task two

RDD编程

RDD创建的两种方式

1. 读取外部数据集。本地加载数据集，或从HDFS/HBASE/Cassandra/Amazon S3等外部数据源加载数据集。spark支持文本文件，sequencefile(hadoop提供的sequencefile是一个由二进制序列化过的key/value的字节流组成的文本存储文件)文件和其它符合hadoop inputformat格式的文件。

   Spark采用textFile()方法来从文件系统中加载数据创建RDD，该方法把文件的URI作为参数，这个URI可以是本地文件系统的地址，或者是分布式文件系统HDFS的地址，或者是Amazon S3的地址等等。

   ```
   lines = sc.textFile("file:///usr/local/spark/mycode/rdd/word.txt")    # 从本地读取文件
   lines = sc.textFile("hdfs://localhost:9000/user/hadoop/word.txt")    # 从hdfs读取文件
   ```

   在使用Spark读取文件时，需要说明以下几点： （1）如果使用了本地文件系统的路径，那么，必须要保证在所有的worker节点上，也都能够采用相同的路径访问到该文件，比如，可以把该文件拷贝到每个worker节点上，或者也可以使用网络挂载共享文件系统。 （2）textFile()方法的输入参数，可以是文件名，也可以是目录，也可以是压缩文件等。比如，textFile("/my/directory"), textFile("/my/directory/*.txt"), and textFile("/my/directory/*.gz"). （3）textFile()方法也可以接受第2个输入参数（可选），用来指定分区的数目。默认情况下，Spark会为HDFS的每个block创建一个分区（HDFS中每个block默认是128MB）。你也可以提供一个比block数量更大的值作为分区数目，但是，你不能提供一个小于block数量的值作为分区数目。

2. 调用sparkcontext的parallelize方法，从driver中一个已经存在的集合(数组)上创建

   ```
   >>> nums = [1,2,3,4,5]
   >>> rdd = sc.parallelize(nums)
   ```

RDD的操作

RDD被创建好以后，在后续使用过程中一般会发生两种操作：

- 转换（Transformation）： 基于现有的数据集创建一个新的数据集。

  对于RDD而言，每一次转换操作都会产生不同的RDD，供给下一个“转换”使用。转换得到的RDD是惰性求值的，也就是说，整个转换过程只是记录了转换的轨迹，并不会发生真正的计算，只有遇到行动操作时，才会发生真正的计算，开始从血缘关系源头开始，进行物理的转换操作。

  - filter(func)：筛选出满足函数func的元素，并返回一个新的数据集
  - map(func)：将每个元素传递到函数func中，并将结果返回为一个新的数据集
  - flatMap(func)：与map()相似，但每个输入元素都可以映射到0或多个输出结果
  - groupByKey()：应用于(K,V)键值对的数据集时，返回一个新的(K, Iterable)形式的数据集
  - reduceByKey(func)：应用于(K,V)键值对的数据集时，返回一个新的(K, V)形式的数据集，其中的每个值是将每个key传递到函数func中进行聚合
- 行动（Action）：在数据集上进行运算，返回计算值。

行动操作是真正触发计算的地方。Spark程序执行到行动操作时，才会执行真正的计算，从文件中加载数据，完成一次又一次转换操作，最终，完成行动操作得到结果。 下面列出一些常见的行动操作（Action API）：

- count() 返回数据集中的元素个数
- collect() 以数组的形式返回数据集中的所有元素
- first() 返回数据集中的第一个元素
- take(n) 以数组的形式返回数据集中的前n个元素
- reduce(func) 通过函数func（输入两个参数并返回一个值）聚合数据集中的元素
- foreach(func) 将数据集中的每个元素传递到函数func中运行

持久化

在Spark中，RDD采用惰性求值的机制，每次遇到行动操作，都会从头开始执行计算。如果整个Spark程序中只有一次行动操作，这当然不会有什么问题。但是，在一些情形下，我们需要多次调用不同的行动操作，这就意味着，每次调用行动操作，都会触发一次从头开始的计算。这对于迭代计算而言，代价是很大的，迭代计算经常需要多次重复使用同一组数据。

```
>>> list = ["Hadoop","Spark","Hive"]
>>> rdd = sc.parallelize(list)
>>> print(rdd.count()) //行动操作，触发一次真正从头到尾的计算
3
>>> print(','.join(rdd.collect())) //行动操作，触发一次真正从头到尾的计算
Hadoop,Spark,Hive
```

