packaging tests

This project contains tests that verify the distributions we build work correctly on the operating systems we support. They're intended to cover the steps a user would take when installing and configuring an Elasticsearch distribution. They're not intended to have significant coverage of the behavior of Elasticsearch's features.

There are two types of tests in this project. The old tests live in `src/test/` and are written in Bats, which is a flavor of bash scripts that run as unit tests. These tests are deprecated because Bats is unmaintained and cannot run on Windows.

The new tests live in `src/main/` and are written in Java. Like the old tests, this project's tests are run inside the VM, not on your host. All new packaging tests should be added to this set of tests if possible.

Running these tests

See the section in TESTING.asciidoc

Adding a new test class

When gradle runs the packaging tests on a VM, it runs the full suite by default. To add a test class to the suite, add its `class` to the `@SuiteClasses` annotation in PackagingTestCase.java. If a test class is added to the project but not to this annotation, it will not run in CI jobs. The test classes are run in the order they are listed in the annotation.

Choosing which distributions to test

Distributions are represented by enum values which know if they are compatible with the platform the tests are currently running on. To skip a test if the distribution it's using isn't compatible with the current platform, put this assumption in your test method or in a `@Before` method

```
assumeTrue(distribution.packaging.compatible);
```

Similarly if you write a test that is intended only for particular platforms, you can make an assumption using the constants and methods in Platforms.java

```
assumeTrue("only run on windows", Platforms.WINDOWS);

assumeTrue("only run if using systemd", Platforms.isSystemd());
```

Writing a test that covers multiple distributions

It seems like the way to do this that makes it the most straightforward to run and reproduce specific test cases is to create a test case class with an abstract method that provides the distribution

```
public abstract class MyTestCase {
  @Test
  public void myTest() { /* do something with the value of #distribution() */ }
  abstract Distribution distribution();
}
```

and then for each distribution you want to test, create a subclass

```
public class MyTestDefaultTar extends MyTestCase {
  @Override
  Distribution distribution() { return Distribution.DEFAULT_TAR; }
}
```

That way when a test fails the user gets told explicitly that `MyTestDefaultTar` failed, and to reproduce it they should run that class. See ArchiveTestCase and its children for an example of this.

Running external commands

In general it's probably best to avoid running external commands when a good Java alternative exists. For example most filesystem operations can be done with the java.nio.file APIs. For those that aren't, use an instance of Shell

This class runs scripts in either bash with the `bash -c <script>` syntax, or in powershell with the `powershell.exe -Command <script>` syntax.

```
Shell sh = new Shell();

// equivalent to `bash -c 'echo $foo; echo $bar'`
sh.bash("echo $foo; echo $bar");

// equivalent to `powershell.exe -Command 'Write-Host $foo; Write-Host $bar'`
sh.powershell("Write-Host $foo; Write-Host $bar");
```

Notes about powershell

Powershell scripts for the most part have backwards compatibility with legacy cmd.exe commands and their syntax. Most of the commands you'll want to use in powershell are Cmdlets which generally don't have a one-to-one mapping with an executable file.

When writing powershell commands in this project it's worth testing them by hand, as sometimes when a script can't be interpreted correctly it will fail silently.