调度程序DAG、查询优化器和物理执行引擎，这几个概念在后边会一一介绍，总而言之就是Spark出乎意料的快。除了性能好，Spark还异常亲民，你可以用Java写、用Scala写、用Python写，同时也支持R、SQL，上手非常简单，用惯了python DataFrame的可以在Spark找到对应的DataFrames库，用惯了SQL的分析员也可以在里头找到SQL，机器学习工程师也照样可以使用MLlib进行建模。

在Spark中，有一个核心概念叫RDD（Resilient Distributed Dataset），基本所有的操作都是围绕其展开的，所以第一节我们先讲解RDD编程的核心概念和基本操作，当然这之前要先按照官网提供的安装教程进行安装好Spark。如果没有集群，可以先在单机版上练习。

RDD 核心概念

Spark中，所有的数据操作归纳起来就三种：

- RDD的创建（create）
- RDD的转化（transformation）
- RDD的行动（action）

那什么是RDD呢？

RDD是Resilient Distributed Dataset的简称，首先Dataset意味着RDD是一个数据集，但它与我们常见的数据集格式不同，他是弹性的（Resilient），如何理解弹性，就是在集群上的某一节点失效时可以高效地重建数据集，RDD就像海绵一样有弹性似的，在被挤压之后仍可以恢复完整，即是容错的。

在容错这一点上，Spark采用了记录数据更新而不是数据检查点的方式，因为数据检查点方式会消耗大量的存储资源，但若更新达到一定的数量，记录数据更新的成本也很高。因此，RDD只支持粗粒度的转化，我们后面会看到RDD都是在大量记录上执行的单个操作。

而分布式（Distributed）指的是每个RDD都被分为多个分区，这些分区运行在集群中的不同节点上，这样就保证了其负载均衡（多台机器负载）、扩展性强（多台机器扩展）的优点。

以上这些概念比较抽象，我们来看一个具体的RDD从创建到转化再到行动的实例，从实例出发了解RDD编程:

假如我们有一个存储在HDFS上的文件，其路径为 `hdfs:///adalgo/profile/20190615/*` ，每一行的格式如下所示，\t左边为用户id，\t右边为用户的兴趣分类（单个人没有重复的key），以json格式存储。假设总共有一亿多行这样的数据（id没有重复），我们想要得到的最终结果是每一个类别的人数。

```
['00000f9d\t{"cat":{"football":2,"basketball":3}]
```

以下是本节作为示例的代码：

```
%pyspark
import json

## 创建
profile =  sc.textFile("hdfs:///adalgo/profile/20190615/*")

## 转化
profile_2 = profile.filter(lambda x:len(x.split('\t'))==2).map(lambda x: x.split('\t')[

## 行动
profile_2.take(100)

## 转化
scat = profile_2.flatMap(lambda x:x.split(' '))
scat_count = ad_scat.map(lambda x:(x,1)).reduceByKey(lambda x,y: x+y)
ad_scat_count.toDF().orderBy("_2")

## 行动
scat_count.show(100)
```

接下来我们一边讲解概念，一边解读这段代码。

RDD 创建

一般我们可以通过两种方式创建RDD。

一种是在驱动器程序中并行的对象集合，简单说就是一个我们希望传入的列表或者元组：

num_rdd = sc.parallelize([1,2,3])

忽然冒出了一个sc，这是一个什么玩意儿？

每一个spark程序都是由driver program发起集群并行操作的，当我们启动了spark shell时，就自动创建了一个SparkContext对象，即sc变量，可以用它来创建RDD。

但这种方式一般就是在平时测试的时候用，在真实生产场景中很少用到，因为这种方式需要将整个数据集先放到driver程序所在的机器的内存中。

另一种是从外部存储中读取数据，比如HDFS/Amazon S3，具体的会在后面的章节介绍，我们的程序实例中就是从HDFS文件系统读取的数据集：

```
1   user_ad_profile =  sc.textFile("hdfs:///adalgo/profile/20190615/*")
```

