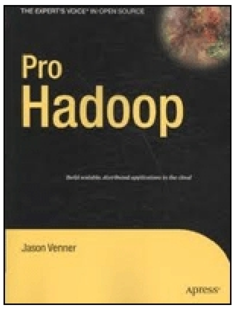


Pro Hadoop

by Jason Venner

Apress. (c) 2009. Copying Prohibited.

Chapter 7: Unit Testing and Debugging

Overview

Two questions echo endlessly in the dark hours of small, enclosed areas lit by the dim glow of display screens: "Is it working?" and "How did *that* happen?" This chapter is about answering those questions.

"Is it working?" will be addressed by writing unit tests. The agile programming model suggests starting with unit tests and building the code afterwards. I generally try to follow this model, although under pressure, I have shifted to just writing the code, usually to my later regret. Testing the lower-level APIs in your applications is a known skill set. This chapter will cover unit tests that actually run MapReduce jobs at a small scale.

There are several ways to determine what is happening in a running program. The first level is through examining the log messages or other job output data. This requires detailed understanding of the output and may not provide sufficient information to isolate the problem. An alternative is to put custom code into the application to trigger different behavior or logging around the area of code that is in question. The most comprehensive method is to attach to the running application with a debugger and step through the execution of the code. This chapter will cover interactive debugging, using the Eclipse platform to provide the graphical interface.

Unit Testing MapReduce Jobs

MapReduce jobs, by their very nature, don't lend themselves to the traditional unit testing model. The common approach is to verify that all of the lower-level APIs are working correctly through their own unit tests. Next, you build small test datasets that have known outcomes and run them on a simulated MapReduce cluster, and then examine the output. These tests, when run on the simulated clusters, tend to be quite slow—on the order of minutes per test—due to the cluster setup and teardown times. If a real cluster is available for testing, the test run time will be shorter, but the tests must be coordinated among the cluster users.

The unit tests covered here are built on the Hadoop basic test case class `org.apache.hadoop.mapred.ClusterMapReduceTestCase`. This class provides a basic JUnit 3 test base that will start and stop a mini-HDFS with two DataNodes and a NameNode, and a mini-MapReduce cluster with two TaskTrackers and a JobTracker. This is a complete cluster, with web consoles for the JobTracker and the NameNode. All of the servers will run on the local machine, and the ports will be chosen from the free ports on the machine.

Because `ClusterMapReduceTestCase` is a JUnit 3-style test class, at least through Hadoop 0.19.0, it does not support annotations. The virtual cluster will be established and torn down for each test that any derived classes execute.

JUnit 4 supports the annotations `@BeforeClass` and `@AfterClass`, which allow for the cluster setup and teardown to happen one time per test class, providing each test with a clean cluster and saving significant wall clock time if many tests are run by the class. The JUnit 4-compliant delegate class demonstrated in this chapter allows `ClusterMapReduceTestCase` to be used with JUnit 4, so you can define when the virtual clusters are created and destroyed.

The JAR Files that Come with Your Hadoop Distribution

In the archive that contains a Hadoop release are a number of prebuild release-specific JAR files. These JAR files are named in the form `hadoop-`*`major release`*`-`*`minor release`*`-`*`component name`*`.jar`.

The standard Hadoop JARs are found in the root directory of the installation. For Hadoop 0.19.0, the JAR files are `hadoop-0.19.0-core.jar`. This JAR is for the component `core`, for the major `release 0.19`, and the minor release `1`.

The root directory will contain JAR files for Ant, Core, examples, test, and tools. The `contrib.component` JARs also follow the same naming convention.

Each component may have an associated `lib` directory containing JAR files on which the component depends. By convention, the `lib` directory is located in the same directory as the component JAR.

In this chapter, I refer to the JARs as `hadoop-rel-`*`component`*`.jar`, where *`component`* is replaced with the actual component name, such as `hadoop-<rel>-core.jar` for the Hadoop Core JAR.

Requirements for Using ClusterMapReduceTestCase

The `ClusterMapReduceTestCase` class, like all Hadoop Core classes, makes strong assumptions about the runtime environment. For Hadoop Core unit tests, and for running standard jobs, the Hadoop Ant environment and the `bin/hadoop` script configure the runtime environment for the unit test or job, respectively. The unit tests that developers write to run in their workspace, or those created for build automation tools, do not generally have this luxury and must set up the runtime environment directly.

The `ClusterMapReduceTestCase` starts a virtual Hadoop cluster on which the tests are run. If the cluster does not start successfully, the test case will fail, without exercising the classes being tested. A startup failure is commonly due to a configuration issue, either with the runtime classpath or server or cluster configuration file. In particular, the NameNode and the JobTracker use Jetty to provide web servers for their web UIs. The error messages relating to the Jetty web server start failures do not provide sufficient information for the novice to resolve the configuration problem.

For developers running the tests from their IDE, it is not uncommon to load the Hadoop source code into the workspace, in place of using `hadoop-<rel>-core.jar`. Most of the Hadoop classes require configuration information that is provided in the distribution's `config/hadoop-default.xml` file. A copy of `hadoop-default.xml` is also bundled into the `hadoop-<rel>-core.jar`. When this configuration information is absent, the virtual cluster behavior is unpredictable.

The following are the requirements for using `ClusterMapReduceTestCase` in a unit test:

`hadoop-<rel>-core jar`: This is required for basic Hadoop classes. Include this JAR in the build path of your project and in the runtime classpath for the JUnit execution environment.

`hadoop-<rel>- test.jar`: This provides `ClusterMapReduceTestCase` and supporting classes. Include this JAR in the build path of your project and in the runtime classpath for the JUnit execution environment.

`lib/*.jar`: This provides the required services for the Hadoop Core classes. Include the needed JARs in the build path of your project and in the runtime classpath for the JUnit execution environment. It is often simpler to just include all of the JARs in the `lib` directory.

`lib/jetty-ext/*.jar`: This provides the additional classes that Jetty requires for the web consoles that will be run for the virtual cluster.

`hadoop.log.dir`: The framework requires this Java system property to be set to the path to an existing writable directory. This must be defined at test case start time, or by using the call `System.setProperty ("hadoop.log.dir", "path")`, before the first call to `ClusterMapReduceTestCase.setup()`. The absence of a valid `hadoop.log.dir` system property results in a `NullPointerExceptions` or `IOException` being thrown by the test case during cluster setup.

`javax.xml.parsers.SAXParserFactory`: This must be Xerces, not SAX, or a validating parser error will be thrown. The correct value for this property is `org.apache.xerces.jaxp.SAXParserFactoryImpl`. It may be specified by the following argument on the JVM command line:

```
D javax.xml.parsers.SAXParserFactory= ➥
com.sun.org.apache.xerces.internal.jaxp.SAXParserFactoryImpl
```

Alternatively, you can call the following before the first call to `ClusterMapReduceTestCase.setup()`:

```
System.setProperty("javax.xml.parsers.SAXParserFactory"," ➥
com.sun.org.apache.xerces.internal.jaxp.SAXParserFactoryImpl")
```

Let's look at some ways to check whether the requirements for using `ClusterMapReduceTestCase` have been met.

Troubles with Jetty, the HTTP Server for the Web UI

Jetty requires a validating parser that can handle its XML usage. Most parsers will do fine, but some will fail.

Note I was working on a large application that had a complex classpath. For unit testing, the entire classpath, including the Hadoop JARs, were folded together. All of a sudden, the unit tests started failing with an exception thrown by Jetty. The Saxon JAR was in the classpath before the Jetty JAR, so it was being used to deliver the XML parsers, and the parser was not validating.

A couple of system properties control the XML parser that applications will get:

`javax.xml.parsers.SAXParserFactory`: This contains the class name of the factory for constructing XML parsers. Set this to `com.sun.org.apache.xerces.internal.jaxp.SAXParserFactoryImpl`. This is the Sun JDK default and works well with Jetty. It avoids issues with alternative parsers in the classpath.

`org.mortbay.xml.XmlParser.NotValidating`: This instructs Jetty to not validate the XML configuration data. You should set it to `false`. Validated XML is good, and these are parsed only at job start time.

These properties may be set by passing them as arguments to the JVM via the `-Dproperty=value` syntax, or by calling `System.setProperty(name,value);` when performing setup for the test.

Hadoop Core servers rely on Jetty to provide web services, which are then available for internal and external use. If the Jetty JAR is missing from the classpath, the servers will not start. Listing 7-1 shows the log lines that indicate this.

Listing 7-1: Log Lines That Indicate No Jetty JAR Is in the Unit Test Classpath

```
java.io.IOException: Problem starting http server
Caused by: org.mortbay.util.MultiException[java.lang.reflect. ➥
InvocationTargetException, java.lang.reflect.InvocationTargetException, ➥
java.lang.reflect.InvocationTargetException]
```

The Hadoop Core Jetty configurations also require the JARs that are in the `lib/jetty-ext` directory of the installation. If they are not present in the classpath of the unit test, non-descriptive failure-to-start error messages will be generated.

Listings 7-2 through 7-5 shows the various log entries indicating that specific Jetty JARs are missing from the unit test classpath. In these listings, only the relevant exception lines are shown; the stack traces have been removed to aid clarity. The string *XXXXX* represents some TCP port number.

Listing 7-2: Log Lines That Indicate jetty-ext/commons-el.jar Is Not in the Unit Test Classpath

```
java.io.IOException: Call to /0.0.0.0:XXXXX failed on local exception: ➥
Connection refused: no further information
Caused by: java.net.ConnectException: Connection refused: no further information
```

Listing 7-3: Log Lines That Indicate jetty-ext/jasper-runtime.jar Is Not in the Unit Test Classpath

```
java.lang.NoClassDefFoundError: org/apache/jasper/JasperException
Caused by: java.lang.ClassNotFoundException: org.apache.jasper.JasperException
```

Listing 7-4: Log Lines That Indicate jetty-ext/jasper-compiler.jar Is Not in the Unit Test Classpath

```
java.io.IOException: Problem starting http server
Caused by: org.mortbay.util.MultiException[java.lang.ClassNotFoundException: ➥
org.apache.jasper.servlet.JspServlet, java.lang.ClassNotFoundException: ➥
org.apache.jasper.servlet.JspServlet, java.lang.ClassNotFoundException: ➥
org.apache.jasper.servlet.JspServlet]
```

Listing 7-5: Log Lines That Indicate jetty-ext/jsp-api.jar Is Not in the Unit Test Classpath

```
java.lang.NoClassDefFoundError: javax/servlet/jsp/JspFactory
Caused by: java.lang.ClassNotFoundException: javax.servlet.jsp.JspFactory
```

The Hadoop Core JAR Is Missing or Malformed

Particularly for IDE developers, the classpath may have the Hadoop source tree rather than the Hadoop Core JAR. In this case, various required configuration files may be missing, resulting in unexpected failures. Listings 7-6 and 7-7 show the log lines that indicate missing configuration files.

Listing 7-6: Log Lines That Indicate the hadoop-default.xml File Is Missing or Malformed

```
java.lang.NullPointerException
    at org.apache.hadoop.hdfs.server.namenode. ➥
    FSNamesystem.close(FSNamesystem.java:523)
    at org.apache.hadoop.hdfs.server.namenode. ➥
    FSNamesystem.<init>(FSNamesystem.java:293)
```

Listing 7-7: Log Lines That Indicate the Java System Property `hadoop.log.dir` Is Unset

```
ERROR mapred.MiniMRCluster: Job tracker crashed
java.lang.NullPointerException
    at java.io.File.<init>(Unknown Source)
    at org.apache.hadoop.mapred.JobHistory.init(JobHistory.java:143)
    at org.apache.hadoop.mapred.JobTracker.<init>(JobTracker.java:1110)
    at org.apache.hadoop.mapred.JobTracker.startTracker(JobTracker.java:143)
    at org.apache.hadoop.mapred. ➥
    MiniMRCluster$JobTrackerRunner.run(MiniMRCluster.java:96)
    at java.lang.Thread.run(Unknown Source)
```

The `MiniDFSCluster` creates the directories for HDFS storage in the path `build/test/data/dfs/data`, or `build\test\data\dfs\data` under Windows. It will attempt to remove the directory before starting. Listing 7-8 shows the error message that results if the directories cannot be deleted. The typical reason for the failure is that a prior instance of `MiniDFSCluster` is still running.

