Learning Spark_Chapter6

August 3, 2016

1 Chapter 6. Advanced Spark Progamming

1.0.1 소개

이 장에서는 아래 4가지를 배우게 됨 > - Accumulator > - Broadcast variable > - 파티션 단위 Transformation (mapPartitions, mapPartitionsWithIndex, foreachPartition() > - pipe()을 이용한 외부 스크립트 활용

> 샘플 데이터로는 아래와 같은 ham radio call log를 사용.. 하려 했으나 원본 파일을 구할수 없음..
> 대신 Spark 설치시 만들어지는 README.md 파일 사용

```
In [1]: import play.api.libs.json._

        val call = """{"address":"address here", "band":"40m","callsign":"KK6JLK","
        "contactlat":"37.384733","contactlong":"-122.032164",
        "county":"Santa Clara","dxcc":"291","fullname":"MATTHEW McPherrin",
        "id":57779,"mode":"FM","mylat":"37.751952821","mylong":"-122.4208688735"}"

        print(call)
```

```
{"address":"address here", "band":"40m","callsign":"KK6JLK","city":"SUNNYVALE",
"contactlat":"37.384733","contactlong":"-122.032164",
"county":"Santa Clara","dxcc":"291","fullname":"MATTHEW McPherrin",
"id":57779,"mode":"FM","mylat":"37.751952821","mylong":"-122.4208688735"}
```

1.0.2 어큐뮬레이터(Accumulators)

> Spark에서 제공하는 shared variable에는 **accumulator**와 **broadcast variable**가 있다.
>
> **accumulator**는 그 이름이 의미하는 것 처럼
> Cluster에 분산되어 있는 각 worker node에서 일어나는
> 어떤 이벤트의 수를 카운트하여 driver program에 집계하는 것과 같은 목적으로 사용한다.
>
> 물론 RDD Transformation으로도 각 worker node에서 driver program으로 동일한 정보 집계를 구현할 수 있으나
> 원래 목적으로 하는 RDD Transformation에 비해
> 다소 보조적인 목적(디버깅, 모니터링)을 위해 사용하며
> 마치 전체 cluster 전역변수를 사용하는 것과 같이 간단한 문법으로 사용할 수 있다.

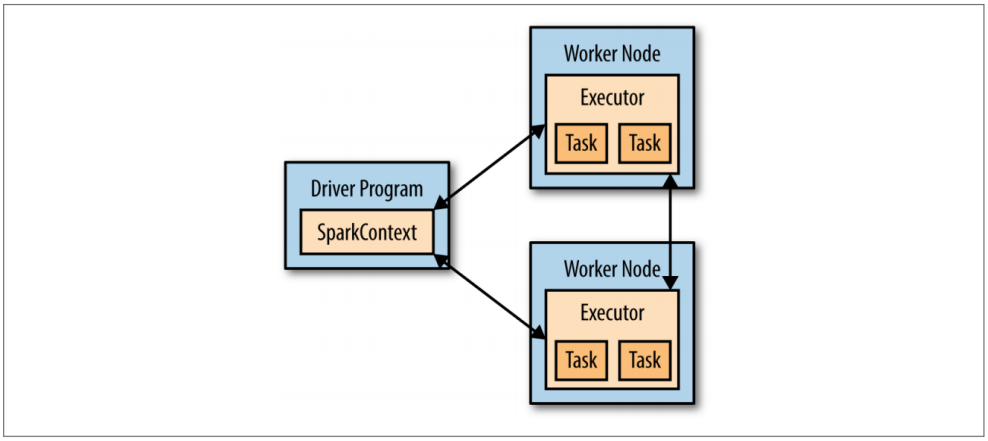


Figure 2-3. Components for distributed execution in Spark

Local image

Example: simple accumulator

accumulator 생성은 아래와 같으며, accumulator에 초기값을 할당하고, 초기값의 Type에 따른 그 동작이 다르다.
(아래 예 에서는 org.apache.spark.Accumulator< $java.lang.Integer$ > type의 object가 생성된다.)

```
val blankLines = sc.accumulator(0)
```

Worker node에서 **accumulator**변수에 += 연산을 이용하여 값을 더할 수 있다.

```
In [2]: val file = sc.textFile("/home/sparkuser/spark-1.6.2-bin-hadoop2.6/README.md")
        file.
          take(10).
          foreach(println)

        print("\nNumber of partitions: " + file.partitions.size)
```

```
# Apache Spark

Spark is a fast and general cluster computing system for Big Data. It provides
high-level APIs in Scala, Java, Python, and R, and an optimized engine that
supports general computation graphs for data analysis. It also supports a
rich set of higher-level tools including Spark SQL for SQL and DataFrames,
MLlib for machine learning, GraphX for graph processing,
and Spark Streaming for stream processing.

<http://spark.apache.org/>

Number of partitions: 1
```

```
In [26]: val blankLines = sc.accumulator(0, "counter")
```

```
        val callSigns = file.flatMap(line => {
          if (line == "") {
            blankLines += 1
          }
          line.split(" ")
        })

        callSigns.saveAsTextFile("./_tmp/output_" + System.currentTimeMillis / 100

        println("Blank lines: " + blankLines.value)
```

```
Blank lines: 35
```

accumulator변수는 worker node에게 write-only라서 value attribute에 접근할 수 없다. 접근하려 할 경우 에러 발생(java.lang.UnsupportedOperationException: Can't read accumu에게 value in task)

