Tuesday, January 27, 2015

Scala on Spark cheatsheet

This is a cookbook for scala programming.

1. Define a object with main function -- Helloworld.

```
object HelloWorld {
  def main(args: Array[String]) {
    println("Hello, world!")
  }
}
```

Execute main function:

```
scala> HelloWorld.main(null)
Hello, world!
```

2. Creating RDDs

- **Parallelized Collections:**

```
val data = Array(1, 2, 3, 4, 5)
val distData = sc.parallelize(data)
```

- **External Datasets:**

```
val distFile = sc.textFile("data.txt")
```

Above command returns the content of the file:

```
scala> distFile.collect()
res16: Array[String] = Array(1,2,3, 4,5,6)
```

SparkContext.wholeTextFiles can return (filename, content).

```
val distFile = sc.wholeTextFiles("/tmp/tmpdir")

scala> distFile.collect()
res17: Array[(String, String)] =
Array((maprfs:/tmp/tmpdir/data3.txt,"1,2,3
4,5,6
"), (maprfs:/tmp/tmpdir/data.txt,"1,2,3
4,5,6
"), (maprfs:/tmp/tmpdir/data2.txt,"1,2,3
4,5,6
"))
```

3. RDD Operations

- **Transformations (which create a new dataset from an existing one)Lazy!**

3.1 map(f:T-U)
Return a new distributed dataset formed by passing each element of the source through a function func.
Example 1: To calculate the length of each line.

```
scala> lines.map(s => s.length).collect
res46: Array[Int] = Array(48, 25, 34, 5, 6, 6, 5, 5, 6)
```

3.2 filter(f:T->Bool)
Return a new dataset formed by selecting those elements of the source on which func returns true.
Example 1: Find the lines which starts with "APPLE":

```
scala> lines.filter(_.startsWith("APPLE")).collect
res50: Array[String] = Array(APPLE)
```

Example 2: Find the lines which contains "test":

```
scala> lines.filter(_.contains("test")).collect
res54: Array[String] = Array("This is a test data text file for Spark to use. ", "To test Scala and Spark, ")
```

3.3 flatMap(func)

Similar to map, but each input item can be mapped to 0 or more output items (so func should return a Seq rather than a single item).
Example 1: Generate the Int List and compare "map" and "flatMap".

```
scala> val intlist = List( 1,2,3,4,5 )
intlist: List[Int] = List(1, 2, 3, 4, 5)

scala> intlist.map(x=>List(x,x*2))
res72: List[List[Int]] = List(List(1, 2), List(2, 4), List(3, 6), List(4, 8), List(5, 10))

scala> intlist.flatMap(x=>(List(x,x*2)))
res73: List[Int] = List(1, 2, 2, 4, 3, 6, 4, 8, 5, 10)
```

Example 2: Use flatMap for map

```
scala> val m = Map(1 -> 2, 2 -> 4, 3 -> 6)
m: scala.collection.immutable.Map[Int,Int] = Map(1 -> 2, 2 -> 4, 3 -> 6)

scala> def h(k:Int, v:Int) = if (v > 2) Some(k->v) else None
h: (k: Int, v: Int)Option[(Int, Int)]

scala> m.flatMap { case (k,v) => h(k,v) }
res76: scala.collection.immutable.Map[Int,Int] = Map(2 -> 4, 3 -> 6)
```

3.4 mapPartitions(func)

Similar to map, but runs separately on each partition (block) of the RDD, so func must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.
To be simple:
map converts each element of the source RDD into a single element of the result RDD by applying a function.
mapPartitions converts each partition of the source RDD into into multiple elements of the result (possibly none).
It can improve performance by reducing new object creation in the map function.
Example 1: We have totally 9 lines here, instead of map each line, we can firstly split all 9 lines into 2 partitions, and then we only need to map twice.
Firstly we need to create a map function which accepts Iterator as inputs and also as returning value.
This function simply return the size of each partition.

```
def myfunc(inputs: Iterator[String]) : Iterator[Int] = {
  var results = List[Int]()
  results .::= (inputs.size)
  results.iterator
}
```