Listing 7-8: Log Lines That Indicate a Unit Test Is Already in Progress

```
java.io.IOException: Cannot remove data directory: build\test\data\dfs\data
    at org.apache.hadoop.hdfs.MiniDFSCluster.<init>(MiniDFSCluster.java:263)
    at org.apache.hadoop.hdfs.MiniDFSCluster.<init>(MiniDFSCluster.java:119)
    at org.apache.hadoop.mapred.ClusterMapReduceTestCase. ➥
    startCluster(ClusterMapReduceTestCase.java:81)
```

The Virtual Cluster Failed to Start

`ClusterMapReduceTestCase` builds a virtual Hadoop cluster on which to run the test cases. By default, this virtual cluster starts six server processes: one NameNode, one JobTracker, two DataNodes, and two TaskTrackers. If the NameNode or the JobTracker did not start, the tests cannot be run.

The HDFS portion of the cluster is started first. It is composed of one NameNode and two DataNodes. If the HDFS fails to start, the MapReduce portion is not started.

The NameNode is kind enough to actually report that it is up, as in this example log line:

```
namenode.NameNode: Namenode up at: localhost/127.0.0.1:XXXXX
```

The DataNodes' state must be deduced from the log messages. The log messages in Listing 7-9 indicate successful startup. (Your timestamps and port allocations will vary.)

Listing 7-9: Log Lines That Indicate Both DataNodes Are Running

```
Starting DataNode 0 with dfs.data.dir:
➥ build\test\data\dfs\data\data1,build\test\data\dfs\data\data2
...
INFO datanode.DataNode: New storage id DS-222715038-192.168.1.12-2232- ➥
1236829361312 is assigned to data-node 127.0.0.1:2232
INFO datanode.DataNode: DatanodeRegistration(127.0.0.1:2232, ➥
storageID=DS-222715038-192.168.1.12-2232-1236829361312, infoPort=2233, ➥
ipcPort=2234)In DataNode.run, data = FSDataset{dirpath= ➥
'C:\Documents and Settings\Jason\My Documents\HadoopBook\ ➥
code\examples\build\test\data\dfs\data\data1\current, ➥
C:\Documents and Settings\Jason\My Documents\HadoopBook\ ➥
code\examples\build\test\data\dfs\data\data2\current'}
INFO datanode.DataNode: using BLOCKREPORT_INTERVAL of 3600000msec ➥
```

```
Initial delay:0msec

Starting DataNode 1 with dfs.data.dir: ➥
build\test\data\dfs\data\data3,build\test\data\dfs\data\data4
...

INFO datanode.DataNode: New storage id DS-2049952137-192.168.1.12-2239- ➥
1236829361718 is assigned to data-node 127.0.0.1:2239
INFO datanode.DataNode: DatanodeRegistration(127.0.0.1:2239, ➥
storageID=DS-2049952137-192.168.1.12-2239-1236829361718, infoPort=2240, ➥
 ipcPort=2241)In DataNode.run, data = FSDataset{dirpath= ➥
'C:\Documents and Settings\Jason\My Documents\HadoopBook\ ➥
code\examples\build\test\data\dfs\data\data3\current, ➥
C:\Documents and Settings\Jason\My Documents\HadoopBook\ ➥
code\examples\build\test\data\dfs\data\data4\current'}
INFO datanode.DataNode: using BLOCKREPORT_INTERVAL of 3600000msec ➥
Initial delay: 0msec

Waiting for the Mini HDFS Cluster to start...
INFO datanode.DataNode: BlockReport of 0 blocks got processed in 0 msecs
INFO datanode.DataNode: Starting Periodic block scanner.
INFO datanode.DataNode: BlockReport of 0 blocks got processed in 0 msecs
INFO datanode.DataNode: Starting Periodic block scanner.
```

The two DataNodes are started after the NameNode is started, and each has a `Starting` line followed by a final line that indicates the `BLOCKREPORT_INTERVAL`. If the lines containing `BLOCKREPORT_INTERVAL` are missing, the DataNode did not start.

As with the HDFS portion, the JobTracker informs you directly that it is running, but the TaskTracker status must be deduced from the logs. Here is the message from a successfully started JobTracker:

```
INFO mapred.JobTracker: Starting RUNNING
```

Listing 7-10 shows the log lines that show the TaskTracker is running. You need two sets of these for full service.

Listing 7-10: Log Lines That Indicate a TaskTracker Is Running

```
mapred.TaskTracker: TaskTracker up at: 0.0.0.0/0.0.0.0:2262
mapred.TaskTracker: Starting tracker tracker_host1.foo.com:0.0.0.0/0.0.0.0:2262
```

If the `hadoop.log.dir` system property is unset, a subprocess of the virtual cluster may crash and leave the test case in limbo. Listing 7-11 shows this error.

Listing 7-11: A Virtual Cluster Server Process Has Crashed

```
ERROR mapred.MiniMRCluster: Job tracker crashed
java.lang.NullPointerException
    at java.io.File.<init>(Unknown Source)
    at org.apache.hadoop.mapred.JobHistory.init(JobHistory.java:143)
    at org.apache.hadoop.mapred.JobTracker.<init>(JobTracker.java:1110)
    at org.apache.hadoop.mapred.JobTracker.startTracker(JobTracker.java:143)
    at org.apache.hadoop.mapred. ➥
    MiniMRCluster$JobTrackerRunner.run(MiniMRCluster.java:96)
    at java.lang.Thread.run(Unknown Source)
```

There should be no `ERROR` level log messages.

The Eclipse framework provides a decent way to run individual or class-based Hadoop unit tests, as well as simple debugging. Occasionally, some state can get lost, particularly in the Windows environment, and the virtual cluster will fail to start. The indication of this will be a series of messages stating that a connection attempt has failed. In particular, if you see connection failure messages that have `/0.0.0.0:`, it is an indication that Eclipse needs a restart, as shown in Listing 7-12.

Listing 7-12: Log Lines That Indicate Eclipse Has Lost State and Needs to Be Restarted

```
INFO ipc.Client: Retrying connect to server: /0.0.0.0:9100. Already tried 0 time(s).
INFO ipc.Client: Retrying connect to server: /0.0.0.0:9100. Already tried 1 time(s).
INFO ipc.Client: Retrying connect to server: /0.0.0.0:9100. Already tried 2 time(s).
INFO ipc.Client: Retrying connect to server: /0.0.0.0:9100. Already tried 3 time(s).
INFO ipc.Client: Retrying connect to server: /0.0.0.0:9100. Already tried 4 time(s).
INFO ipc.Client: Retrying connect to server: /0.0.0.0:9100. Already tried 5 time(s).
INFO ipc.Client: Retrying connect to server: /0.0.0.0:9100. Already tried 6 time(s).
INFO ipc.Client: Retrying connect to server: /0.0.0.0:9100. Already tried 7 time(s).
INFO ipc.Client: Retrying connect to server: /0.0.0.0:9100. Already tried 8 time(s).
INFO ipc.Client: Retrying connect to server: /0.0.0.0:9100. Already tried 9 time(s).
```

Simpler Testing and Debugging with ClusterMapReduceDelegate

When running tests, it is very helpful to be able to interact with the test cluster HDFS, to access the Web GUIs of the various servers, and to examine the log files. This book's downloadable code includes a class `com.apress.hadoop.mapred.test.ClusterMapReduceDelegate` that provides a wrapper around the Hadoop Core test framework class `ClusterMapReduceTestCase`. This delegate class provides a JUnit 4-friendly way to build test classes, and exposes information useful to understanding what is happening in your test. All the test case classes discussed here extend the class `ClusterMapReduceDelegate`.

Core Methods of ClusterMapReduceDelegate

Table 7-1 lists the core methods that any unit test interacting with Hadoop will use. In particular, all `JobConf` objects used by the test cases and classes being tested must be children of the `JobConf` object returned by the `createJobConf()` method.

Table 7-1: Core Methods of ClusterMapReduceDelegate

Method	When to Use	What it does
`setupBeforeClass()`	In the `@BeforeClass` method.	Ensures that the required Hadoop system properties are set to sensible values if they are unset. It will then start the virtual cluster, with 2 TaskTrackers, 2 DataNodes, a NameNode, and a JobTracker. This method will throw an exception or possibly hang if the virtual cluster does not start.
`logConfiguration (JobConf conf, Logger log)`	If you need detailed information on how the virtual cluster is configured.	Dumps the key parameters out of `conf`, and the virtual cluster NameNode and DFS configuration objects to log at level info.
`Configuration getHDFSConfiguration()`	If you or your test need to interact with NameNode. This object has the key parameters such as `fs.default.name` and `dfs.http.address`.	Returns the virtual cluster's HDFS configuration object.
`Configuration getJobTrackerConfiguration ()`	If you or your test need to interact with the JobTracker. This object has key parameters such as `mapred.job.tracker`, `mapred.job.tracker.http.address` and `mapred.system.dir`.	Returns the configuration object used by the `MiniMRCluster` private base class, which sets up the JobTracker for the virtual cluster.
`Path getTestRootDir()`	To determine the root path of the test in HDFS, when you need to examine data files.	Returns the path of the test case in HDFS.
`FileSystem getFileSystem()`	When the test case need to create files or otherwise interact with HDFS.	Returns a file system object constructed for the virtual cluster's HDFS file system.
`JobConf createJobConf ()`	*Must be used to create the `JobConf` object's used by your test cases.*	Creates a `JobConf` object that is correctly configured for the virtual cluster.
`void tearDownAfterClass ()`	In the `@AfterClass` method of your test class.	Stops the virtual cluster.

`void logDefaults (Logger log)`	If you have a problem with the test case or cluster starting.	Writes the locations of the Core Hadoop JAR files and the critical parameters to log at the info level.

All tests will need to call the methods `setupBeforeClass()` and `tearDownAfterClass()` at least once to start and stop the virtual cluster. Any test case that needs to create files in HDFS or access files in HDFS will need to call the method `getFileSystem()` to get a file system object to use for the interactions.

Configuration Parameters for Interacting with Virtual Clusters

Several core parameters are needed for the tester and the test cases, when interacting with the virtual cluster. Table 7-2 details the parameter names, how to get their values, and what to do with them. When debugging test cases, it is very useful to know where the log files are being written, and the web addresses for the NameNode and JobTracker. For accessing files in the virtual cluster's HDFS, the HDFS file system URL must be available.

Table 7-2: Important Configuration Parameters for Interacting with the Virtual Cluster

Parameter	What it is	How to Get it	What to Do with it
`hadoop.log.dir`	The path to the directory log files are written to	`System.getProperties ("hadoop.log.dir");`	Look in this directory for cluster log files, such as user logs for the per task log files.
`fs.default.name`	The URL for HDFS	`getFileSystem().getUri();`	Use this URL to interact with the virtual HDFS from the command line via `bin/hadoop dfs -fs URL` file operations.
`mapred.job.tracker.http.address`	The URL for the virtual cluster JobTracker web interface.	`createJobConf().get ("mapred.job.tracker.http.address");`	Use this URL to view the state of running and finished jobs in the virtual cluster.
`dfs.http.address`	The URL for the virtual cluster NameNode web interface.	`getHDFSConfiguration().get ("dfs.http.address");`	Use this URL to view the state of the virtual cluster HDFS.

Writing a Test Case: SimpleUnitTest

The sample `SimpleUnitTest` test case simply starts a cluster, writes a single file to the cluster HDFS, and reads that file back, verifying the contents are correct. This section will walk through building this unit test. The full code for this example is in the file `SimpleUnitTest.java` of package `com.apress.hadoopbook.examples.ch7` in the downloadable code for this book.

The TestCase Class Declaration

In Listing 7-13, the test case class extends `ClusterMapReduceDelegate` and a `Logger` object is declared. Normally, the `Logger` would be from the class being tested to better enable control over the logging levels. In this sample test case, there is no class being tested, so a logger is created.

Listing 7-13: Class Declaration from SimpleUnitTest.java

```
/** This simple unit test exists to demonstrate the creation and teardown of a
 * virtual cluster and the writing of a test case that uses the created cluster.
 *
 * @author Jason
 *
 */
import com.apress.hadoop.mapred.test.ClusterMapReduceDelegate;
public class SimpleUnitTest extends ClusterMapReduceDelegate {
    public static Logger LOG = Logger.getLogger(SimpleUnitTest.class);
```

The Cluster Start Method

The `startVirtualCluster()`method in `SimpleUnitTest`, shown in Listing 7-14, is used to start the virtual cluster and verify that the cluster has started successfully. `startVirtualCluster()` uses the JUnit 4 annotation `@BeforeClass` to indicate to the JUnit framework that this method must be run one time only, and before any of the test cases in the class are launched. The test also makes two JUnit `assertNotNull` checks, to verify that the cluster configuration information is available. If there are any failures, an exception should be thrown. The JUnit framework will catch the exception and mark the test set as failed.