(각 worker node에는 initial value만 전달되므로 worker node에서는 전체 cluster의 집계 값을 알 수 없음)

```
In [4]: val blankLines = sc.accumulator(0)

        val callSigns = file.flatMap(line => {
          if (line == "") {
            blankLines += 1
            print(blankLines.value)
          }
          line.split(" ")
        })

        callSigns.saveAsTextFile("./_tmp/output_" + System.currentTimeMillis / 1000
        println("Blank lines: " + blankLines.value)
```

```
Out[4]: Name: org.apache.spark.SparkException
        Message: Job aborted due to stage failure: Task 0 in stage 2.0 failed 1 tim
                at org.apache.spark.Accumulable.value(Accumulators.scala:117)
                at $line27.$read$$iwC$$iwC$$iwC$$iwC$$iwC$$iwC$$anonfun$1.apply(<co
                at $line27.$read$$iwC$$iwC$$iwC$$iwC$$iwC$$iwC$$anonfun$1.apply(<co
                at scala.collection.Iterator$$anon$13.hasNext(Iterator.scala:371)
                at scala.collection.Iterator$$anon$11.hasNext(Iterator.scala:327)
                at org.apache.spark.rdd.PairRDDFunctions$$anonfun$saveAsHadoopDatas
                at org.apache.spark.rdd.PairRDDFunctions$$anonfun$saveAsHadoopDatas
                at org.apache.spark.rdd.PairRDDFunctions$$anonfun$saveAsHadoopDatas
                at org.apache.spark.util.Utils$.tryWithSafeFinallyAndFailureCallbac
                at org.apache.spark.rdd.PairRDDFunctions$$anonfun$saveAsHadoopDatas
                at org.apache.spark.rdd.PairRDDFunctions$$anonfun$saveAsHadoopDatas
                at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:6
                at org.apache.spark.scheduler.Task.run(Task.scala:89)
```

```
        at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExec
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExe
        at java.lang.Thread.run(Thread.java:745)

Driver stacktrace:
StackTrace: org.apache.spark.scheduler.DAGScheduler.org$apache$spark$schedu
org.apache.spark.scheduler.DAGScheduler$$anonfun$abortStage$1.apply(DAGSche
org.apache.spark.scheduler.DAGScheduler$$anonfun$abortStage$1.apply(DAGSche
scala.collection.mutable.ResizableArray$class.foreach(ResizableArray.scala:
scala.collection.mutable.ArrayBuffer.foreach(ArrayBuffer.scala:47)
org.apache.spark.scheduler.DAGScheduler.abortStage(DAGScheduler.scala:1418)
org.apache.spark.scheduler.DAGScheduler$$anonfun$handleTaskSetFailed$1.appl
org.apache.spark.scheduler.DAGScheduler$$anonfun$handleTaskSetFailed$1.appl
scala.Option.foreach(Option.scala:236)
org.apache.spark.scheduler.DAGScheduler.handleTaskSetFailed(DAGScheduler.sc
org.apache.spark.scheduler.DAGSchedulerEventProcessLoop.doOnReceive(DAGSche
org.apache.spark.scheduler.DAGSchedulerEventProcessLoop.onReceive(DAGSchedu
org.apache.spark.scheduler.DAGSchedulerEventProcessLoop.onReceive(DAGSchedu
org.apache.spark.util.EventLoop$$anon$1.run(EventLoop.scala:48)
org.apache.spark.scheduler.DAGScheduler.runJob(DAGScheduler.scala:620)
org.apache.spark.SparkContext.runJob(SparkContext.scala:1832)
org.apache.spark.SparkContext.runJob(SparkContext.scala:1845)
org.apache.spark.SparkContext.runJob(SparkContext.scala:1922)
org.apache.spark.rdd.PairRDDFunctions$$anonfun$saveAsHadoopDataset$1.apply$
org.apache.spark.rdd.PairRDDFunctions$$anonfun$saveAsHadoopDataset$1.apply(
org.apache.spark.rdd.PairRDDFunctions$$anonfun$saveAsHadoopDataset$1.apply(
org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:1
org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:1
org.apache.spark.rdd.RDD.withScope(RDD.scala:316)
org.apache.spark.rdd.PairRDDFunctions.saveAsHadoopDataset(PairRDDFunctions.
org.apache.spark.rdd.PairRDDFunctions$$anonfun$saveAsHadoopFile$4.apply$mcV
org.apache.spark.rdd.PairRDDFunctions$$anonfun$saveAsHadoopFile$4.apply(Pai
org.apache.spark.rdd.PairRDDFunctions$$anonfun$saveAsHadoopFile$4.apply(Pai
org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:1
org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:1
org.apache.spark.rdd.RDD.withScope(RDD.scala:316)
org.apache.spark.rdd.PairRDDFunctions.saveAsHadoopFile(PairRDDFunctions.sca
org.apache.spark.rdd.PairRDDFunctions$$anonfun$saveAsHadoopFile$1.apply$mcV
org.apache.spark.rdd.PairRDDFunctions$$anonfun$saveAsHadoopFile$1.apply(Pai
org.apache.spark.rdd.PairRDDFunctions$$anonfun$saveAsHadoopFile$1.apply(Pai
org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:1
org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:1
org.apache.spark.rdd.RDD.withScope(RDD.scala:316)
org.apache.spark.rdd.PairRDDFunctions.saveAsHadoopFile(PairRDDFunctions.sca
org.apache.spark.rdd.RDD$$anonfun$saveAsTextFile$1.apply$mcV$sp(RDD.scala:1
org.apache.spark.rdd.RDD$$anonfun$saveAsTextFile$1.apply(RDD.scala:1436)
org.apache.spark.rdd.RDD$$anonfun$saveAsTextFile$1.apply(RDD.scala:1436)
```

```
org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:1
org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:1
org.apache.spark.rdd.RDD.withScope(RDD.scala:316)
org.apache.spark.rdd.RDD.saveAsTextFile(RDD.scala:1436)
$line28.$read$$iwC$$iwC$$iwC$$iwC$$iwC$$iwC.<init>(<console>:31)
$line28.$read$$iwC$$iwC$$iwC$$iwC$$iwC.<init>(<console>:36)
$line28.$read$$iwC$$iwC$$iwC$$iwC.<init>(<console>:38)
$line28.$read$$iwC$$iwC$$iwC.<init>(<console>:40)
$line28.$read$$iwC$$iwC.<init>(<console>:42)
$line28.$read$$iwC.<init>(<console>:44)
$line28.$read.<init>(<console>:46)
$line28.$read$.<init>(<console>:50)
$line28.$read$.<clinit>(<console>)
$line28.$eval$.<init>(<console>:7)
$line28.$eval$.<clinit>(<console>)
$line28.$eval.$print(<console>)
sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:6
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImp
java.lang.reflect.Method.invoke(Method.java:498)
org.apache.spark.repl.SparkIMain$ReadEvalPrint.call(SparkIMain.scala:1065)
org.apache.spark.repl.SparkIMain$Request.loadAndRun(SparkIMain.scala:1346)
org.apache.spark.repl.SparkIMain.loadAndRunReq$1(SparkIMain.scala:840)
org.apache.spark.repl.SparkIMain.interpret(SparkIMain.scala:871)
org.apache.spark.repl.SparkIMain.interpret(SparkIMain.scala:819)
org.apache.toree.kernel.interpreter.scala.ScalaInterpreter$$anonfun$interpr
org.apache.toree.kernel.interpreter.scala.ScalaInterpreter$$anonfun$interpr
org.apache.toree.global.StreamState$.withStreams(StreamState.scala:81)
org.apache.toree.kernel.interpreter.scala.ScalaInterpreter$$anonfun$interpr
org.apache.toree.kernel.interpreter.scala.ScalaInterpreter$$anonfun$interpr
org.apache.toree.utils.TaskManager$$anonfun$add$2$$anon$1.run(TaskManager.s
java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1
java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:
java.lang.Thread.run(Thread.java:745)
```