Then:

```
scala> lines.count
res12: Long = 9

scala> lines.mapPartitions(myfunc).collect
res9: Array[Int] = Array(2, 7)
```

3.5 mapPartitionsWithIndex(func)

Similar to mapPartitions, but also provides func with an integer value representing the index of the partition, so func must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T.
Example 1: Take above mapPartitions example, and here we have one more "index" as input value for map function.

```
def myfunc2(index: Int, inputs: Iterator[String]) : Iterator[(Int, Int)] = {
  var results = List[(Int,Int)]()
  results .::= (index, inputs.size)
  results.iterator
}
```

Then:

```
scala> lines.mapPartitionsWithIndex(myfunc2).collect
res14: Array[(Int, Int)] = Array((0,2), (1,7))
```

3.6 sample(withReplacement, fraction, seed)

Sample a fraction of the data, with or without replacement, using a given random number generator seed.
Note: Comparing to takeSample, the 2nd parameter of sample() is how much percentage of the total number should be sampled. However the actual number sampled may not be exactly the same.
Example 1: Fraction = 0.5 may not be exactly 50% of total numbers. It may change.

```
scala> val list = sc.parallelize(1 to 9)
list: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[55] at parallelize at <console>:33

scala> list.sample(true, 0.5) .collect
res63: Array[Int] = Array(7, 9)

scala> list.sample(true, 0.5) .collect
res64: Array[Int] = Array(1, 1, 2, 4, 5, 6)
```

Example 2: "withReplacement"=true means output may have duplicate elements, else, it will not.

```
scala> list.sample(true, 1).collect
res70: Array[Int] = Array(4, 4, 4, 4, 8, 8, 9, 9)

scala> list.sample(false, 1).collect
res71: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Example 3: If "seed" does not change, the result will not change.

```
scala> list.sample(true, 0.5, 1).collect
res73: Array[Int] = Array(5, 7, 8, 9, 9)

scala> list.sample(true, 0.5, 1).collect
res74: Array[Int] = Array(5, 7, 8, 9, 9)
```

3.7 union(otherDataset)
Return a new dataset that contains the union of the elements in the source dataset and the argument.
Note: It is the same as operator "++".
Example 1: Union 2 array of Strings.
Unlike "Union" in SQL, here it will not de-duplicate.

```
scala> val list  = sc.parallelize(List("apple", "orange", "banana", "apple", "orange"))
scala> val list2 = sc.parallelize(List("mapr", "cloudera", "hortonworks"))

scala> (list union list2).collect
res4: Array[String] = Array(apple, orange, banana, apple, orange, mapr, cloudera, hortonworks)

scala> (list ++ list2).collect
res5: Array[String] = Array(apple, orange, banana, apple, orange, mapr, cloudera, hortonworks)
```

3.8 intersection(otherDataset)
Return a new RDD that contains the intersection of elements in the source dataset and the argument.
Note: It will do de-duplicate here.
Example 1: Intersection 2 array of Strings. However the result only contains one "apple" although each of the arrays has more than one.

```
scala> val list  = sc.parallelize(List("apple", "orange", "banana", "apple", "orange"))
scala> val list2 = sc.parallelize(List("apple", "mapr", "apple" ))

list.intersection(list2).collect
res7: Array[String] = Array(apple)
```

3.9 distinct([numTasks]))
Return a new dataset that contains the distinct elements of the source dataset.
Example 1: Return distinct values from one array.

```
scala> val list  = sc.parallelize(List("apple", "orange", "banana", "apple", "orange"))
scala> list.distinct.collect
res13: Array[String] = Array(orange, apple, banana)
```

3.10 groupByKey([numTasks])
When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs.
Note: If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using reduceByKey or combineByKey will yield much better performance.
Note: By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional numTasks argument to set a different number of tasks.
Example 1: Group by a list of (K,V) pairs.

