HowToDebugMapReducePrograms

How to Debug Map/Reduce Programs

Debugging distributed programs is always difficult, because very few debuggers will let you connect to a remote program that wasn't run with the proper command line arguments.

1. Start by getting everything running (likely on a small input) in the local runner.
   You do this by setting your job tracker to "local" in your config. The local runner can run under the debugger and runs on your development machine. A very quick and easy way to set this config variable is to include the following line just before you run the job:
   `conf.set("mapred.job.tracker", "local");`
   You may also want to do this to make the input and output files be in the local file system rather than in the Hadoop distributed file system (HDFS):
   `conf.set("fs.default.name", "local");`
   You can also set these configuration parameters in `hadoop-site.xml`. The configuration files `hadoop-default.xml`, `mapred-default.xml` and `hadoop-site.xml` should appear somewhere in your program's class path when the program runs.

   2. Run the small input on a 1 node cluster. This will smoke out all of the issues that happen with distribution and the "real" task runner, but you only have a single place to look at logs. Most useful are the task and job tracker logs. Make sure you are logging at the INFO level or you will miss clues like the output of your tasks.
   3. Run on a big cluster. Recently, I added the keep.failed.task.files config variable that tells the system to keep files for tasks that fail. This leaves "dead" files around that you can debug with. On the node with the failed task, go to the task tracker's local directory and cd to *<local>*/taskTracker/*<taskid>*/work and run

   ```
   % hadoop org.apache.hadoop.mapred.IsolationRunner ../job.xml
   ```

   This will run the failed task in a single jvm, which can be in the debugger, over precisely the same input.

There is also a configuration variable (keep.task.files.pattern) that will let you specify a task to keep by name, even if it doesn't fail. Other than that, logging is your friend.

Other Java Debugging hints

To print information about the state of the threads in a Java program including:

- Call stack
- Locks held
- Deadlocks

Send a QUIT signal to the Java process:

```
kill -QUIT <pid>
```

The output is sent to stdout.

If that doesn't work because the output is being sent to /dev/null, you can also use the commands:

- **jps**: List your java processes
- **jstack**: Get the call stack for a given Java process

These commands were first included in Sun's Java 1.5.

Setting Task Status

The map and reduce interfaces both include a parameter 'Reporter reporter'. The method Reporter.setStatus(String status) changes the displayed status of the map task and is visable on the jobtracker web page.

This can be extremely useful to display debug information about the current record being handled, or setting certain debug flags about the status of the mapper. While running locally on a small data set can find many bugs, large data sets may contain pathological cases that are otherwise unexepcted. This method of debugging can help catch those cases.

How to debug Hadoop Pipes programs

In order to debug Pipes programs you need to keep the downloaded commands.

First, to keep the TaskTracker from deleting the files when the task is finished, you need to set either keep.failed.task.files (set it to true if the interesting task always fails) or keep.task.files.pattern (set to a regex that includes the interesting task name).

Second, your job should set hadoop.pipes.command-file.keep to true in the JobConf. This will cause all of the tasks in the job to write their command stream to a file in the working directory named downlink.data. This file will contain the JobConf, the task information, and the task input, so it may be large. But it provides enough information that your executable will run without any interaction with the framework.

Third, go to the host where the problem task ran, go into the work directory and

```
setenv hadoop.pipes.command.file downlink.data
```

and run your executable under the debugger or valgrind. It will run as if the framework was feeding it commands and data and produce a output file downlink.data.out with the binary commands that it would have sent up to the framework. Eventually, I'll probably make the downlink.data.out file into a text-based format, but for now it is binary. Most problems however, will be pretty clear in the debugger or valgrind, even without looking at the generated data.

The following sections are applicable only for Hadoop 0.16.0 and above

Run a debug script when Task fails

When map/reduce task fails, there is a facility provided, via user-provided scripts, for doing post-processing on task logs i.e task's stdout, stderr, syslog. The stdout and stderr of the user-provided debug script are printed on the diagnostics. These outputs are displayed on job UI on demand.

For pipes, a default script is run which processes core dumps under gdb, prints stack trace and gives info about running threads.

In the following sections we discuss how to submit debug script along with the job. We also discuss what the default behavior is. For submiting debug script, first it has to distributed. Then the script has to supplied in Configuration.

How to submit debug script file

To submit the debug script file, first put the file in dfs.

The file can be distributed by setting the property "mapred.cache.files" with value <path>#<script-name>. For more than one file, they can be added as comma seperated paths. The script file needs to be symlinked.

This property can also be set by APIs DistributedCache.addCacheFile(URI,conf) and DistributedCache.setCacheFiles(URIs,conf) where URI is of the form "hdfs://host:port/<absolutepath>#<script-name>". For Streaming, the file can be added through command line option -cacheFile. To create symlink for the file, the property "mapred.create.symlink" is set to "yes". This can also be set by DistributedCache.createSymLink

How to submit debug script

A quick way to submit debug script is to set values for the properties "mapred.map.task.debug.script" and "mapred.reduce.task.debug.script" for debugging map task and reduce task respectively. These properties can also be set by APIs JobConf.setMapDebugScript JobConf.setReduceDebugScript. The script is given task's stdout, stderr, syslog, jobconf files as arguments. The debug command, run on the node where the map/reduce failed, is:

```
$script $stdout $stderr $syslog $jobconf
```

For streaming, debug script can be submitted with command-line options -mapdebug, -reducedebug for debugging mapper and reducer respectively.

Pipes programs have the c++ program name as a fifth argument for the command. Thus for the pipes programs the command is

```
$script $stdout $stderr $syslog $jobconf $program
```

Here is an example on how to submit a script

```
    jobConf.setMapDebugScript("./myscript");
    DistributedCache.createSymlink(jobConf);
    DistributedCache.addCacheFile("/debug/scripts/myscript#myscript");
```

Default Behavior

The default behavior for failed map/reduce tasks is

For Java programs:
Stdout, stderr are shown on job UI. Stack trace is printed on diagnostics.

For Pipes:
Stdout, stderr are shown on the job UI. If the failed task has core file, Default gdb script is run which prints info abt threads: thread Id and function in which it was running when task failed. And prints stack trace where task has failed.

For Streaming:
Stdout, stderr are shown on the Job UI. The exception details are shown on task diagnostics.