Example: using transformation

아래와 같이 RDD Transformation으로 동일한 일을 할수도 있으나,
본래 하려는 작업을 수행하면서 부가적인 디버깅/모니터링을 위해 **accumulator**를 사용한다.

```
In [8]: file_part.
          filter(line => line == "").
          count()
```

```
Out[8]: 35
```

Example: 활용

README.md 파일을 이용해 책 예제 **6-5***를 테스트

파일의 line이 빈줄일 경우 invalidLineCount를 증가시키고, 내용이 있는 줄 일 경우 valid-LineCount를 증가

실제 하려는 일은 Transformation으로 내용이 있는 줄을 filter하여 저장하는 것인데,
이 과정에서 내용이 있는 줄이 임계값이상일 경우 결과를 파일로 저장

```
In [34]: val file = sc.textFile("/home/sparkuser/spark-1.6.2-bin-hadoop2.6/README.m
         file.
           take(10).
           foreach(println)

# Apache Spark

Spark is a fast and general cluster computing system for Big Data. It provides
high-level APIs in Scala, Java, Python, and R, and an optimized engine that
supports general computation graphs for data analysis. It also supports a
rich set of higher-level tools including Spark SQL for SQL and DataFrames,
MLlib for machine learning, GraphX for graph processing,
and Spark Streaming for stream processing.

<http://spark.apache.org/>


In [10]: file.
           filter(line => line != "").
           take(10).
           foreach(println)

# Apache Spark
Spark is a fast and general cluster computing system for Big Data. It provides
high-level APIs in Scala, Java, Python, and R, and an optimized engine that
supports general computation graphs for data analysis. It also supports a
rich set of higher-level tools including Spark SQL for SQL and DataFrames,
MLlib for machine learning, GraphX for graph processing,
and Spark Streaming for stream processing.
<http://spark.apache.org/>
## Online Documentation
You can find the latest Spark documentation, including a programming
```

위와 같은 일을 filter 과정에서
어떤 값에 따라 처리를 컨트롤 하고 싶은 경우 아래와 같이 할수 있다.

```
In [31]: val validLineCount = sc.accumulator(0)
         val invalidLineCount = sc.accumulator(0)

         def validateLine(line: String): Boolean = {
```

```
            if(line != ""){
              validLineCount += 1
              true
            } else{
              invalidLineCount += 1
              false
            }
          }
```

```
In [29]: val validLineRDD = file.filter(validateLine)

         println(s"Valid lines: " + validLineCount.value + ", " + "Invalid lines: "

         if( validLineCount.value > 1.5 * invalidLineCount.value) {
           println("Saving result as file...")
           validLineRDD.saveAsTextFile("./_tmp/output_" + System.currentTimeMillis
         } else{
           print("Too many empty lines")
         }
```

```
Valid lines: 0, Invalid lines: 0

Too many empty lines
```

