The Hadoop Ecosystem Builder HOWTO

03 July 2014

This document is a quick start guide for build a Lambda Architecture ecosystem package set using The Hadoop Ecosystem Builder - Buildoop tool.

1. Preface

This document was written to help bring the system integrators a guide for build a full package set using the building system Buildoop from scratch. The aim of this document is make a YUM repository ready for a full Lambda Architecture deployment, and a guide for deploy this system manually using the YUM tool.

The automatic deployment is out of scope of this document. For this purpose the “The Hadoop Deploy System - Deploop - HOWTO” will soon be available.

1.1. Copyright

1.2. Disclaimer

Use the information in this document at your own risk. I disavow any potential liability for the contents of this document. Use of the concepts, examples, and/or other content of this document is entirely at your own risk.

All copyrights are owned by their owners, unless specifically noted otherwise. Use of a term in this document should not be regarded as affecting the validity of any trademark or service mark.

Naming of particular products or brands should not be seen as endorsements.

You are strongly recommended to take a backup of your system before major installation and backups at regular intervals.

2. Introduction

2.1. What is the Hadoop Ecosystem Builder Project?

The Hadoop Ecosystem Builder Project (Buildoop) is an effort to create a tool for system integrators .

2.2. What is wrong with Apache BigTop?

The Linux development community utilizes two important (some out argue most important) testing techniques in its normal operations: Design and Code Inspections. The intent of LTP is to support this by giving developers an ever growing set of tools to help identify any operational problems in their code that may be missed by human review. One of the toughest categories of problems to catch with inspection is that of interaction of features. With a continuously improving set of tests and tools, developers can get an indication of whether their changes may have broken some other functionality.

There is no such thing as a perfect test base. It is only useful it if keeps up with new and changing functionality, and if it actually gets used.

2.3. Development in progress

Not at this time. We are more interested in functional, regression, and stress testing the Linux kernel. Benchmarking may be workable to compare the performance among kernel versions.

3. Structure

The basic building block of the test project is atest case that consists of a single action and a verification that the action worked. The result of the test case is usually restricted to PASS/FAIL.

Atest program is a runnable program that contains one or more test cases. Test programs often understand command line options which alter their behavior. The options could determine the amount of memory tested, the location of temporary files, the type of network packet used, or any other useful parameter.

Test tags are used to pair a unique identifier with a test program and a set of command line options. Test tags are the basis for test suites.

4. Writing Tests

Writing a test case is a lot easier than most people think. Any code that you write to examine how a part of the kernel works can be adapted into a test case. All that is needed is a way to report the result of the action to the rest of the world. There are several ways of doing this, some more involved than others.

4.1. Exit Style Tests

Probably the simplest way of reporting the results of a test case is the exit status of your program. If your test program encounters unexpected or incorrect results, exit the test program with a non-zero exit status, i.e.exit(1). Conversely, if your program completes as expected, return a zero exit status, i.e.exit(0). Any test driver should be able to handle this type of error reporting. If a test program has multiple test cases you won't know which test case failed, but you will know the program that failed.

4.2. Formatted Output Tests

The next easiest way of reporting the results is to write the results of each test case to standard output. This allows for the testing results to be more understandable to both the tester and the analysis tools. When the results are written in a standard way, tools can be used to analyze the results.

5. Testing Tools

The Linux Test Project has not yet decided on a "final" test harness. We have provided a simple solution withltp-pan to make due until a complete solution has been found/created that compliments the Linux kernel development process. Several people have said we should use such and such a test harness. Until we find we need a large complex test harness, we will apply the KISS concept.

5.1. Ltp-pan

ltp-pan is a simple test driver with the ability to keep track of orphaned processes and capture test output. It works by reading a list of test tags and command lines and runs them. By default ltp-pan will select a command randomly from the list of test tags, wait for it to finish. Through command line options you can run through the entire list sequentially, run n tests, keep n test running at all times, and buffer test output. Ltp-pan can be nested to create very complex test environments.