这样我们就可以接着对这个RDD进行操作了。

RDD 操作与惰性求值

RDD有两种操作：转化操作和行动操作。那如何定义转化和行动操作呢？转化操作返回的是一个新的RDD，而行动操作则是返回结果或把结果写入外部系统，触发实际的计算。

这样说还是有点抽象，看实例中就是一连串的转化操作，首先 `.filter(lambda x:len(x.split('\t'))==2)` 是从 `user_ad_profile_1` 中筛选出用 `\t` 分隔后长度为2的行，把一些不合法的行也去掉，否则后边就会报错；之后的 `.map(lambda x: x.split('\t')[1])` 是取出第2个元素，用 `.map(lambda x: json.loads(x))` 来加载json格式元素，在最后用 `.map(lambda x:''.join(x['ad_scat'].keys()))` 取出自字典所有的key返回一个空格分隔的列表，具体的每个函数的作用我们会详细介绍。以上这些操作返回的其实都是经过转化操作作用在各个元素上然后生成一个新的RDD，但并没有执行真正的计算操作。这就是之后会提到的惰性求值。

```
1   ## 转化
2   profile_2 = profile.filter(lambda x:len(x.split('\t'))==2).map(lambda x: x.split('\t')[1
```

而行动操作就是会有真正的计算，比如下面 `.take(100)` 是从 `profile_2` 中返回200个元素。

```
1   ## 行动
2   profile_2.take(100)
```

这里面有一个很重要的概念：惰性求值。比如在函数式编程语言中，表达式往往不在它被绑定到变量之后就立即求值，而是在该值被取用的时候求值，对应到Spark中就是在调用行动操作之前Spark不会开始计算，无论是读取数据还是转化操作，都是如此。

在执行操作时，Spark会记录下当前执行操作的指令列表：在读取数据时，Spark即使执行了sc.textFile()的操作，也不会真正的读取数据出来，等到行动操作时才会读取。在执行转化操作时，操作也不会立即执行，它只是记录了一个计算操作的指令列表。

那为何要惰性求值呢？因为如果每经过一次转化操作都触发真正的计算，将会有系统负担，而惰性求值会将多个转化操作合并到一起，抵消不必要的步骤后，在最后必要的时才进行运算，获得性能的提升同时又减轻系统运算负担。

接下来让我们看一下常见的一些转化操作和行动操作。

RDD 转化操作

- 基本转化操作：以num_rdd = sc.parallelize([‘Hello New World’,’Hello China’])为例

函数名	目的	示例	结果
map()	将函数应用于每一个元素中，返回值构成新的RDD	num_rdd.map(lambda x: x.lower())	[‘hello new world’,’hello china’]
flatMap()	把元素内容铺展开来，将函数作用于所有的元素内容	num_rdd.flatMap(lambda x: x.split(‘ ‘))	[‘Hello’,’New’,’World’,’Hello’,’China’]
filter()	元素过滤	num_rdd.map(lambda x:len(x.split(‘ ‘))==2)	[‘Hello China’]
distinct()	去重	num_rdd.distinct()	[‘Hello New World’,’Hello China’]

- 集合转换操作，以rdd_1=[1,2,3],rdd_2=[3,4,5]为例

函数名	目的	示例	结果
union()	合并两个RDD所有元素（不去重）	rdd1.union(rdd2)	[1,2,3,3,4,5]
intersection()	求两个RDD的交集	rdd_1.intersection(rdd2)	[3]
substract()	移除在RDD2中存在的RDD1元素	rdd_1.substract(rdd2)	[1,2]

函数名	目的	示例	结果
cartesian()	求两个RDD的笛卡尔积	rdd_1.cartesian(rdd2)	[(1,3),(1,4),(1,5)...(3,5)]

RDD 行动操作

基本行动操作，以rdd = [1,2,3,3]为例

函数名	目的	示例	结果
collect()	收集并返回RDD中所有元素，往往在单元测试时使用，要求数据可放入单台机器内存	rdd.collect()	[1,2,3,3]
count()	RDD中元素的个数	rdd.count()	4
countByValue()	各元素出现的个数	rdd.countByValue()	[(1,1),(2,1),(3,2)]
take(num)	从RDD中返回前num个元素，用于单元测试和快速调试	rdd.take(2)	[1,2]
top(num)	返回降序排序最前面的num个元素，	rdd.take(2)	[3,3]
reduce(f)	并行整合RDD中所有元素，返回一个同一类型元素	rdd.reduce(lambda x: x+y)	9
fold(zeroValue)(f)	与reduce一样，不过需要提供初始值	rdd.fold(0)(lambda x,y: x+y)	9
aggregate(zeroValue)(seqOp , combOp)	与reduce相似，不过返回不同类型的元素	rdd.aggregate((0,0)) (lambda x,y: (x[0] + y, x[1] + 1), lambda x,y: (x[0] + y[0], x[1] + y[1]))	[9,4]
foreach(f)	给每个元素使用给定的函数，结果不需发回本地	rdd.foreach(f)	无