Listing 7-14: Cluster Setup Method with JUnit 4 @BeforeClass Annotation

```
/** This is the JUnit4 before class, cluster initialization method.
 *
 * This method starts the cluster and performs simple validation of the working ➥
    state of the cluster.
 *
 * Under some failure cases usually related to incorrect CLASSPATH ➥
    configuration, this method may never complete.
 *
 * If all of the test cases in your file can share a cluster, use ➥
    the @BeforeClass annotation.
 * @throws Exception
 */
@BeforeClass
public static void startVirtualCluster() throws Exception
{
    /** Turn down the cluster logging to filter the noise out. Do this if ➥
           the test is basically working. */

    setupTestClass();
    /** Verify that there is a JobConf object for the cluster. */
    assertNotNull("Cluster initialized Correctly", getConf());
    /** Verify that the file system object is available. */
    assertNotNull("Cluster has a file system", getFs());
}
```

The `setupTestClass()` call is a method on the `ClusterMapReduceDelegate` and actually starts the virtual cluster and collects the configuration information. It also will set a small number of required system parameters in the configuration object returned by `getConf()`, if those parameters are currently unset.

The Cluster Stop Method

`SimpleUnitTest.stopVirtualCluster()`,shown in Listing 7-15, uses the JUnit 4 annotation `@AfterClass` to indicate to the JUnit framework that it is to be called after the last test in the class has been run. It is essentially the `finally` clause for the test class.

Listing 7-15: Cluster Stop Method with JUnit 4 @AfterClass Annotation

```
/** This is the JUnit4 after class tear down method.
 * This stops the cluster, and would perform any needed cleanup.
 * If all of the test cases in a file can share a cluster use the @AfterClass ➥
    annotation.
 * @throws Exception
 */
@AfterClass
public static void stopVirtualCluster() throws Exception {
    teardownTestClass();
}
```

`teardownTestCase()` is a method on the `ClusterMapReduceDelegate` class that will terminate the virtual cluster and clear the cached cluster information.

The Actual Test

The unit test, shown in Listing 7-16, writes a string to a newly created file in the virtual cluster HDFS, and then reads the string back from the file to verify that the same string can be read back. The test has the standard stylized framework you will see in all of the sample code. Any code that allocates objects that hold system-level file descriptors is done in a `try` block. The `try` block has a `finally` clause where the system-level file descriptors are closed. This pattern, if rigorously applied, will greatly reduce job failures when the jobs are running at large scales.

Listing 7-16: The Actual Test Code with the JUnit 4 @Test Annotation

```
/** A very simple unit test that uses the virtual cluster.
 *
 * The test case writes a single file to HDFS and reads it back, verifying ➥
    the file contents.
 *
 * @throws Exception
 */
@Test
public void createFileInHdfs() throws Exception
{
    final FileSystem fs = getFs();
    assertEquals( "File System is hdfs", "hdfs", fs.getUri().getScheme());

    Path testFile = new Path("testFile");
    FSDataOutputStream out = null;
    FSDataInputStream in = null;
    final String testData = "HelloWorld";

    try {
        /** Create our test file and write our test string to it. The writeUTF ➥
         method writes some header information to the file. */
        out = fs.create(testFile,false);
        out.writeUTF(testData);
        /** With HDFS the file really doesn't exist until after it has been
          * closed. */
        out.close();
        out = null;

        /** Verify that the file exists. Open it and read the data back
          * and verify the data. */
        assertTrue( "Test File " + testFile + " exists", fs.exists(testFile));
        in = fs.open(testFile);
        String readBack = in.readUTF();
        assertEquals("Read our test data back: " + testData, testData, readBack);
        in.close();
        in=null;

    } finally {
        /** Our traditional finally when descriptors were opened
          * to ensure they are closed. */
        Utils.closeIf(out);
        Utils.closeIf(in);
    }
}
```

The test grabs the `FileSystem` object out of the base class using the following call:

```
final FileSystem fs = getFs();
```

This ensures that the file operations will be on the virtual cluster's HDFS.

As a double layer of paranoia, the following line verifies that the file system is in fact an HDFS file system:

```
assertEquals( "File System is hdfs", "hdfs", fs.getUri().getScheme());
```

The following line will create `testFile`, if there is not an existing file by that name. The file handle is returned and stored in `out`.

```
out = fs.create(testFile,false);
```

The following line writes the string `testData` with a small header to `testFile`.

```
out.writeUTF(testData);
```

At least through Hadoop 0.19.1, files do not really become available until after they are closed; therefore, the file is closed via `out.close()`. Since the `try` block will also call the `close()` method on `out`, if `out` is not set to `null`, `out` is set to `null` to avoid a duplicate `close()`.

The following line will trigger an exception if `fs.exists(testFile)` is not `true`. This verifies that the file exists in the file system.

```
assertTrue( "Test File " + testFile + " exists", fs.exists(testFile));
```

At this point in the test case, the file has been created and is known to exist. The test case will now open the file and read the contents, verifying that the contents are exactly what was written.

To open the file, rather than to create it, the following line is used:

```
in = fs.open(testFile);
```

The first line in the next snippet reads back one UTF8 string, and the next line verifies that the expected data was read.

```
String readBack = in.readUTF();
assertEquals("Read our test data back: " + testData, testData, readBack);
```

A Test Case That Launches a MapReduce Job

In this example, we will go over a test case that actually calls a MapReduce job and examines the output. My favorite initial testing tool is the `PiEstimator` example in the `hadoop-<rel>-examples` JAR, which is the class for which this unit test is built. The `PiEstimator` class, as it stands, is not unit test-friendly, and very little information can be extracted. The only thing that can be done to verify the result is to examine the estimated value of pi.

As is common in unit tests, this test case declares that it is in the same package as the class under test:

```
package org.apache.hadoop.examples;
```

The full text file is `PiEstimatorTest.java`.

The `PiEstimator` test class started life as a copy of `SimpleUnitTest,java`. This copy was then modified to highlight the relevant details for the `PiEstimator` test case. The `PiEstimator.startVirtualCluster` method has been modified to reduce the logging verbosity of the virtual cluster server processes, as shown in Listing 7-17.

Listing 7-17: Reduction in Logging Level for the Virtual Cluster

```
/** Turn down the cluster logging to filter the noise out.
  * Do this if the test is basically working. */
final String rootLogLevel = System.getProperty("virtual.cluster.logLevel","WARN");
final String testLogLevel = System.getProperty("test.log.level", "INFO");
LOG.info("Setting Log Level to " + rootLogLevel);
LogManager.getRootLogger().setLevel(Level.toLevel(rootLogLevel));

/** Turn up the logging on this class and the delegate. */
LOG.setLevel(Level.toLevel(testLogLevel));
ClusterMapReduceDelegate.LOG.setLevel(Level.toLevel(testLogLevel));
```

No changes have been made to the `stopVirtualCluster()` method.

> **Note** If you wish to interact with the HDFS or the JobTracker web interface, it is necessary to put a breakpoint on `stopVirtualCluster()`, to prevent the cluster from being torn down. When the debugger breaks there, the various servers are still running and available. The NameNode web GUI is not known to work correctly with

Hadoop 0.19.0 under the virtual cluster.

The actual test case is very different from `SimpleUnitTest,java`. The method under test is `launch()`, and the test method is `testLaunch()`. The preamble, shown in Listing 7-18, logs a couple of key pieces of information to enable the test runner to interact with the virtual cluster services. The JobTracker URL will let you interact with the running or finished job, examine the task outputs, and look at the job counters. The HDFS URL will let you interact directly with the file system to view the data files.

Listing 7-18: Test Member Preamble with Useful Debugging Information

```
@Test
public void testLaunch() throws Exception {
    final FileSystem fs = getFs();
    final JobConf testBaseConf = getConf();
    LOG.info( "The HDFS url is " + fs.getUri());
    LOG.info( "The Jobtracker URL is " + getJobtrackerURL());
    LOG.info( "The Namenode URL is " + getNamenodeURL());
```

Note When interacting with the JobTracker web interface, the URLs for the task log files are generated with fictitious names of the form `hostX.foo.com`. Replace the fictitious hostname with `localhost`, and you will be able to fetch the task logs. Alter `http://host0.foo.com:3126/tasklog?taskid=attempt_200903130041_0001_m_000000_0&all=true` to `http://localhost:3126/tasklog?taskid=attempt_200903130041_0001_m_000000_0&all=true`.

Listing 7-19 constructs a new `JobConf` object out of the test default configuration via `JobConf conf = new JobConf(testBaseConf)`. This is a highly recommended practice, as the object the method `ClusterMapReduceDelegate.getConf()` returns is shared by all test cases of that virtual cluster instance. It is actually the `JobConf` object returned by `ClusterMapReduceTestCase.createJobConf()` method. As a debugging nicety, the next line tells the TaskTrackers to send all task output to the console of the process that submitted the job. This is set using the configuration parameter `jobclient.output.filter` to `ALL`.

Listing 7-19: Set Up the JobConf Object for the Class Tested

```
/** Make a new {@link JobConf} object that is set up to
  * ensure that the jar containing {@link PiEstimator}
 * is available to the TaskTrackers.
 *
 * Note: It is very bad practice to modify the configuration given back by getConf()
 * as the returned object is shared among all tests in the Test file.
 */
JobConf conf = new JobConf(testBaseConf);
/** Make all task output come to the console of the unit test. */
conf.set("jobclient.output.filter","ALL");

/** Ensure that hadoop- -examples.jar is pushed into the DistributedCache
  * and made available to the TaskTrackers.
 *
 */
conf.setJarByClass(PiEstimator.class);
```

Tip It is always wise to ensure that the JAR that contains your MapReduce classes is part of the classpath for tasks, and the `conf.setJarByClass(PiEstimator.class)` call ensures that.

The Magic of Task Output Filtering

The configuration parameter `jobclient.output.filter` specifies what output, if any, from the tasks are printed on the console of the job submitter. The valid values are as follows:

- `ALL`: Return all task output.
- `NONE`: Return no task output.

- `KILLED`: Return output from tasks that are killed.
- `FAILED`: Return output from tasks that failed.
- `SUCCEEDED`: Return output from tasks that succeeded

The default value is `FAILED`, and only failed tasks have their output printed.

The `PiEstimator` instance needs to be created and configured, as shown in Listing 7-20.

Listing 7-20: Preparing the PiEstimator Instance to Be Run

```
/** Create the PiEstimator object and initialize it with our conf object. */
PiEstimator toTest = new PiEstimator();
toTest.setConf(conf);
```

The `launch()` method is invoked, and the results are tested, as shown in Listing 7-21. This requires knowledge of the proper arguments to the method.

Listing 7-21: Actually Calling the PiEstimator.launch() Method and Testing the Result

```
int maps = 10;
long samples = 1000;
double result = toTest.launch(maps, samples, null, null);
LOG.info("The computed result for pi is " + result);
assertTrue("Result Pi >3 ", result > 3);
assertTrue("Result Pi <4 ", result < 4);
```

This completes the walk-through of a unit test that invokes a MapReduce job.

Running the Debugger on MapReduce Jobs

There are several basic strategies for running a MapReduce job under the debugger. Here, we'll start with simplest and move to the more complex methods.

Increasing the MapReduce Job Timeout Length

When running MapReduce jobs under a debugger, it is important to drastically increase the value of the configuration key `mapred.task.timeout`. The default value is `600000`, or 10 minutes. When you are single-stepping through a map or reduce task, it is common for more than 10 minutes to pass. If the value has not been lengthened, the task you are debugging will be killed.

You can set the task timeout length to a large long value via the `bin/hadoop jar` command line, using `-Dmapred.task.timeout`, after the main class specification, if the job uses the `GenericOptionsParser` that the `ToolRunner` class provides. For example, to set a 2-hour timeout, uses the value `7200000`, as follows:

```
bin/hadoop jar job.jar main.class -Dmapred.task.timeout=7200000 other arguments
```

This value is parsed as a long, so values up to `9223372036854775807` will work on 32-bit JVMs.

Running an Entire MapReduce Job in a Single JVM

The normal process of job submission involves the `JobClient` class storing all of the relevant information about the job in HDFS, and then making an RPC call to the JobTracker to submit the job for execution. The `JobClient` determines the address of the JobTracker through the configuration value of the configuration key `mapred.job.tracker`. This configuration key may have the value of `local`, in which case the entire MapReduce job will be run in the JVM that is submitting the job. This is ideal for debugging small-scale problems.