accumulator는 RDD Transformation과 마찬가지로 lazy evaluated.
즉 action을 실행하는 시점에야 그 값을 집계된다.

```
In [32]: val validLineRDD = file.filter(validateLine)

         validLineRDD.count()

         println(s"Valid lines: " + validLineCount.value + ", " + "Invalid lines: "

         if( validLineCount.value > 1.5 * invalidLineCount.value) {
           println("Saving result as file...")
           validLineRDD.saveAsTextFile("./_tmp/output_" + System.currentTimeMillis
         } else{
           print("Too many empty lines")
         }
```

```
Valid lines: 60, Invalid lines: 35

Saving result as file...
```

```
In [36]: val file = sc.textFile("/home/sparkuser/spark-1.6.2-bin-hadoop2.6/README.m

         val validLineCount = sc.accumulator(0)
         val invalidLineCount = sc.accumulator(0)
```

```
        file.foreach(line => {
          if(line != "") validLineCount += 1
          else invalidLineCount += 1
        })

        val validLineRDD = file.filter(line => line != "")

        println(s"Valid lines: " + validLineCount.value + ", " + "Invalid lines: "

        if( validLineCount.value > 1.5 * invalidLineCount.value) {
          println("Saving result as file...")
          validLineRDD.saveAsTextFile("./_tmp/output_" + System.currentTimeMillis
        } else{
          print("Too many empty lines")
        }
```

```
Valid lines: 60, Invalid lines: 35

Saving result as file...
```

1.0.3 어큐뮬레이터와 장애 내구성(Accumulators and Fault Tolerance)

Transformation뿐만 아니라 action에서도 **accumulator**연산을 사용할 수 있는데,
Spark는 Action에서 사용하는 **accumulator**연산의 신뢰성을 보장하지만
Transformation에서 사용하는 **accumulator**연산의 신뢰성은 보장하지 않는다.

[원문] **For accumulator updates performed inside actions only, Spark guarantees** that each task's update to the accumulator will only be applied once, i.e. restarted tasks will not update the value. **In transformations, users should be aware** of that each task's update may be applied more than once if tasks or job stages are re-executed.
http://spark.apache.org/docs/latest/programming-guide.html#accumulators

Transformation은 여러 장애 상황에서 여러번 수행될 수 있다.
예를들어 Node crash가 일어나면 해당 task들을 다른 node에서 다시 실행하게 되고,
느린 task의 경우 복재본이 만들어져 다른 node에서 수행되기도 하고,
Cashed 되었으나 자주 사용되지 않는 RDD의 일부는 메모리에서 내려가게 되는데 재사용시 일부 Task가 재실행되기도 한다.

이런 여러 가지 fault tolerance 상황에서 transformation이 여러번 실행될 수 있는데,
이때 accumulator값이 중복 집계되는 문제가 발생할 수 있다.
[참고] http://imranrashid.com/posts/Spark-Accumulators/

accumulator값의 신뢰성이 보장되어야 하는 경우 **action**에 accumulator연산을 집어 넣는 것이 좋다.

driver program은 accumulator instance의 복제본을 모든 task에 전달하고,
계산이 끝나고 driver program으로 다시 반환되므로
accumulator instance의 용량이 크고 Task수가 많아질 경우 효율적이지 않다.
따라서 accumulator는 가능한 가볍고 간단한 목적을 위해 사용하는 것이 좋다.

```
In [37]: val file = sc.textFile("/home/sparkuser/spark-1.6.2-bin-hadoop2.6/README.m

         val validLineCount = sc.accumulator(0)
         val invalidLineCount = sc.accumulator(0)

         file.foreach(line => {
           if(line != "") validLineCount += 1
           else invalidLineCount += 1
         })

         val validLineRDD = file.filter(line => line != "")

         println(s"Valid lines: " + validLineCount.value + ", " + "Invalid lines: "

         if( validLineCount.value > 1.5 * invalidLineCount.value) {
           println("Saving result as file...")
           validLineRDD.saveAsTextFile("./_tmp/output_" + System.currentTimeMillis
         } else{
           print("Too many empty lines")
         }
```

```
Valid lines: 60, Invalid lines: 35

Saving result as file...
```

1.0.4 사용자 지정 어큐뮬레이터(Custom Accumulators)

Spark에서 기본적으로 제공하는 (built-in) accumulator는 Int, Double, Long, Float 이 있다.
앞서 예에서 사용한 accumulator는 $org.apache.spark.Accumulator < java.lang.Integer >$ type이
었다.

다른 연산이나 데이터 구조를 이용하여 집계하기 위하여 별도의 accumulator를 정의할 수 있다.
이를 위해서는 **AccumulatorParam** class를 상속받은 class/object를 정의해야 한다.

Example: Partition & custom accumulator

HashMap 형태의 accumulator를 사용하고자 한다.
예를들어 acc += ("a", 1) 라고 할 경우 HashMap에 동일 key가 있다면 value += 1이 수행되고
동일 key가 없다면 "a"-> 1이 추가되는 accumulator를 필요로 한다고 해 보자.

```
In [5]: //Original Source: https://gist.github.com/fedragon/b22e5d1eee4803c86e53

        import org.apache.spark.{ AccumulableParam, SparkConf }
        import org.apache.spark.serializer.JavaSerializer
        import scala.collection.mutable.{ HashMap => MutableHashMap }
```

```
/*
 * Allows a mutable HashMap[String, Int] to be used as an accumulator in Sp
 * Whenever we try to put (k, v2) into an accumulator that already contains
 * will be a HashMap containing (k, v1 + v2).
 *
 * Would have been nice to extend GrowableAccumulableParam instead of redef
 * private to the spark package.
 */
object HashMapParam extends AccumulableParam[MutableHashMap[String, Int],

  def addAccumulator(acc: MutableHashMap[String, Int], elem: (String, Int))
    val (k1, v1) = elem
    acc += acc.find(_._1 == k1).map {
      case (k2, v2) => k2 -> (v1 + v2)
    }.getOrElse(elem)

    acc
  }

  /*
   * This method is allowed to modify and return the first value for effici
   *
   * @see org.apache.spark.GrowableAccumulableParam.addInPlace(r1: R, r2: R
   */
  def addInPlace(acc1: MutableHashMap[String, Int], acc2: MutableHashMap[St
    acc2.foreach(elem => addAccumulator(acc1, elem))
    acc1
  }

  /*
   * @see org.apache.spark.GrowableAccumulableParam.zero(initialValue: R):
   */
  def zero(initialValue: MutableHashMap[String, Int]): MutableHashMap[Strin
    val ser = new JavaSerializer(new SparkConf(false)).newInstance()
    val copy = ser.deserialize[MutableHashMap[String, Int]](ser.serialize(i
    copy.clear()
    copy
  }
}
```