```
scala> val kv = sc.parallelize( List(("apple", 1), ("orange", 2), ("banana", 3), ("apple", 2)) )

scala> kv.groupByKey.collect
res14: Array[(String, Iterable[Int])] = Array((orange,ArrayBuffer(2)), (apple,ArrayBuffer(1, 2)), (banana,ArrayBuffer(3)))
```

3.11 reduceByKey(func, [numTasks])
When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function func, which must be of type (V,V) => V. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.
Example 1: Take above example to calculate the total sum of value for each key.

```
scala> kv.reduceByKey(_ + _ ).collect
res19: Array[(String, Int)] = Array((orange,2), (apple,3), (banana,3))
```

3.12 aggregateByKey(zeroValue)(seqOp, combOp, [numTasks])
When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.
Example 1:Take above example to debug the functions.

```
scala> kv.aggregateByKey("")(  ( (a,b) => "DEBUG:" + "a=" + a + " b=" + b ), ((v1, v2) => v1 + " and " + v2) ).collect
res34: Array[(String, String)] = Array((orange,DEBUG:a= b=2), (apple,DEBUG:a= b=1 and DEBUG:a= b=2), (banana,DEBUG:a= b=3))
```

3.13 sortByKey([ascending], [numTasks])
When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument.
Example 1:Take above example to sort by key asc or desc.

```
scala> kv.sortByKey(true).collect
res35: Array[(String, Int)] = Array((apple,2), (apple,1), (banana,3), (orange,2))
```

```
scala> kv.sortByKey(false).collect
res36: Array[(String, Int)] = Array((orange,2), (banana,3), (apple,2), (apple,1))
```

3.14 join(otherDataset, [numTasks])
When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are also supported through leftOuterJoin and rightOuterJoin.
Example 1: Inner Join

```
scala> val kv = sc.parallelize( List(("apple", 1), ("orange", 2), ("banana", 3), ("apple", 2)) )
scala> val kw = sc.parallelize( List(("apple", 999), ("orange", 222) ))

scala> kv.join(kw).collect
res37: Array[(String, (Int, Int))] = Array((orange,(2,222)), (apple,(1,999)), (apple,(2,999)))
```

Example 2: Left Outer Join

```
scala> kv.leftOuterJoin(kw).collect
res38: Array[(String, (Int, Option[Int]))] = Array((orange,(2,Some(222))), (apple,(2,Some(999))), (apple,(1,Some(999))), (banana,(3,None)))
```

Example 3: Right Outer Join

```
scala> kv.rightOuterJoin(kw).collect
res39: Array[(String, (Option[Int], Int))] = Array((orange,(Some(2),222)), (apple,(Some(1),999)), (apple,(Some(2),999)))
```

3.15 cogroup(otherDataset, [numTasks])
When called on datasets of type (K, V) and (K, W), returns a dataset of (K, Iterable<V>, Iterable<W>) tuples. This operation is also called groupWith.
Example 1: 2 RDDs GroupWith.

```
scala> kv.cogroup(kw).collect
res40: Array[(String, (Iterable[Int], Iterable[Int]))] = Array((orange,(CompactBuffer(2),CompactBuffer(222))), (apple,(CompactBuffer(2, 1),Comp
```

Example 1: 3 RDDs GroupWith.

```
scala> val kw2 = sc.parallelize( List(("banana", 123), ("banana", 456) ))

scala> kv.cogroup(kw,kw2).collect
res41: Array[(String, (Iterable[Int], Iterable[Int], Iterable[Int]))] = Array((orange,(CompactBuffer(2),CompactBuffer(222),CompactBuffer())), (
```

3.16 cartesian(otherDataset)
When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements). Example 1: 2 RDDs cartesian sets.