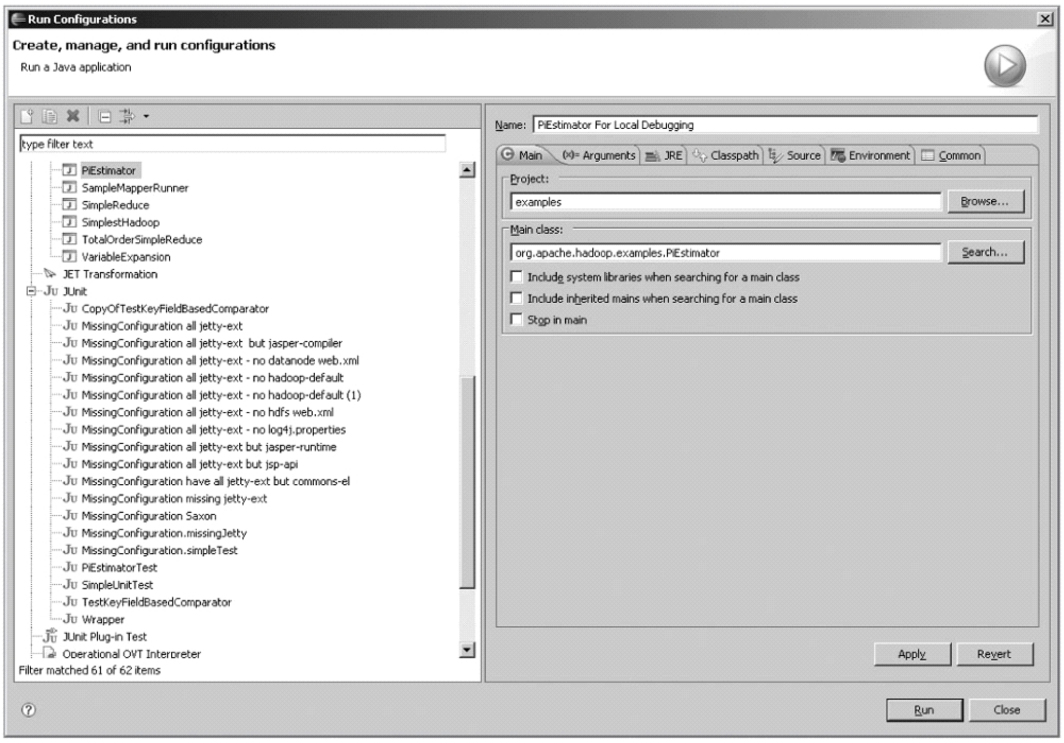
There are some restrictions to this technique. The cause of most of the restrictions is that the map and reduce tasks do not run in their own JVMs. There is no way to change the JVM working directory, classpath, or other command-line configured options. A significant result of this is that the `DistributedCache` behavior is very different. If your job relies on the `DistributedCache`, this method will not be a good debugging choice. The lifetime of your classes will be longer than you expect, and you may experience unexpected results due to prior variable initialization for static variables.

The number of reduce tasks is limited to zero or one. If your job requires more than one reduce task, this method will not be a good debugging choice.

The example in this section uses our old friend `PiEstimator`, the example that comes with Hadoop Core, as the MapReduce job.

Figures 7-1 through 7-4 are guides to configuring a run/debug profile for a MapReduce application so that it may be run using the `LocalJobRunner`, JobTracker, in a single JVM.

The classpath must include the Core JAR, the example JAR (since the test case comes from this JAR), and all of the JARs from `lib` and `lib/jetty-ext`, as shown in Figure 7-1. You must explicitly specify all of the JARs required for your job here. In this example, Eclipse is not configured to load the native compression codec libraries.

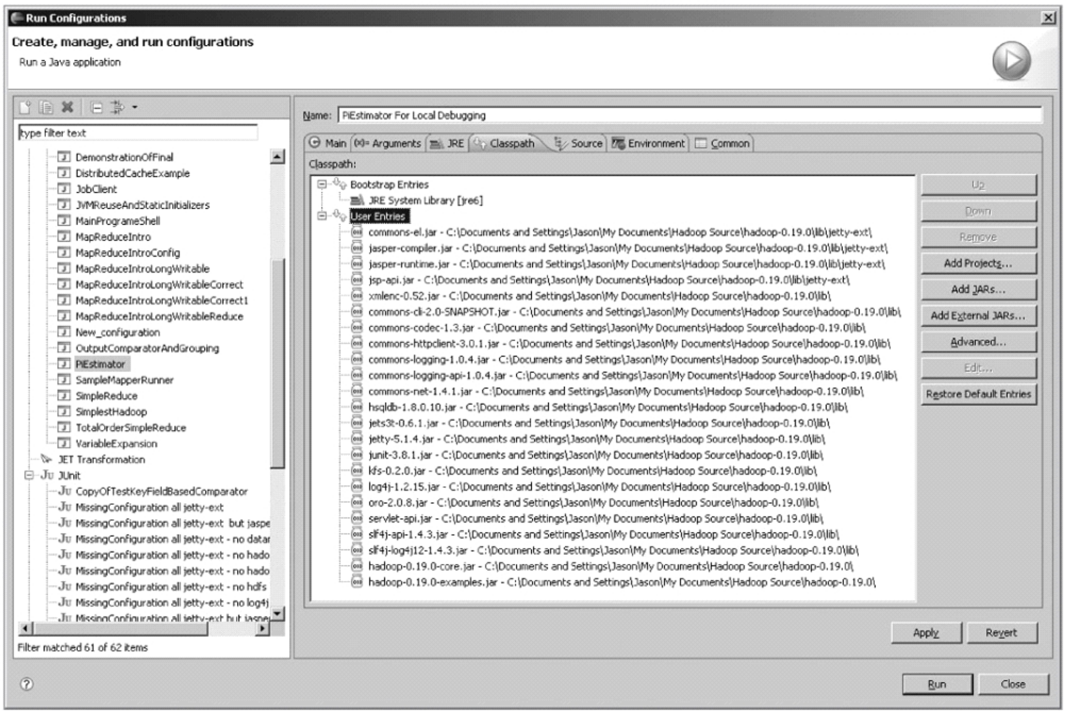


```
PiEstimator
SampleMapperRunner
SimpleReduce
SimplestHadoop
TotalOrderSimpleReduce
VariableExpansion
JET Transformation
JUnit
  CopyOfTestKeyFieldBasedComparator
  MissingConfiguration all jetty-ext
  MissingConfiguration all jetty-ext  but jasper-compiler
  MissingConfiguration all jetty-ext - no datanode web.xml
  MissingConfiguration all jetty-ext - no hadoop-default
  MissingConfiguration all jetty-ext - no hadoop-default (1)
  MissingConfiguration all jetty-ext - no hdfs web.xml
  MissingConfiguration all jetty-ext - no log4j.properties
  MissingConfiguration all jetty-ext but jasper-runtime
  MissingConfiguration all jetty-ext but jsp-api
  MissingConfiguration have all jetty-ext but commons-el
  MissingConfiguration missing jetty-ext
  MissingConfiguration Saxon
  MissingConfiguration.missingJetty
  MissingConfiguration.simpleTest
  PiEstimatorTest
  SimpleUnitTest
  TestKeyFieldBasedComparator
  Wrapper
JUnit Plug-in Test
Operational OVT Interpreter
Filter matched 61 of 62 items
```

```
Name: PiEstimator For Local Debugging
Project: examples
Main class: org.apache.hadoop.examples.PiEstimator
[ ] Include system libraries when searching for a main class
[ ] Include inherited mains when searching for a main class
[ ] Stop in main
```

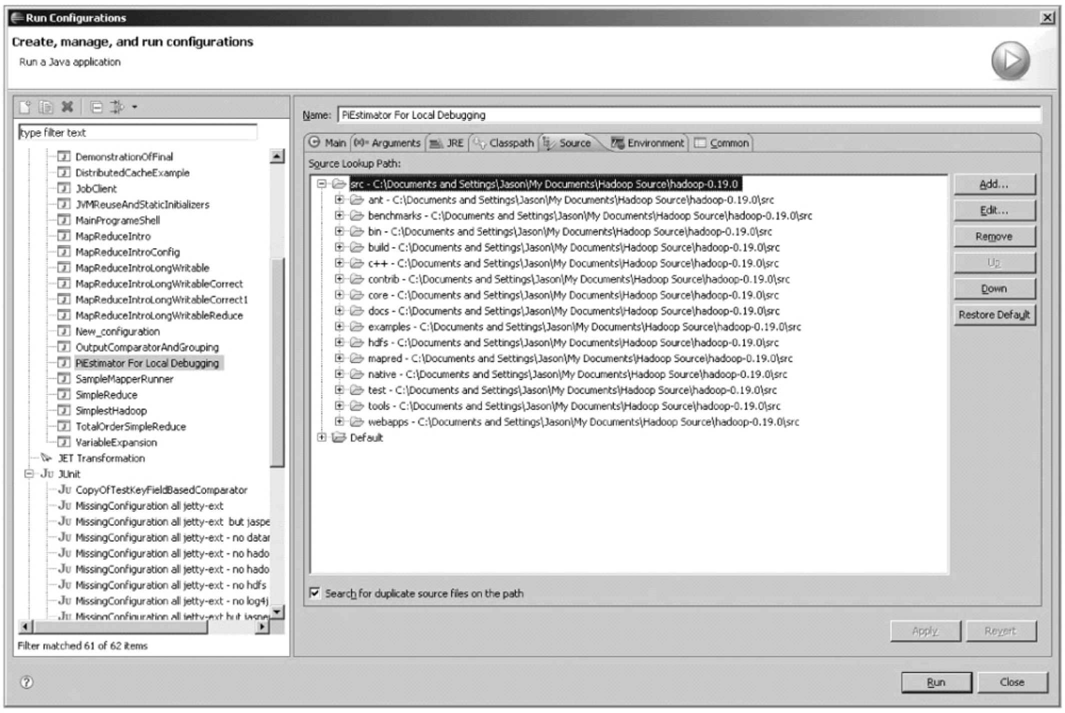
Figure 7-1: Creating a run configuration for the PiEstimator in Eclipse

What is the point of using a visual debugger if the source code is not available? You will need to set up any source paths required for your application, as shown in Figure 7-3.



```
Run Configurations
Create, manage, and run configurations
Run a Java application

type filter text
    DemonstrationOfFinal
    DistributedCacheExample
    JobClient
    JVMReuseAndStaticInitializers
    MainProgrameShell
    MapReduceIntro
    MapReduceIntroConfig
    MapReduceIntroLongWritable
    MapReduceIntroLongWritableCorrect
    MapReduceIntroLongWritableCorrect1
    MapReduceIntroLongWritableReduce
    New_configuration
    OutputComparatorAndGrouping
    PiEstimator
    SampleMapperRunner
    SimpleReduce
    SimplestHadoop
    TotalOrderSimpleReduce
    VariableExpansion
  JET Transformation
  JUnit
    CopyOfTestKeyFieldBasedComparator
    MissingConfiguration all jetty-ext
    MissingConfiguration all jetty-ext  but jaspe
    MissingConfiguration all jetty-ext - no datar
    MissingConfiguration all jetty-ext - no hado
    MissingConfiguration all jetty-ext - no hado
    MissingConfiguration all jetty-ext - no hdfs
    MissingConfiguration all jetty-ext - no log4j
Filter matched 61 of 62 items

Name: PiEstimator For Local Debugging
Main | Arguments | JRE | Classpath | Source | Environment | Common
Classpath:
Bootstrap Entries
    JRE System Library [jre6]
User Entries
    commons-el.jar - C:\Documents and Settings\Jason\My Documents\Hadoop Source\hadoop-0.19.0\lib\jetty-ext\
    jasper-compiler.jar - C:\Documents and Settings\Jason\My Documents\Hadoop Source\hadoop-0.19.0\lib\jetty-ext\
    jasper-runtime.jar - C:\Documents and Settings\Jason\My Documents\Hadoop Source\hadoop-0.19.0\lib\jetty-ext\
    jsp-api.jar - C:\Documents and Settings\Jason\My Documents\Hadoop Source\hadoop-0.19.0\lib\jetty-ext\
    xmlenc-0.52.jar - C:\Documents and Settings\Jason\My Documents\Hadoop Source\hadoop-0.19.0\lib\
    commons-cli-2.0-SNAPSHOT.jar - C:\Documents and Settings\Jason\My Documents\Hadoop Source\hadoop-0.19.0\lib\
    commons-codec-1.3.jar - C:\Documents and Settings\Jason\My Documents\Hadoop Source\hadoop-0.19.0\lib\
    commons-httpclient-3.0.1.jar - C:\Documents and Settings\Jason\My Documents\Hadoop Source\hadoop-0.19.0\lib\
    commons-logging-1.0.4.jar - C:\Documents and Settings\Jason\My Documents\Hadoop Source\hadoop-0.19.0\lib\
    commons-logging-api-1.0.4.jar - C:\Documents and Settings\Jason\My Documents\Hadoop Source\hadoop-0.19.0\lib\
    commons-net-1.4.1.jar - C:\Documents and Settings\Jason\My Documents\Hadoop Source\hadoop-0.19.0\lib\
    hsqldb-1.8.0.10.jar - C:\Documents and Settings\Jason\My Documents\Hadoop Source\hadoop-0.19.0\lib\
    jets3t-0.6.1.jar - C:\Documents and Settings\Jason\My Documents\Hadoop Source\hadoop-0.19.0\lib\
    jetty-5.1.4.jar - C:\Documents and Settings\Jason\My Documents\Hadoop Source\hadoop-0.19.0\lib\
    junit-3.8.1.jar - C:\Documents and Settings\Jason\My Documents\Hadoop Source\hadoop-0.19.0\lib\
    kfs-0.2.0.jar - C:\Documents and Settings\Jason\My Documents\Hadoop Source\hadoop-0.19.0\lib\
    log4j-1.2.15.jar - C:\Documents and Settings\Jason\My Documents\Hadoop Source\hadoop-0.19.0\lib\
    oro-2.0.8.jar - C:\Documents and Settings\Jason\My Documents\Hadoop Source\hadoop-0.19.0\lib\
    servlet-api.jar - C:\Documents and Settings\Jason\My Documents\Hadoop Source\hadoop-0.19.0\lib\
    slf4j-api-1.4.3.jar - C:\Documents and Settings\Jason\My Documents\Hadoop Source\hadoop-0.19.0\lib\
    slf4j-log4j12-1.4.3.jar - C:\Documents and Settings\Jason\My Documents\Hadoop Source\hadoop-0.19.0\lib\
    hadoop-0.19.0-core.jar - C:\Documents and Settings\Jason\My Documents\Hadoop Source\hadoop-0.19.0\
    hadoop-0.19.0-examples.jar - C:\Documents and Settings\Jason\My Documents\Hadoop Source\hadoop-0.19.0\

Up | Down | Remove | Add Projects... | Add JARs... | Add External JARs... | Advanced... | Edit... | Restore Default Entries
Apply | Revert
Run | Close
```

