October 2019

JAVA INSTRUMENTATION OPTIMIZATION TECHNIQUES

Ted Hulick

JAVA INSTRUMENTATION OPTIMIZATION TECHNIQUES

AUTHORS:
Ted Hulick

ABSTRACT

Techniques are described herein for reducing overhead, risk, and code maintenance for Java® instrumentation required to monitor customer-facing applications. These techniques avoid transforms and retransforms, thereby eliminating risk and overhead as there are no transformers registered with the Java Virtual Machine (JVM). This may use the same call mechanism as a typical transform, and therefore requires no code changes in instrumentation. The "dummy" transform may always obtain a real class (not a null) that has not been initialized. Thus, a user may add a static block or other one-time-only objects. These techniques are also portable, because the only hook is in the core JVM classloader, and compact, because very little code is required for the agent itself.

DETAILED DESCRIPTION

The Java agent instrumentation package has existed for years. There are over approximately fifty different Java agents in the market today, most notably in the Application Performance Management (APM) space. Java agents vary greatly in how they instrument and how often they instrument. In the APM market, the bar is high in terms of resource consumption since the agents are meant to monitor resource use rather than to create additional resource use.

There are several common Java agent functions that may impact resource usage, such as the time required to initialize the agent before the application starts up, the number of classes that are "retransformed" so they can be reviewed with a class structure, the time required to review a class to determine if it should be instrumented, and the amount of time for which instrumented methods in a class "block" in the application. There are also other issues such as methods that are called only once immediately after class load and that need to be instrumented to observe the call.

Most Java instrumentation uses transforms and retransforms to instrument classes which can cause performance issues and in some versions of Java can lead to a complete Java Virtual Machine (JVM) crash. For example, there have been in excess of fifty Java 8 crashes related to class retransformation, creating a need to reduce the transformations required to achieve the necessary level of instrumentation. In addition, the initial transform is called with no class reference (i.e., it does not exist). Oftentimes other class references do not exist either, and frequently a method is called once immediately after the class is loaded and a retransform cannot be issued in time to instrument the class before the method is called. One example is a Runnable (run()) to launch a thread that occurs immediately after class load.

For a typical Java agent, the Java agent entry is put into a special jar file which specifies a "premain class" in the manifest contained in the jar file. The JVM boots up and then executes the "premain method in the specified premain class," which temporarily blocks the application from loading. The premain class generally initializes the agent by loading agent jars into a newly created and isolated class loader that does not interfere with the application. In addition, the premain usually adds a "transformer," which is a class that contains the transform method and is called after premain when new classes are loaded.

The premain may evaluate any classes already loaded using "getAllClassesLoaded" from an instrumentation handle. Selected classes are retransformed and run through the transform method. After returning from the premain, the application may continue loading and the application main class is called. As new classes load in the application, they are also run through the transform method and are evaluated, and can be instrumented as well. The transform returns null (e.g., use original) or new bytes (e.g., instrumented) back to the JVM. There may be several reasons why a class is not instrumented immediately and might need to be "retransformed." For example, a rule may be added after the initial startup, the class superclasses/interfaces may not have been loaded yet, or only bytes but not the class itself may yet exist.

A "retransform" cycle may "retransform" queued classes to be run back through the transform for evaluation and instrumentation. Generally, instrumented methods call to the instrumentation core in the entry of the method body, and call before exit from the method body. Optionally, the method may be "wrapped" to catch exceptions that occur. The calls are usually static and eventually transfer to "handlers" or "interceptors" that know how to consume the arguments and the context depending on the purpose of the instrumentation.

The no transform Java agent described herein is designed to avoid transforms and retransforms in the agent, thereby eliminating overhead and risk. Modification of a loaded class may be enabled in time to instrument a one-time called method, such as a "run" (Runnable). Furthermore, a transform method is supplied to the agent, but that method is only called after a class has been created and before it has been initialized. This removes two of the issues that create a need to retransform, and also guarantees that the user can instrument methods called immediately after class load and only called one time, such as "run()" async calls.

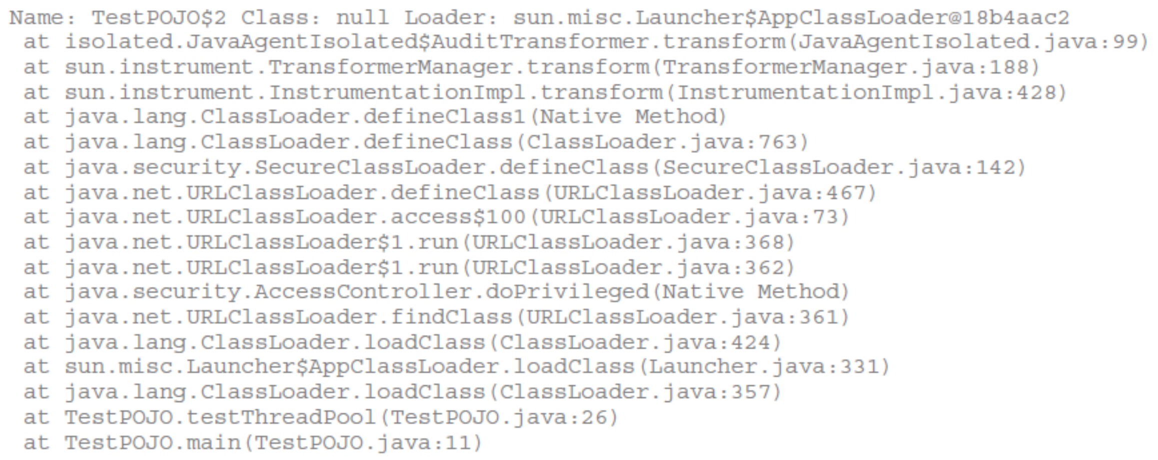
There are three main components of the Java agent described herein, each in separate jar: boot proxy, main, and agent. Boot proxy receives callbacks from java.lang.ClassLoader.defineClass() when a class is defined and not yet initialized (via the clinit() method). "Main" is the premain entry point for the agent. It creates and isolates the agent classloader so there is no conflict with the application. The instrumentation takes place at the agent.

