Table 6-1. GenericOptionsParser and ToolRunner options

Option	Description
`-D property=value`	Sets the given Hadoop configuration property to the given value. Overrides any default or site properties in the configuration and any properties set via the `-conf` option.
`-conf filename ...`	Adds the given files to the list of resources in the configuration. This is a convenient way to set site properties or to set a number of properties at once.
`-fs uri`	Sets the default filesystem to the given URI. Shortcut for `-D fs.defaultFS=uri`.
`-jt host:port`	Sets the YARN resource manager to the given host and port. (In Hadoop 1, it sets the jobtracker address, hence the option name.) Shortcut for `-D yarn.resourcemanager.address=host:port`.
`-files file1,file2,...`	Copies the specified files from the local filesystem (or any filesystem if a scheme is specified) to the shared filesystem used by MapReduce (usually HDFS) and makes them available to MapReduce programs in the task's working directory. (See "Distributed Cache" on page 274 for more on the distributed cache mechanism for copying files to machines in the cluster.)
`-archives archive1,archive2,...`	Copies the specified archives from the local filesystem (or any filesystem if a scheme is specified) to the shared filesystem used by MapReduce (usually HDFS), unarchives them, and makes them available to MapReduce programs in the task's working directory.
`-libjars jar1,jar2,...`	Copies the specified JAR files from the local filesystem (or any filesystem if a scheme is specified) to the shared filesystem used by MapReduce (usually HDFS) and adds them to the MapReduce task's classpath. This option is a useful way of shipping JAR files that a job is dependent on.

Writing a Unit Test with MRUnit

The map and reduce functions in MapReduce are easy to test in isolation, which is a consequence of their functional style. MRUnit is a testing library that makes it easy to pass known inputs to a mapper or a reducer and check that the outputs are as expected. MRUnit is used in conjunction with a standard test execution framework, such as JUnit, so you can run the tests for MapReduce jobs in your normal development environment. For example, all of the tests described here can be run from within an IDE by following the instructions in "Setting Up the Development Environment" on page 144.

Mapper

The test for the mapper is shown in Example 6-5.

Example 6-5. Unit test for MaxTemperatureMapper

```
import java.io.IOException;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mrunit.mapreduce.MapDriver;
import org.junit.*;

public class MaxTemperatureMapperTest {

  @Test
  public void processesValidRecord() throws IOException, InterruptedException {
    Text value = new Text("0043011990999991950051518004+68750+023550FM-12+0382" +
                          // Year ^^^^
        "99999V0203201N00261220001CN9999999N9-00111+99999999999");
                        // Temperature ^^^^^
    new MapDriver<LongWritable, Text, Text, IntWritable>()
      .withMapper(new MaxTemperatureMapper())
      .withInput(new LongWritable(0), value)
      .withOutput(new Text("1950"), new IntWritable(-11))
      .runTest();
  }
}
```

The idea of the test is very simple: pass a weather record as input to the mapper, and check that the output is the year and temperature reading.

Since we are testing the mapper, we use MRUnit's `MapDriver`, which we configure with the mapper under test (`MaxTemperatureMapper`), the input key and value, and the expected output key (a `Text` object representing the year, 1950) and expected output value (an `IntWritable` representing the temperature, −1.1°C), before finally calling the `runTest()` method to execute the test. If the expected output values are not emitted by the mapper, MRUnit will fail the test. Notice that the input key could be set to any value because our mapper ignores it.

Proceeding in a test-driven fashion, we create a `Mapper` implementation that passes the test (see Example 6-6). Because we will be evolving the classes in this chapter, each is put in a different package indicating its version for ease of exposition. For example, `v1.MaxTemperatureMapper` is version 1 of `MaxTemperatureMapper`. In reality, of course, you would evolve classes without repackaging them.