앞서 README.md 파일에서 빈줄의 수를 count 했었는데,
각 partiton에서 몇건이나 이런 빈줄이 발생하는지 추적하고 싶다면
방금 정의한 HashMapParam를 이용하여 아래와 같이 해 볼 수 있다.

```
In [38]: val file_part = sc.textFile("/home/sparkuser/spark-1.6.2-bin-hadoop2.6/REA
         file_part.
           take(10).
           foreach(println)
```

```
        print("\nNumber of partitions: " + file_part.partitions.size)
```

```
Spark is built using [Apache Maven](http://maven.apache.org/).
The easiest way to start using Spark is through the Scala shell:
```

```
["Specifying the Hadoop Version"](http://spark.apache.org/docs/latest/building-spar
# Apache Spark
```

```
Number of partitions: 11
```

```
In [39]: import scala.collection.mutable.HashMap
         import org.apache.spark.TaskContext

         val mapAcc = sc.accumulable(new HashMap[String, Int])(HashMapParam)

         val callSigns = file_part.flatMap(line => {
           if (line == "") {
             val ctx = TaskContext.get
             val stageId = ctx.stageId
             val partId = ctx.partitionId
             val hostname = ctx.taskMetrics.hostname

             mapAcc += (s"Stage: $stageId, Partition: $partId, Host: $hostname", 1)
           }

           line.split(" ")
         })

         callSigns.saveAsTextFile("./_tmp/output_" + System.currentTimeMillis / 100

         mapAcc.value.foreach{case(key, value) => println(key + " -> " + value)}
```

```
Stage: 52, Partition: 0, Host: localhost -> 5
Stage: 52, Partition: 5, Host: localhost -> 3
Stage: 52, Partition: 1, Host: localhost -> 2
Stage: 52, Partition: 10, Host: localhost -> 2
Stage: 52, Partition: 6, Host: localhost -> 2
Stage: 52, Partition: 2, Host: localhost -> 5
Stage: 52, Partition: 7, Host: localhost -> 1
Stage: 52, Partition: 3, Host: localhost -> 4
Stage: 52, Partition: 8, Host: localhost -> 3
Stage: 52, Partition: 4, Host: localhost -> 4
```

```
Stage: 52, Partition: 9, Host: localhost -> 4
```

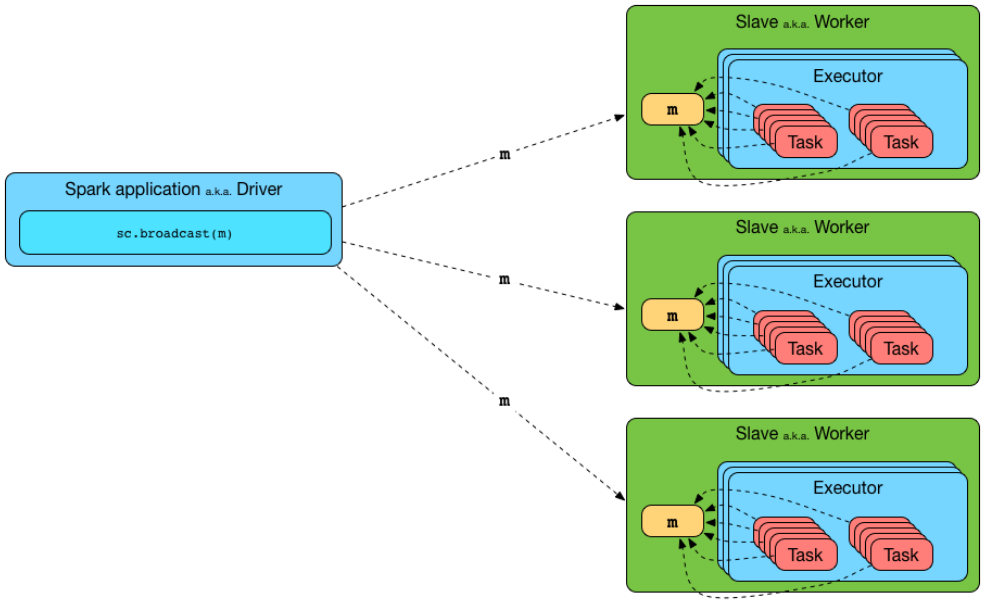
1.0.5 브로드캐스트 변수(Broadcast Variables)

Broadcast variable을 이용하여
Task들이 공통적으로 그리고 반복적으로 사용하는
read-only variable을 각 worker에 cashing할 수 있다.

예를들어 각 dictionary에 있는 단어의 등장 횟수를 집계할 경우나
머신러닝에서 각 Partition에 흩어져 있는 sample에 대한 예측값을 구하기 위하여 모든 task
에서 동일한 parameter vector를 사용해야 하는 경우 등에
효과적으로 사용될 수 있다.

SparkContext.broadcast method를 이용하여 생성하며,
value property를 이용하여 값에 접근

mutable 변수를 broadcast variable로 넘겨줄 수 있고,
각 worker node에서 이 값을 변경할 수도 있지만
변경이 다른 노드에는 영향을 주지 않는다.



