# Question1: write to multiple outputs

**How can you write to multiple outputs dependent on the key using Spark in a single Job.**

## Answer:

import org.apache.hadoop.io.NullWritable

import org.apache.spark.\_

import org.apache.spark.SparkContext.\_

import org.apache.hadoop.mapred.lib.MultipleTextOutputFormat

class RDDMultipleTextOutputFormat extends MultipleTextOutputFormat[Any, Any] {

override def generateActualKey(key: Any, value: Any): Any =

NullWritable.get()

override def generateFileNameForKeyValue(key: Any, value: Any, name: String): String =

key.asInstanceOf[String]

}

object Split {

def main(args: Array[String]) {

val conf = new SparkConf().setAppName("Split" + args(1))

val sc = new SparkContext(conf)

sc.textFile("input/path")

.map(a => (k, v)) // Your own implementation

.partitionBy(new HashPartitioner(num))

.saveAsHadoopFile("output/path", classOf[String], classOf[String],

classOf[RDDMultipleTextOutputFormat])

spark.stop()

}

}

Just saw similar answer above, but actually we don't need customized partitions. The MultipleTextOutputFormat will create file for each key. It is ok that multiple record with same keys fall into the same partition.

new HashPartitioner(num), where the num is the partition number you want. In case you have a big number of different keys, you can set number to big. In this case, each partition will not open too many hdfs file handlers.

# Question2:Serialization problem

This is a working code example:

object working extends App {

val list = List(1,2,3)

val rddList = Spark.ctx.parallelize(list)

//calling function outside closure

val after = rddList.map(someFunc(\_))

def someFunc(a:Int) = a+1

after.collect().map(println(\_))

}

This is the non-working example :

object NOTworking extends App {

new testing().doIT

}

//adding extends Serializable wont help

class testing {

val list = List(1,2,3)

val rddList = Spark.ctx.parallelize(list)

def doIT = {

//again calling the fucntion someFunc

val after = rddList.map(someFunc(\_))

//this will crash (spark lazy)

after.collect().map(println(\_))

}

def someFunc(a:Int) = a+1

}

## Answer1:

I don't think the other answer is entirely correct. [RDDs are indeed serializable](https://github.com/apache/spark/blob/master/core/src/main/scala/org/apache/spark/rdd/RDD.scala#L78), so this is not what's causing your task to fail.

Spark is a distributed computing engine and its main abstraction is a resilient distributed dataset (**RDD**), which can be viewed as a distributed collection. Basically, RDD's elements are partitioned across the nodes of the cluster, but Spark abstracts this away from the user, letting the user interact with the RDD (collection) as if it were a local one.

Not to get into too many details, but when you run different transformations on a RDD (map, flatMap, filter and others), your transformation code (closure) is:

1. serialized on the driver node,
2. shipped to the appropriate nodes in the cluster,
3. deserialized,
4. and finally executed on the nodes

You can of course run this locally (as in your example), but all those phases (apart from shipping over network) still occur. [This lets you catch any bugs even before deploying to production]

What happens in your second case is that you are calling a method, defined in class testing from inside the map function. Spark sees that and since methods cannot be serialized on their own, Spark tries to serialize *the whole* testing class, so that the code will still work when executed in another JVM. You have two possibilities:

Either you make class testing serializable, so the whole class can be serialized by Spark:

import org.apache.spark.{SparkContext,SparkConf}

object Spark {

val ctx = new SparkContext(new SparkConf().setAppName("test").setMaster("local[\*]"))

}

object NOTworking extends App {

new Test().doIT

}

class Test extends java.io.Serializable {

val rddList = Spark.ctx.parallelize(List(1,2,3))

def doIT() = {

val after = rddList.map(someFunc)

after.collect().foreach(println)

}

def someFunc(a: Int) = a + 1

}

or you make someFunc function instead of a method (functions are objects in Scala), so that Spark will be able to serialize it:

import org.apache.spark.{SparkContext,SparkConf}

object Spark {

val ctx = new SparkContext(new SparkConf().setAppName("test").setMaster("local[\*]"))

}

object NOTworking extends App {

new Test().doIT

}

class Test {

val rddList = Spark.ctx.parallelize(List(1,2,3))

def doIT() = {

val after = rddList.map(someFunc)

after.collect().foreach(println)

}

val someFunc = (a: Int) => a + 1

}

Similar, but not the same problem with class serialization can be of interest to you and you can read on it [in this Spark Summit 2013 presentation](http://spark-summit.org/wp-content/uploads/2013/10/McDonough-spark-tutorial_spark-summit-2013.pptx).