Example 6-6. First version of a Mapper that passes MaxTemperatureMapperTest

```
public class MaxTemperatureMapper
    extends Mapper<LongWritable, Text, Text, IntWritable> {

  @Override
```

```
  public void map(LongWritable key, Text value, Context context)
      throws IOException, InterruptedException {

    String line = value.toString();
    String year = line.substring(15, 19);
    int airTemperature = Integer.parseInt(line.substring(87, 92));
    context.write(new Text(year), new IntWritable(airTemperature));
  }
}
```

This is a very simple implementation that pulls the year and temperature fields from the line and writes them to the `Context`. Let's add a test for missing values, which in the raw data are represented by a temperature of `+9999`:

```
    @Test
    public void ignoresMissingTemperatureRecord() throws IOException,
        InterruptedException {
      Text value = new Text("0043011990999991950051518004+68750+023550FM-12+0382" +
                            // Year ^^^^
          "99999V0203201N00261220001CN9999999N9+99991+99999999999");
                           // Temperature ^^^^^
      new MapDriver<LongWritable, Text, Text, IntWritable>()
        .withMapper(new MaxTemperatureMapper())
        .withInput(new LongWritable(0), value)
        .runTest();
    }
```

A `MapDriver` can be used to check for zero, one, or more output records, according to the number of times that `withOutput()` is called. In our application, since records with missing temperatures should be filtered out, this test asserts that no output is produced for this particular input value.

The new test fails since `+9999` is not treated as a special case. Rather than putting more logic into the mapper, it makes sense to factor out a parser class to encapsulate the parsing logic; see Example 6-7.

Example 6-7. A class for parsing weather records in NCDC format

```
public class NcdcRecordParser {

  private static final int MISSING_TEMPERATURE = 9999;

  private String year;
  private int airTemperature;
  private String quality;

  public void parse(String record) {
    year = record.substring(15, 19);
    String airTemperatureString;
    // Remove leading plus sign as parseInt doesn't like them (pre-Java 7)
    if (record.charAt(87) == '+') {
      airTemperatureString = record.substring(88, 92);
```

```
    } else {
      airTemperatureString = record.substring(87, 92);
    }
    airTemperature = Integer.parseInt(airTemperatureString);
    quality = record.substring(92, 93);
  }

  public void parse(Text record) {
    parse(record.toString());
  }

  public boolean isValidTemperature() {
    return airTemperature != MISSING_TEMPERATURE && quality.matches("[01459]");
  }

  public String getYear() {
    return year;
  }

  public int getAirTemperature() {
    return airTemperature;
  }
}
```

The resulting mapper (version 2) is much simpler (see Example 6-8). It just calls the parser's `parse()` method, which parses the fields of interest from a line of input, checks whether a valid temperature was found using the `isValidTemperature()` query method, and, if it was, retrieves the year and the temperature using the getter methods on the parser. Notice that we check the quality status field as well as checking for missing temperatures in `isValidTemperature()`, to filter out poor temperature readings.

Another benefit of creating a parser class is that it makes it easy to write related mappers for similar jobs without duplicating code. It also gives us the opportunity to write unit tests directly against the parser, for more targeted testing.

Example 6-8. A Mapper that uses a utility class to parse records

```
public class MaxTemperatureMapper
    extends Mapper<LongWritable, Text, Text, IntWritable> {

  private NcdcRecordParser parser = new NcdcRecordParser();

  @Override
  public void map(LongWritable key, Text value, Context context)
      throws IOException, InterruptedException {

    parser.parse(value);
    if (parser.isValidTemperature()) {
```

```
      context.write(new Text(parser.getYear()),
          new IntWritable(parser.getAirTemperature()));
    }
  }
}
```

With the tests for the mapper now passing, we move on to writing the reducer.

Reducer

The reducer has to find the maximum value for a given key. Here's a simple test for this feature, which uses a `ReduceDriver`:

```
    @Test
    public void returnsMaximumIntegerInValues() throws IOException,
        InterruptedException {
      new ReduceDriver<Text, IntWritable, Text, IntWritable>()
        .withReducer(new MaxTemperatureReducer())
        .withInput(new Text("1950"),
            Arrays.asList(new IntWritable(10), new IntWritable(5)))
        .withOutput(new Text("1950"), new IntWritable(10))
        .runTest();
    }
```

We construct a list of some `IntWritable` values and then verify that `MaxTemperatureReducer` picks the largest. The code in Example 6-9 is for an implementation of `MaxTemperatureReducer` that passes the test.