上述代码执行过程中，前后共触发了两次从头到尾的计算。

实际上，可以通过持久化（缓存）机制避免这种重复计算的开销。可以使用persist()方法对一个RDD标记为持久化，之所以说“标记为持久化”，是因为出现persist()语句的地方，并不会马上计算生成RDD并把它持久化，而是要等到遇到第一个行动操作触发真正计算以后，才会把计算结果进行持久化，持久化后的RDD将会被保留在计算节点的内存中被后面的行动操作重复使用。

persist()的圆括号中包含的是持久化级别参数，比如，persist(MEMORY_ONLY)表示将RDD作为反序列化的对象存储于JVM中，如果内存不足，就要按照LRU原则替换缓存中的内容。persist(MEMORY_AND_DISK)表示将RDD作为反序列化的对象存储在JVM中，如果内存不足，超出的分区将会被存放在硬盘上。一般而言，使用cache()方法时，会调用persist(MEMORY_ONLY)。

```
>>> list = ["Hadoop","Spark","Hive"]
>>> rdd = sc.parallelize(list)
>>> rdd.cache()  //会调用persist(MEMORY_ONLY)，但是，语句执行到这里，并不会缓存rdd，这是rdd还没有被计算生成
>>> print(rdd.count()) //第一次行动操作，触发一次真正从头到尾的计算，这时才会执行上面的rdd.cache()，把这个rdd放到缓存中
3
>>> print(','.join(rdd.collect())) //第二次行动操作，不需要触发从头到尾的计算，只需要重复使用上面缓存中的rdd
Hadoop,Spark,Hive
```

最后，可以使用unpersist()方法手动地把持久化的RDD从缓存中移除。

分区

RDD是弹性分布式数据集，通常RDD很大，会被分成很多个分区，分别保存在不同的节点上。RDD分区的一个分区原则是使得分区的个数尽量等于集群中的CPU核心（core）数目。

对于不同的Spark部署模式而言（本地模式、Standalone模式、YARN模式、Mesos模式），都可以通过设置spark.default.parallelism这个参数的值，来配置默认的分区数目，一般而言：

- 本地模式：默认为本地机器的CPU数目，若设置了local[N],则默认为N；
- Apache Mesos：默认的分区数为8；
- Standalone或YARN：在“集群中所有CPU核心数目总和”和“2”二者中取较大值作为默认值；

对于parallelize而言，如果没有在方法中指定分区数，则默认为spark.default.parallelism，比如：

```
>>> array = [1,2,3,4,5]
>>> rdd = sc.parallelize(array,2) #设置两个分区
```

对于textFile而言，如果没有在方法中指定分区数，则默认为min(defaultParallelism,2)，其中，defaultParallelism对应的就是spark.default.parallelism。 如果是从HDFS中读取文件，则分区数为文件分片数(比如，128MB/片)。

键值对RDD

键值对RDD的创建

1. 从文件中加载

```
>>>  lines = sc.textFile("file:///usr/local/spark/mycode/pairrdd/word.txt")
>>> pairRDD = lines.flatMap(lambda line : line.split(" ")).map(lambda word : (word,1))
>>> pairRDD.foreach(print)
(i,1)
(love,1)
(hadoop,1)
(i,1)
(love,1)
(Spark,1)
(Spark,1)
(is,1)
(fast,1)
(than,1)
(hadoop,1)
```

2. 通过并行集合（列表）创建RDD

```
>>> list = ["Hadoop","Spark","Hive","Spark"]
>>> rdd = sc.parallelize(list)
>>> pairRDD = rdd.map(lambda word : (word,1))
>>> pairRDD.foreach(print)
(Hadoop,1)
(Spark,1)
(Hive,1)
(Spark,1)
```

常用的键值对转换操作

- reduceByKey(func)

```
>>> pairRDD.reduceByKey(lambda a,b : a+b).foreach(print)
(Spark,2)
(Hive,1)
(Hadoop,1)
```

