How to Run Spark Application

Contents

1 Intro **2**

2 How to Install Spark on a Local Machine? **2**
- 2.1 On Ubuntu 14.04 2

3 How to Run Spark Application on a Local Machine? **3**
- 3.1 Write Application Code 3
- 3.2 Compile Application Code 3
- 3.3 Run Application 4

4 Details about Submitting Applications **5**
- 4.1 Bundling Application's Dependencies 5
- 4.2 Launching Applications with spark-submit 5
- 4.3 Master URLs 6

5 How to Run Spark Application on EC2? **7**
- 5.1 Before You Start 7
- 5.2 Launch a Cluster 7
- 5.3 Running Applications 8
  - 5.3.1 Write Application Code 8
  - 5.3.2 Compile Application Code 9
  - 5.3.3 Deploy Code 9
  - 5.3.4 Run Application 9
  - 5.3.5 Check Output 10
  - 5.3.6 Uploading Input Data to EC2 Instance 10
- 5.4 Terminating a Cluster 10

6 References **11**

1 Intro

This tutorial simply concatenates the parts of documents from the following links:

- Quick Start
  `http://spark.apache.org/docs/latest/quick-start.html`
- Submitting Applications
  `http://spark.apache.org/docs/1.4.1/submitting-applications.html`
- Cluster Mode Overview
  `http://spark.apache.org/docs/1.4.1/cluster-overview.html`
- Running Spark on EC2
  `http://spark.apache.org/docs/1.4.1/ec2-scripts.html`

2 How to Install Spark on a Local Machine?

2.1 On Ubuntu 14.04

1. Install the latest version of Java Development Kit.
2. Install the latest version of Scala.
3. Download and unzip spark-1.4.1-bin-hadoop2.6.tgz, which is prebuilt Spark for Hadoop 2.6 or later.
4. Try running Spark interactive shell, inside the spark-1.4.1-bin-hadoop2.6 directory, by typing:
   ```
   $ ./bin/spark-shell
   ```

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.

5 How to Run Spark Application on EC2?

The `spark-ec2` script, located inside `spark-1.4.1-bin-hadoop2.6/ec2/` directory on your local machine, allows you to launch, manage, and shut down Spark clusters on Amazon EC2. It automatically sets up Spark and HDFS on the cluster for you.

5.1 Before You Start

EC2 Key Pair
Create an EC2 key pair so that you can SSH into a master or slave instances in a Spark cluster later after you launch the cluster. This can be done through AWS console. When the private key is downloaded to your local machine, set the permissions for the private key file to 400 and move the file to a safe location. For example:

```
$ sudo chmod 400 jung-keypair-useast1.pem
$ mv jung-keypair-useast1.pem ~/.ssh
```

AWS Access Keys
Create an AWS access keys from AWS console and set the environment variables `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` to your Amazon access key ID and secret access key. For example, inside your `~/.bashrc` file, add the lines below:

```
export AWS_ACCESS_KEY_ID=ABCDE1234567890
export AWS_SECRET_ACCESS_KEY=AaBbCcDdEe1234567890!@#$%^&*()
```

5.2 Launch a Cluster

Here is an example of how you would launch a Spark cluster with 3 slave nodes and 1 master node:

```
$ /home/jung/spark-1.4.1-bin-hadoop2.6/ec2/spark-ec2 \
    --key-pair=jung-keypair-useast1 \
    --identity-file=/home/jung/.ssh/jung-keypair-useast1.pem \
    --region=us-east-1 \
    --instance-type=m3.2xlarge \
    --slaves=3 \
    launch jung-ec2-useast1-mycluster
```

Run `/home/jung/spark-1.4.1-bin-hadoop2.6/ec2/spark-ec2 --help` to see more usage options. Here are some of the options:

- `--slaves=SLAVES`
  specifies the number of slaves to launch (default: 1).
- `--key-pair=KEY_PAIR`
  specifies which key pair to use on instances.
- `--identity-file=IDENTITY_FILE`
  specifies the SSH private key file to use for logging into instances.

- `--instance-type=INSTANCE_TYPE`
  specifies the type of instance to launch (default: `m1.large`, which has 2 cores and 7.5 GB RAM).

- `--region=REGION`
  specifies an EC2 region to launch instances in. The region should be the same as the region where you have created your EC2 key pair.

- `--zone=ZONE`
  specifies an availability zone to launch instances in.

After you launch a Spark cluster, you can monitor the instances through AWS console or `http://<master-public-dns>:8080`. The `<master-public-dns>` can be obtained from AWS console as well.

5.3 Running Applications

5.3.1 Write Application Code

Let's use the same application code that we have used in **How to Run Spark Application on a Local Machine** section but with a small tweak.