Example 6-9. Reducer for the maximum temperature example

```
public class MaxTemperatureReducer
    extends Reducer<Text, IntWritable, Text, IntWritable> {

  @Override
  public void reduce(Text key, Iterable<IntWritable> values, Context context)
      throws IOException, InterruptedException {

    int maxValue = Integer.MIN_VALUE;
    for (IntWritable value : values) {
      maxValue = Math.max(maxValue, value.get());
    }
    context.write(key, new IntWritable(maxValue));
  }
}
```

Running Locally on Test Data

Now that we have the mapper and reducer working on controlled inputs, the next step is to write a job driver and run it on some test data on a development machine.

Running a Job in a Local Job Runner

Using the `Tool` interface introduced earlier in the chapter, it's easy to write a driver to run our MapReduce job for finding the maximum temperature by year (see `MaxTemperatureDriver` in Example 6-10).

Example 6-10. Application to find the maximum temperature

```
public class MaxTemperatureDriver extends Configured implements Tool {

  @Override
  public int run(String[] args) throws Exception {
    if (args.length != 2) {
      System.err.printf("Usage: %s [generic options] <input> <output>\n",
          getClass().getSimpleName());
      ToolRunner.printGenericCommandUsage(System.err);
      return -1;
    }

    Job job = new Job(getConf(), "Max temperature");
    job.setJarByClass(getClass());

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.setMapperClass(MaxTemperatureMapper.class);
    job.setCombinerClass(MaxTemperatureReducer.class);
    job.setReducerClass(MaxTemperatureReducer.class);

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    return job.waitForCompletion(true) ? 0 : 1;
  }

  public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(new MaxTemperatureDriver(), args);
    System.exit(exitCode);
  }
}
```

`MaxTemperatureDriver` implements the `Tool` interface, so we get the benefit of being able to set the options that `GenericOptionsParser` supports. The `run()` method constructs a `Job` object based on the tool's configuration, which it uses to launch a job. Among the possible job configuration parameters, we set the input and output file paths; the mapper, reducer, and combiner classes; and the output types (the input types are determined by the input format, which defaults to `TextInputFormat` and has `LongWritable` keys and `Text` values). It's also a good idea to set a name for the job (`Max temperature`) so that you can pick it out in the job list during execution and after it has

completed. By default, the name is the name of the JAR file, which normally is not particularly descriptive.

Now we can run this application against some local files. Hadoop comes with a local job runner, a cut-down version of the MapReduce execution engine for running MapReduce jobs in a single JVM. It's designed for testing and is very convenient for use in an IDE, since you can run it in a debugger to step through the code in your mapper and reducer.

The local job runner is used if `mapreduce.framework.name` is set to `local`, which is the default.[1]

From the command line, we can run the driver by typing:

```
% mvn compile
% export HADOOP_CLASSPATH=target/classes/
% hadoop v2.MaxTemperatureDriver -conf conf/hadoop-local.xml \
  input/ncdc/micro output
```

Equivalently, we could use the `-fs` and `-jt` options provided by `GenericOptionsParser`:

```
% hadoop v2.MaxTemperatureDriver -fs file:/// -jt local input/ncdc/micro output
```

This command executes `MaxTemperatureDriver` using input from the local *input/ncdc/micro* directory, producing output in the local *output* directory. Note that although we've set `-fs` so we use the local filesystem (`file:///`), the local job runner will actually work fine against any filesystem, including HDFS (and it can be handy to do this if you have a few files that are on HDFS).

We can examine the output on the local filesystem:

```
% cat output/part-r-00000
1949    111
1950    22
```

Testing the Driver

Apart from the flexible configuration options offered by making your application implement `Tool`, you also make it more testable because it allows you to inject an arbitrary `Configuration`. You can take advantage of this to write a test that uses a local job runner to run a job against known input data, which checks that the output is as expected.

There are two approaches to doing this. The first is to use the local job runner and run the job against a test file on the local filesystem. The code in Example 6-11 gives an idea of how to do this.

1. In Hadoop 1, `mapred.job.tracker` determines the means of execution: `local` for the local job runner, or a colon-separated host and port pair for a jobtracker address.