- groupByKey()

groupByKey()的功能是，对具有相同键的值进行分组。比如，对四个键值对(“spark”,1)、(“spark”,2)、(“hadoop”,3)和(“hadoop”,5)，采用groupByKey()后得到的结果是：(“spark”,(1,2))和(“hadoop”,(3,5))。

```
>>> pairRDD.groupByKey()
PythonRDD[11] at RDD at PythonRDD.scala:48
>>> pairRDD.groupByKey().foreach(print)
('spark', <pyspark.resultiterable.ResultIterable object at 0x7f1869f81f60>)
('hadoop', <pyspark.resultiterable.ResultIterable object at 0x7f1869f81f60>)
('hive', <pyspark.resultiterable.ResultIterable object at 0x7f1869f81f60>)
```

- keys()

keys()只会把键值对RDD中的key返回形成一个新的RDD。比如，对四个键值对(“spark”,1)、(“spark”,2)、(“hadoop”,3)和(“hadoop”,5)构成的RDD，采用keys()后得到的结果是一个RDD[Int]，内容是{“spark”,”spark”,”hadoop”,”hadoop”}。

```
>>> pairRDD.keys()
PythonRDD[20] at RDD at PythonRDD.scala:48
>>> pairRDD.keys().foreach(print)
Hadoop
Spark
Hive
Spark
```

- values()

与keys()类似，不过只使用value构成RDD

- sortByKey()：sortByKey()的功能是返回一个根据键排序的RDD。
- mapValues(func)

我们经常会遇到一种情形，我们只想对键值对RDD的value部分进行处理，而不是同时对key和value进行处理。对于这种情形，Spark提供了mapValues(func)，它的功能是，对键值对RDD中的每个value都应用一个函数，但是，key不会发生变化。比如，对四个键值对(“spark”,1)、(“spark”,2)、(“hadoop”,3)和(“hadoop”,5)构成的pairRDD，如果执行pairRDD.mapValues(lambda x : x+1)，就会得到一个新的键值对RDD，它包含下面四个键值对(“spark”,2)、(“spark”,3)、(“hadoop”,4)和(“hadoop”,6)。

```
>>> pairRDD.mapValues(lambda x : x+1)
PythonRDD[38] at RDD at PythonRDD.scala:48
>>> pairRDD.mapValues( lambda x : x+1).foreach(print)
(Hadoop,2)
(Spark,2)
(Hive,2)
(Spark,2)
```

- join

join(连接)操作是键值对常用的操作。“连接”(join)这个概念来自于关系数据库领域，因此，join的类型也和关系数据库中的join一样，包括内连接(join)、左外连接(leftOuterJoin)、右外连接(rightOuterJoin)等。最常用的情形是内连接，所以，join就表示内连接。

```
>>> pairRDD1 = sc.parallelize([('spark',1),('spark',2),('hadoop',3),('hadoop',5)])
>>> pairRDD2 = sc.parallelize([('spark','fast')])
>>> pairRDD1.join(pairRDD2)
PythonRDD[49] at RDD at PythonRDD.scala:48
>>> pairRDD1.join(pairRDD2).foreach(print)
```

共享变量

在默认情况下，当Spark在集群的多个不同节点的多个任务上并行运行一个函数时，它会把函数中涉及到的每个变量，在每个任务上都生成一个副本。但是，有时候，需要在多个任务之间共享变量，或者在任务（Task）和任务控制节点（Driver Program）之间共享变量。为了满足这种需求，Spark提供了两种类型的变量：广播变量（broadcast variables）和累加器（accumulators）。广播变量用来把变量在所有节点的内存之间进行共享。累加器则支持在所有不同节点之间进行累加计算（比如计数或者求和）。

广播变量

广播变量（broadcast variables）允许程序开发人员在每个机器上缓存一个只读的变量，而不是为机器上的每个任务都生成一个副本。通过这种方式，就可以非常高效地给每个节点（机器）提供一个大的输入数据集的副本。Spark的“actioin”操作会跨越多个阶段（stage），对于每个阶段内的所有任务所需要的公共数据，Spark都会自动进行广播。通过广播方式进行传播的变量，会经过序列化，然后在被任务使用时再进行反序列化。这就意味着，显式地创建广播变量只有在下面的情形中是有用的：当跨越多个阶段的那些任务需要相同的数据，或者当以反序列化方式对数据进行缓存是非常重要的。

