第5章 RDD编程(RDD Programming) 讲课脚本

ㅁ설치시 문제:
$netstat -tlnp |grep 18080
$sudo kill -9 18080

슬라이드1: 标题슬라이드

大家好！今天我们要学习第5章 RDD编程，也就是RDD Programming。

RDD是什么呢？RDD是Resilient Distributed Dataset的缩写，是Spark最核心的数据结构。可以理解为弹性分布式数据集。

今天的课程中，我们将详细学习如何创建和使用RDD。

슬라이드2: 课程概要

首先，让我们看看今天要学习的内容。

第一个是**RDD编程基础(RDD Programming Basics)**。我们将学习RDD是什么、如何创建、如何操作等基础知识。

第二个是**键值对RDD(Key-Value Pair RDD)**。这是由键值对组成的特殊形式的RDD。

第三个是**数据读写(Data Read/Write)**，我们将学习如何从文件读取数据和保存数据的方法。

最后是**综合案例(Comprehensive Case Study)**，通过实际例子来综合运用前面学到的内容。

슬라이드3: RDD编程基础 详细目录

在RDD编程基础中，我们将具体学习这些内容：

5.1.1 RDD创建(RDD Creation) - 如何创建RDD

5.1.2 RDD操作(RDD Operations) - 可以对RDD进行哪些运算

5.1.3 持久化(Persistence) - 将RDD保存到内存或磁盘的方法

5.1.4 分区(Partitioning) - 将RDD分成多个片段的方法

5.1.5 一个综合实例(A Comprehensive Example) - 综合例子

让我们逐个学习。zhú gè xué xí

Bingxing

슬라이드4: RDD创建方法

现在如何创建RDD。

创建RDD的方法主要有两种：

第一种是从文件系统中加载数据创建RDD(Creating RDD by loading data from file system)。

从已有的文件中读取数据来创建RDD。

第二种是通过并行集合（列表）创建RDD(Creating RDD through parallel collections)。

将程序中的列表或数组等集合转换为RDD的方法

슬라이드5: 从文件系统创建RDD

我们先详细看第一种方法。

Spark采用textFile()方法来从文件系统中加载数据创建RDD(Spark uses the textFile() method to load data from the file system to create RDD)

这个textFile()方法的参数是URI，这个URI可以是多种类型：

- **本地文件系统的地址(Local file system address)** - 本地计算机的文件
- **分布式文件系统HDFS的地址(HDFS distributed file system address)** - Hadoop文件系统的文件
- **Amazon S3的地址(Amazon S3 address)** 等等 :S3: simple storage service

可以从非常多样的地方读取数据。

슬라이드6: 从本地文件创建RDD示例

让我们看一个实际例子。

```
>>> lines = sc.textFile("file:///usr/local/spark/mycode/rdd/word.txt")
>>> lines.foreach(print) 혹은 >>>print(lines.collect())
```

执行这个代码会得到：

Hadoop is good

Spark is fast

Spark is better

这样的结果。

$ 실행방법01: ~/spark_multimode.sh
$ 책실행방법02:cd /usr/local/hadoop
$ cd /usr/local/spark
$ ./bin/pyspark 엔터치고 기다림

정지는 quit()

可以看到，word.txt文件中的三行文本分别成为了RDD的元素。文件的一行成为RDD的一个元素(element)。

右边的图很好地展示了这个过程。textFile()方法读取文件内容并转换为RDD的过程。

슬라이드7: SparkContext对象

这里重要的是**sc** 这个对象，它就是**SparkContext**对象。

这是pyspark启动的时候自动创建的。

运行pyspark时会自动创建，可以直接使用。

但是**如果是编写独立应用程序(If you're writing a standalone application)**，也就是编写独立程序时，需要直接创建SparkContext：

```
from pyspark import SparkConf, SparkContext
conf = SparkConf().setMaster("local").setAppName("My App")
sc = SparkContext(conf = conf)
```

这样配置就可以了。

슬라이드8: 在PyCharm中运行示例 {여기에서부터 실습에 들어갑니다.}

如果想在PyCharm等IDE中进行调试，可以编写这样的完整代码：

```
# coding:utf8
from pyspark import SparkConf, SparkContext
if __name__ == '__main__':
    conf = SparkConf().setMaster("local[*]").setAppName("My App")
    sc = SparkContext(conf = conf)
    lines = sc.textFile("file:///usr/local/spark/mycode/rdd/word.txt")
    lines.foreach(print)
     sc.stop()
```

这里local[*]**表示使用本地所有可用的核心，最后用**sc.stop()来关闭SparkContext也很重要

슬라이드9: 从HDFS加载数据

现在我们来 看**从分布式文件系统HDFS中加载数据(Loading data from HDFS distributed file system)**的方法。

下面三种方法的结果是相同的：

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

三条语句是完全等价的(These three statements are completely equivalent)。如果HDFS设置为默认文件系统，只写文件名也可以。

```
$hdfs dfs -put /home/Hadoop/word.txt /user/Hadoop
$hdfs dfs -ls
```

```
from pyspark import SparkConf, SparkContext
if __name__=='__main__':
    conf=SparkConf().setMaster("local").setAppName("Page 92 HDFS")
    sc=SparkContext(conf=conf)
    sc.setLogLevel('WARN')
    lines=sc.textFile('/user/hadoop/word.txt')
    lines.foreach(print)
    sc.stop()
```