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

object GenerateNames {
    val SPARK_MASTER = "spark://ec2-54-158-141-105.compute-1.amazonaws.com:7077"
    val HDFS = "hdfs://ec2-54-158-141-105.compute-1.amazonaws.com:9000"
    val outputDir = HDFS + "/output/part"
    def main(args: Array[String]) {
        val conf = new SparkConf()
            .setMaster(SPARK_MASTER)
            .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)
        }
    }
}
```

The only changes that I made here are:

```
SPARK_MASTER = "spark://ec2-54-158-141-105.compute-1.amazonaws.com:7077"
HDFS = "hdfs://ec2-54-158-141-105.compute-1.amazonaws.com:9000"
```

where `ec2-54-158-141-105.compute-1.amazonaws.com` is the public DNS of the master node, which could be found in the AWS console after you launch a cluster. And the `HDFS` is the master node's access point to HDFS, in which we will persist ouput.

5.3.2 Compile Application Code

On your local machine, inside the `/home/jung/sparkapp/` directory, which contains application source code, type: `$ sbt package` to create an uber JAR.

5.3.3 Deploy Code

Let's deploy our application to our Spark cluster. First, we need to `scp` the JAR file we created to the master instance.

```
$ scp -i /home/jung/.ssh/jung-keypair-useast1.pem \
    sparkapp_2.11-0.1.jar \
    ec2-user@ec2-54-158-141-105.compute-1.amazonaws.com:/home/ec2-user/
```

Then, `ssh` into the master instance.

```
$ ssh -i /home/jung/.ssh/jung-keypair-useast1.pem \
    ec2-user@ec2-54-158-141-105.compute-1.amazonaws.com
```

If you want to see event logs after your application is finished, you need to modify the `/root/spark/conf/spark-defaults.conf` file as below:

```
# add line
spark.eventLog.enabled      true
```

Then, disseminate the configuration file to worker nodes by typing:

```
$ sudo /root/spark-ec2/copy-dir /root/spark/conf
```

5.3.4 Run Application

Finally, we can run the application that we deployed on the cluster. Inside the master node, type:

```
$ /root/spark/bin/spark-submit \
    --class GenerateNames \
    /home/ec2-user/sparkapp_2.11-0.1.jar
```

5.3.5 Check Output

Let's check the `output` directory inside the HDFS.

```
$ sudo /root/ephemeral-hdfs/bin/hadoop fs -ls /
```

This command will print out something like this:

```
Warning: $HADOOP_HOME is deprecated.

Found 1 items
drwxr-xr-x - root supergroup 0 2015-03-09 06:24 /output
```

Let's copy the directory into `/home/ec2-user/`.

```
$ sudo /root/ephemeral-hdfs/bin/hadoop fs -get \
    /output /home/ec2-user
```

Now you can view names that your application generated.
Also, here are some `hdfs` commands that you might find them useful:

```
# removes all files in hdfs
$ sudo /root/ephemeral-hdfs/bin/hadoop fs -rmr /*

# puts foo.txt into hdfs
$ sudo /root/ephemeral-hdfs/bin/hadoop fs -put \
    /home/ec2-user/foo.txt /

# retrieves the size of all data in hdfs
$ sudo /root/ephemeral-hdfs/bin/hadoop fs -du -s -h /
```

In addition to checking outputs in the HDFS, you can also view Spark's web UI on a browser by giving the following address:
`spark://ec2-54-158-141-105.compute-1.amazonaws.com:8080` , where
`ec2-54-158-141-105.compute-1.amazonaws.com` is the public DNS of the master node.

5.3.6 Uploading Input Data to EC2 Instance

If you have an input data and would like to upload it to your EC2 instance, you could use the same way as we deployed our Spark application to the Spark cluster.

```
$ scp -i /home/jung/.ssh/jung-keypair-useast1.pem \
    some_input_file.txt \
    ec2-user@ec2-54-158-141-105.compute-1.amazonaws.com:/home/ec2-user/
```

5.4 Terminating a Cluster

To terminate the cluster, type:

```
$ /home/jung/spark-1.4.1-bin-hadoop2.6/ec2/spark-ec2 destroy CLUSTER_NAME
```

where `CLUSTER_NAME` is `jung-ec2-useast1-mycluster` for our example.

6 References

[1] http://spark.apache.org/docs/latest/quick-start.html

[2] http://spark.apache.org/docs/1.4.1/submitting-applications.html

[3] http://spark.apache.org/docs/1.4.1/cluster-overview.html

[4] http://spark.apache.org/docs/1.4.1/ec2-scripts.html