**Introduction**

In the previous parts of this series, we have shown how to write functions as both [combinations of dplyr verbs](https://jozef.io/r202-spark-r-dplyr-verbs/), [SQL query generators](https://jozef.io/r203-spark-r-sql/) that can be executed by Spark and [how to use the lower-level API](https://jozef.io/r204-spark-r-invoke-scala/) to invoke methods on Java object references from R.

In this fifth part, we will look into more details around sparklyr’s invoke() API, investigate available methods for different classes of objects using the Java reflection API and look under the hood of the sparklyr interface mechanism with invoke logging.

**Contents**

1. [Preparation](https://jozef.io/r205-spark-r-invoke-scala-2/#preparation)
2. [Examining available methods from R](https://jozef.io/r205-spark-r-invoke-scala-2/#examining-available-methods-from-r)
3. [Using the Java reflection API to list the available methods](https://jozef.io/r205-spark-r-invoke-scala-2/#using-the-java-reflection-api-to-list-the-available-methods)
4. [Investigating DataSet and SparkContext class methods](https://jozef.io/r205-spark-r-invoke-scala-2/#investigating-dataset-and-sparkcontext-class-methods)
5. [How sparklyr communicates with Spark, invoke logging](https://jozef.io/r205-spark-r-invoke-scala-2/#how-sparklyr-communicates-with-spark-invoke-logging)
6. [Conclusion](https://jozef.io/r205-spark-r-invoke-scala-2/#conclusion)
7. [References](https://jozef.io/r205-spark-r-invoke-scala-2/#references)

**Preparation**

The full setup of Spark and sparklyr is not in the scope of this post, please check the [first one](https://jozef.io/r201-spark-r-1/#setting-up-spark-with-r-and-sparklyr) for some setup instructions and a ready-made Docker image.

If you have docker available, running

docker run -d -p 8787:8787 -e PASSWORD=pass --name rstudio jozefhajnala/sparkly:add-rstudio

Should make RStudio available by navigating to [http://localhost:8787](http://localhost:8787/) in your browser. You can then use the user name rstudio and password pass to login and continue experimenting with the code in this post.

# Load packages

suppressPackageStartupMessages({

library(sparklyr)

library(dplyr)

library(nycflights13)

})

# Connect and copy the flights dataset to the instance

sc <- sparklyr::spark\_connect(master = "local")

tbl\_flights <- dplyr::copy\_to(sc, nycflights13::flights, "flights")

**Examining available methods from R**

If you did not do so, it is recommended to read the [previous part](https://jozef.io/r204-spark-r-invoke-scala/) of this series before this one to get a quick overview of the invoke() API.

**Using the Java reflection API to list the available methods**

The invoke() interface is powerful, but also a bit hidden from the eyes as we do not immediately know what methods are available for which object classes. We can circumvent that using the getMethods method which (in short) returns an array of Method objects reflecting public member methods of the class.

For instance, retrieving a list of methods for the org.apache.spark.SparkContext class:

mthds <- sc %>% spark\_context() %>%

invoke("getClass") %>%

invoke("getMethods")

head(mthds)

## [[1]]

##

## java.lang.reflect.Method

## public org.apache.spark.util.CallSite [org.apache.spark.SparkContext.org](http://org.apache.spark.SparkContext.org)$apache$spark$SparkContext$$creationSite()

##

## [[2]]

##

## java.lang.reflect.Method

## public org.apache.spark.SparkConf [org.apache.spark.SparkContext.org](http://org.apache.spark.SparkContext.org)$apache$spark$SparkContext$$\_conf()

##

## [[3]]

##

## java.lang.reflect.Method

## public org.apache.spark.SparkEnv [org.apache.spark.SparkContext.org](http://org.apache.spark.SparkContext.org)$apache$spark$SparkContext$$\_env()

##

## [[4]]

##

## java.lang.reflect.Method

## public scala.Option [org.apache.spark.SparkContext.org](http://org.apache.spark.SparkContext.org)$apache$spark$SparkContext$$\_progressBar()