슬라이드10: 通过并行集合创建RDD

第二种方法是**通过并行集合（列表）创建RDD(Creating RDD through parallel collections)**

```
from pyspark import SparkConf, SparkContext
import logging
if __name__=='__main__':
    conf=SparkConf().setMaster("local").setAppName("Page 93 App")
    sc=SparkContext(conf=conf)
    sc.setLogLevel("WARN")
    array=range(1,6)
    rdd=sc.parallelize(array)
    rdd.foreach(print)
    sc.stop()结果：
```

1　　2　　3　　4　　5

可以调用SparkContext的parallelize方法(You can call SparkContext's parallelize method)，将普通的Python列表转换为RDD。

右边的图显示了数组[1,2,3,4,5]转换为RDD的过程。每个元素都成为RDD的元素。

먼저 하둡에 파일을 업로드(파일 복사후에)

$hdfs dfs -put /home/hadoop/word.txt /user/hadoop/

혹시 안된면 yarn 도 실행

- 중요한 것은 /user/hadoop은 HDFS내 파일

슬라이드11: RDD操作的种类

现在我们创建了RDD，来看看可以用它做什么。

RDD操作主要分为三种：

1. **转换操作(Transformation Operations)**
2. **行动操作(Action Operations)**
3. **惰性机制(Lazy Evaluation)**

让我们逐个详细说明。 **ràng wǒ men zhú gè xiáng xì shuō míng**

슬라이드12: 转换操作 (Transformation Operations)

我们先看**转换操作(Transformation Operations)**。

有个重要特征：

- 对**于RDD而言，每一次转换操作都**会**产生不同的RDD(For RDD, each transformation operation produces a different RDD)**
- **转换得到的RDD是惰性求值的(The transformed RDD is lazily evaluated)**

什么意思呢？就是说，每次进行转换操作时都会创建新的RDD，但实际上不会立即计算，只是记录进行了什么操作。**只有遇到行动操作时，才**会**发生**真**正的计算(Only when an action operation is encountered will actual computation occur)**。

右边的图表显示，从Input开始经过多次转换，最终在执行Action时进行所有计算。

示例说明：

教**授给学生一次性布置了三个商店的任务**：

- **“今天之内完成：**
  1. **超市** → 买1盒牛奶、6个鸡蛋
  2. **文具店** → 买2本笔记本、3支笔
  3. **水果店** → 买5个苹果、1串香蕉

这时，学**生只是把所有任务记在备忘录里**，并没有立即出门。

如果教**授突然改口说“超市只要买牛奶”**，学生只需要**修改备忘录里的超市任务**，而其他商店的任务仍然未执行。

最后，当学**生**真**正出门时（ = 需要结果时）**，才会**一次性完成所有跑腿任务**。

☑ **这种“不立即执行指令，而是延迟到必要时才一次性处理”的方式，就是惰性求值！**

슬라이드13: 主要转换操作API

让我们用表格整理一下常用的转换操作：

- **filter(func)**: 筛选满足条件的元素
- **map(func)**: 对每个元素应用函数进行转换
- **mapPartitions(func)**: 对每个分区应用函数
- **flatMap(func)**: 类似map但会将结果展平
- **groupBy(func)**: 按指定规则分组

- **groupByKey()**：在键值对中将相同键分组
- **reduceByKey(func)**：合并相同键的值

我们将通过实际例子来说明每一个。

슬라이드14: 更多转换操作

另外还有这些操作：

- **sortBy(func)**：排序
- **distinct**：去重
- **union**：合并两个RDD
- **intersection**：交集
- **subtract**：差集
- **zip**：将两个RDD合并为键值对

真的有很多种操作吧？让我们通过实际例子逐个学习。

슬라이드15: filter()操作示例

我们先看**filter(func)**操作。

```
from pyspark import SparkConf, SparkContext
if __name__=='__main__':
    conf=SparkConf().setMaster("local").setAppName("Page 94 Filter")
    sc=SparkContext(conf=conf)
    sc.setLogLevel("WARN")
    rdd=sc.textFile('file:///usr/local/spark/mycode/rdd/word.txt')
    rddWithSpark=rdd.filter(lambda rdd:"Spark" in rdd)
    rddWithSpark.foreach(print)
     sc.stop()
```

结果：

Spark is better

Spark is fast

筛选出满足函数func的元素(Filter out elements that satisfy function func)。

这里筛选出包含"Spark"单词的行。

右边的图表显示，原来有3行，但除了"Hadoop is good"，只剩下包含"Spark"的两行。

슬라이드16: map()操作示例1

```
现在看**map(func)**操作。
from pyspark import SparkConf, SparkContext
import logging

if __name__=='__main__':
    conf=SparkConf().setMaster("local").setAppName("Page 94 Map")
    sc=SparkContext(conf=conf)
    sc.setLogLevel("WARN")
    array=range(1,6)
    rdd=sc.parallelize(array)
    rddMap=rdd.map(lambda rdd:rdd+30)
    rddMap.foreach(print)
     sc.stop()
```