```
scala> val a = sc.parallelize(List(1,2,3))
scala> val b = sc.parallelize(List(4,5,6))
scala> a.cartesian(b).collect
res42: Array[(Int, Int)] = Array((1,4), (1,5), (1,6), (2,4), (3,4), (2,5), (2,6), (3,5), (3,6))
```

3.17 pipe(command, [envVars])
Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script. RDD elements are written to the process's stdin and lines output to its stdout are returned as an RDD of strings.
Example 1: Split a List to 2 partitions, and the command will be executed from each partition.

```
scala> val list  = sc.parallelize(List("apple", "orange", "banana", "mapr", "cloudera" , "hortonworks") , 2)

scala> list.pipe("tail -1").collect
res34: Array[String] = Array(banana, hortonworks)

scala> list.pipe("tail -2").collect
res35: Array[String] = Array(orange, banana, cloudera, hortonworks)
```

3.18 coalesce(numPartitions)
Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset.
Example 1: Reduce above "list" from 2 partitions to 1 partition. Comparing the differences.

```
scala> list.coalesce(1, false).pipe("tail -1").collect
res37: Array[String] = Array(hortonworks)

scala> list.coalesce(1, false).pipe("tail -2").collect
res38: Array[String] = Array(cloudera, hortonworks)
```

3.19 repartition(numPartitions)
Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.
Example 1: Increase above "list" from 2 partitions to 6 partitions. Comparing the differences.

```
scala> list.repartition(6).pipe("tail -1").collect
res39: Array[String] = Array(hortonworks, apple, orange, banana, mapr, cloudera)
```

- **Actions (which return a value to the driver program after running a computation on the dataset)**

3.20 reduce(f: (T, T) => T): T
Aggregate the elements of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
Example 1: Calculate the sum of int from 1 to 9.

```
scala> val a = sc.parallelize(1 to 9)
a: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[3] at parallelize at <console>:33

scala> a.reduce(_ + _)
res5: Int = 45
```

3.21 collect()

Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
Example 1: Collect all elements of List and return an Array.

```
scala> val list = sc.parallelize(List("apple", "orange", "banana"))
list: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[4] at parallelize at <console>:33

scala> list.collect
res6: Array[String] = Array(apple, orange, banana)
```

3.22 count()

Return the number of elements in the dataset.
Example 1: Return number of above list.

```
scala> list.count
res7: Long = 3
```

3.23 first()

Return the first element of the dataset (similar to take(1)).
Example 1: Return first element of above list. Similar as "limit 1" in SQL.

```
scala> list.first
res8: String = apple
```

3.24 take(n)

Return an array with the first n elements of the dataset. Note that this is currently not executed in parallel. Instead, the driver program computes all the elements.
Example 1: Return first 2 elements of above list. Similar as "limit s" in SQL.

```
scala> list.take(2)
res9: Array[String] = Array(apple, orange)
```

3.25 takeSample(withReplacement, num, [seed])

Return an array with a random sample of num elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.
Note: It returns an Array instead of RDD.
Example 1: "withReplacement"=true means output may have duplicate elements, else, it will not.

```
scala> val list = sc.parallelize(1 to 9)
list: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[55] at parallelize at <console>:33

scala> list.takeSample(true, 10)
res59: Array[Int] = Array(5, 8, 2, 4, 8, 9, 9, 1, 4, 5)

scala> list.takeSample(false, 10)
res60: Array[Int] = Array(3, 8, 2, 6, 1, 7, 5, 9, 4)
```

Example 2: If "seed" does not change, the result will not change.

```
scala>  list.takeSample(true, 5, 1)
res61: Array[Int] = Array(8, 6, 3, 8, 9)

scala>  list.takeSample(true, 5, 1)
res62: Array[Int] = Array(8, 6, 3, 8, 9)

scala>  list.takeSample(true, 5, 2)
res63: Array[Int] = Array(3, 8, 4, 8, 9)
```

3.26 takeOrdered(n, [ordering])

Return the first n elements of the RDD using either their natural order or a custom comparator.
Example 1: Similar as "order by limit n" in SQL.