##

## [[5]]

##

## java.lang.reflect.Method

## public scala.Option [org.apache.spark.SparkContext.org](http://org.apache.spark.SparkContext.org)$apache$spark$SparkContext$$\_ui()

##

## [[6]]

##

## java.lang.reflect.Method

## public org.apache.spark.rpc.RpcEndpointRef [org.apache.spark.SparkContext.org](http://org.apache.spark.SparkContext.org)$apache$spark$SparkContext$$\_heartbeatReceiver()

We can see that the invoke() chain has returned a list of Java object references, each of them of class java.lang.reflect.Method. This is a good result, but the output is not very user-friendly from the R user perspective. Let us write a small wrapper that will return a some of the method’s details in a more readable fashion, for instance the return type and an overview of parameters:

getMethodDetails <- function(mthd) {

returnType <- mthd %>% invoke("getReturnType") %>% invoke("toString")

params <- mthd %>% invoke("getParameters")

params <- vapply(params, invoke, "toString", FUN.VALUE = character(1))

c(returnType = returnType, params = paste(params, collapse = ", "))

}

Finally, to get a nice overview, we can make another helper function that will return a named list of methods for an object’s class, including their return types and overview of parameters:

getAvailableMethods <- function(jobj) {

mthds <- jobj %>% invoke("getClass") %>% invoke("getMethods")

nms <- vapply(mthds, invoke, "getName", FUN.VALUE = character(1))

res <- lapply(mthds, getMethodDetails)

names(res) <- nms

res

}

**Investigating DataSet and SparkContext class methods**

Using the above defined function we can explore the methods available to a DataFrame reference, showing a few of the names first:

dfMethods <- tbl\_flights %>% spark\_dataframe() %>%

getAvailableMethods()

# Show some method names:

dfMethodNames <- sort(unique(names(dfMethods)))

head(dfMethodNames, 20)

## [1] "agg" "alias"

## [3] "apply" "as"

## [5] "cache" "checkpoint"

## [7] "coalesce" "col"

## [9] "collect" "collectAsArrowToPython"

## [11] "collectAsList" "collectToPython"

## [13] "colRegex" "columns"

## [15] "count" "createGlobalTempView"

## [17] "createOrReplaceGlobalTempView" "createOrReplaceTempView"

## [19] "createTempView" "crossJoin"

If we would like to see more details we can now investigate further, for instance show different parameter interfaces for the agg method, showing that the agg method has the following parameter interfaces:

sort(vapply(

dfMethods[names(dfMethods) == "agg"],

`[[`, "params",

FUN.VALUE = character(1)

))

## agg

## "java.util.Map arg0"

## agg

## "org.apache.spark.sql.Column arg0, org.apache.spark.sql.Column... arg1"

## agg

## "org.apache.spark.sql.Column arg0, scala.collection.Seq arg1"

## agg

## "scala.collection.immutable.Map arg0"

## agg

## "scala.Tuple2 arg0, scala.collection.Seq> arg1"

Similarly, we can look at a SparkContext class and show some available methods that can be invoked:

scMethods <- sc %>% spark\_context() %>%

getAvailableMethods()

scMethodNames <- sort(unique(names(scMethods)))

head(scMethodNames, 60)