Figure 7-2: Configuring the classpath



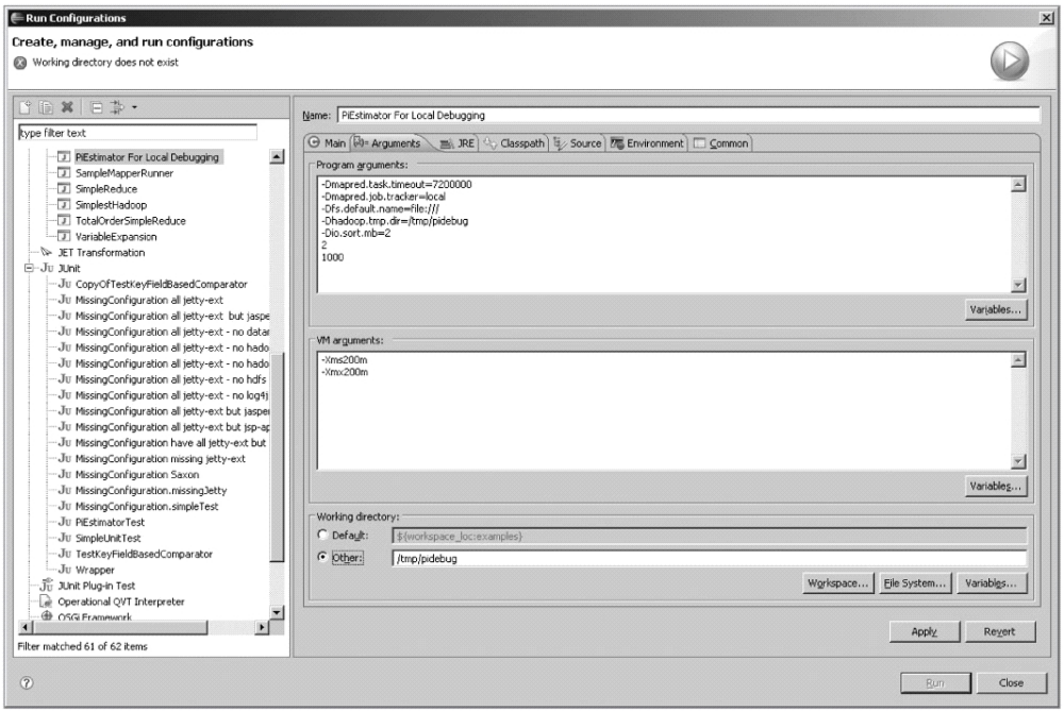
```
Run Configurations
Create, manage, and run configurations
Run a Java application

type filter text
    DemonstrationOfFinal
    DistributedCacheExample
    JobClient
    JVMReuseAndStaticInitializers
    MainProgrameShell
    MapReduceIntro
    MapReduceIntroConfig
    MapReduceIntroLongWritable
    MapReduceIntroLongWritableCorrect
    MapReduceIntroLongWritableCorrect1
    MapReduceIntroLongWritableReduce
    New_configuration
    OutputComparatorAndGrouping
    PiEstimator For Local Debugging
    SampleMapperRunner
    SimpleReduce
    SimplestHadoop
    TotalOrderSimpleReduce
    VariableExpansion
  JET Transformation
  JUnit
    CopyOfTestKeyFieldBasedComparator
    MissingConfiguration all jetty-ext
    MissingConfiguration all jetty-ext  but jaspe
    MissingConfiguration all jetty-ext - no datar
    MissingConfiguration all jetty-ext - no hado
    MissingConfiguration all jetty-ext - no hado
    MissingConfiguration all jetty-ext - no hdfs
    MissingConfiguration all jetty-ext - no log4j
Filter matched 61 of 62 items

Name: PiEstimator For Local Debugging
Main | Arguments | JRE | Classpath | Source | Environment | Common
Source Lookup Path:
src - C:\Documents and Settings\Jason\My Documents\Hadoop Source\hadoop-0.19.0
    ant - C:\Documents and Settings\Jason\My Documents\Hadoop Source\hadoop-0.19.0\src
    benchmarks - C:\Documents and Settings\Jason\My Documents\Hadoop Source\hadoop-0.19.0\src
    bin - C:\Documents and Settings\Jason\My Documents\Hadoop Source\hadoop-0.19.0\src
    build - C:\Documents and Settings\Jason\My Documents\Hadoop Source\hadoop-0.19.0\src
    c++ - C:\Documents and Settings\Jason\My Documents\Hadoop Source\hadoop-0.19.0\src
    contrib - C:\Documents and Settings\Jason\My Documents\Hadoop Source\hadoop-0.19.0\src
    core - C:\Documents and Settings\Jason\My Documents\Hadoop Source\hadoop-0.19.0\src
    docs - C:\Documents and Settings\Jason\My Documents\Hadoop Source\hadoop-0.19.0\src
    examples - C:\Documents and Settings\Jason\My Documents\Hadoop Source\hadoop-0.19.0\src
    hdfs - C:\Documents and Settings\Jason\My Documents\Hadoop Source\hadoop-0.19.0\src
    mapred - C:\Documents and Settings\Jason\My Documents\Hadoop Source\hadoop-0.19.0\src
    native - C:\Documents and Settings\Jason\My Documents\Hadoop Source\hadoop-0.19.0\src
    test - C:\Documents and Settings\Jason\My Documents\Hadoop Source\hadoop-0.19.0\src
    tools - C:\Documents and Settings\Jason\My Documents\Hadoop Source\hadoop-0.19.0\src
    webapps - C:\Documents and Settings\Jason\My Documents\Hadoop Source\hadoop-0.19.0\src
Default

[✓] Search for duplicate source files on the path

Add... | Edit... | Remove | Up | Down | Restore Default
Apply | Revert
Run | Close
```

Figure 7-3: Configuring the source path

You also need to set up the following arguments, as shown in Figure 7-4:

- `-Dmapred.task.timeout=7200000`: Ensure that your tasks are not killed by the framework for not responding.
- `-Dmapred.job.tracker=local`: Run the MapReduce job using the `LocalJobRunner`, entirely in this JVM.
- `-Dfs.default.name=file:///`: Use the local file system for all storage.
- `-Dhadoop.tmp.dir=/tmp/pidebug`: Store all working files in this directory.
- `-Dio.sort.mb=2`: Allocate only 2MB for the merge-sort working space.

- `2`: Run two map tasks.
- `1000`: Generate 1,000 random samples in each map task.



Name: PiEstimator For Local Debugging

Program arguments:

```
-Dmapred.task.timeout=7200000
-Dmapred.job.tracker=local
-Dfs.default.name=file:///
-Dhadoop.tmp.dir=/tmp/pidebug
-Dio.sort.mb=2
2
1000
```

VM arguments:

```
-Xms200m
-Xmx200m
```

Working directory: ( ) Default: ${workspace_loc:examples} [✓ Other:] /tmp/pidebug

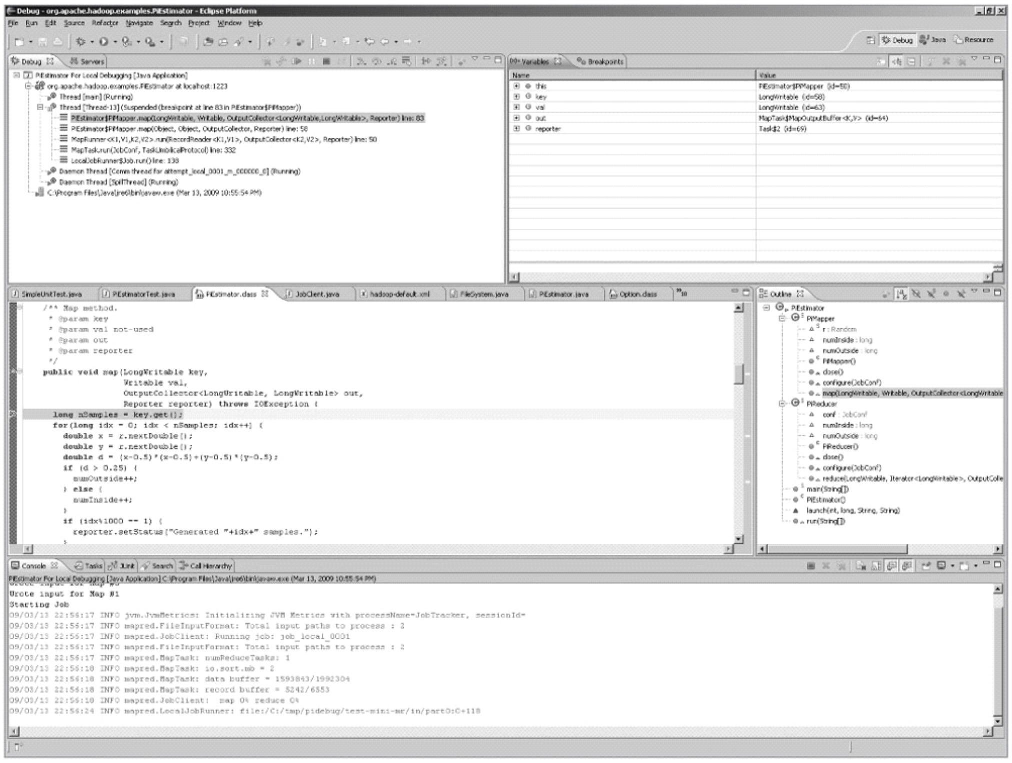
Figure 7-4: Configuring the command-line arguments for debugging. Note the need to make the working directory.

