Better testing in HotSpot

Leo Korinth

Testing behaviour defined by VM flags

When testing certain HotSpot features, VM flags must be used to enable certain behaviour. Many features, especially in the GC, compiler and runtime areas can not be enabled programmatically using function calls. Instead of enabling a behaviour to an existing VM programmatically, a new VM must be forked, and the feature must be enabled using command line flags. It would be extremely hard to change many things after startup, so many things must be set at VM startup.

When testing a specific feature in hotspot — GC for example — this is natural. We need to disable compressed class pointers if we want to test non-compressed class pointers when class pointers are compressed by default. How many such tests do we need to write to cover usage of non-compressed class pointers? There is also another dimension to this problem; certain flags *interact*. Are we to write tests for infinite combinations of flags?

Many bugs are found in test cases designed to test *unrelated* things. Would it not be great if we could reuse — almost — all test cases and run them with our VM flag combination of interest?

The rest of this post will discuss how to propagate VM options to a testcase, and by doing so enabling the reuse of the test for testing flag combinations.

Adding VM flags from a test suite

All test cases in OpenJDK written in Java are written in the test framework jtreg. Runtime flags can be given in the testcase itself, but can also be *added* to the testcase by the one running the testcase. Jtreg supports running a testcase with these *extra* runtime flags by giving the argument `-javaoption(s)` to jtreg or modifying the `make` command line with something like `JTREG='JAVA_OPTIONS=...'`

VM flags can also be modified using `-vmoption(s)` or `JTREG='VM_OPTIONS=...'`; *this is almost never what you want to do*. The reason is that these flags will not only be propagated to the test case but also to `javac` and other tools when they are spawned by jtreg.

How VM flags are propagated to a test case

Jtreg test cases are annotated with tags in the beginning of the test cases. These annotations are not *real* Java annotations, although they look very similar. They are Java comments that are parsed by jtreg.

When specifying the `@run` tag, options can be added describing how to run the testcase. Two main versions exists:

`/othervm` (spawn a new VM with each testcase)

`/agentvm` (reuse a pool of test VMs). This version has two sub modes that can be set with `defaultExecMode` in `TEST.ROOT` for the test suite, and/or can be set by a command line option to jtreg.

same VM use the same VM to compile and run tests
different VM use different VMs to compile and run tests (not to be confused with '/othervm')

What is important to understand is that *all* these methods will propagate your `JAVA_OPTIONS` to the VM your test case is run in, when spawning a new testcase. A new VM will be spawned if the command line options have changed when using `agentvm`.

However, this is the behaviour *only* when running the `main` action. If you instead use the `driver` action of the `@run` tag, *most* VM options set by `JAVA_OPTIONS` will be filtered out. I think this is not documented, but the relevant behaviour can be found in `filterJavaOpts` in the `DriverAction` of jtreg. Thus:

- Use `main` for your *normal* test cases that are spawned directly by jtreg.
- If your code is a *driver*, that is, glue code that is used to launch a new JVM, you should use the `driver` action to filter JAVA/JVM options to the glue *driver* process. You should then use `createTestJavaProcessBuilder` to fork your test, and by that propagate the flags to your *test* JVM.

There is a way to check for which flags was given to the VM being tested from within the testcase. In the test description a `@require` tag can be added. It will test for VM properties, such as active flags.

When VM flags are not propagated to a VM

Although jtreg itself propagates `JAVA_OPTIONS`, many testcases spawn new VMs from *within* a testcase. This is done for multiple reasons: one may be interested in inspecting the spawned VM or parse its output, or maybe a permutation of VM flags should be tested. There exists two main APIs for spawning new Java test processes (and some helper versions calling these as well):

- createLimitedTestJavaProcessBuilder (does not propagate VM flags):
  + *This is easy to use* (no conflicting `JAVA_OPTIONS`, no need to `@require` absence of conflicting flags)

  + stable test cases (no dynamic behaviour from clashes between `JAVA_OPTIONS` and the additional options added by the testcase)
  – no propagation of flags, *bad test coverage*
- createTestJavaProcessBuilder (does propagate VM flags, behaves as `@run`):
  + propagation of flags — thus much better test coverage
  – unfortunately, as `JAVA_FLAGS` are propagated, flag combinations not considered can cause a test crash or timeout
  – complicated `@require` flags can mitigate some problems, but adds complexity
  – complicated `@require` flags can cause the test to be `skipped` by mistake, an easy mistake that is hard to find.