## [1] "$lessinit$greater$default$3" "$lessinit$greater$default$4"

## [3] "$lessinit$greater$default$5" "accumulable"

## [5] "accumulableCollection" "accumulator"

## [7] "addedFiles" "addedJars"

## [9] "addFile" "addJar"

## [11] "addSparkListener" "applicationAttemptId"

## [13] "applicationId" "appName"

## [15] "assertNotStopped" "binaryFiles"

## [17] "binaryFiles$default$2" "binaryRecords"

## [19] "binaryRecords$default$3" "broadcast"

## [21] "cancelAllJobs" "cancelJob"

## [23] "cancelJobGroup" "cancelStage"

## [25] "checkpointDir" "checkpointDir\_$eq"

## [27] "checkpointFile" "clean"

## [29] "clean$default$2" "cleaner"

## [31] "clearCallSite" "clearJobGroup"

## [33] "collectionAccumulator" "conf"

## [35] "createSparkEnv" "dagScheduler"

## [37] "dagScheduler\_$eq" "defaultMinPartitions"

## [39] "defaultParallelism" "deployMode"

## [41] "doubleAccumulator" "emptyRDD"

## [43] "env" "equals"

## [45] "eventLogCodec" "eventLogDir"

## [47] "eventLogger" "executorAllocationManager"

## [49] "executorEnvs" "executorMemory"

## [51] "files" "getAllPools"

## [53] "getCallSite" "getCheckpointDir"

## [55] "getClass" "getConf"

## [57] "getExecutorIds" "getExecutorMemoryStatus"

## [59] "getExecutorThreadDump" "getLocalProperties"

**Using helpers to explore the methods**

We can also use the helper functions to investigate more. For instance, we see that there is a getConf method avaiable to us. Looking at the object reference however does not provide useful information, so we can list the methods for that class and look for "get" methods that would show us the configuration:

spark\_conf <- sc %>% spark\_context() %>% invoke("conf")

spark\_conf\_methods <- spark\_conf %>% getAvailableMethods()

spark\_conf\_get\_methods <- spark\_conf\_methods %>%

names() %>%

grep(pattern = "get", ., value = TRUE) %>%

sort()

spark\_conf\_get\_methods

## [1] "get" "get" "get"

## [4] "getAll" "getAllWithPrefix" "getAppId"

## [7] "getAvroSchema" "getBoolean" "getClass"

## [10] "getDeprecatedConfig" "getDouble" "getenv"

## [13] "getExecutorEnv" "getInt" "getLong"

## [16] "getOption" "getSizeAsBytes" "getSizeAsBytes"

## [19] "getSizeAsBytes" "getSizeAsGb" "getSizeAsGb"

## [22] "getSizeAsKb" "getSizeAsKb" "getSizeAsMb"

## [25] "getSizeAsMb" "getTimeAsMs" "getTimeAsMs"

## [28] "getTimeAsSeconds" "getTimeAsSeconds" "getWithSubstitution"

We see that there is a getAll method that could prove useful, returning a list of tuples and taking no arguments as input:

# Returns a list of tuples, takes no arguments:

spark\_conf\_methods[["getAll"]]

## returnType params

## "class [Lscala.Tuple2;" ""

# Invoke the `getAll` method and look at part of the result

spark\_confs <- spark\_conf %>% invoke("getAll")

spark\_confs <- vapply(spark\_confs, invoke, "toString", FUN.VALUE = character(1))

sort(spark\_confs)[c(2, 3, 12, 14)]

## [1] "([spark.app.name](http://spark.app.name" \t "_blank),sparklyr)"

## [2] "(spark.driver.host,localhost)"

## [3] "(spark.sql.catalogImplementation,hive)"

## [4] "(spark.submit.deployMode,client)"

Looking at [the Scala documentation for the getAll method](https://spark.apache.org/docs/1.6.0/api/java/org/apache/spark/SparkConf.html#getAll()), we actually see that there is information missing on our data – the classes of the objects in the tuple, which in this case is scala.Tuple2[].

We could therefore improve our helper to be more detailed in the return value information.

**Unexported helpers provided by sparklyr**

The sparklyr package itself provides facilities of nature similar to those above, looking at some of them, even though they are not exported:

sparklyr:::jobj\_class(spark\_conf)

## [1] "SparkConf" "Object"

sparklyr:::jobj\_info(spark\_conf)$class

## [1] "org.apache.spark.SparkConf"

capture.output(sparklyr:::jobj\_inspect(spark\_conf)) %>% head(10)