Once the directory `/tmp/pidebug` has been made, the Run button in the bottom-right corner of the Run Configurations window will be active. Click this button just to verify that the job will work in this setting. The output should be as follows:

```
Number of Maps = 2 Samples per Map = 1000
Wrote input for Map #0
Wrote input for Map #1
Starting Job
jvm.JvmMetrics: Initializing JVM Metrics with processName=JobTracker, sessionId=
mapred.FileInputFormat: Total input paths to process : 2
mapred.JobClient: Running job: job_local_0001
mapred.FileInputFormat: Total input paths to process : 2
mapred.MapTask: numReduceTasks: 1
mapred.MapTask: io.sort.mb = 2
mapred.MapTask: data buffer = 1593843/1992304
mapred.MapTask: record buffer = 5242/6553
mapred.MapTask: Starting flush of map output
mapred.MapTask: Finished spill 0
mapred.TaskRunner: Task:attempt_local_0001_m_000000_0 is done. ➥
And is in the process of commiting
mapred.LocalJobRunner: Generated 1 samples.
mapred.TaskRunner: Task 'attempt_local_0001_m_000000_0' done.
mapred.MapTask: numReduceTasks: 1
mapred.MapTask: io.sort.mb = 2
mapred.MapTask: data buffer = 1593843/1992304
mapred.MapTask: record buffer = 5242/6553
mapred.MapTask: Starting flush of map output
mapred.MapTask: Finished spill 0
mapred.TaskRunner: Task:attempt_local_0001_m_000001_0 is done. ➥
And is in the process of commiting
mapred.LocalJobRunner: Generated 1 samples.
mapred.TaskRunner: Task 'attempt_local_0001_m_000001_0' done.
mapred.Merger: Merging 2 sorted segments
mapred.Merger: Down to the last merge-pass, with 2 segments left ➥
of total size: 76 bytes
mapred.TaskRunner: Task:attempt_local_0001_r_000000_0 is done. ➥
And is in the process of commiting
mapred.LocalJobRunner:
mapred.TaskRunner: Task attempt_local_0001_r_000000_0 is allowed to commit now
mapred.FileOutputCommitter: Saved output of task 'attempt_local_0001_r_ ➥
000000_0' to file:/C:/tmp/pidebug/test-mini-mr/out
mapred.LocalJobRunner: reduce > reduce
mapred.TaskRunner: Task 'attempt_local_0001_r_000000_0' done.
mapred.JobClient: Job complete: job_local_0001
mapred.JobClient: Counters: 11
mapred.JobClient:   File Systems
mapred.JobClient:     Local bytes read=450833
mapred.JobClient:     Local bytes written=503689

mapred.JobClient:   Map-Reduce Framework
mapred.JobClient:     Reduce input groups=2
mapred.JobClient:     Combine output records=0
mapred.JobClient:     Map input records=2
mapred.JobClient:     Reduce output records=0
mapred.JobClient:     Map output bytes=64
mapred.JobClient:     Map input bytes=48
mapred.JobClient:     Combine input records=0
mapred.JobClient:     Map output records=4
Job Finished in 1.547 seconds
mapred.JobClient:     Reduce input records=4
Estimated value of PI is 3.132
```

The final line, with the estimated value of pi, indicates that the environment is correctly configured.

For Figure 7-5, a breakpoint was set in the map task, and the job launched via the Debug As Java Application menu item.



```
PiEstimator For Local Debugging [Java Application]
  org.apache.hadoop.examples.PiEstimator at localhost:1223
    Thread [main] (Running)
    Thread [Thread-13] (Suspended (breakpoint at line 83 in PiEstimator$PiMapper))
      PiEstimator$PiMapper.map(LongWritable, Writable, OutputCollector<LongWritable,LongWritable>, Reporter) line: 83
      PiEstimator$PiMapper.map(Object, Object, OutputCollector, Reporter) line: 58
      MapRunner<K1,V1,K2,V2>.run(RecordReader<K1,V1>, OutputCollector<K2,V2>, Reporter) line: 50
      MapTask.run(JobConf, TaskUmbilicalProtocol) line: 332
      LocalJobRunner$Job.run() line: 138
    Daemon Thread [Comm thread for attempt_local_0001_m_000000_0] (Running)
    Daemon Thread [SpillThread] (Running)
  C:\Program Files\Java\jre6\bin\javaw.exe (Mar 13, 2009 10:55:54 PM)
```

```
    /** Map method.
     * @param key
     * @param val not-used
     * @param out
     * @param reporter
     */
    public void map(LongWritable key,
                    Writable val,
                    OutputCollector<LongWritable, LongWritable> out,
                    Reporter reporter) throws IOException {
      long nSamples = key.get();
      for(long idx = 0; idx < nSamples; idx++) {
        double x = r.nextDouble();
        double y = r.nextDouble();
        double d = (x-0.5)*(x-0.5)+(y-0.5)*(y-0.5);
        if (d > 0.25) {
          numOutside++;
        } else {
          numInside++;
        }
        if (idx%1000 == 1) {
          reporter.setStatus("Generated "+idx+" samples.");
        }
```

```
PiEstimator For Local Debugging [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Mar 13, 2009 10:55:54 PM)
Wrote input for Map #1
Starting Job
09/03/13 22:56:17 INFO jvm.JvmMetrics: Initializing JVM Metrics with processName=JobTracker, sessionId=
09/03/13 22:56:17 INFO mapred.FileInputFormat: Total input paths to process : 2
09/03/13 22:56:17 INFO mapred.JobClient: Running job: job_local_0001
09/03/13 22:56:17 INFO mapred.FileInputFormat: Total input paths to process : 2
09/03/13 22:56:17 INFO mapred.MapTask: numReduceTasks: 1
09/03/13 22:56:18 INFO mapred.MapTask: io.sort.mb = 2
09/03/13 22:56:18 INFO mapred.MapTask: data buffer = 1593843/1992304
09/03/13 22:56:18 INFO mapred.MapTask: record buffer = 5242/6553
09/03/13 22:56:18 INFO mapred.JobClient:  map 0% reduce 0%
09/03/13 22:56:24 INFO mapred.LocalJobRunner: file:/C:/tmp/pidebug/test-mini-mr/in/part0:0+118
```

Figure 7-5: Eclipse with PiEstimator stopped in the map task

The job is configured to use the local file system for all working storage, and you may examine the files using the normal file system tools.

In this case, with the `PiEstimator` test stopped in the first line of the first map task, you can see that the working directory in the local file system contains a number of interesting files.

```
mapred/local/localRunner/.job_local_0001.xml.crc
mapred/local/localRunner/.split.dta.crc
mapred/local/localRunner/job_local_0001.xml
mapred/local/localRunner/split.dta
mapred/system/job_local_0001/.job.jar.crc
mapred/system/job_local_0001/.job.split.crc
mapred/system/job_local_0001/.job.xml.crc
mapred/system/job_local_0001/job.jar
mapred/system/job_local_0001/job.split
mapred/system/job_local_0001/job.xml
test-mini-mr/in/.part0.crc
test-mini-mr/in/.part1.crc
test-mini-mr/in/part0
test-mini-mr/in/part1
```

These files contain the following:

- `mapred/local/localRunner/job_local_0001.xml` contains the textual representation of the localized `JobConf` object for the task in progress, which is the task that is stopped in the debugger.
- `mapred/system/job_local_0001/job.xml` contains the textual representation of the `JobConf` object for the job. The file names that end in `.crc` are checksum files written out by the framework to provide data integrity checking.
- `mapred/local/localRunner/split.dta` contains the input split class and the data fields of the split that this task is to use as input.
- `mapred/system/job_local_0001/job.split` contains the list of input splits. It is prepared by the class `JobClient` as part of the job submission process.
- `test-mini-mr/in/part0` and `test-mini-mr/in/part1` are the input files to the job, prepared by the `PiEstimator` class. The `PiEstimator` class writes out `SequenceFiles` for its input, and these binary files may be examined with the command-line Hadoop tool via `bin/hadoop dfs -fs file:/// -text /tmp/pidebug/test-mini-mr/in/part1`.

Note `-fs file:///` indicates that the local file system is to be used, and `-text` causes the input file to be read as a `SequenceFile` and the key/value pairs printed via their respective `toString()` methods.

The input files `part0` and `part1` each has a single record of `1000, 0`. Let's examine the contents of `part0` of the input to demonstrate this.

```
> bin/hadoop dfs -fs file:/// -text /tmp/pidebug/test-mini-mr/in/part0
```

```
1000    0
```

Debugging a Task Running on a Cluster

Hadoop Core, as of version 0.19.0, does not provide any tools to specify which tasks of a job to enable Java debugging services on, nor does Hadoop Core provide a way to indicate on which host and port such a task might be listening for remote debugging connections. To debug tasks running on a cluster, the JVM parameters for the task have remote debugging enabled via the command-line arguments. The core issue is to arrange for the JVM of the server or task that is to be debugged to have the additional command-line arguments that enable the Java Platform Debugger Architecture (JPDA) servers. Table 7-3 describes the parameters to the JPDA debugging agent, `agentlib:jdwp`, that must be enabled in the task JVM to allow connections from debuggers.

Table 7-3: Configuration Parameters for the Debugger Agent

Parameter	Description

`suspend`	Set this to `suspend=y` to suspend the task on start; otherwise, the task will run normally until a debugger connects to it.
`address`	Set this to `address=0.0.0.0` to have the debugger agent bind to the machine wild-card address at an allocated port. This address is specific to IP version 4.
`launch`	A program to invoke when the debugger initializes. The program receives two arguments: the transport, `dt_socket`, and the allocated port. Only a program name is accepted as a value. This program may be used to provide notification that the task has engaged the debugger agent and the port it is listening on for a debugger connection.
`onthrow`	Do not initialize the debugger agent until an exception with the parameter value is thrown. In the example `onthrow=java.io.IOException`, the debugger agent would not initialize and bind a port until an `IOException` was thrown. At this point, any program defined by `launch` would be executed. Program execution will be stopped, while the agent waits for a debugger connection.
`onuncaught`	Works like `onthrow`, except that the exception must not be caught.

Tip The Sun VM Invocation Options guide at `http://java.sun.com/javase/6/docs/technotes/guides/jpda/ conninv.html#Invocation` provides many details on how the command-line arguments for debugging may be constructed.

This section will consider only using the debugger configured for TCP/socket-based transport, à *la* remote debugging, in Eclipse.

The configuration parameter that needs to be set is `mapred.child.java.opts`, which may be set at the job level. The JPDA invocation arguments need to be added to the value for this parameter. The parameters that are most relevant to the user are `suspend` and `address`. The `suspend=y` setting forces the JVM to be stopped just before the `main()` method is invoked. The value `suspend=n` may also be used, in which case the JVM will run normally. The parameter `address=0.0.0.0` instructs the debugger interface in the JVM to allocate a free port and to listen for connections on the wildcard address of the local machine. The port that is allocated will be printed on the standard output.

Note Unless the cluster is configured for one map task per machine, and the job has no reduce tasks specified, there will be multiple tasks running on any given TaskTracker machine. This precluded specifying a fixed port as part of the address parameter. If only a single task will run at a time, a port may be specified via `address=host:port`, where `host` is optional.

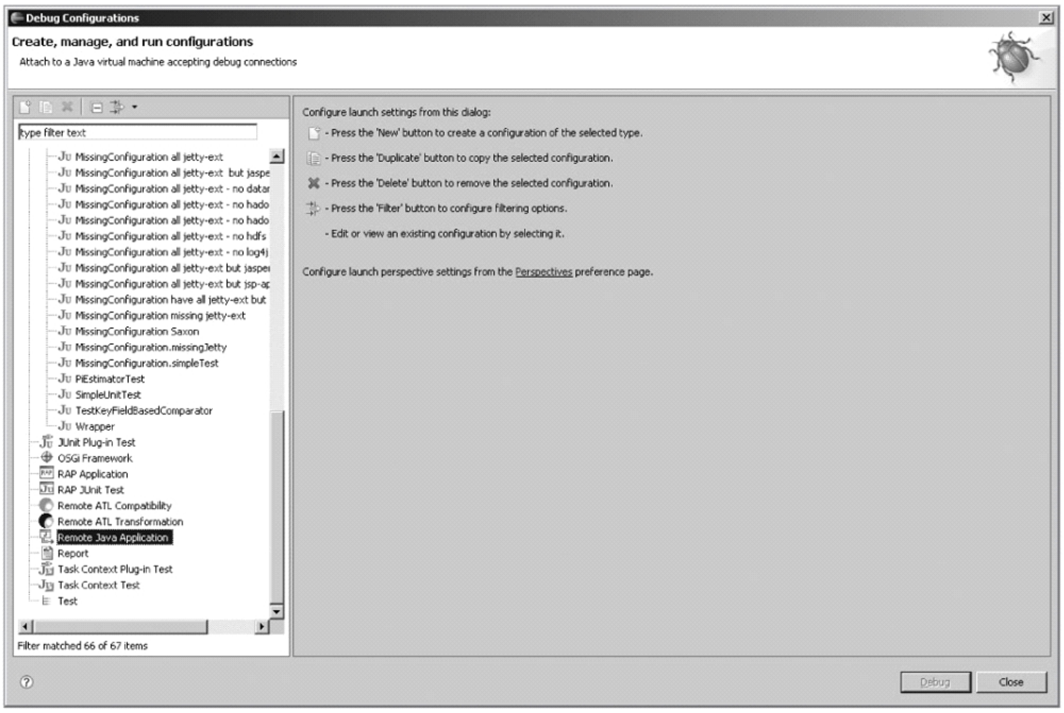
Let's start a Hadoop job with remote debugging enabled.

```
HADOOP_OPTS=-agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=0.0.0.0 ➥
bin/hadoop  dfs -fs file:/// -text /tmp/pidebug/test-mini-mr/in/part0
```

```
Listening for transport dt_socket at address: 59348
```

It is best to have only the option `suspend` set to `y` for all of the task JVMs, isolate this option to a single machine in the cluster, to avoid requiring extensive interaction with each task, at minimum, connecting to the task in the debugger to resume execution.

