HowToDevelopUnitTests

h1. How to develop Hadoop Tests This page contains Hadoop testing and test development guidelines. h2. How Hadoop Unit Tests Work Hadoop Unit tests are all designed to work on a local machine, rather than a full-scale Hadoop cluster. The ongoing for for that is in Apache Bigtop. The unit tests work by creating a miniDFS, MiniYARN and MiniMR clusters -as appropriate. These all run the code of the specific services. h3. MiniDFSCluster {{org.apache.hadoop.hdfs.MiniDFSCluster}} Emulates an HDFS cluster with the given number of (emulated) datanodes. After creating one via its builder API; you can build up the HDFS URI {{"hdfs://localhost:" + miniDFSCluster.getNameNodePort()}}. This can be used as the base URI for filesystem operations. {code:language=java} File baseDir = new File("./target/hdfs/" + testName).getAbsoluteFile(); FileUtil.fullyDelete(baseDir); conf.set(MiniDFSCluster.HDFS_MINIDFS_BASEDIR, baseDir.getAbsolutePath()); MiniDFSCluster.Builder builder = new MiniDFSCluster.Builder(conf); MiniDFSCluster hdfsCluster = builder.build(); String hdfsURI = "hdfs://localhost:"+ hdfsCluster.getNameNodePort() + "/"; {code} h3. MiniYARNCluster {{org.apache.hadoop.yarn.server.MiniYARNCluster}} Starts the YARN Services in the JVM, with the given number of simulated Node Managers. You can then submit work to the [ResourceManager]. The actual AMs (and any containers they themselves execute code in) are actually executed in separate processes -as on a real YARN cluster. The key difference is that the classpath of the test JVM is passed down to the spawned processes (how? Which environment variable?) so that they pick up the same version of the Hadoop JARs. {code:language=java} YarnConfiguration clusterConf = new YarnConfiguration(); conf.setInt (YarnConfiguration.RM_SCHEDULER_MINIMUM_ALLOCATION_MB, 64); conf.setClass(YarnConfiguration.RM_SCHEDULER, FifoScheduler.class, ResourceScheduler.class); miniCluster = new MiniYARNCluster(name, noOfNodeManagers, numLocalDirs, numLogDirs); miniCluster.init(conf); miniCluster.start(); //once the cluster is created, you can get its configuration //with the binding details to the cluster added from the minicluster YarnConfiguration appConf = new YarnConfiguration(miniCluster.getConfig()); {code} The results of a test run end up saved into the filesystem, where then can be retrieved by hand. {noformat} cat target/TestKillAM/TestKillAM-logDir-nm-0_0/application_1378993847080_0001/container_1378993847080_0001_01_000001/out.txt {noformat} # The output is not merged into the JUnit results (if anyone can fix this, code would be welcome) # The output is formatted by whatever logging tools and configuration the AM and its containers use -such as the specific version of {{Apache Log4J}} and {{log4j.properties}} are on the classpath. # The name of the base directory and logdir is determined by the name given to the test cluster -unique cluster names per test classes are invaluable. # The more node managers you create, the more log directories you will have to look into. A single NM is easier to work with. # the application- and container- directory names vary every run. # You can {{tail -f}} the {{out.txt}} and {{err.txt}} files while the tests are running. # {{jps -v}} will list the running applications; {{kill}} can then be used to kill the processes, and so test the application's resilience to failures. \\ It's a bit inelegant to work with, but functional. The ability list and terminate the processes makes writing failure simulation tests possible -which is important as production applications need to be designed to handle failures of child containers. h3. MiniMRYarnCluster {{org.apache.hadoop.mapreduce.v2.MiniMRYarnCluster}} This adds an MR History Server to the {{MiniYarnCluster}}, and extends the cluster configuration to refer to it. MR applications can then easily talk to the RM to submit jobs, with the history being preserved. h3. Using the Mini clusters in tests The clusters take time to set up and tear down, so should only be created once per test class, in a {{@BeforeClass}}-tagged static class method. in an {{@AfterClass}} they should be stopped. {{MiniDFSCluster.shutdown()}} and via the {{stop()}} method in the YARN clusters. h2. Writing JUnit Tests h3. Cheat sheet of tests development for JUnit v4 Hadoop has been using JUnit4 for a while now, however it seems that many new tests are still being developed for JUnit v3. It is partially JUnit's fault because for the false sense of backward compatibility all v3 {{junit.framework}} classes are packaged along with v4 classes and it all is called {{junit-4.10.jar}}. This is necessary to permit mixing of the old and new tests, and to allow the new v4 tests to run under the existing JUnit test runners in IDEs and build tools. Here's the short list of traps one need to be aware and not to develop yet another JUnit v3 test case * YES, new unit tests HAVE to be developed for JUnit v4. No patches which add v3 test case classes will be approved. * DO NOT use {{junit.framework}} imports * DO use only {{org.junit}} imports * DO NOT {{extends [TestCase]}} (now, you can create your own test class structures if needed!) * DO use {{@Test}} annotation to highlight what methods represent your test cases * DO NOT put a JUnit 3.x JAR on your classpath. \\ Other Hadoop Test case requirements * DO begin all your test classes with the word {{Test}}. This is used to select test cases to be executed. * DO NOT give non Test classes classnames starting with the word {{Test}}. This confuses the test runner. * DO give test classes methods meaningful names, ones that help people to diagnose problems from remote test runs * DO NOT assume that Test methods run in any specific order (Java 7 does re-order the methods), or even that more than one test method is run. * DO log information that is useful to diagnose why tests failed * But DO NOT generate megabytes of log data at INFO or above, as that can overload test