```
scala> val list = sc.parallelize(List("apple", "orange", "banan", "APPLE", "BABY","cat", "1"  , "3" , "9" ))
list: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[64] at parallelize at <console>:33

scala> list.takeOrdered(3)
res67: Array[String] = Array(1, 3, 9)

scala> list.takeOrdered(5)
res68: Array[String] = Array(1, 3, 9, APPLE, BABY)
```

3.27 saveAsTextFile(path)

Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file.
Example 1: Save above list on HDFS.

```
list.saveAsTextFile("/tmp/tmpout")

# hadoop fs -ls /tmp/tmpout
Found 3 items
```

```
-rwxr-xr-x   3 root root          0 2015-02-28 02:23 /tmp/tmpout/_SUCCESS
-rwxr-xr-x   3 root root         25 2015-02-28 02:23 /tmp/tmpout/part-00000
-rwxr-xr-x   3 root root         15 2015-02-28 02:23 /tmp/tmpout/part-00001
# hadoop fs -cat /tmp/tmpout/part-00000
apple
orange
banan
APPLE
# hadoop fs -cat /tmp/tmpout/part-00001
BABY
cat
1
3
9
```

3.28 saveAsSequenceFile(path)
(Java and Scala) Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that either implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).
Example 1: Save a key-value pair as sequence file on HDFS.

```
val a = sc.parallelize(Array(("apple",1), ("orange",2), ("banana",3)))
a.saveAsSequenceFile("/tmp/tmpoutseq")

# hadoop fs -ls /tmp/tmpoutseq
Found 3 items
-rwxr-xr-x   3 root root          0 2015-02-28 02:27 /tmp/tmpoutseq/_SUCCESS
-rw-r--r--   3 root root        103 2015-02-28 02:27 /tmp/tmpoutseq/part-00000
-rw-r--r--   3 root root        123 2015-02-28 02:27 /tmp/tmpoutseq/part-00001
```

3.29 saveAsObjectFile(path)
(Java and Scala) Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using SparkContext.objectFile().
Example 1: Save and load an object file on HDFS.

```
scala> val list = sc.parallelize(List("apple", "orange", "banan"))
list: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[7] at parallelize at <console>:12

scala> list.saveAsObjectFile("/tmp/tmpobj")

scala> val newlist = sc.objectFile[String]("/tmp/tmpobj")
newlist: org.apache.spark.rdd.RDD[String] = FlatMappedRDD[11] at objectFile at <console>:12

scala> newlist.collect
res4: Array[String] = Array(orange, banan, apple)
```

3.30 countByKey()
Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
Example 1: Count the List of KV pairs.

```
scala> val kv = sc.parallelize( List(("apple", 1), ("orange", 2), ("banana", 3), ("apple", 2)) )
kv: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[15] at parallelize at <console>:12

scala> kv.countByKey
res6: scala.collection.Map[String,Long] = Map(banana -> 1, apple -> 2, orange -> 1)
```

3.31 foreach(func)
Run a function func on each element of the dataset. This is usually done for side effects such as updating an accumulator variable (see below) or interacting with external storage systems.
Example 1: Calculate the sum of 1 to 4.

```
scala> val accum = sc.accumulator(0)
accum: org.apache.spark.Accumulator[Int] = 0

scala> sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum += x)

scala> accum.value
res24: Int = 10
```

In all, here is an example to calculate the total length of the file.

```
val lines = sc.textFile("data.txt")
val lineLengths = lines.map(s => s.length)
val totalLength = lineLengths.reduce((a, b) => a + b)
```

- **Persist and Unpersist RDD**

Persist can save the RDD in memory or disk in this application after the first time it is computed.

```
lineLengths.persist()
lineLengths.unpersist()
```

Example 1: persist() an object in different levels.