Figures 7-6, 7-7, and 7-8 show our friend `PiEstimator` being run on a small cluster. The JPDA arguments will be passed via the command line to the child JVM, and JVM reuse will be explicitly disabled to avoid complications. Figure 7-6 shows the Eclipse setup for a remote debugging session.



Debug Configurations

Create, manage, and run configurations

Attach to a Java virtual machine accepting debug connections

type filter text

- MissingConfiguration all jetty-ext
- MissingConfiguration all jetty-ext but jaspe
- MissingConfiguration all jetty-ext - no datar
- MissingConfiguration all jetty-ext - no hado
- MissingConfiguration all jetty-ext - no hado
- MissingConfiguration all jetty-ext - no hdfs
- MissingConfiguration all jetty-ext - no log4j
- MissingConfiguration all jetty-ext but jasper
- MissingConfiguration all jetty-ext but jsp-ap
- MissingConfiguration have all jetty-ext but
- MissingConfiguration missing jetty-ext
- MissingConfiguration Saxon
- MissingConfiguration.missingJetty
- MissingConfiguration.simpleTest
- PiEstimatorTest
- SimpleUnitTest
- TestKeyFieldBasedComparator
- Wrapper
- JUnit Plug-in Test
- OSGi Framework
- RAP Application
- RAP JUnit Test
- Remote ATL Compatibility
- Remote ATL Transformation
- Remote Java Application
- Report
- Task Context Plug-in Test
- Task Context Test
- Test

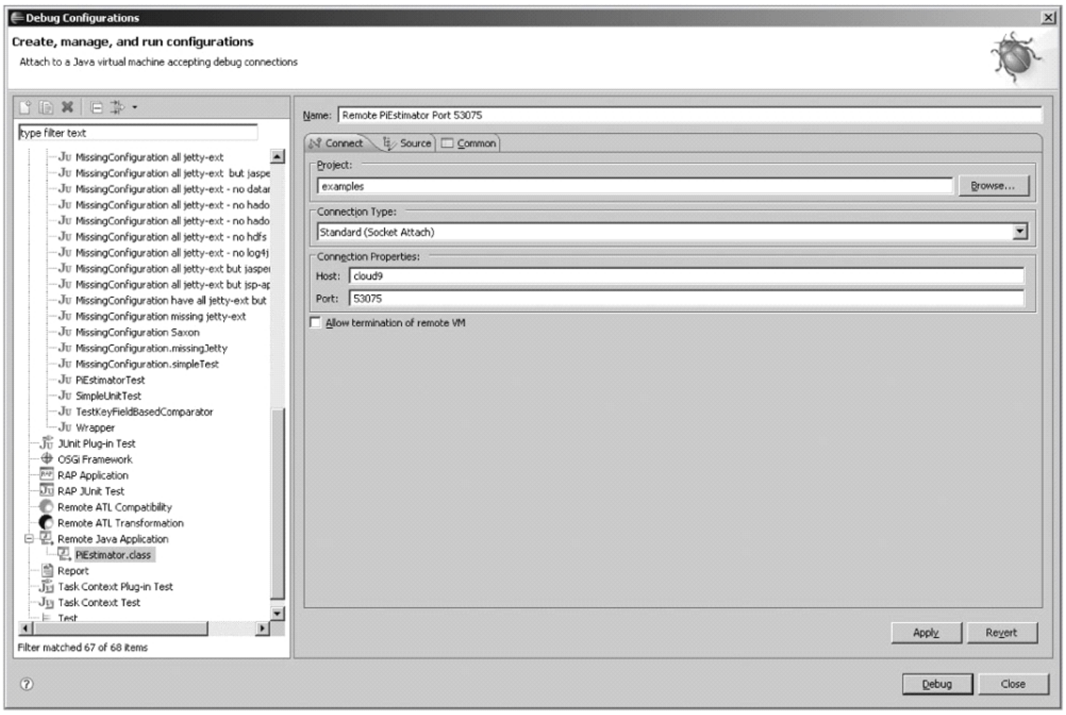
Filter matched 66 of 67 items

Configure launch settings from this dialog:

- Press the 'New' button to create a configuration of the selected type.
- Press the 'Duplicate' button to copy the selected configuration.
- Press the 'Delete' button to remove the selected configuration.
- Press the 'Filter' button to configure filtering options.
- Edit or view an existing configuration by selecting it.

Configure launch perspective settings from the Perspectives preference page.

Figure 7-6: Setting up Eclipse for a remote debugging session



Debug Configurations

Create, manage, and run configurations

Attach to a Java virtual machine accepting debug connections

type filter text

- MissingConfiguration all jetty-ext
- MissingConfiguration all jetty-ext but jaspe
- MissingConfiguration all jetty-ext - no datar
- MissingConfiguration all jetty-ext - no hado
- MissingConfiguration all jetty-ext - no hado
- MissingConfiguration all jetty-ext - no hdfs
- MissingConfiguration all jetty-ext - no log4j
- MissingConfiguration all jetty-ext but jasper
- MissingConfiguration all jetty-ext but jsp-ap
- MissingConfiguration have all jetty-ext but
- MissingConfiguration missing jetty-ext
- MissingConfiguration Saxon
- MissingConfiguration.missingJetty
- MissingConfiguration.simpleTest
- PiEstimatorTest
- SimpleUnitTest
- TestKeyFieldBasedComparator
- Wrapper
- JUnit Plug-in Test
- OSGi Framework
- RAP Application
- RAP JUnit Test
- Remote ATL Compatibility
- Remote ATL Transformation
- Remote Java Application
  - PiEstimator.class
- Report
- Task Context Plug-in Test
- Task Context Test
- Test

Filter matched 67 of 68 items

Name: Remote PiEstimator Port 53075

Connect | Source | Common

Project: examples

Connection Type: Standard (Socket Attach)

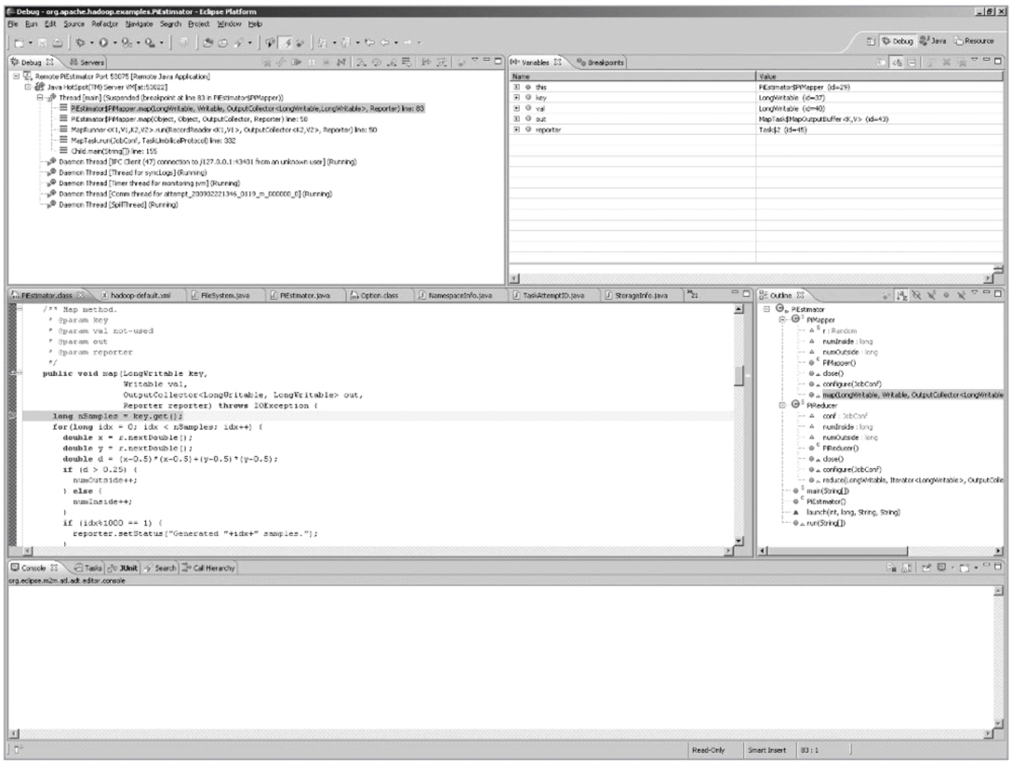
Connection Properties:

Host: cloud9

Port: 53075

Allow termination of remote VM

Figure 7-7: Specifying the port and host to connect to. This is configured per task and needs to be determined from task log files.



```
Remote PiEstimator Port 53075 [Remote Java Application]
  Java HotSpot(TM) Server VM[at:53022]
    Thread [main] (Suspended (breakpoint at line 83 in PiEstimator$PiMapper))
      PiEstimator$PiMapper.map(LongWritable, Writable, OutputCollector<LongWritable,LongWritable>, Reporter) line: 83
      PiEstimator$PiMapper.map(Object, Object, OutputCollector, Reporter) line: 58
      MapRunner<K1,V1,K2,V2>.run(RecordReader<K1,V1>, OutputCollector<K2,V2>, Reporter) line: 50
      MapTask.run(JobConf, TaskUmbilicalProtocol) line: 332
      Child.main(String[]) line: 155
    Daemon Thread [IPC Client (47) connection to /127.0.0.1:43401 from an unknown user] (Running)
    Daemon Thread [Thread for syncLogs] (Running)
    Daemon Thread [Timer thread for monitoring jvm] (Running)
    Daemon Thread [Comm thread for attempt_200902221346_0119_m_000000_0] (Running)
    Daemon Thread [SpillThread] (Running)
```

Name	Value
this	PiEstimator$PiMapper (id=29)
key	LongWritable (id=37)
val	LongWritable (id=40)
out	MapTask$MapOutputBuffer<K,V> (id=43)
reporter	Task$2 (id=45)

```
    /** Map method.
     * @param key
     * @param val not-used
     * @param out
     * @param reporter
     */
    public void map(LongWritable key,
                    Writable val,
                    OutputCollector<LongWritable, LongWritable> out,
                    Reporter reporter) throws IOException {
      long nSamples = key.get();
      for(long idx = 0; idx < nSamples; idx++) {
        double x = r.nextDouble();
        double y = r.nextDouble();
        double d = (x-0.5)*(x-0.5)+(y-0.5)*(y-0.5);
        if (d > 0.25) {
          numOutside++;
        } else {
          numInside++;
        }
        if (idx%1000 == 1) {
          reporter.setStatus("Generated "+idx+" samples.");
        }
```

```
PiEstimator
  PiMapper
    r : Random
    numInside : long
    numOutside : long
    PiMapper()
    close()
    configure(JobConf)
    map(LongWritable, Writable, OutputCollector<LongWritable
  PiReducer
    conf : JobConf
    numInside : long
    numOutside : long
    PiReducer()
    close()
    configure(JobConf)
    reduce(LongWritable, Iterator<LongWritable>, OutputColle
  main(String[])
  PiEstimator()
  launch(int, long, String, String)
  run(String[])
```

Debug - org.apache.hadoop.examples.PiEstimator - Eclipse Platform
File Run Edit Source Refactor Navigate Search Project Window Help
Debug Java Resource
Debug Servers
Variables Breakpoints
PiEstimator.class hadoop-default.xml FileSystem.java PiEstimator.java Option.class NamespaceInfo.java TaskAttemptID.java StorageInfo.java
Outline
Console Tasks JUnit Search Call Hierarchy
org.eclipse.m2m.atl.adt.editor.console
Read-Only Smart Insert 83 : 1

Figure 7-8: A remotely connected session to a map task

In the dialog box in Figure 7-7, New has been clicked, and the title of the debug session changed to `Remote PiEstimator 53075`. The Source tab has been configured identically to the earlier source configuration shown in Figure 7-3, with the `hadoop/src` directory and its subfolders. Then you click the Apply and Debug buttons, to save this session and connect to the task. Figure 7-8 shows the `PiEstimator` task in the debugger, stopped at a breakpoint in the `map()` method.

To determine the port to connect to, if no `launch` program has been provided, requires finding the per-task stdout log file. At present, this step is manual and requires issuing shell commands on one of the cluster machines.

The job ID must be determined. This is available via the JobTracker web interface or via the command-line tool:

```
bin/hadoop job -list
```

In this case, two tasks are suspended and waiting for the debugger, one at 192.168.1.2:43004 and the other at 192.168.1.2:35403. These values are what you would put into the Host and Port fields of the Remote Debugging configuration dialog box. The `slaves.sh` command will execute its command-line arguments as a `shell` command on each machine in the `conf/slaves` file, the output of these commands will have the generating host's IP address prefixed to the output lines.