Ltp-pan uses an*active file*, also called a*zoo file* to keep track of which tests are currently running. This file holds the pid, tag, and a portion of the command line. When you start ltp-pan it becomes a test tag in itself, thus it requires a name for itself. Ltp-pan updates the active file to show which test tags are currently running. When a test tag exits, ltp-pan will overwrite the first character with a '#'. The active file can be shared between multiple instances of ltp-pan so you know which tests were running when the system crashes by looking at one file.

A*ltp-pan file* contains a list of test tags for ltp-pan to run. The format of a ltp-pan file is as follows:

```
testtag testprogram -o one -p two other command line options
# This is a comment. It is a good idea to describe the test
# tags in your ltp-pan file. Tests programs can have different
# behaviors depending on the command line options so it is
# helpful to describe what each test tag is meant to verify or # provoke.
# Some more test cases
mm01 mmap001 -m 10000
# 40 Mb mmap() test.
# Creates a 10000 page mmap, touches all of the map, sync's
```

```
# it, and munmap()s it.
mm03 mmap001 -i 0 -I 1 -m 100
# repetitive mmapping test.
# Creates a one page map repetitively for one minute.
dup02 dup02
# Negative test for dup(2) with bad fd
kill09 kill09
# Basic test for kill(2)
fs-suite01 ltp-pan -e -a fs-suite01.zoo -n fs-suite01 -f runtest/fs
# run the entire set of file system tests
```

The test tags are simple identifiers, no spaces are allowed. The test of the line is the program to run, which is done using execvp(3). Lines starting with '#' are comments and ignored by ltp-pan. It is a good practice to include descriptions with your test tags so you can have a reminder what a certain obscure test tag tries to do.

5.1.1. Examples

The most basic way to run ltp-pan is by passing the test program and parameters on the command line. This will run the single program once and wrap the output.

```
$ ltp-pan -a ltp.zoo -n tutor sleep 4
<<<test_start>>>
tag=cmdln stime=971450564
cmdline="sleep 4"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
<<<execution_status>>>
duration=103341903 termination_type=exited termination_id=0 corefile=no cutime=0 cstime=0
<<<test_end>>>
$ cat ltp.zoo
#9357,tutor,pan/ltp-pan -a ltp.zoo -n tutor sleep 4
#9358,cmdln,sleep 4
$
```

5.1.1.1. How it works

This example shows the two parameters that are always required by ltp-pan, the active file and a test tag for ltp-pan. The“sleep 4” on the end of the command line is a test program and parameters that ltp-pan should run. This test is given the tag“cmdln.” Ltp-pan will run one test randomly, which ends up being cmdln since it is the only test that we told ltp-pan about.

In the active file,ltp.zoo, ltp-pan writes the pid, test tag, and part of the command line for the currently running tests. The command lines are truncated so each line will fit on an 80 column display. When a test tag finishes, ltp-pan will place a '#' at the beginning of the line to mark it as available. Here you can see that cmdln and tutor, the name we gave ltp-pan, ran to completion. If the computer hangs, you can read this file to see which test programs were running.

We have run one test once. Let's do something a little more exciting. Let's run one test several times, at the same time.

```
$ ltp-pan -a ltp.zoo -n tutor -x 3 -s 3 -O /tmp sleep 1
<<<test_start>>>
tag=cmdln stime=971465653
cmdline="sleep 1"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>

<<<execution_status>>>
duration=103326814 termination_type=exited termination_id=0 corefile=no
cutime=1 cstime=0
<<<test_end>>>
<<<test_start>>>
tag=cmdln stime=971465653
cmdline="sleep 1"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>

<<<execution_status>>>
duration=103326814 termination_type=exited termination_id=0 corefile=no
cutime=0 cstime=1
<<<test_end>>>
<<<test_start>>>
tag=cmdln stime=971465653
cmdline="sleep 1"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>

<<<execution_status>>>
duration=103326814 termination_type=exited termination_id=0 corefile=no
cutime=0 cstime=0
<<<test_end>>>
```

5.1.1.2. How it works

In this example we run another fake test from the command line, but we run it three times (-s 3) and keep three test tags active at the same time (-x 3). The -O parameter is a directory where temporary files can be created to buffer the output of each test tag. You can see in the output that cmdln ran three times. If the -O option were omitted, your test output would be mixed, making it almost worthless.