结果：

11

13

12

14

15

map(func)操作将**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**

给每个数字加30，创建了新的RDD。

슬라이드17: map()操作示例2

map的另一个例子：

```
from pyspark import SparkConf, SparkContext
if __name__=='__main__':
    conf=SparkConf().setMaster("local").setAppName("Page 94 Filter")
    sc=SparkContext(conf=conf)
    sc.setLogLevel("WARN")
    rdd=sc.textFile('file:///usr/local/spark/mycode/rdd/word.txt')
    words=rdd.map(lambda rdd:rdd.split(' '))
    words.foreach(print)
     sc.stop()
```

结果：

['Hadoop', 'is', 'good']

['Spark', 'is', 'fast']

['Spark', 'is', 'better']

这里将每行按空格分割成单词列表。字符串转换成了列表。

슬라이드18: 使用普通函数代替Lambda函数

在上面的例子中，我们传递给map()算子的是一个Lambda表达式
(In the example above, we passed a Lambda expression to the map() operator)。但也可以定义普通函数来使用，而不是Lambda：

```
from pyspark import SparkConf, SparkContext
if __name__=='__main__':
    conf=SparkConf().setMaster("local").setAppName("Page 93 App")
    sc=SparkContext(conf=conf)
    sc.setLogLevel("WARN")
    array=range(1,6)
     rdd=sc.parallelize(array)

    def add(data):
        return data+40

    rddWithMap=rdd.map(add)
    rddWithMap.foreach(print)
     sc.stop()
```

这样可以让代码更易读。

슬라이드19: mapPartitions()操作

mapPartitions(func)操作比较特殊。

map算子是对RDD中的每一个元素进行操作，mapPartitions算子则是对RDD中的每个分区的迭代器进行操作(The map operator operates on each element in the RDD, while the mapPartitions operator operates on the iterator of each partition in the RDD)。

例如，在有100个数据的分区中：

- 使用map时函数执行100次
- 使用mapPartitions时函数只执行1次

如果map执行的过程中还需要创建对象，比如创建Redis连接或者JDBC连接等，mapPartitions会更高效。每个分区只需要创建一次连接。

缺点是**如果一个分区有很多数据，一次函数处理可能会导致内存溢出**

슬라이드20: mapPartitions()实际示例

让我们看实际代码：

```
if __name__=='__main__':
    conf=SparkConf().setMaster("local").setAppName("Page 93 App")
    sc=SparkContext(conf=conf)
    sc.setLogLevel("WARN")
    array=range(1,10)
    rdd=sc.parallelize(array,3) #设置3个分区

    def myFunction(iter):
        result = list()
        print("======Processing Element:",iter)
        for it in iter:
            result.append(it*2)
        return result

    rdd2=rdd.mapPartitions(myFunction)
    print(rdd2.collect())
    sc.stop()
```

```
from pyspark import SparkConf, SparkContext

if __name__=='__main__':
    conf=SparkConf().setMaster("local").setAppName("Page 93 App")
    sc=SparkContext(conf=conf)
    sc.setLogLevel("WARN")
    array=range(1,10)
    rdd=sc.parallelize(array,3) #设置3个分区

    def myFunction2(x):
        result = list()
        print(f"======Processing Element:{x}")
        return x*2

    rdd2=rdd.map(myFunction2)
    print(rdd2.collect())
    sc.stop()
```

슬라이드21: mapPartitions()执行结果

执行结果：

==========

==========

[2, 4, 6, 8]

从程序执行结果可以看出，由于在程序中设置了两个分区，所以myFunction被调用了2次，而不是4次(From the program execution results, we can see that since two partitions were set in the program, myFunction was called 2 times, not 4 times)。

"=========="输出了两次，说明每个分区调用了一次函数。

슬라이드22: flatMap()操作

flatMap(func)操作非常有用。

```
#coding:utf8
from pyspark import SparkConf, SparkContext
if __name__=='__main__':
    conf=SparkConf().setMaster("local").setAppName("Page 94 Filter")
    sc=SparkContext(conf=conf)
    sc.setLogLevel("WARN")
    rdd=sc.textFile('file:///usr/local/spark/mycode/rdd/word.txt')
    words=rdd.flatMap(lambda word:word.split(" "))
    print(words.collect())
    sc.stop()
```

右边的图表很好地显示了过程：

1. **第1步:map()** - 首先将每行分割成单词数组
2. **第2步:拍扁(flat)** - 然后将所有数组合并成一个

结果是嵌套结构被展平，按单词成为单个元素。

슬라이드23: groupBy()操作

我们看**groupBy()**操作：

```
#coding:utf8
from pyspark import SparkConf, SparkContext
if __name__=='__main__':
    conf=SparkConf().setMaster("local").setAppName("Page 94 Filter")
    sc=SparkContext(conf=conf)
    sc.setLogLevel("WARN")
    mylist=['Hadoop','Spark','Storm','Flink','Flume']
    rdd=sc.parallelize(mylist)
    rddGroup=rdd.groupBy(lambda x:x[0])
    print(rddGroup.collect())
    # 각 그룹의 키와 값을 출력 (打印每个组的键和值)
    for (key, iterable) in rddGroup.collect():
        print(f"Key: {key}, Values: {list(iterable)}")
    sc.stop()
```