Let's see an example of determining the ports and hosts of the suspended tasks:

```
> bin/hadoop job -list
```

```
1 job currently running
JobId   State   StartTime   UserName    Priority    SchedulingInfo
job_200902221346_0119   1   1237025200426   jason   NORMAL
```

```
> cd logs
../bin/slaves.sh cat $PWD/userlogs/*200902221346_0119*/stdout
```

```
192.168.1.119: cat: /home/jason/src/hadoop-0.19.0/logs/ ➥
userlogs/attempt_200902221346_0119_r_000000_0/stdout: ➥
No such file or directory ➥
192.168.1.119: cat: /home/jason/src/hadoop-0.19.0/logs/ ➥
userlogs/attempt_200902221346_0119_r_000002_0/stdout: ➥
```

```
No such file or directory
192.168.1.2: Listening for transport dt_socket at address: 43004
192.168.1.2: Listening for transport dt_socket at address: 35403
```

Note The port numbers change for each task, and there may be inconsistency in the ports shown in the screenshots and those mentioned in the text.

Rerunning a Failed Task

The `IsolationRunner` provides a way of rerunning a task out of a failed job. Normally, the framework immediately removes all local task specific data when a task finishes.

Configuring the Job or Cluster to Save the Task Local Working Directory

Two configuration keys provide some control over the ability to rerun a task. The value of the configuration key `keep.task.files.pattern` is a Java regular expression, which is matched against task files. Any task that matches this pattern will not have its task files removed. The effective code used for this is as follows:

```
alwaysKeepTaskFiles =
  Pattern.matches(conf.getKeepTaskFilesPattern(), task.getTaskID().toString()
```

To save the results of all map tasks, a pattern `*_m.*` would work. To match all reduces tasks, `*_r_*`.

The other option is to set the value of the configuration key `keep.failed.tasks.files` to `true`. Any task that fails will not be subject to cleanup.

Caution Nothing will reclaim this space, which may be quite large. Do this with care and clean up afterwards.

Determining the Location of the Task Local Working Directory

The root directory set for the local working areas is stored in the configuration under the key `mapred.local.dir`, and may be a comma-separated list of directories. Running the `find` command on this set of directories looking for files named `job.xml` will present a set of candidate tasks to be run via the `IsolationRunner`.

```
> find /tmp/hadoop-0.19.0-jason/mapred/local/ -wholename '*attempt*/job.xml' -print
```

```
/tmp/hadoop-0.19.0-jason/mapred/local/taskTracker/jobcache/ ➥
job_200902221346_0119/attempt_200902221346_0119_m_000000_0/job.xml
```

Running a Job with a Keep Pattern and Debugging via the IsolationRunner

Again, this example uses our old friend the `PiEstimator` job. It demonstrates how to run it so that the map task local file space is left in intact. Then you find the `job.xml` files that can be run via the `IsolationRunner` and run one of them in a way that will enable the use of the debugger.

Here's how to put it all together for the `IsolationRunner`:

```
> bin/hadoop jar hadoop-0.19.0-examples.jar pi ➥
-D keep.task.files.pattern=".*_m.*" 2 1000
```

```
...cd
Estimated value of PI is 3.102
```

```
> find /tmp/hadoop-0.19.0-jason/mapred/local/ -wholename '*attempt*/job.xml' -print
```

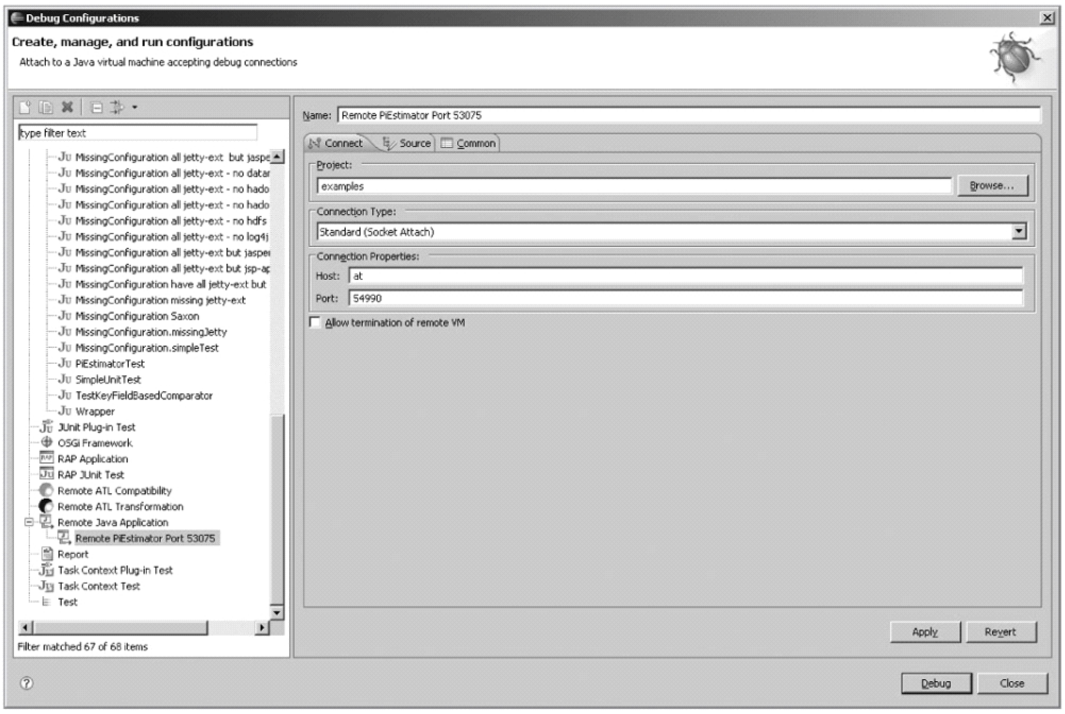
```
/tmp/hadoop-0.19.0-jason/mapred/local/taskTracker/jobcache/ ➥
job_200902221346_0120/attempt_200902221346_0120_m_000002_0/job.xml
/tmp/hadoop-0.19.0-jason/mapred/local/taskTracker/jobcache/ ➥
job_200902221346_0120/attempt_200902221346_0120_m_000003_0/job.xml
> cd /tmp/hadoop-0.19.0-jason/mapred/local/taskTracker/jobcache/ ➥
job_200902221346_0120/attempt_200902221346_0120_m_000003_0/
```

```
> HADOOP_OPTS=-agentlib:jdwp=transport=dt_socket,server=y,address=0.0.0.0 ➥
```

```
~/src/hadoop-0.19.0/bin/hadoop org.apache.hadoop.mapred.IsolationRunner ../job.xml
```

```
Listening for transport dt_socket at address: 54990
```

Once the child JVM is configured and waiting, Eclipse must be configured to connect to it. Figure 7-9 shows the Eclipse Debug Configuration window for setting up a remote debugging connection. The Host field must be filled in with the host on which the JVM is running, and the Port field filled in with the value from the `Listening for transport dt_socket at address:` *XXXXX* output line from the JVM. Figure 7-10 shows Eclipse connected to the running JVM of the `PiExample` test case.



Debug Configurations

Create, manage, and run configurations

Attach to a Java virtual machine accepting debug connections

type filter text

- MissingConfiguration all jetty-ext but jaspe
- MissingConfiguration all jetty-ext - no datar
- MissingConfiguration all jetty-ext - no hado
- MissingConfiguration all jetty-ext - no hado
- MissingConfiguration all jetty-ext - no hdfs
- MissingConfiguration all jetty-ext - no log4j
- MissingConfiguration all jetty-ext but jasper
- MissingConfiguration all jetty-ext but jsp-ap
- MissingConfiguration have all jetty-ext but
- MissingConfiguration missing jetty-ext
- MissingConfiguration Saxon
- MissingConfiguration.missingJetty
- MissingConfiguration.simpleTest
- PiEstimatorTest
- SimpleUnitTest
- TestKeyFieldBasedComparator
- Wrapper
- JUnit Plug-in Test
- OSGi Framework
- RAP Application
- RAP JUnit Test
- Remote ATL Compatibility
- Remote ATL Transformation
- Remote Java Application
  - Remote PiEstimator Port 53075
- Report
- Task Context Plug-in Test
- Task Context Test
- Test

Filter matched 67 of 68 items

Name: Remote PiEstimator Port 53075

Connect | Source | Common

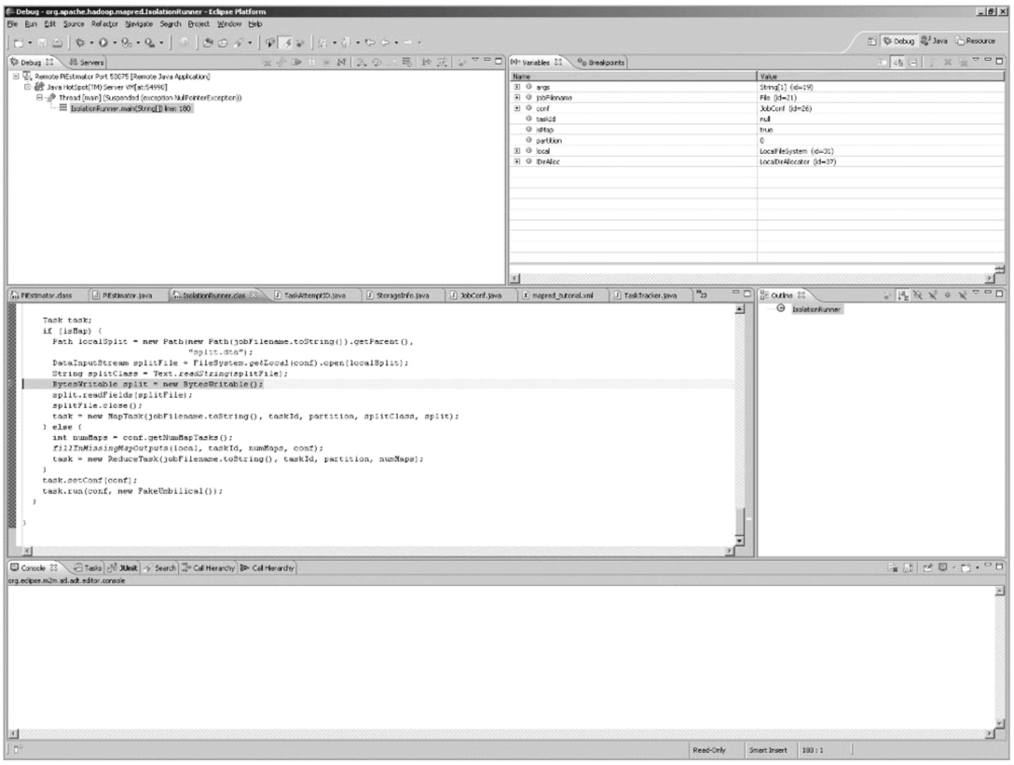
Project: examples

Connection Type: Standard (Socket Attach)

Connection Properties:
Host: at
Port: 54990

☐ Allow termination of remote VM

Figure 7-9: Eclipse setup to connect to port 54990 on the specified host



Debug - org.apache.hadoop.mapred.IsolationRunner - Eclipse Platform

```
Remote PIEstimator Port 53075 [Remote Java Application]
  Java HotSpot(TM) Server VM[at:54990]
    Thread [main] (Suspended (exception NullPointerException))
      IsolationRunner.main(String[]) line: 180
```

Name	Value
args	String[1] (id=19)
jobFilename	File (id=21)
conf	JobConf (id=26)
taskId	null
isMap	true
partition	0
local	LocalFileSystem (id=31)
lDirAlloc	LocalDirAllocator (id=37)

```
    Task task;
    if (isMap) {
      Path localSplit = new Path(new Path(jobFilename.toString()).getParent(),
                                 "split.dta");
      DataInputStream splitFile = FileSystem.getLocal(conf).open(localSplit);
      String splitClass = Text.readString(splitFile);
      BytesWritable split = new BytesWritable();
      split.readFields(splitFile);
      splitFile.close();
      task = new MapTask(jobFilename.toString(), taskId, partition, splitClass, split);
    } else {
      int numMaps = conf.getNumMapTasks();
      fillInMissingMapOutputs(local, taskId, numMaps, conf);
      task = new ReduceTask(jobFilename.toString(), taskId, partition, numMaps);
    }
    task.setConf(conf);
    task.run(conf, new FakeUmbilical());
  }

}
```

Figure 7-10: Eclipse connected to the IsolationRunner

Now you may set your breakpoints, debug, and explore.

Summary

It is very important to know that your code is correct. This chapter has provided you with the techniques needed to build unit tests for your MapReduce jobs. MapReduce applications can also be test-driven.

This chapter also demonstrated three ways to run MapReduce applications under the Eclipse debugger. This will enable you to understand problems that occur only at scale on your large clusters, as well as explore how things work directly on your local development machine.