(출처: https://jaceklaskowski.gitbooks.io/mastering-apache-spark/content/spark-broadcast.html)

Example:

유입되는 로그(file_new_log)에서 특정 단어들(bv_lookup_words)이 포함된 문장만을 처리한다고 해 보자.

```
In [125]: val bv_lookup_words = sc.broadcast(Set("spark", "you", "mllib", "python")

In [126]: val file_new_log = sc.textFile("/home/sparkuser/spark-1.6.2-bin-hadoop2.6
```

```
            file_new_log.
              take(10).
              foreach(println)

Spark is built using [Apache Maven](http://maven.apache.org/).
The easiest way to start using Spark is through the Scala shell:



["Specifying the Hadoop Version"](http://spark.apache.org/docs/latest/building-spar
# Apache Spark


In [127]: file_new_log.
              map{ line =>
                val words = line.toLowerCase().split(" ").toSet;
                val common = words & bv_lookup_words.value;
                if(common.size > 0) (common, line)
                else (Set(), line)
              }.
              filter{case (k, v) => k.size > 0}.
              take(100).
              foreach(println)

(Set(spark),Spark is built using [Apache Maven](http://maven.apache.org/).)
(Set(spark),The easiest way to start using Spark is through the Scala shell:)
(Set(spark),# Apache Spark)
(Set(spark),To build Spark and its example programs, run:)
(Set(spark),Spark is a fast and general cluster computing system for Big Data. It p
(Set(you, spark),You can find the latest Spark documentation, including a programmi
(Set(you),(You do not need to do this if you downloaded a pre-built package.))
(Set(you),You can set the MASTER environment variable when running examples to subm
(Set(spark),rich set of higher-level tools including Spark SQL for SQL and DataFram
(Set(spark),Spark uses the Hadoop core library to talk to HDFS and other Hadoop-sup
(Set(mllib),MLlib for machine learning, GraphX for graph processing,)
(Set(spark),Testing first requires [building Spark](#building-spark). Once Spark is
(Set(spark),and Spark Streaming for stream processing.)
(Set(python),## Interactive Python Shell)
(Set(you),locally with one thread, or "local[N]" to run locally with N threads. You
(Set(you, spark),Hadoop, you must build Spark against the same version that your cl
(Set(spark),## Building Spark)
(Set(spark),Spark also comes with several sample programs in the `examples` directo
(Set(you, python),Alternatively, if you prefer Python, you can use the Python shell
```

위의 작업을 join transformation을 이용하여 수행할 수도 있지만,
분산 환경에서는 상대적으로 큰 테이블(fact table)과 작은 테이블(dimension)을 join하는

경우
map을 이용한 join을 사용하는 것이 효과적(map-size join이라 함)
(참고: http://dmtolpeko.com/2015/02/20/map-side-join-in-spark/)

1.0.6 브로드캐스트 최적화(Optimizing Broadcasts)

앞서 broadcast variable을 이용한 구현을
일반 변수를 사용하여 아래와 같이 바꿔볼 수 있다.

```
In [128]: val lookup_words = Set("spark", "you", "mllib", "python")

In [129]: val file_new_log = sc.textFile("/home/sparkuser/spark-1.6.2-bin-hadoop2.6

          file_new_log.
            take(10).
            foreach(println)


Spark is built using [Apache Maven](http://maven.apache.org/).
The easiest way to start using Spark is through the Scala shell:



["Specifying the Hadoop Version"](http://spark.apache.org/docs/latest/building-spar
# Apache Spark



In [130]: file_new_log.
            map{ line =>
              val words = line.toLowerCase().split(" ").toSet;
              val common = words & lookup_words;
              if(common.size > 0) (common, line)
              else (Set(), line)
            }.
            filter{case (k, v) => k.size > 0}.
            take(100).
            foreach(println)

(Set(spark),Spark is built using [Apache Maven](http://maven.apache.org/).)
(Set(spark),The easiest way to start using Spark is through the Scala shell:)
(Set(spark),# Apache Spark)
(Set(spark),To build Spark and its example programs, run:)
(Set(spark),Spark is a fast and general cluster computing system for Big Data. It p
(Set(you, spark),You can find the latest Spark documentation, including a programmi
(Set(you),(You do not need to do this if you downloaded a pre-built package.))
(Set(you),You can set the MASTER environment variable when running examples to subm
(Set(spark),rich set of higher-level tools including Spark SQL for SQL and DataFram
```

```
(Set(spark),Spark uses the Hadoop core library to talk to HDFS and other Hadoop-sup
(Set(mllib),MLlib for machine learning, GraphX for graph processing,)
(Set(spark),Testing first requires [building Spark](#building-spark). Once Spark is
(Set(spark),and Spark Streaming for stream processing.)
(Set(python),## Interactive Python Shell)
(Set(you),locally with one thread, or "local[N]" to run locally with N threads. You
(Set(you, spark),Hadoop, you must build Spark against the same version that your cl
(Set(spark),## Building Spark)
(Set(spark),Spark also comes with several sample programs in the `examples` directo
(Set(you, python),Alternatively, if you prefer Python, you can use the Python shell
```

위와 같은 구현에서는 lookup_words의 복사본이 모든 task에 직렬화되어(serialized) 전달되고, 각 task를 실행하기 직전에 역직렬화(deserialized)된다.

물론 broadcast variable을 사용할 경우에도 비슷한 과정을 거치지만 worker 단위로 한번만 직렬화 개체를 전달하고 역직렬화(deserialized)가 한번만 필요하므로 반복사용되는 데이터의 경우 효과적이다.