- Using a ltp-pan file to run multiple tests
- Nesting ltp-pan

For more information on ltp-pan see the man pagedoc/man1/ltp-pan.1.

5.2. Scanner

Ltp-scanner is a results analysis tool that understands the*rts* style output whichltp-pan generates by default. It will produce a table summarizing which tests passed and which failed.

5.3. The Quick-hitter Package

Many of the tests released use the Quick-hitter test package to perform tasks like create and move to a temporary directory, handle some common command line parameters, loop, run in parallel, handle signals, and clean up.

There is an example test case,doc/examples/quickhit.c, which shows how the quick-hitter package can be used. The file is meant to be a supplement to the documentation, not a working test case. Use any of the tests intests/ as a template.

6. To Do

There are a lot of things that still need to be done to make this a complete kernel testing system. The following sections will discuss some of the to do items in detail.

6.1. Configuration Analysis

While the number of configuration options for the Linux kernel is seen as a strength to developers and users alike, it is a curse to testers. To create a powerful automated testing system, we need to be able to determine what the configuration on the booted box is and then determine which tests should be run on that box.

The Linux kernel has hundreds of configuration options that can be set to compile the kernel. There are more options that can be set when you boot the kernel and while it is running. There are also many patches that can be applied to the kernel to add functionality or change behavior.

6.2. Result Comparison

A lot of testing will be done in the life of the Linux Test Project. Keeping track of the results from all the testing will require some infrastructure. It would be nice to take that output from a test machine, feed it to a program and receive a list of items that broke since the last run on that machine, or were fixed, or work on another test machine but not on this one.

7. Contact information and updates

URL: http://ltp.sourceforge.net/mailing list: ltp-list@lists.sourceforge.netlist archive: https://sourceforge.net/mailarchive/foru
list

Questions and comments should be sent to the LTP mailing list at ltp-list@lists.sourceforge.net. To subscribe, please go to https://lists.sourceforge.net/lists/listinfo/ltp-list.

The source is also available via CVS. See the web site for a web interface and check out instructions.

8. Glossary

Test

> IEEE/ANSI[1]: (i) An activity in which a system or component is executed under specified conditions, the results are observed or record, and an evaluation is made of some aspect of the system or component. (ii) A set of one or more test cases.

Test Case

> A test assertion with a single result that is being verified. This allows designations such as PASS or FAIL to be applied to a single bit of functionality. A single test case may be one of many test cases for testing the complete functionality of a system. IEEE/ANSI: (i)A set of test inputs, execution conditions, and expected results developed for a particular objective. (ii) The smallest entity that is always executed as a unit, from beginning to end.

Test Driver

> A program that handles the execution of test programs. It is responsible for starting the test programs, capturing their output, and recording their results. Ltp-pan is an example of a test driver.

Test Framework

> A mechanism for organizing a group of tests. Frameworks may have complex or very simple API's, drivers and result logging mechanisms. Examples of frameworks are TETware and DejaGnu.

Test Harness

> A Test harness is the mechanism that connects a test program to a test framework. It may be a specification of exit codes, or a set of libraries for formatting messages and determining exit codes. In TETware, the tet_result() API is the test harness.

Test Program

> A single invokable program. A test program can contain one or more test cases. The test harness's API allows for reporting/analysis of the individual test cases.

Test Suite

> A collection of tests programs, assertions, cases grouped together under a framework.

Test Tag

> An identifier that corresponds to a command line which runs a test. The tag is a single word that matches a test program with a set of command line arguments.

9. GNU Free Documentation License

GNU Free Documentation License Version 1.3, 3 November 2008 in this link:

http://www.gnu.org/licenses/fdl.html

Notes

Kit, Edward, Software Testing in the Real World: Improving the Process. P. 82. ACM Press, 1995.