可以通过调用SparkContext.broadcast(v)来从一个普通变量v中创建一个广播变量。这个广播变量就是对普通变量v的一个包装器，通过调用value方法就可以获得这个广播变量的值，具体代码如下:

```
>>> broadcastVar = sc.broadcast([1, 2, 3])
>>> broadcastVar.value
[1,2,3]
```

这个广播变量被创建以后，那么在集群中的任何函数中，都应该使用广播变量broadcastVar的值，而不是使用v的值，这样就不会把v重复分发到这些节点上。此外，一旦广播变量创建后，普通变量v的值就不能再发生修改，从而确保所有节点都获得这个广播变量的相同的值。

累加器

累加器是仅仅被相关操作累加的变量，通常可以被用来实现计数器（counter）和求和（sum）。Spark原生地支持数值型（numeric）的累加器，程序开发人员可以编写对新类型的支持。如果创建累加器时指定了名字，则可以在Spark UI界面看到，这有利于理解每个执行阶段的进程。

一个数值型的累加器，可以通过调用SparkContext.accumulator()来创建。运行在集群中的任务，就可以使用add方法来把数值累加到累加器上，但是，这些任务只能做累加操作，不能读取累加器的值，只有任务控制节点（Driver Program）可以使用value方法来读取累加器的值。 下面是一个代码实例，演示了使用累加器来对一个数组中的元素进行求和：

```
>>> accum = sc.accumulator(0)
>>> sc.parallelize([1, 2, 3, 4]).foreach(lambda x : accum.add(x))
>>> accum.value
10
```

文件数据读写

文件系统的数据读写

除了可以对本地文件系统进行读写以外，Spark还支持很多其他常见的文件格式（如文本文件、JSON、SequenceFile等）和文件系统（如HDFS、Amazon S3等）和数据库（如MySQL、HBase、Hive等）。数据库的读写我们将在Spark SQL部分介绍，因此，这里只介绍文件系统的读写和不同文件格式的读写。

本地文件系统的数据读写

```
>>> textFile = sc.textFile("file:///usr/local/spark/mycode/wordcount/word.txt")    # 读取本地文件
>>> textFile.first()    # 显示第一行
>>> textFile.saveAsTextFile("file:///usr/local/spark/mycode/wordcount/writeback.txt")
# 保存，保存的是个文件夹里面的part-0000和word.txt内容一样，读取这个文件夹的时候自动识别part-0000这个文件
```

分布式文件系统HDFS的数据读写

```
>>> val textFile = sc.textFile("hdfs://localhost:9000/user/hadoop/word.txt")
>>> textFile.first()
```

```
# 对于HDFS系统，以下三条命令完全一样
>>> val textFile = sc.textFile("hdfs://localhost:9000/user/hadoop/word.txt")
>>> val textFile = sc.textFile("/user/hadoop/word.txt")
>>> val textFile = sc.textFile("word.txt")
```

不同文件格式的读写

文本文件

实际上，我们在上面演示的都是文本文件的读写，因此，这里不再赘述，只是简单再总结一下。 把本地文件系统中的文本文件加载到RDD中的语句如下：

```
>>> rdd = sc.textFile("file:///usr/local/spark/mycode/wordcount/word.txt")
```

当我们给textFile()函数传递一个“包含完整路径的文件名”时，就会把这个文件加载到RDD中。如果我们给textFile()函数传递的不是文件名，而是一个目录，则该目录下的所有文件内容都会被读取到RDD中。

关于把RDD中的数据保存到文本文件，可以采用如下语句：

```
>>> rdd.saveAsTextFile("file:///usr/local/spark/mycode/wordcount/outputFile")
```

正像上面我们已经介绍的那样，我们在saveAsTextFile()函数的参数中给出的是目录，不是文件名，RDD中的数据会被保存到给定的目录下。

JSON

```
{"name":"Michael"}
{"name":"Andy", "age":30}
{"name":"Justin", "age":19}
```

```
>>> jsonStr =
sc.textFile("file:///usr/local/spark/examples/src/main/resources/people.json")
>>> jsonStr.foreach(print)
{"name":"Michael"}
{"name":"Andy", "age":30}
{"name":"Justin", "age":19}
```