Classloading and Forking under Maven Surefire

This page discusses classloading and forking under Maven Surefire, especially as regards the differences between Maven Surefire 2.3 and Maven Surefire 2.4.

Executive Summary

useSystemClassLoader changed between Surefire 2.3 and Surefire 2.4. The default was useSystemClassLoader=false, but now the default is useSystemClassLoader=true. If you're having problems, try turning it back off to see if that helps.

What problem does Surefire solve?

How do we launch your tests in an isolated classpath, where that classpath may be quite large?

Initially, the problem seems simple enough. Just launch Java with a classpath, like this:

```
java -classpath foo.jar:bar.jar MyApp
```

But there's a problem here: on some operating systems (Windows), there's a limit on how long you can make your command line, and therefore a limit on how long you can make your classpath. The limit is different on different versions of Windows; in some versions only a few hundred characters are allowed, in others a few thousand, but the limit can be pretty severe in either case.

How do people solve this problem in general?

There are two ways to workaround this problem; both of them are tricky and can cause other problems in some cases.

1. **Isolated Classloader**: One workaround is to use an isolated classloader. Instead of launching MyApp directly, we can launch some other app (a "booter") with a much shorter classpath. We can then create a new java.lang.ClassLoader (usually a java.net.URLClassLoader) with your classpath configured. The booter can then load up MyApp from the classloader; when MyApp refers to other classes, they will be automatically loaded from our isolated classloader.

The problem with using an isolated classloader is that your classpath isn't *really* correct, and some apps can detect this and object. For example, the system property "java.class.path" won't include your jars; if your app notices this, it could cause a problem.

There's another similar problem with using an isolated classloader: any class may call the static method ClassLoader.getSystemClassLoader() and attempt to load classes out of that classloader, instead of using the default classloader. Classes often do this if they need to create classloaders of their own.... Unfortunately, Java-based web application servers like Jetty, Tomcat, BEA WebLogic and IBM WebSphere are very likely to try to escape the confines of an isolated classloader.

2. **Manifest-Only Jar**: Another workaround is to use a "manifest-only jar." In this case, you create a temporary jar that's almost completely empty, except for a META-INF/MANIFEST.MF file. Java manifests can contain attributes that the Java VM will honor as directives; for example, you can have a "Class-Path" attribute, which contains a list of other jars to add to the classpath. So then you can run your code like this:

```
java -classpath booter.jar MyApp
```

This is a bit more realistic, because in this case the system classloader, the thread context classloader and the default classloader are all the same; there's no possibility of "escaping" the classloader. But this is still a weird simulation of a "normal" classpath, and it's still possible for apps to notice this. Again, java.class.path may not be what you'd expect ("why does it contain only one jar?"). Additionally, it's possible to query the system classloader to get the list of jars back out of it; your app may be confused if it finds only our booter.jar there!

Advantages/Disadvantages of each solution

If your app tries to interrogate its own classloader for a list of jars, it may work better under an isolated classloader than it would with a manifest-only jar. However, if your app tries to escape its default classloader, it may not work under an isolated classloader at all.

One advantage of using an isolated classloader is that it's the only way to use an isolated classloader without forking a separate process, running

all of the tests in the same process as Maven itself. But that itself can be pretty risky, especially if Maven is running embedded in your IDE!

What does Surefire do?

Surefire provides a mechanism for using either strategy. The parameter that determines this is called "useSystemClassLoader". If useSystemClassLoader is true, then we use a manifest-only jar; otherwise, we use an isolated classloader.

The default value for useSystemClassLoader changed between Surefire 2.3 and Surefire 2.4, which was a pretty significant change. In Surefire 2.3, useSystemClassLoader was false by default, and we used an isolated classloader. In Surefire 2.4, useSystemClassLoader is true by default. No value works for everyone, but we think this default is an improvement; a bunch of hard-to-diagnose bugs get better when we useSystemClassLoader=true.

Unfortunately, if this value is set incorrectly for your app, you're going to have a problem on your hands that can be quite difficult to diagnose. You might even be forced to read a long wiki article like this one. 😃

If you're having problems when upgrading from an older version of Surefire to a newer version, try setting useSystemClassLoader=false to see if that helps. You can do that with the POM snippet below, or by setting "-Dsurefire.useSystemClassLoader=false".

```
<project>
  [...]
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <configuration>
          <useSystemClassLoader>false</useSystemClassLoader>
        </configuration>
      </plugin>
    </plugins>
  </build>
  [...]
</project>
```

Debugging Classpath Problems

If you've read this far, you're probably fully equipped to diagnose problems that may occur during classloading. Here's some general tips to try:

- Run mvn with --debug (aka -X) to get more detailed output
- Check your forkMode. If forkMode=never, it's impossible to use the system classloader; we have to use an isolated classloader.
- If useSystemClassLoader=true, look at the surefire booter jar. Open it up (it's just a zip) and read its manifest.
- Run mvn with -Dmaven.surefire.debug, and attach to the running process with a debugger.