上面代码的含义是，根据单词的首字母进行分组，首字母相同的单词会被分到同一个组中

结果可以看到分成了'S', 'H', 'F'组。相同首字母的单词被分到一起。

슬라이드24: groupByKey()操作

groupByKey()操作用于键值对：

```
#coding:utf8
from pyspark import SparkConf, SparkContext
if __name__=='__main__':
    conf=SparkConf().setMaster("local").setAppName("Page 94 Filter")
    sc=SparkContext(conf=conf)
    sc.setLogLevel("WARN")
    mylist=[('Hadoop',1),('is',1),('good',1),('Spark',1),('is',1),('fast',1),
            ('Spark',1),('is',1),('better',1)]
    rdd=sc.parallelize(mylist)
    rddGroupByKey=rdd.groupByKey()
    print(rddGroupByKey.collect())
    # 각 그룹의 키와 값을 출력 (打印每个组的键和值)
    for (key, iterable) in rddGroupByKey.collect():
        print(f"Key: {key}, Values: {list(iterable)}")
    sc.stop()
```

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

相同键的值被分组到一起。

슬라이드25: groupByKey()图表

右边的图表很好地显示了groupByKey()操作的过程。

原来有("Hadoop",1), ("is",1), ("good",1)这样的单个键值对，groupByKey()操作后，相同键被分组为("is",(1,1,1)), ("Spark",(1,1))这样的形式。

"is"出现3次，"Spark"出现2次，所以对应的值被分组。

슬라이드26: reduceByKey()操作

```
from pyspark import SparkConf, SparkContext
if __name__=='__main__':
    conf=SparkConf().setMaster("local").setAppName("Page 94 Filter")
    sc=SparkContext(conf=conf)
    sc.setLogLevel("WARN")
    rdd = sc.textFile('file:///usr/local/spark/mycode/rdd/word.txt')
    rdd=rdd.flatMap(lambda x:x.split(" "))
    rddList=rdd.map(lambda x:(x,1))
    rddreduceByKey=rddList.reduceByKey(lambda a,b:a+b)
    rddreduceByKey.foreach(print)
    sc.stop()
```

结果：

('good', 1)

('Hadoop', 1)

('better', 1)

('Spark', 2)

('fast', 1)

('is', 3)

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中进行聚合后得到的结果

슬라이드27: reduceByKey()图表

reduceByKey()操作过程的图表更清楚。

相同键的值通过lambda a,b:a+b函数合并。"is"的情况是1+1+1=3，"Spark"的情况是1+1=2。

슬라이드28: reduceByKey()工作原理

让我们更详细地看reduceByKey()的工作原理。

rdd.reduceByKey(lambda a,b:a+b)

当有<"spark",<1,1,1>>这样的数据时，reduce函数如何工作在下面的图表中显示。

元素按1-2-3-4-5顺序进入，lambda a,b:a+b函数重复应用得到最终结果的过程。

슬라이드29: sortBy()操作

可以用**sortBy(func)**操作进行排序：

```
>>> rdd1 = sc.parallelize([3, 1, 2, 4],1)  # 设置为1个分区保证全局排序
>>> rdd2 = rdd1.sortBy(lambda x:x)
>>> rdd2.foreach(print)
```

升序结果：

1

2

3

4

要降序的话：

```
>>> rdd2 = rdd1.sortBy(lambda x:x,False)  # False参数表示降序
```

根据一定的规则对数据进行排序(Sort data according to certain rules)。这里设置为1个分区保证全局排序的结果。

슬라이드30: distinct操作

distinct操作去除重复：

>>> rdd1 = sc.parallelize(["Flink","Spark","Spark"])

>>> rdd2 = rdd1.distinct()

>>> rdd2.foreach(print)

结果：

Flink

Spark

distinct操作会对**RDD内部的元素进行去重，**并**把去重后的元素放到新的RDD中**

右边的图表显示，原来3个元素中重复的"Spark"被去除，只剩2个元素。

```
#coding:utf8
from pyspark import SparkConf, SparkContext
if __name__=='__main__':
    conf=SparkConf().setMaster("local").setAppName("Page 94 Distinct()")
    sc=SparkContext(conf=conf)
    sc.setLogLevel("WARN")
    rdd = sc.textFile('file:///usr/local/spark/mycode/rdd/word.txt')
    rdd=rdd.flatMap(lambda x:x.split(" "))
    rddDistinct=rdd.distinct()
    rddDistinct.foreach(print)
    sc.stop()
```

슬라이드31: union操作

union操作合并两个RDD：

>>> rdd1 = sc.parallelize([1,2,3])

>>> rdd2 = sc.parallelize([3,4,5])

>>> rdd3 = rdd1.union(rdd2)

>>> rdd3.collect()

[1, 2, 3, 3, 4, 5]

union操作会对两个RDD进行并集运算，并返回新的RDD，整个过程不会对元素进行去重(The union operation performs a union operation on two RDDs and returns a new RDD, and the entire process does not deduplicate elements)。