runners. Print the low-level log data at debug level. * DO NOT swallow or wrap exceptions, throw them from your test methods. * AVOID looking for hard coded error strings in your test, instead have the production classes export their strings as constants, which your test methods can then reference directly. * DO NOT assume the port numbers that Hadoop services will come up on; ask the mini clusters for the values. * DO NOT assume the external internet is reachable. * DO NOT assume that DNS works, and especially not rDNS. * DO NOT assume that there is only one NIC. * DO NOT assume that hostnames clock or timezone are correct \\ You can make tests that work with external networks (the 'Blobstore' tests) do this -but they are designed to be skipped if the login credentials are missing. If you do anything similar, the tests must be optional Assumptions you can make * Whoever is running the test can understand Java stack traces and network errors. * There is a network card, the machine has a name and Java can determine that hostname. * \\ \\ h2. Assertions Because your test asserts your will be using need to be statically imported either one by one, i.e. {code:language=java} import static org.junit.Assert.assertTrue; {code} or all of them at once {code:language=java} import static org.junit.Assert.*; {code} It is also possible to cheat and extend the Assert class itself {code:language=java} import org.junit.Assert; public class TestSomething extends Assert { } {code} The final tactic is half-way between JUnit 3.x and the JUnit 4 styles; the Hadoop team is yet to come down against it, though they reserve the right. h3. Effective Assertions # Use the JUnit assertions, not the Java {{assert}} statement. # In equality tests, place the expected value first # Give assertions meaningful error messages. \\ h4. Bad {code:language=java} /** a test */ @Test public void testBuildVersion() { Namenode nn = getNameNode(); assertNotNull(nn); NamespaceInfo info = nn.versionRequest() ; assertEquals(info.getBuildVersion(),"32"); } {code} This test doesn't include any details as to why a test fails, so if you do a test run you find out the name and the line of a test and are left looking that up in the source to work out what went wrong. Some explanations help. The {{assertEquals()}} test will have some meaningful message, but because the variable comes before the constant, the text will be wrong. h3. Good {code:language=java} /** * Test that the build version is OK */ @Test public void testBuildVersion() { Namenode nn = getNameNode(); assertNotNull("No namenode", nn); NamespaceInfo info = nn.versionRequest() ; assertEquals("Build version wrong", "32", info.getBuildVersion()); } {code} When any of the equals assertions fail, the error text includes the text inserted in the assertion, and the expected and equals values. You don't need to explicitly include them. For all assertions, providing hints as to what is wrong is good. h2. Logging All Hadoop test cases run on a classpath which contains commons-logging; use the logging APIs just as you would in Hadoop's own codebase {code:language=java} import org.apache.commons.logging.Log; import org.apache.commons.logging.LogFactory; import org.junit.Test; public class TestSomething { private static final Log LOG = LogFactory.getLog(TestSomething.class); } {code} Don't go overboard in logging at info level, as it can be buffered in the test runners (especially the XML one) and lead to out of memory problems. Log the details at debug level which can then be turned on for specific tests causing problems. h2. Exception Handling in Tests Test methods can declare that they throw {{Throwable}}. There is no need to catch exceptions and wrap them in JUnit {{RuntimeException}} instances. h3. Bad {code:language=java} @Test public void testCode() { try { doSomethingThatFails(); catch(IOException ioe) { fail("something went wrong"); } } {code} h3. good {code:language=java} @Test public void testCode() throws Throwable { doSomethingThatFails(); } {code} This leaves less code around (lower maintenance costs), and ensures that any failure gets reported with a full stack trace. h2. Let Unit Tests be "Unit" tests Avoid starting servers (including Jetty and Mini\{DFS|MR\}Clusters) in *unit* tests, as they take tens of seconds to start for each test (HDFS and MapReduce tests already take many hours mostly due to these servers starts). Use them only in cross component *functional* or *integration* (system) tests (cf. [HADOOP-6399|https://issues.apache.org/jira/browse/HADOOP-6399]). Try to use one of the lighter weight [test doubles|http://www.martinfowler.com/bliki/TestDouble.html] for collaborating components for the component under test. Hadoop has adopted the [Mockito|http://mockito.googlecode.com/svn/tags/latest/javadoc/org/mockito/Mockito.html] library for easy mock and stub creation. h2. Skipping tests with Assume Some tests only work on specific platforms, or if a specific configuration property is set. Hadoop's older JUnit3 tests avoided these tests running by having {{{if(!condition) \{ return;\} }}} statements -which skipped the tests but hid the fact that they weren't running. Similarly, some tests were not given classnames beginning with {{Test}} so they wouldn't even be run. This wouldn't generate false positives, but it would hide the fact that the tests weren't being run. The JUnit4 approach is to to use the {{Assume}} class {code:language=java} Assume.assumeTrue("No login provided", null != conf.get("test.login.password")); {code} This raises an exception that is picked up by the test runner and converted into a _Skipped_ exception: the fact that the test was skipped is now explicitly displayed in the test results. Explicitly skipping tests this way makes it clearer in the code what is going on, and people running the tests can now see what tests are not being run. h2. References * [Quick tutorial|http://code.google.com/p/t2framework/wiki/JUnitQuickTutorial] on the JUnit website.