따라서 network overhead를 줄일 수 있고 직렬화(serialization) 포멧을 사용하는 것이 매우 중요하다.

8장에서 $Kyro$라는 빠른 직렬화 라이브러리에 대해 배우게 될 것이다.

1.0.7 파티션별로 작업하기(Working on a Per-Partition Basis)

RDD의 각 element에 대하여 어떤 작업을 하는 것이 아니라
각 Partition에 대하여 한번식 어떤 작업이 해야하는 경우
예를들어 DB connection이나 난수생성등의 일은 data element단위로 필요한 일이 아니라 partition단위로 필요한 작업니다.

이런 partition단위의 일을 쉽게 해 주는 것이 **mapPartitions, mapPartitionsWithIndex, foreachPartition**함수이다.

```
In [150]: import scala.collection.mutable.HashMap
          import org.apache.spark.TaskContext

          val file_new_log = sc.textFile("/home/sparkuser/spark-1.6.2-bin-hadoop2.6

          file_new_log.
            mapPartitions(lineIter => {

              val work2do = "Common work for partition " + TaskContext.get.partit

              lineIter.toList.map(line => work2do + " -> " + line).iterator
            }
          ).
          take(15).
          foreach(println)
```

```
Common work for partition 0 ->
Common work for partition 0 -> You can set the MASTER environment variable when run
Common work for partition 0 -> building for particular Hive and Hive Thriftserver c
Common work for partition 1 -> # Apache Spark
Common work for partition 1 -> ## Interactive Scala Shell
Common work for partition 1 -> examples to a cluster. This can be a mesos:// or spa
Common work for partition 1 ->
Common work for partition 2 ->
Common work for partition 2 ->
Common work for partition 2 -> "yarn" to run on YARN, and "local" to run
Common work for partition 2 -> ## Configuration
Common work for partition 3 -> Spark is a fast and general cluster computing system
Common work for partition 3 -> The easiest way to start using Spark is through the
Common work for partition 3 -> locally with one thread, or "local[N]" to run locall
Common work for partition 3 ->
```

```
In [151]: import scala.collection.mutable.HashMap
          import org.apache.spark.TaskContext

          val file_new_log = sc.textFile("/home/sparkuser/spark-1.6.2-bin-hadoop2.6

          file_new_log.
            mapPartitionsWithIndex((idx, lineIter) => {

                val work2do = "Common work for partition " + idx

                lineIter.toList.map(line => work2do + " -> " + line).iterator
              }
            ).
            take(15).
            foreach(println)
```

```
Common work for partition 0 ->
Common work for partition 0 -> You can set the MASTER environment variable when run
Common work for partition 0 -> building for particular Hive and Hive Thriftserver c
Common work for partition 1 -> # Apache Spark
Common work for partition 1 -> ## Interactive Scala Shell
Common work for partition 1 -> examples to a cluster. This can be a mesos:// or spa
Common work for partition 1 ->
Common work for partition 2 ->
Common work for partition 2 ->
Common work for partition 2 -> "yarn" to run on YARN, and "local" to run
Common work for partition 2 -> ## Configuration
Common work for partition 3 -> Spark is a fast and general cluster computing system
Common work for partition 3 -> The easiest way to start using Spark is through the
Common work for partition 3 -> locally with one thread, or "local[N]" to run locall
Common work for partition 3 ->
```

1.0.8 외부 프로그램과 파이프로 연결하기(Piping to External Programs)

pipe 함수를 이용하여
RDD의 데이터를 (Scala, Java, Python이 아니더라도) 스크립트에 전달하고
처리된 결과값을 받아올 수 있다.

SparkContext.addFile(path) 와 같이 context에 file을 추가하면,
driver node에 존재하는 script file이 각 worker node에 복제된다.

각 node에서 file의 위치는 SparkFiles.getRootDirectory를 이용하여 알수 있다.

또한 SparkFiles.get(name)으로 각 node에서 file을 복제해갈 수 있다.
물론 수동 혹은 다른 방법으로 각 worker node의 SparkFiles.getRootDirectory 경로에 위치시켜도 된다.

예제로 문장을 받아 " "으로 나눈 후 단어 수를 반환하는 아래의 R script를 테스트

```
#!/usr/bin/Rscript
f <- file("stdin")
open(f)

while(length(line <- readLines(f, n=1)) > 0) {
  len <- length(strsplit(line, " ")[[1]])
  write(len, stdout())
}
```

```
In [1]: val file_new_log = sc.textFile("/home/sparkuser/spark-1.6.2-bin-hadoop2.6/R

        file_new_log.
          filter(line => line != "").
          take(10).
          foreach(println)
```

```
Spark is built using [Apache Maven](http://maven.apache.org/).
The easiest way to start using Spark is through the Scala shell:
["Specifying the Hadoop Version"](http://spark.apache.org/docs/latest/building-spar
# Apache Spark
To build Spark and its example programs, run:
    ./bin/pyspark
    ./bin/run-example SparkPi
    MASTER=spark://host:7077 ./bin/run-example SparkPi
Please see the guidance on how to
for detailed guidance on building for a particular distribution of Hadoop, includir
```

```
In [2]: import org.apache.spark.SparkFiles

        val rScript = "/home/sparkuser/work/jupyter/_etc/withSpark.R"
        val rScriptName = "withSpark.R"
        sc.addFile(rScript)
```