重要的是，不进行去重。可以看到3出现了两次。

```
#coding:utf8
from pyspark import SparkConf, SparkContext
if __name__=='__main__':
    conf=SparkConf().setMaster("local").setAppName("Page 94 union()")
    sc=SparkContext(conf=conf)
    sc.setLogLevel("WARN")
    rdd = sc.textFile('file:///usr/local/spark/mycode/rdd/word.txt')
    rdd=rdd.flatMap(lambda x:x.split(" "))
    rdd1=rdd.distinct()
    rdd2= sc.parallelize(['Name','is','Spark'])
    rdd3=rdd1.union(rdd2)
    rdd3.foreach(print)
    sc.stop()
```

슬라이드32: intersection操作

intersection操作求交集：

>>> rdd1 = sc.parallelize([1,2,3])

>>> rdd2 = sc.parallelize([3,4,5])

>>> rdd3 = rdd1.intersection(rdd2)

>>> rdd3.collect()

[3]

intersection操作会对两**个RDD进行交集**运**算(The intersection operation performs an intersection operation on two RDDs)**。只有两个RDD共同包含的元素才会包含在结果中。

这里只有3是共同的，所以结果是[3]。

```
#coding:utf8
from pyspark import SparkConf, SparkContext
if __name__=='__main__':
    conf=SparkConf().setMaster("local").setAppName("Page 94intersection()")
    sc=SparkContext(conf=conf)
    sc.setLogLevel("WARN")
    rdd = sc.textFile('file:///usr/local/spark/mycode/rdd/word.txt')
    rdd=rdd.flatMap(lambda x:x.split(" "))
    rdd1=rdd.distinct()
    rdd2= sc.parallelize(['Name','is','Spark'])
    rdd3=rdd1.intersection(rdd2)
    rdd3.foreach(print)
    sc.stop()
```

슬라이드33: subtract操作

subtract操作求差集：

>>> rdd1 = sc.parallelize([1,2,3])

>>> rdd2 = sc.parallelize([3,4,5])

>>> rdd3 = rdd1.subtract(rdd2)

>>> rdd3.collect()

[1, 2]

subtract操作会对两**个RDD进行差集**运**算(The subtract operation performs a difference operation on two RDDs)**。

最终返回的是在rdd1中存在但不在rdd2中的元素(The final result contains elements that exist in rdd1 but not in rdd2)。

rdd1中有但rdd2中没有的1和2成为结果。

```
#coding:utf8
from pyspark import SparkConf, SparkContext
if __name__=='__main__':
    conf=SparkConf().setMaster("local").setAppName("Page 94 subtract()")
    sc=SparkContext(conf=conf)
    sc.setLogLevel("WARN")
    rdd = sc.textFile('file:///usr/local/spark/mycode/rdd/word.txt')
    rdd=rdd.flatMap(lambda x:x.split(" "))
    rdd1=rdd.distinct()
    rdd2= sc.parallelize(['Name','is','Spark'])
    rdd3=rdd1.subtract(rdd2)
    rdd3.foreach(print)
    sc.stop()
```

슬라이드34: zip操作

zip操作将两个RDD合并为键值对：

>>> rdd1 = sc.parallelize([1,2,3])

>>> rdd2 = sc.parallelize(["Hadoop","Spark","Flink"])

>>> rdd3 = rdd1.zip(rdd2)

>>> rdd3.collect()

[(1, 'Hadoop'), (2, 'Spark'), (3, 'Flink')]

zip操作会把两个RDD中的元素以键值对的形式进行合并(The zip operation combines elements from two RDDs in key-value pair form)。

重要注意事项：**需要确保两个RDD中的元素个数是相同的(You need to ensure that the number of elements in both RDDs is the same)**。

```
#coding:utf8
from pyspark import SparkConf, SparkContext
if __name__=='__main__':
    conf=SparkConf().setMaster("local").setAppName("Page 94 zip()")
    sc=SparkContext(conf=conf)
    sc.setLogLevel("WARN")
    rdd = sc.textFile('file:///usr/local/spark/mycode/rdd/word.txt')
    rdd=rdd.flatMap(lambda x:x.split(" "))
    rdd1=rdd.distinct()
    rdd2= sc.parallelize(range(rdd1.count()))
    rdd3=rdd1.zip(rdd2)
    rdd3.foreach(print)
    sc.stop()
```

슬라이드35: Action操作

现在我们来学习**行动操作(Action Operations)**。

行动操作是真正触发计算的地方(Action operations are where computation is actually triggered)。到目前为止学的转换操作实际上不进行计算，只记录要进行什么操作，遇到Action操作时才开始真正的计算。

主要Action操作的表格整理：

- **count()**: 计算元素个数
- **collect()**: 将所有元素作为数组获取
- **first()**: 获取第一个元素
- **take(n)**: 获取前n个元素
- **reduce(func)**: 将所有元素合并为一个
- **foreach(func)**: 对每个元素应用函数
- **countByKey()**: 按键计算个数
- **aggregate()**: 复杂的聚合操作

슬라이드36: Action操作实际示例

通过实际例子看Action操作：

```
>>> rdd = sc.parallelize([1,2,3,4,5])
>>> rdd.count()
5
>>> rdd.first()
1
>>> rdd.take(3)
[1, 2, 3]
>>> rdd.reduce(lambda a,b:a+b)
15
>>> rdd.collect()
[1, 2, 3, 4, 5]
>>> rdd.foreach(lambda elem:print(elem))
1
2
3
4
5
```