其中 `aggregate` 和 `fold` 函数的理解会稍难一些，可以查阅资料深入了解。

持久化

上面说到，RDD是惰性求值的，而我们会重复使用同一个RDD，而如果简单的对其调用行动操作，Spark每次都会重算RDD，资源消耗很大。比如加入我们上述说的实例中，有以下的执行：

```
1   input = sc.parallelize([1,2,3])
```

```
2   print result.count()
3   print result.collect()
```

这就多次计算同一个RDD了，为了避免这种情况，我们可以使用persist()对其持久化存储，而且可以根据需求选择不同的持久化级别，一般内存成分多的速度会快一些，磁盘部分多的速度稍慢一些，如下：

级别	使用空间	CPU时间	是否在内存中	是否在磁盘上
MEMORY_ONLY	高	低	是	否
MEMORY_ONLY_SER	低	高	是	否
MEMORY_AND_DISK	高	中	部分	部分
MEMORY_AND_DISK_SER	低	高	部分	部分
DISK_ONLY	低	高	否	是

比如下面的这个例子：

```
1   input = sc.parallelize([1,2,3])
2   result = input.persist(StorageLevel.DISK_ONLY)
3   print result.count()
4   print result.collect()
```

若想把持久化的RDD从缓存中移除，可以使用unpersist()方法。

实例讲解

最后把一开始的实例讲解一下：

```
1   %pyspark
2   import json
3
4   ## 创建
5   profile =  sc.textFile("hdfs:///adalgo/profile/20190615/*")
6
7   ## 转化
8   profile_2 = profile.filter(lambda x:len(x.split('\t'))==2).map(lambda x: x.split('\t')[
9
10  ## 行动
11  profile_2.take(100)
12
13  ## 转化
14  scat = profile_2.flatMap(lambda x:x.split(' '))
```

```
scat_count = ad_scat.map(lambda x:(x,1)).reduceByKey(lambda x,y: x+y)
ad_scat_count.toDF().orderBy("_2")

## 行动
scat_count.show(100)
```

首先 `.filter(lambda x:len(x.split('\t'))==2)` 是从 `user_ad_profile_1` 中筛选出用 `\t` 分隔后长度为2的行，把一些不合法的行也去掉，否则后边就会报错；之后的 `.map(lambda x: x.split('\t')[1])` 是取出第2个元素，用 `.map(lambda x: json.loads(x))` 来加载json格式元素，在最后用 `.map(lambda x:' '.join(x['ad_scat'].keys()))` 取出自字典所有的key返回一个空格分隔的列表。

`profile_2.flatMap(lambda x:x.split(' '))` 是把所有单词分割开，铺展开为一个列表；`ad_scat.map(lambda x:(x,1))` 则对单词进行单个计数，即都记为1； `reduceByKey(lambda x,y: x+y)` 是根据单词作为key，对其总计数； `.toDF().orderBy("_2")` 则是将其DataFrame化，并按第二列排序，这样就完成了一个简单的单词计数，同时也是各个类别人数计数的功能。

以上就是Spark RDD中最重要的一些概念讲解，包括RDD的基本概念、创建、转化操作、行动操作与持久化，下一章节我们会继续跟着快速大数据分析一起看一下Spark并行聚合、分组操作，即键值对的操作。

参考资料：《Spark快速大数据分析》

----------本文结束🐾感谢您的阅读----------

欢迎关注个人公众号【应统联盟】，一起交流学习算法、编程与大数据。

感谢打赏，您的支持将鼓励我继续创作！

赏

RDD编程 # Spark

林中路（9）：阿玛蒂亚·森 | 身份与暴力：命运的幻象

广告&推荐面面观（1）：经久不衰的逻辑回归