Hadoop Programming

Now we will see how to develop an Hadoop program in a terminal. In order to manage the complexities of Java and Hadoop dependencies linking, We will leverage Maven.

In the following we will see how:

- Compile an Hadoop program on your machine using Maven and prepare a JAR file to be executed on Hadoop.
- Run the Hadoop program on your cluster's machines.

Hadoop and Maven

For running a Hadoop job written in Java, we need to create a jar file with the compiled classes and also include other dependencies of our code. This can be very time consuming if we do not automatise the tasks.

Apache Maven (http://maven.apache.org/) allows a project to build using its *project object model* (POM) and a set of plugins that are shared by all projects using Maven, providing a uniform **build system**.

If you want to install Maven on your virtual machines, please use the following command:

```
sudo apt-get install maven
```

Let see how to configure a Apache Maven `pom.xml` file to obtain a single jar with code plus dependencies ready to be executed on our Hadoop installation.

1. Start with an empty `pom.xml` file

Let's start with a simple `pom.xml`. `archetype:generate` can make the work for us to start the configuration process. It creates the folder structure and a `pom.xml` with the minimum data required. Change `groupId` and `artifactId` with your requirements.

```
mvn archetype:generate -DgroupId=it.unipi.hadoop -DartifactId=wordcount \
                       -DarchetypeArtifactId=maven-archetype-quickstart \
                       -DinteractiveMode=false
```

2. Maven directory layout

Maven will create a folder named after the provided `artifactId` (in our case, `wordcount`), including a minimal `pom.xml` file and a folder structure like the following:

```
wordcount/src
├── main
│   └── java
│       └── it
│           └── unipi
│               └── hadoop
│                   └── App.java
└── test
    └── java
        └── it
            └── unipi
                └── hadoop
                        └── AppTest.java
```

Delete/ignore the `test` folder, as well as the `App.java` file. We will write our own Java file.

```
cd wordcount
rm -rf src/test
rm -rf src/main/java/it/unipi/hadoop/App.java
```

3. Update POM

Add the plugin configuration and the following. Version numbers can vary, we currently use version 3.1.3.

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.0</version>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
        <encoding>${project.build.sourceEncoding}</encoding>
      </configuration>
    </plugin>

    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-jar-plugin</artifactId>
      <configuration>
        <archive>
          <manifest>
            <addClasspath>true</addClasspath>
          </manifest>
        </archive>
      </configuration>
    </plugin>
  </plugins>
</build>

<dependencies>

  <dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-mapreduce-client-jobclient</artifactId>
    <version>3.1.3</version>
  </dependency>

  <dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-common</artifactId>
    <version>3.1.3</version>
  </dependency>

  <dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-hdfs-client</artifactId>
```

```
      <version>3.1.3</version>
    </dependency>

    <dependency>
      <groupId>org.apache.hadoop</groupId>
      <artifactId>hadoop-mapreduce-client-app</artifactId>
      <version>3.1.3</version>
    </dependency>

    [...]

  </dependencies>
```

4. Write code

You can write the source code of your application with any text editor. Here we will use the GNU `nano` (https://www.nano-editor.org) editor.

```
nano src/main/java/it/unipi/hadoop/WordCount.java
```

Edit the Java file with content, then close the file (Ctrl+O followed by Ctrl+X).

```
package it.unipi.hadoop;

import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;

public class WordCount {

  public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }

  public static class IntSumReducer
       extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context
                       ) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
```

```
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
    if (otherArgs.length < 2) {
      System.err.println("Usage: wordcount <in> [<in>...] <out>");
      System.exit(2);
    }
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    for (int i = 0; i < otherArgs.length - 1; ++i) {
      FileInputFormat.addInputPath(job, new Path(otherArgs[i]));
    }
    FileOutputFormat.setOutputPath(job,
      new Path(otherArgs[otherArgs.length - 1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

5. Compile and package

In the folder containing your `pom.xml` file, run the following command.

```
mvn clean package
```

If compilation and packaging runs smoothly, we will get a new `target` folder, containing the jar file to use to dispatch our application on any Hadoop cluster.

This jar file must be copied to one of your virtual machines hosting the Hadoop cluster.

```
scp target/wordcount-1.0-SNAPSHOT.jar hadoop@<vm ip address>:
```

How to execute a Hadoop programs from Terminal

To test the Hadoop program we just wrote, we will use a small input data set called `pg100.txt` (../data/pg100.txt).

1. Open a terminal and run the following commands:

```
hadoop fs -put pg100.txt pg100.txt
hadoop jar wordcount-1.0-SNAPSHOT.jar it.unipi.hadoop.WordCount pg100.txt output
```

2. Run the following command:

```
hadoop fs -ls output
```

   You should see an output file for each reducer. Since there was only one reducer for this job, you should only see one `part-r-*` file. Note that sometimes the files will be called `part-NNNNN`, and sometimes they'll be called `part-r-NNNNN`.

3. Run the following command:

```
hadoop fs -cat output/part* | head
```

   You should see the output.