可以直接确认每个操作产生什么结果。

```
#coding:utf8
from pyspark import SparkConf, SparkContext
if __name__=='__main__':
    conf=SparkConf().setMaster("local").setAppName("Page 94 Filter")
    sc=SparkContext(conf=conf)
    sc.setLogLevel("WARN")
    rdd = sc.textFile('file:///usr/local/spark/mycode/rdd/word.txt')
```

```
rdd=rdd.flatMap(lambda x:x.split(" "))
print(f'action:first()->{rdd.first()}')
print(f'action:take(n)->{rdd.take(3)}')
print(f'action:countByKey()->{rdd.countByKey()}')
rdd.foreach(print)
sc.stop()
```

슬라이드37: countByKey()和aggregate()示例

countByKey()操作示例：

>>> rdd = sc.parallelize([("hadoop",1),("spark",1),("spark",1)])

>>> result = rdd.countByKey()

>>> print(result)

defaultdict(<class 'int'>, {'hadoop': 1, 'spark': 2})

按键计算有多少个。

aggregate()操作示例：

>>> rdd1 = sc.parallelize([1, 2, 3, 4], 2)

>>> result = rdd1.aggregate(0,lambda a,b:a+b,lambda a,b:a+b)

>>> print(result)

10

aggregate是用于复杂聚合操作的函数。

```
from pyspark import SparkConf, SparkContext

# Spark配置（本地模式）
conf = SparkConf().setAppName("简单Aggregate示例").setMaster("local")
sc = SparkContext(conf=conf)

# 创建简单的数字列表
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
numbers_rdd = sc.parallelize(numbers, 3)  # 分成3个分区

# 使用aggregate()计算总和与数量
# 初始值：（总和，数量）= (0, 0)
initial_value = (0, 0)

# 在每个分区中执行的函数（seqOp）
def seq_op(accumulator, number):
```

```
    (current_sum, current_count) = accumulator
    return (current_sum + number, current_count + 1)

# 合并分区结果的函数（combOp）
def comb_op(accumulator1, accumulator2):
    (sum1, count1) = accumulator1
    (sum2, count2) = accumulator2
    return (sum1 + sum2, count1 + count2)

# 执行aggregate
total_sum, total_count = numbers_rdd.aggregate(initial_value, seq_op, comb_op)

print("="*50)
print(f"数字列表: {numbers}")
print(f"总和: {total_sum}")
print(f"数量: {total_count}")
print(f"平均值: {total_sum/total_count:.2f}")
print("="*50)

sc.stop()
```

슬라이드38: 惰性求值(Lazy Evaluation)

我们来学习**惰性机制(Lazy Evaluation)**。

所谓的"惰性机制"是指，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，才会触发"从头到尾"的真正的计算(The so-called "lazy mechanism" means that the entire transformation process only records the transformation trajectory and does not perform actual computation. Only when an action operation is encountered will actual end-to-end computation be triggered)。

示例：

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

>>> lineLengths = lines.map(lambda s:len(s))

>>> totalLength = lineLengths.reduce(lambda a,b:a+b)

>>> print(totalLength)

这里前两行实际上不进行计算，reduce() Action执行时才进行所有计算。

```
#coding:utf8
from pyspark import SparkConf, SparkContext
if __name__=='__main__':
    conf=SparkConf().setMaster("local").setAppName("Page 94 Filter")
    sc=SparkContext(conf=conf)
    sc.setLogLevel("WARN")
    rdd = sc.textFile('file:///usr/local/spark/mycode/rdd/word.txt')
    lineLengths=rdd.map(lambda x:len(x))
    lineLengths.foreach(print)
    totalLength=lineLengths.reduce(lambda   a,b:a+b)
    print(totalLength)
    sc.stop()
```

슬라이드39: 持久化(Persistence) - 问题

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

这可能成为问题。例如：

```
from pyspark import SparkConf, SparkContext

if __name__=='__main__':
    conf=SparkConf().setMaster("local").setAppName("Page 93 App")
    sc=SparkContext(conf=conf)
    sc.setLogLevel("WARN")
    list=["Hadoop","Spark","Hive"]
    rdd=sc.parallelize(list)
    print(f"action 1: {rdd.count()}") # 第一个Action - 从头到尾计算
    print(f"action 2: {','.join(rdd.collect())}")  # 第二个Action - 又从头到尾计算

    sc.stop()
```

对同一个RDD执行多次Action时，每次都要从头重新计算，效率低下。

슬라이드40: 持久化解决方法

可以通过持久化（缓存）机制避免这种重复计算的开销(This repeated computation overhead can be avoided through persistence (caching) mechanisms)。

重要要点：

- **可以使用persist()方法对一个RDD标记为持久化(You can use the persist() method to mark an RDD for persistence)**
- **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(The reason we say "mark for persistence" is that where the persist() statement appears, the RDD will not be immediately computed and persisted)**
- **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(The persisted RDD will be retained in the memory of compute nodes for reuse by subsequent action operations)**

슬라이드41: 持久化级别

persist()的圆括号中包含的是持久化级别参数(The parentheses of persist() contain persistence level parameters)：