From a *superficial* glance, `createLimitedTestJavaProcessBuilder` seems easier to use, and it is thus *understandable* that the limited API is used much more often. 699 occurrences versus 201 as can be seen below:

```
cd openjdk/test
git switch --detach 86623aa
grep -R createLimitedTestJavaProcessBuilder | wc -l
699
grep -R createTestJavaProcessBuilder | wc -l
201
```

The heavy usage of this limited API that does not propagate `JAVA_OPTIONS` compelled us to change the name and documentation to make it more obvious that one version is *limited* when it comes to test coverage. Although jtreg documentation asks us to respect command line compiler and VM options, it is easier to write tests correctly if you get help from naming. Especially if you draw your inspiration from already existing testcases that more often than not, use the "wrong" API.

Problems when not propagating VM flags

Although createLimitedTestJavaProcessBuilder is easier to use, it comes with problems.

Test coverage

If we spawn new processes without materializing `JAVA_OPTIONS`, those flags might never be tested in combinations with flags that we *do* test in the test case. At least at Oracle, we test each test suite with a range of flag combinations. This is of course expensive, both in equipment and in test time, but it is required to ensure quality. Each test case that does not propagate `JAVA_OPTIONS` will not benefit from this flag rotation.

Test performance

If we test a certain test case using a number of flag combinations without materializing `JAVA_OPTIONS`, not only do we miss the opportunity of testing these combinations, but we waste energy and money by running effectively the same test case several times.

If there is a hard need for not propagating `JAVA_OPTIONS`, this is uncommon, you can `@require` the property `vm.flagless`. With `vm.flagless` you will not get good coverage, but at least the test case will not be run on all flag combinations.

Writing good test cases

If we put a bit more work in the test cases, *and accept that we will have crashes with flags that are not tested in the pipeline*, we can make our test cases use the non-limited createTestJavaProcessBuilder. It is more work and certain flag combinations might not work, *but we will be able to test flags*.

The API `createLimitedTestJavaProcessBuilder` will test only *one of countless* flag combinations. Most testcases could probably benefit being tested with the interpreter, without compressed oops or maybe with another GC. Most bugs are found in testcases not created to test for the specific bug.

If we further mark certain test cases with `@keyword flag-sensitive` we might filter out test failures easily.

Problems and Solutions

Many of the test cases add their own flags. One can add `@require` lines so that the test case flags do not conflict with `JAVA_OPTIONS`. Unfortunately, adding too complicated `@require` lines might add to the risk of *never* running the test case by. This is problematic because it is easy to do this slightly wrong and extremely easy to not find that the tests are skipped. Although test cases do show as *skipped* if they are filtered out by `@require` lines, they look exactly as if they passed normally if they throw the SkippedException.

Certain test flags *must* be `@required` away. For example, you can not say that you want to use two kinds of GCs. So if you spawn a new process with a certain GC, you must ensure that another GC was not propagated through `JAVA_OPTIONS`.

There is also the case when a flag does not conflict with another flag because of flag parsing, but will modify its behaviour. This is a bit trickier; we will not get an as easy to diagnose bug. A test case might fail or time out as a result — sometimes intermittently.

Worth noting in this discussion is that `JAVA_OPTIONS` are *prepended* on the command line (from my understanding both from `@run` tags and from the function calls that spawn new VMs). This is interesting because — usually, but

not always — the latter arguments wins if multiple arguments of the same type are given. This behaviour makes it slightly easier to write test cases. It has its pros and cons. Most often the flag your test case uses will *override* a conflicting flag given in `JAVA_OPTIONS`. However, this also highly limits the way we are able to *effectively* propagating flags to test cases.

If a testcase is known to be fragile, it can be marked with `@keyword flag-sensitive` and then easily filtered out. GC tests need to know the size of heap and generations and will not work if these test preconditions change. One might mark these tests with `flag-sensitive` and if someone tests large test suites with a flag that modify these preconditions, that tester might filter out these fragile tests. It is not a perfect solution, *but it is no worse than not testing the flags to begin with.*

Conclusion

Although testcases are somewhat easier to write — and made stable — if we do not propagate `JAVA_OPTIONS`, *extra care should be taken* to actually propagate `JAVA_OPTIONS`. Our goal must be to test well, and for hotspot tests that means that many flags must be tested. Ignoring flags will not lead to good testing.

Many tests need no changes at all, and can move from the *limited* API to the one that propagates VM flags without further changes. When changes are needed, we can guard against certain flags using `@require` lines. Often that is not even needed as jtreg prepends `JAVA_OPTIONS` instead of appending them.

If a testcase is fragile when flags change, one may opt to mark it using `@keyword flag-sensitive`.