3 How to Run Spark Application on a Local Machine?

3.1 Write Application Code

Here is an example application code that generates 4 million random alphanumeric string with length 5 and persists them into `outputDir`.

```
/* GenerateNames.scala */
import org.apache.spark.SparkContext
import org.apache.spark.SparkConf
import scala.util.Random

object GenerateNames {
    val outputDir = "/home/jung/sparkapp/output/part"

    def main(args: Array[String]) {
        val conf = new SparkConf()
            .setMaster("local[3]")
            .setAppName("GenerateNames")

        val sc = new SparkContext(conf)

        for (partition <- 0 to 3) {
            val data = Seq.fill(1000000)(Random.alphanumeric.take(5).mkString)
            sc.parallelize(data, 1).saveAsTextFile(outputDir + "_" + partition)
        }
    }
}
```

3.2 Compile Application Code

Our application depends on the Spark API, so we'll also include an sbt configuration file, `build.sbt`, which describes about the dependency. Also, this file adds a repository that Spark API depends on:

```
/* build.sbt */
name := "SparkApp"

version := "0.1"

scalaVersion := "2.11.6"

libraryDependencies += "org.apache.spark" %% "spark-core" % "1.4.1"
```

For `sbt` to work correctly, we will need to layout `GenerateNames.scala` and `build.sbt` files according to the typical directory structure. Your directory layout should look something like below when you type `find` command inside your application directory.

```
# Inside /home/jung/sparkapp/ directory:
$ find .
.
./build.sbt
./src
./src/main
./src/main/scala
./src/main/scala/GenerateNames.scala
```

Once that is in place, we can create a JAR package containing the application's code.

```
# Package a jar containing your application.
# Inside /home/jung/sparkapp/ directory:
$ sbt package
...
[info] Packaging {..}/{..}/target/scala-2.11/sparkapp_2.11-0.1.jar
[info] Done packaging.
[success] Total time: ...
```

3.3 Run Application

Finally, we can run the application using:
`/home/jung/spark-1.4.1-bin-hadoop2.6/bin/spark-submit` script.

```
$ /home/jung/spark-1.4.1-bin-hadoop2.6/bin/spark-submit \
    --class GenerateNames
    /home/jung/sparkapp/target/scala-2.11/sparkapp_2.11-0.1.jar
```

Inside the `/home/jung/output/` directory, we can see that there are 4 directories:

```
part_0  part_1  part_2  part_3
```

Each directory contains:

```
part_00000  _SUCCESS
```

And each `part_00000` contains 1 million names.

4 Details about Submitting Applications

In the previous section, we showed you how to run a simple Spark application on a local machine. In this section, we will explain about the details of submitting a Spark application.

4.1 Bundling Application's Dependencies

In the above example, we wrote `build.sbt` file and used `sbt package` command to create a assembly jar (`sparkapp_2.11-0.1.jar` in our example). Why do we need this process? The reason is because if your code (`GenerateNames.scala` in our example) depends on other projects, such as Spark, you will need to package them alongside your application in order to distribute the code to a Spark cluster (which is our final goal of this tutorial).

4.2 Launching Applications with spark-submit

Once you have an assembled jar, you can call the `spark-submit` script to launch the application. This script takes care of setting up the classpath with Spark and its dependencies, and can support different cluster managers and deploy modes that Spark supports:

```
$ /home/jung/spark-1.4.1-bin-hadoop2.6/bin/spark-submit \
    --class MAIN_CLASS \
    --master MASTER_URL \
    --conf KEY=VALUE \
    ...  # other options
    APPLICATION_JAR \
    [APPLICATION_ARGUMENTS]
```

Some of the commonly used options are:

- `--class`: The entry point for your application.
- `--master`: The master URL for the cluster.
- `--deploy-mode`: Whether to deploy your driver on the worker nodes (`cluster`) or locally as an external client (`client`) (default: `client`).
  - `--deploy-mode client`
  - `--deploy-mode cluster`
- `--total-executor-cores`: The total number of cores worker nodes can have.
  - `--total-executor-cores 3`
- `--executor-memory`: The size of memory each worker node can have.
  - `--executor-memory 512m`
  - `--executor-memory 2g`
- `--conf`: Configuration.
  - `spark.executor.extraJavaOptions=-XX:+PrintGCDetails`
  - `spark.executor.extraJavaOptions=-XX+:PrintGCTimeStamps`

  - `spark.executor.extraJavaOptions=-XX:+HeapDumpOnOutOfMemoryError`

- `APPLICATION_JAR`: Path to a bundled jar including your application and all dependencies. The URL must be globally visible inside of your cluster.
- `APPLICATION_ARGUMETNS`: Arguments passed to the main method of your main class, if any.

4.3 Master URLs

The master URL passed to Spark can be in one of the following formats:

- `local`: Run Spark locally with one worker thread.
- `local[K]`: Run Spark locally with `K` worker threads (ideally, set this to the number of cores on your machines).
- `local[*]`: Run Spark locally with as many worker threads as logical cores on your machine.
- `spark://HOST:PORT`: Connect to the given Spark standalone cluster master. The port must be whichever one you master is configured to use, which is 7077 by default.