Example 6-11. A test for MaxTemperatureDriver that uses a local, in-process job runner

```
@Test
public void test() throws Exception {
  Configuration conf = new Configuration();
  conf.set("fs.defaultFS", "file:///");
  conf.set("mapreduce.framework.name", "local");
  conf.setInt("mapreduce.task.io.sort.mb", 1);

  Path input = new Path("input/ncdc/micro");
  Path output = new Path("output");

  FileSystem fs = FileSystem.getLocal(conf);
  fs.delete(output, true); // delete old output

  MaxTemperatureDriver driver = new MaxTemperatureDriver();
  driver.setConf(conf);

  int exitCode = driver.run(new String[] {
      input.toString(), output.toString() });
  assertThat(exitCode, is(0));

  checkOutput(conf, output);
}
```

The test explicitly sets `fs.defaultFS` and `mapreduce.framework.name` so it uses the local filesystem and the local job runner. It then runs the `MaxTemperatureDriver` via its `Tool` interface against a small amount of known data. At the end of the test, the `checkOutput()` method is called to compare the actual output with the expected output, line by line.

The second way of testing the driver is to run it using a "mini-" cluster. Hadoop has a set of testing classes, called `MiniDFSCluster`, `MiniMRCluster`, and `MiniYARNCluster`, that provide a programmatic way of creating in-process clusters. Unlike the local job runner, these allow testing against the full HDFS, MapReduce, and YARN machinery. Bear in mind, too, that node managers in a mini-cluster launch separate JVMs to run tasks in, which can make debugging more difficult.

You can run a mini-cluster from the command line too, with the following:

```
% hadoop jar \
  $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-*-tests.jar \
  minicluster
```

Mini-clusters are used extensively in Hadoop's own automated test suite, but they can be used for testing user code, too. Hadoop's `ClusterMapReduceTestCase` abstract class provides a useful base for writing such a test, handles the details of starting and stopping

the in-process HDFS and YARN clusters in its `setUp()` and `tearDown()` methods, and generates a suitable `Configuration` object that is set up to work with them. Subclasses need only populate data in HDFS (perhaps by copying from a local file), run a MapReduce job, and confirm the output is as expected. Refer to the `MaxTemperatureDriverMiniTest` class in the example code that comes with this book for the listing.

Tests like this serve as regression tests, and are a useful repository of input edge cases and their expected results. As you encounter more test cases, you can simply add them to the input file and update the file of expected output accordingly.

Running on a Cluster

Now that we are happy with the program running on a small test dataset, we are ready to try it on the full dataset on a Hadoop cluster. Chapter 10 covers how to set up a fully distributed cluster, although you can also work through this section on a pseudo-distributed cluster.

Packaging a Job

The local job runner uses a single JVM to run a job, so as long as all the classes that your job needs are on its classpath, then things will just work.

In a distributed setting, things are a little more complex. For a start, a job's classes must be packaged into a *job JAR file* to send to the cluster. Hadoop will find the job JAR automatically by searching for the JAR on the driver's classpath that contains the class set in the `setJarByClass()` method (on `JobConf` or `Job`). Alternatively, if you want to set an explicit JAR file by its file path, you can use the `setJar()` method. (The JAR file path may be local or an HDFS file path.)

Creating a job JAR file is conveniently achieved using a build tool such as Ant or Maven. Given the POM in Example 6-3, the following Maven command will create a JAR file called *hadoop-examples.jar* in the project directory containing all of the compiled classes:

```
% mvn package -DskipTests
```

If you have a single job per JAR, you can specify the main class to run in the JAR file's manifest. If the main class is not in the manifest, it must be specified on the command line (as we will see shortly when we run the job).

Any dependent JAR files can be packaged in a *lib* subdirectory in the job JAR file, although there are other ways to include dependencies, discussed later. Similarly, resource files can be packaged in a *classes* subdirectory. (This is analogous to a Java *Web application archive*, or WAR, file, except in that case the JAR files go in a *WEB-INF/lib* subdirectory and classes go in a *WEB-INF/classes* subdirectory in the WAR file.)