```
scala> import org.apache.spark.storage.StorageLevel
import org.apache.spark.storage.StorageLevel
```

```
scala> kv.persist(StorageLevel.MEMORY_ONLY)
res9: kv.type = ParallelCollectionRDD[0] at parallelize at <console>:12

scala> kv.getStorageLevel
res10: org.apache.spark.storage.StorageLevel = StorageLevel(false, true, false, true, 1)

scala> kv.unpersist()
res13: kv.type = ParallelCollectionRDD[0] at parallelize at <console>:12

scala> kv.getStorageLevel
res14: org.apache.spark.storage.StorageLevel = StorageLevel(false, false, false, false, 1)

scala> kv.persist(StorageLevel.DISK_ONLY)
res15: kv.type = ParallelCollectionRDD[0] at parallelize at <console>:12

scala> kv.getStorageLevel
res16: org.apache.spark.storage.StorageLevel = StorageLevel(true, false, false, false, 1)
```

4. Functions to Spark

- **Pass reference of a function**

Example to add "hello" to each element in the RDD.

```
def sayhello(s: String): String = "Hello " + s
lines.map(sayhello)
```

Result:

```
scala> lines.collect
res31: Array[String] = Array(1,2,3, 4,5,6)
scala> lines.map(sayhello).collect
res32: Array[String] = Array(Hello 1,2,3, Hello 4,5,6)
```

- **Anonymous functions**

So simple way to do above stuff is:

```
lines.map(x => "Hello " + x)
```

5. KeyValue pairs

- **reduceByKey**

```
val lines = sc.textFile("data.txt")
val pairs = lines.map(s => (s, 1))
val counts = pairs.reduceByKey((a, b) => a + b)
```

Sample data:

```
# cat data.txt
This is a test data text file for Spark to use.
To test Scala and Spark,
we need to repeat again and again.
apple
orange
banana
APPle
APPLE
ORANGE
```

Sample result:

```
scala> counts.collect
res41: Array[(String, Int)] = Array((orange,1), (APPLE,1), (ORANGE,1), (apple,1), ("This is a test data text file for Spark to use. ",1), (APPl
```

- **sortByKey**

```
scala>  counts.sortByKey().collect
res43: Array[(String, Int)] = Array((APPLE,1), (APPle,1), (ORANGE,1), ("This is a test data text file for Spark to use. ",1), ("To test Scala a
```

6. Shared Variables

- **Broadcast Variables**

Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks.

```
scala> val broadcastVar = sc.broadcast(Array(1, 2, 3))
broadcastVar: org.apache.spark.broadcast.Broadcast[Array[Int]] = Broadcast(0)

scala> broadcastVar.value
res18: Array[Int] = Array(1, 2, 3)
```

- **Accumulators**

Accumulators are variables that are only “added” to through an associative operation and can therefore be efficiently supported in parallel. They can be used to implement counters (as in MapReduce) or sums.

```
scala> val accum = sc.accumulator(0)
accum: org.apache.spark.Accumulator[Int] = 0

scala> sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum += x)

scala> accum.value
res24: Int = 10
```

==

Related Posts

- How To Use Scala On Spark To Load Data Into Hbase/MapRDB -- Normal Load Or Bulk Load.
- How To Control The Parallelism Of Spark Job
- Using Spark Job To Upload Files To AWS S3 With Server Side Encryption Enabled
- Quickstart For Kafka Connecting To Spark Streaming On MapR Cluster
- Spark Fails To Parse A Json Object With Multiple Lines
- Understanding PageRank Algorithm In Scala On Spark
- Difference Between Spark HiveContext And SQLContext

Posted by OpenKB at 2:20 PM

Labels: scala, Spark

No comments:

Post a Comment

Prev Page Home Next Page

Subscribe to: Post Comments (Atom)

Popular Posts

Scala on Spark cheatsheet
This is a cookbook for scala programming. 1. Define a object with main function -- Helloworld. object HelloWorld { def main(args: Array...

How to control the file numbers of hive table after inserting data on MapR-FS.