## [1] ""

## [2] " org.apache.spark.SparkConf"

## [3] " org.apache.spark.SparkConf@7ec389e7"

## [4] "Fields:"

## [5] ""

## [6] " java.lang.reflect.Field"

## [7] " private final java.util.concurrent.ConcurrentHashMap [org.apache.spark.SparkConf.org](http://org.apache.spark.SparkConf.org)$apache$spark$SparkConf$$settings"

## [8] ""

## [9] " java.lang.reflect.Field"

## [10] " private transient org.apache.spark.internal.config.ConfigReader [org.apache.spark.SparkConf.org](http://org.apache.spark.SparkConf.org)$apache$spark$SparkConf$$reader"

**How sparklyr communicates with Spark, invoke logging**

Now that we have and overview of the invoke() interface, we can take a look under the hood of sparklyr and see how it actually communicates with the Spark instance. In fact, the communication is a set of invocations that can be very different depending on which of the approches we choose for our purposes.

To obtain the information, we use the sparklyr.log.invoke property. We can choose one of the following 3 values based on our preferences:

* TRUE will use message() to communicate short info on what is being invoked
* "cat" will use cat() to communicate short info on what is being invoked
* "callstack" will use message() to communicate short info on what is being invoked and the callstack

We will use TRUE in our article to keep the output short and easily manageable. First, we will close the previous connection and create a new one with the configuration containing the sparklyr.log.invoke set to TRUE, and copy in the flights dataset:

sparklyr::spark\_disconnect(sc)

## NULL

config <- sparklyr::spark\_config()

config$sparklyr.log.invoke <- TRUE

suppressMessages({

sc <- sparklyr::spark\_connect(master = "local", config = config)

tbl\_flights <- dplyr::copy\_to(sc, nycflights13::flights, "flights")

})

**Using dplyr verbs translated with dbplyr**

Now that the setup is complete, we use the dplyr verb approach to retrieve the count of rows and look the invocations that this entails:

tbl\_flights %>% dplyr::count()

## Invoking sql

## Invoking sql

## Invoking columns

## Invoking isStreaming

## Invoking sql

## Invoking isStreaming

## Invoking sql

## Invoking sparklyr.Utils collect

## Invoking columns

## Invoking schema

## Invoking fields

## Invoking dataType

## Invoking toString

## Invoking name

## Invoking sql

## Invoking columns

## # Source: spark [?? x 1]

## n

##

## 1 336776

We see multiple invocations do the sql method and also the columns method. This makes sense since the dplyr verb approach actually works by translating the commands into Spark SQL via dbplyr and then sends those translated commands to Spark via that interface.

**Using DBI to send queries**

Similarly, we can investigate the invocations that happen when we try to retrieve the same results via the DBI interface:

DBI::dbGetQuery(sc, "SELECT count(1) AS n FROM flights")

## Invoking sql

## Invoking isStreaming

## Invoking sparklyr.Utils collect

## Invoking columns

## Invoking schema

## Invoking fields

## Invoking dataType

## Invoking toString

## Invoking name

## n

## 1 336776

We see slightly fewer invocations compared to the above dplyr approach, but the output is also less processed.

**Using the invoke interface**

Looking at the invocations that get executed using the invoke() interface:

tbl\_flights %>% spark\_dataframe() %>% invoke("count")

## Invoking sql

## Invoking count

## [1] 336776

We see that the amount of invocations is much lower, where the top 3 invocations come from the first part of the pipe. The invoke("count") part translated to exactly one invocation to the count method. We see therefore that the invoke() interface is indeed a more lower-level interface that invokes methods as we request them, with little to none overhead related to translations and other effects.

**Redirecting the invoke logs**

When running R applications that use Spark as a calculation engine, it is useful to get detailed invoke logs for debugging and diagnostic purposes. Implementing such mechanisms, we need to take into consideration how R handles the invoke logs produced by sparklyr. In simple terms, the invoke logs produced when using

* TRUE and "callstack" are created using message(), which means they get sent to the stderr() connection by default
* "cat" are created using cat(), so they get sent to stdout() connection by default

This info can prove useful when redirecting the log information from standard output and standard error to different logging targets.