As a side note, you can rewrite rddList.map(someFunc(\_)) to rddList.map(someFunc), they are exactly the same. Usually, the second is preferred as it's less verbose and cleaner to read.

EDIT (2015-03-15): [SPARK-5307](https://issues.apache.org/jira/browse/SPARK-5307) introduced **SerializationDebugger** and Spark 1.3.0 is the first version to use it. It adds serialization path to a *NotSerializableException*. When a NotSerializableException is encountered, the debugger visits the object graph to find the path towards the object that cannot be serialized, and constructs information to help user to find the object.

In OP's case, this is what gets printed to stdout:

Serialization stack:

- object not serializable (class: testing, value: testing@2dfe2f00)

- field (class: testing$$anonfun$1, name: $outer, type: class testing)

- object (class testing$$anonfun$1, <function1>)

## Answer2:

[Grega's answer](http://stackoverflow.com/a/22596875/2698114) is great in explaining why the original code does not work and two ways to fix the issue. However, this solution is not very flexible; consider the case where your closure includes a method call on a non-Serializable class that you have no control over. You can neither add the Serializable tag to this class nor change the underlying implementation to change the method into a function.

[Nilesh](http://stackoverflow.com/a/23053760/2698114) presents a great workaround for this, but the solution can be made both more concise and general:

def genMapper[A, B](f: A => B): A => B = {

val locker = com.twitter.chill.MeatLocker(f)

x => locker.get.apply(x)

}

This function-serializer can then be used to automatically wrap closures and method calls:

rdd map genMapper(someFunc)

This technique also has the benefit of not requiring the additional Shark dependencies in order to access KryoSerializationWrapper, since Twitter's Chill is already pulled in by core Spark

# Question3:OOME

Spark java.lang.OutOfMemoryError: Java heap space

## Answer:

few suggestions:

* If your nodes have 6g, then use 6g rather than 4g, spark.executor.memory=6g. Make sure**you're using all the memory** by checking the UI (it will say how much mem you're using)
* Try using more partitions, you should have 2 - 4 per CPU. IME increasing the number of partitions is often the easiest way to make a program more stable (and often faster). For huge amounts of data you may need way more than 4 per CPU, I've had to use 8000 partitions in some cases!
* Decrease the **fraction of memory reserved for caching**, using spark.storage.memoryFraction. If you don't use cache() or persist in your code, this might as well be 0. It's default is 0.6, which means you only get 0.4 \* 4g memory for your heap. IME reducing the mem frac often makes OOMs go away. **UPDATE:** From spark 1.6 apparently we will no longer need to play with these values, spark will determine them automatically.
* Similar to above but **shuffle memory fraction**. If your job need much shuffle memory then set it to a lower value (this might cause your shuffles to spill to disk which can have catastrophic impact on speed). Sometimes when it's a shuffle operation that's OOMing you need to do the opposite i.e. set it to something large, like 0.8, or make sure you allow your shuffles to spill to disk (it's the default since 1.0.0).
* Watch out for **memory leaks**, these are often caused by accidentally closing over objects you don't need in your lambdas. The way to diagnose is to look out for the "task serialized as XXX bytes" in the logs, if XXX is larger than a few k or more than an MB, you may have a memory leak. See <http://stackoverflow.com/a/25270600/1586965>
* Related to above; use **broadcast variables** if you really do need large objects.