- **persist(MEMORY_ONLY)**：将RDD保存在JVM内存中，内存不足时按LRU替换
- **persist(MEMORY_AND_DISK)**：保存在内存中，内存不足时保存到磁盘
- **cache()**：一般等同于persist(MEMORY_ONLY)
- **unpersist()**：手动移除持久化

슬라이드42: 持久化使用示例

看实际例子：

```
from pyspark import SparkConf, SparkContext

if __name__ == '__main__':
    conf = SparkConf().setMaster("local").setAppName("Page 93 App")
    sc = SparkContext(conf=conf)
    sc.setLogLevel("WARN")
    list = ["Hadoop", "Spark", "Hive"]
    rdd = sc.parallelize(list)

    rdd.cache()  # 持久化标记 - 还没有计算

    print(f"action 1: {rdd.count()}")  # 第一个Action - 从头到尾计算
    print(f"action 2: {','.join(rdd.collect())}")  # 第二个Action - 又从头到尾计算

    rdd.unpersist()
    sc.stop() 现在第二个Action不再从头重新计算，而是使用缓存的结果。
```

슬라이드43: 分区(Partitioning) - 概念

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 is a resilient distributed dataset. Usually RDD is very large and will be divided into many partitions, which are stored on different nodes)。

分区的作用(Functions of Partitioning)：

1) 增加并行度(Increase parallelism) - 可以在多个节点同时处理。

右边的图表显示rdd1被分成多个分区(p1, p2, p3, p4, p5)，分别保存在不同的WorkerNode上。

슬라이드44: 分区的优点 - 减少通信开销

2) 减少通信开销(Reduce communication overhead)

例如考虑UserData和Events表进行连接时。

不分区的话，所有数据都要通过网络传输，通信成本很高。

但如果将具有相同键的数据放在同一个节点上，大部分连接操作可以在本地执行，大大减少网络通信。

슬라이드45: 正确分区的效果

采用分区以后对UserData和Events两个表进行连接操作(Join operation on UserData and Events tables after using partitioning)可以看到：

具有相同UserID的数据在同一个分区中，大部分连接在本地进行，只有少部分需要网络通信。

这样可以大大提高性能。

슬라이드46: RDD分区原则

RDD分区的一个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(For different Spark deployment modes)：

- **本地模式(Local mode)**: 默认为本地机器的CPU数，设置local[N]时为N个
- **Apache Mesos**: 默认分区数为8
- **Standalone或YARN**: 集群所有CPU核心数和2中的较大值

这样设置的原因是让每个CPU核心能够处理一个分区，最大化并行处理效率。

슬라이드47: 设置分区个数 - 手动指定

创建RDD时手动指定分区个数(Manually specify the number of partitions when creating RDD)：

sc.textFile(path, partitionNum)

这里path是文件路径，partitionNum是分区个数。

示例：

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

这样数据就会分成2个分区保存。

슬라이드48: 设置分区个数 - 重新分区

通过转换操作得到新RDD时，直接调用repartition方法即可(When getting a new RDD through transformation operations, directly call the repartition method)：

```
>>> data = sc.parallelize([1,2,3,4,5],2)
>>> len(data.glom().collect())  # 确认当前分区数
2
>>> rdd = data.repartition(1)  # 重新分区为1个
>>> len(rdd.glom().collect())  # 重新分区后的分区数
1
```

glom()方法将每个分区的数据制作成数组的函数。用它可以确认分区个数。

슬라이드49: 自定义分区

Spark提供了自带的HashPartitioner（哈希分区）与RangePartitioner（区域分区）(Spark provides built-in HashPartitioner and RangePartitioner)，大多数情况下这些就足够了。

但是**Spark也支持自定义分区方式(Spark also supports custom partitioning methods)**，也就是利用领域知识创建更高效的分区方法。

슬라이드50: 自定义分区示例

看实际例子：

根据key值的最后一位数字，写到不同的文件(Write to different files based on the last digit of the key value)：

- 10 → part-00000
- 11 → part-00001
- ...
- 19 → part-00009

这样根据键的最后一位数字保存到不同文件。

슬라이드51: 自定义分区代码

实际实现代码：

```
from pyspark import SparkConf, SparkContext


def MyPartitioner(key):
    print("MyPartitioner is running")
    print('The key is %d' % key)
    return key%10


def main():
    print("The main function is running")
    conf = SparkConf().setMaster("local").setAppName("MyApp")
    sc = SparkContext(conf = conf)
    data = sc.parallelize(range(10),5)
    data.map(lambda x:(x,1)) ₩
```

```
        .partitionBy(10,MyPartitioner) ₩
        .map(lambda x:x[0]) ₩
        .saveAsTextFile("file:///usr/local/spark/mycode/rdd/partitioner")

if __name__ == '__main__':
    main()
```