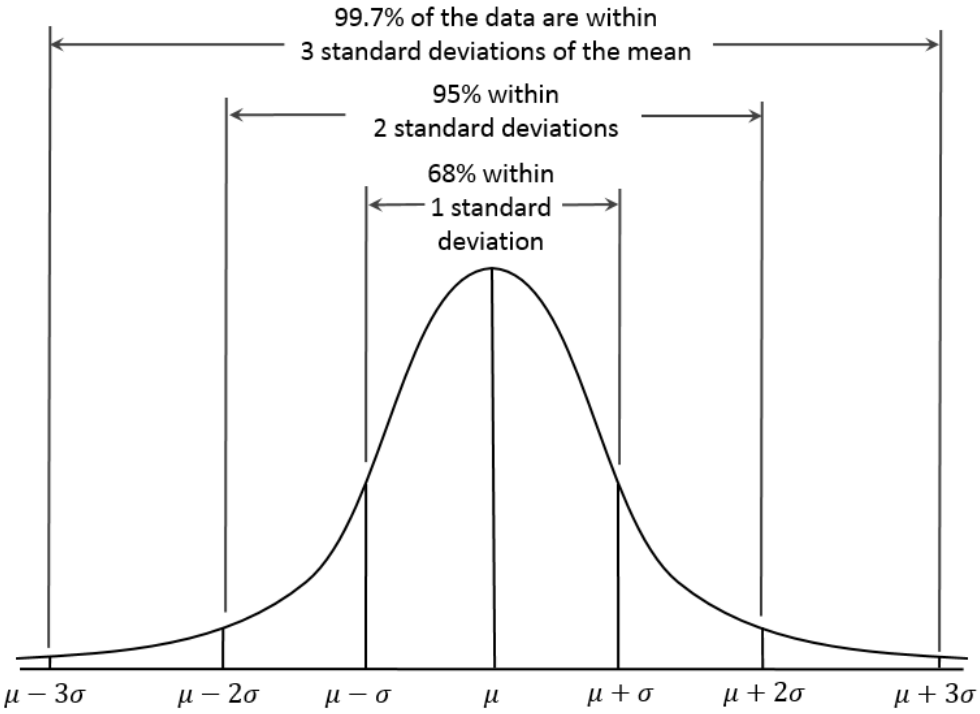
```
In [3]: file_new_log.
          filter(line => line != "").
          pipe(Seq(SparkFiles.get(rScriptName))).
          collect().toList
```

```
Out[3]: List(6, 12, 4, 3, 8, 5, 6, 7, 7, 12, 3, 5, 8, 8, 14, 8, 10, 6, 13, 13, 10,
```

1.0.9 수치 RDD 연산들(Numeric RDD Operations)

수치 데이터를 포함하는 RDD의 기술통계량(descriptive statistics)을 계산하기 위한 함수들을 제공한다.
count(), mean(), sum(), max(), min(), variance(), sampleVariance(), stdev(), sampleStddev()

예제로 문장내 단어의 수가
파일내 문장들의 단어수 분포의
평균을 중심으로 68% 내에 있는 문장들을 필터링 해 보자.



Local image

```
In [25]: val file_new_log = sc.textFile("/home/sparkuser/spark-1.6.2-bin-hadoop2.6/
```

```
file_new_log.
  filter(line => line != "").
  map(line => (line.split(" ").length, line)).
  take(10).
  foreach(println)
```

```
(6,Spark is built using [Apache Maven](http://maven.apache.org/).)
(12,The easiest way to start using Spark is through the Scala shell:)
(4,["Specifying the Hadoop Version"](http://spark.apache.org/docs/latest/building-s
(3,# Apache Spark)
(8,To build Spark and its example programs, run:)
(5,    ./bin/pyspark)
(6,    ./bin/run-example SparkPi)
(7,    MASTER=spark://host:7077 ./bin/run-example SparkPi)
(7,Please see the guidance on how to)
(12,for detailed guidance on building for a particular distribution of Hadoop, incl
```

```
In [26]: val lineLen = file_new_log.
           filter(line => line != "").
           map(line => line.split(" ").length).
           cache()
```

```
In [28]: val stddev = lineLen.stdev()
         val mean = lineLen.mean()
         println("stddev: " + stddev + ", mean: " + mean)
```

```
stddev: 3.653613127971938, mean: 7.866666666666665
```

```
In [29]: file_new_log.
           filter(line => line != "").
           map(line => (line.split(" ").length, line)).
           filter{case (k, v) => math.abs(k - mean) < stddev}.
           take(10).
           foreach(println)
```

```
(6,Spark is built using [Apache Maven](http://maven.apache.org/).)
(8,To build Spark and its example programs, run:)
(5,    ./bin/pyspark)
(6,    ./bin/run-example SparkPi)
(7,    MASTER=spark://host:7077 ./bin/run-example SparkPi)
(7,Please see the guidance on how to)
(5,    ./bin/spark-shell)
(8,[run tests for a module, or individual tests](https://cwiki.apache.org/confluenc
(8,building for particular Hive and Hive Thriftserver distributions.)
(8,    build/mvn -DskipTests clean package)
```

```
In [30]: file_new_log.
           filter(line => line != "").
           map(line => (line.split(" ").length, line)).
           filter{case (k, v) => math.abs(k - mean) > stddev}.
           take(10).
           foreach(println)
```

```
(12,The easiest way to start using Spark is through the Scala shell:)
(4,["Specifying the Hadoop Version"](http://spark.apache.org/docs/latest/building-s
(3,# Apache Spark)
(12,for detailed guidance on building for a particular distribution of Hadoop, incl
(3,## Online Documentation)
(14,Spark is a fast and general cluster computing system for Big Data. It provides)
(13,Many of the example programs print usage help if no params are given.)
(13,high-level APIs in Scala, Java, Python, and R, and an optimized engine that)
(2,## Configuration)
(13,(You do not need to do this if you downloaded a pre-built package.))
```