# Question4:

Apr 27, 2016 12:14:10 PM org.apache.catalina.loader.WebappClassLoaderBase loadClass

INFO: Illegal access: this web application instance has been stopped already. Could not load org.apache.spark.deploy.client.AppClient$ClientEndpoint$$anon$2$$anonfun$run$1. The eventual following stack trace is caused by an error thrown for debugging purposes as well as to attempt to terminate the thread which caused the illegal access, and has no functional impact.

java.lang.IllegalStateException

at org.apache.catalina.loader.WebappClassLoaderBase.loadClass(WebappClassLoaderBase.java:1743)

at org.apache.catalina.loader.WebappClassLoaderBase.loadClass(WebappClassLoaderBase.java:1701)

at org.apache.spark.deploy.client.AppClient$ClientEndpoint$$anon$2.run(AppClient.scala:125)

at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:471)

at java.util.concurrent.FutureTask.runAndReset(FutureTask.java:304)

at java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.access$301(ScheduledThreadPoolExecutor.java:178)

at java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.run(ScheduledThreadPoolExecutor.java:293)

at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)

at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)

at java.lang.Thread.run(Thread.java:745)

Apr 27, 2016 12:14:13 PM org.apache.catalina.loader.WebappClassLoaderBase loadClass

INFO: Illegal access: this web application instance has been stopped already. Could not load org.apache.spark.Heartbeat. The eventual following stack trace is caused by an error thrown for debugging purposes as well as to attempt to terminate the thread which caused the illegal access, and has no functional impact.

java.lang.IllegalStateException

at org.apache.catalina.loader.WebappClassLoaderBase.loadClass(WebappClassLoaderBase.java:1743)

at org.apache.catalina.loader.WebappClassLoaderBase.loadClass(WebappClassLoaderBase.java:1701)

at java.lang.Class.forName0(Native Method)

at java.lang.Class.forName(Class.java:274)

at akka.util.ClassLoaderObjectInputStream.resolveClass(ClassLoaderObjectInputStream.scala:18)

at java.io.ObjectInputStream.readNonProxyDesc(ObjectInputStream.java:1612)

at java.io.ObjectInputStream.readClassDesc(ObjectInputStream.java:1517)

at java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:1771)

at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1350)

at java.io.ObjectInputStream.defaultReadFields(ObjectInputStream.java:1990)

at java.io.ObjectInputStream.readSerialData(ObjectInputStream.java:1915)

at java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:1798)

at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1350)

at java.io.ObjectInputStream.readObject(ObjectInputStream.java:370)

at akka.serialization.JavaSerializer$$anonfun$1.apply(Serializer.scala:136)

at scala.util.DynamicVariable.withValue(DynamicVariable.scala:57)

at akka.serialization.JavaSerializer.fromBinary(Serializer.scala:136)

at akka.serialization.Serialization$$anonfun$deserialize$1.apply(Serialization.scala:104)

at scala.util.Try$.apply(Try.scala:161)

at akka.serialization.Serialization.deserialize(Serialization.scala:98)

at akka.remote.MessageSerializer$.deserialize(MessageSerializer.scala:23)

at akka.remote.DefaultMessageDispatcher.payload$lzycompute$1(Endpoint.scala:58)

at akka.remote.DefaultMessageDispatcher.payload$1(Endpoint.scala:58)

at akka.remote.DefaultMessageDispatcher.dispatch(Endpoint.scala:76)

at akka.remote.EndpointReader$$anonfun$receive$2.applyOrElse(Endpoint.scala:935)

at akka.actor.Actor$class.aroundReceive(Actor.scala:467)

at akka.remote.EndpointActor.aroundReceive(Endpoint.scala:411)

at akka.actor.ActorCell.receiveMessage(ActorCell.scala:516)

at akka.actor.ActorCell.invoke(ActorCell.scala:487)

at akka.dispatch.Mailbox.processMailbox(Mailbox.scala:238)

at akka.dispatch.Mailbox.run(Mailbox.scala:220)

at akka.dispatch.ForkJoinExecutorConfigurator$AkkaForkJoinTask.exec(AbstractDispatcher.scala:397)

at scala.concurrent.forkjoin.ForkJoinTask.doExec(ForkJoinTask.java:260)

at scala.concurrent.forkjoin.ForkJoinPool$WorkQueue.runTask(ForkJoinPool.java:1339)

at scala.concurrent.forkjoin.ForkJoinPool.runWorker(ForkJoinPool.java:1979)

at scala.concurrent.forkjoin.ForkJoinWorkerThread.run(ForkJoinWorkerThread.java:107)

## Answer:

spark.checkpoint.dir配置成了hadoop standby节点，导致故障。