MyPartitioner函数是自定义分区函数。用键除以10的余数来决定分区。

```
#coding:utf8
from pyspark import SparkConf, SparkContext
import os
import shutil

def MyPartitioner(key):
    print('MyPartitioner is running')
    print('the key is %d' %key)
    return key%10
def main():
    output_path = '/usr/local/spark/mycode/rdd/partitioner'
    if os.path.exists(output_path):
        shutil.rmtree(output_path)

    print('The main function is running')
    conf=SparkConf().setMaster("local").setAppName("p108. 5.1.4")
    sc=SparkContext(conf=conf)
    sc.setLogLevel("WARN")
    rdd=sc.parallelize(range(10),5)
    rdd.map(lambda x:(x,1)).partitionBy(10,MyPartitioner).map(lambda x:x[0])₩
    .saveAsTextFile('file:///usr/local/spark/mycode/rdd/partitioner')

if __name__ == '__main__':
    main()
```

파티션 0: [(0,1), (1,1)]

파티션 1: [(2,1), (3,1)]

파티션 2: [(4,1), (5,1)]

파티션 3: [(6,1), (7,1)]

파티션 4: [(8,1), (9,1)]

파티션 0에서 실행:

MyPartitioner(0) # 0 % 10 = 0

MyPartitioner(1) # 1 % 10 = 1

파티션 1에서 실행:

MyPartitioner(2) # 2 % 10 = 2

MyPartitioner(3) # 3 % 10 = 3

파티션 2에서 실행:

MyPartitioner(4) # 4 % 10 = 4

MyPartitioner(5) # 5 % 10 = 5

파티션 3에서 실행:

MyPartitioner(6) # 6 % 10 = 6

MyPartitioner(7) # 7 % 10 = 7

파티션 4에서 실행:

MyPartitioner(8) # 8 % 10 = 8

MyPartitioner(9) # 9 % 10 = 9

파티션 0: [(0,1)]

파티션 1: [(1,1)]

파티션 2: [(2,1)]

파티션 3: [(3,1)]

파티션 4: [(4,1)]

파티션 5: [(5,1)]

파티션 6: [(6,1)]

파티션 7: [(7,1)]

파티션 8: [(8,1)]

파티션 9: [(9,1)]

슬라이드52: 自定义分区执行

运行程序会得到：

The main function is running

MyPartitioner is running

The key is 0

MyPartitioner is running

The key is 1

...

MyPartitioner is running

The key is 9

可以看到对每个键都调用了MyPartitioner函数。

슬라이드53: 综合示例 - 单词频数统计

现在让我们综合运用到目前为止学的内容，做**词频统计(Word Frequency Count)**示例。

```
>>> lines = sc.textFile("file:///usr/local/spark/mycode/rdd/word.txt")
>>> wordCount = lines.flatMap(lambda line:line.split(" ")) ₩
...                         .map(lambda word:(word,1)) ₩
...                         .reduceByKey(lambda a,b:a+b)
>>> print(wordCount.collect())
[('good', 1), ('Spark', 2), ('is', 3), ('better', 1), ('Hadoop', 1), ('fast', 1)]
```

这个代码是计算文本文件中每个单词出现多少次的程序。

슬라이드54: 单词频数统计过程

右边的图表很好地显示了整个过程：

1. **sc.textFile()**: 从文件按行读取
2. **flatMap()**: 将每行分割成单词并展平
3. **map()**: 将每个单词转换为(单词, 1)对
4. **reduceByKey()**: 计算相同单词的频数合计

最终计算出每个单词的频数。

```
#coding:utf8
from pyspark import SparkConf, SparkContext
if __name__=='__main__':
    conf=SparkConf().setMaster("local").setAppName("P109-5.1.5")
    sc=SparkContext(conf=conf)
    sc.setLogLevel('WARN')
    lines=sc.textFile('file:///usr/local/spark/mycode/rdd/word.txt')
    wordFlatMap=lines.flatMap(lambda line:line.split(" "))
    wordMap=wordFlatMap.map(lambda word:(word,1))
    wordReduceByKey=wordMap.reduceByKey(lambda a,b:a+b)
    result=wordReduceByKey.collect()
    print(result)
    sc.stop()
```

슬라이드55: 实际集群环境

在实际应用中，单词文件可能非常大，会被保存到分布式文件系统HDFS中，Spark和Hadoop会统一部署在一个集群上

在实际运营环境中，会像这样在多个节点上进行分布式处理。Spark Master协调作业，各个Worker节点进行实际处理。

슬라이드56: 集群中的执行过程

最后的图表显示了在实际集群中如何进行单词频数统计。各个节点处理本地数据生成部分结果，最终在Spark Master中合并所有结果计算总的单词频数。

例如：

- 节点1: a:1, b:3, c:1
- 节点2: a:1, e:2, c:1, d:1
- 节点3: a:2, b:2, c:1

这些部分结果合并得到最终结果a:4, b:5, c:3, d:1, e:2。

总结

让我们总结一下今天学到的内容：

1. **RDD创建**: 用textFile()从文件，用parallelize()从集合
2. **转换操作**: filter, map, flatMap, groupBy, reduceByKey等等
3. **行动操作**: count, collect, reduce等等
4. **惰性求值**: 直到调用行动时才进行实际计算的机制
5. **持久化**: 用cache()或persist()保存中间结果进行重用
6. **分区**: 将数据分成多个片段提高并行处理效率

好好理解和运用这些概念，就能使用Spark高效处理大容量数据。

下次课我们将更详细地学习键值对RDD和数据读取/写入。

有什么问题吗？