At the end of the premain call, java.lang.ClassLoader defineClass exits are instrumented to call this on every class creation. The return class ($_ is converted at runtime) and all arguments are passed to defineClass (protectionDomain): boot.JavaAgentBoot.callTransformer($_,$args), where $args is the byte array. The user may call getAllLoadedClasses() and instrument those loaded classes (mainly boot classes) before premain using redefineClasses() instead of retransfrom(). The getAllLoadedClasses() method may be similar to the Java class file transformer, which is documented at:
docs.oracle.com/javase/7/docs/api/java/lang/instrument/ClassFileTransformer.html.

Afterwards, the java.lang.Classloader.defineClass may call the boot.JavaAgentBoot.callTransformer, which calls a registered transformer method (e.g., similar to ClassFileTransformer) in the JavaAgentBoot (not in the JVM). The className contains a "tag" that identifies that this was not a typical transform. The tag informs a user of the difference in case it is desired to use this for both a callback and a real transform. The tag may be stripped off the className before processing transform(ClassLoader loader, String className, Class<?> classBeingRedefined, ProtectionDomain protectionDomain, byte[] classfileBuffer).

The implementation of this method may transform the supplied class file and return a new replacement class file. The transform may operate similarly to a typical transform and provide back null or new bytes. The callTransformer method calls redefineClasses if the class bytes were specified to be modified.

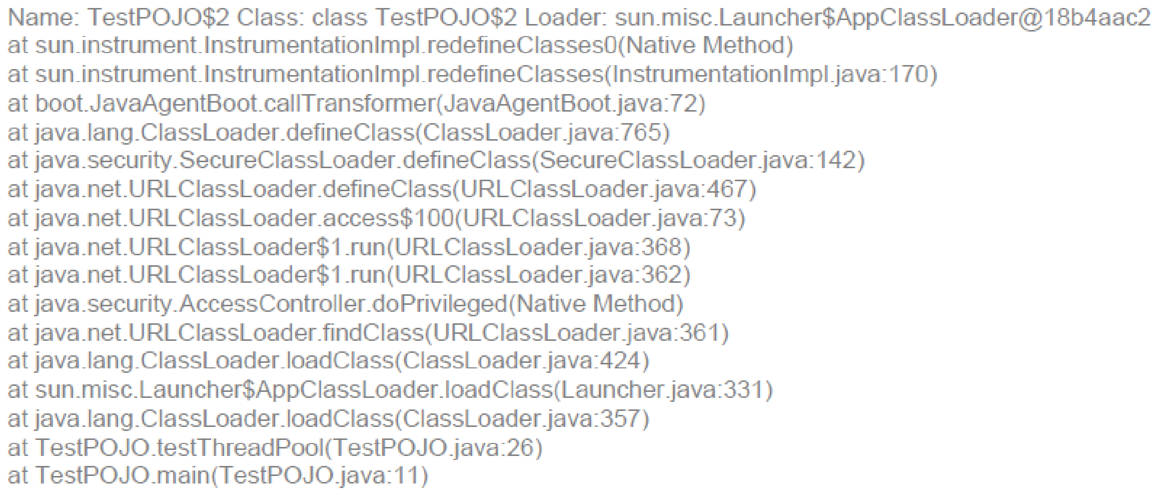
Figure 1 below illustrates an example call stack for a typical transform. In this example, the class is null and the transform was called by the JVM.



```
Name: TestPOJO$2 Class: null Loader: sun.misc.Launcher$AppClassLoader@18b4aac2
 at isolated.JavaAgentIsolated$AuditTransformer.transform(JavaAgentIsolated.java:99)
 at sun.instrument.TransformerManager.transform(TransformerManager.java:188)
 at sun.instrument.InstrumentationImpl.transform(InstrumentationImpl.java:428)
 at java.lang.ClassLoader.defineClass1(Native Method)
 at java.lang.ClassLoader.defineClass(ClassLoader.java:763)
 at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:142)
 at java.net.URLClassLoader.defineClass(URLClassLoader.java:467)
 at java.net.URLClassLoader.access$100(URLClassLoader.java:73)
 at java.net.URLClassLoader$1.run(URLClassLoader.java:368)
 at java.net.URLClassLoader$1.run(URLClassLoader.java:362)
 at java.security.AccessController.doPrivileged(Native Method)
 at java.net.URLClassLoader.findClass(URLClassLoader.java:361)
 at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
 at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:331)
 at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
 at TestPOJO.testThreadPool(TestPOJO.java:26)
 at TestPOJO.main(TestPOJO.java:11)
```

Figure 1

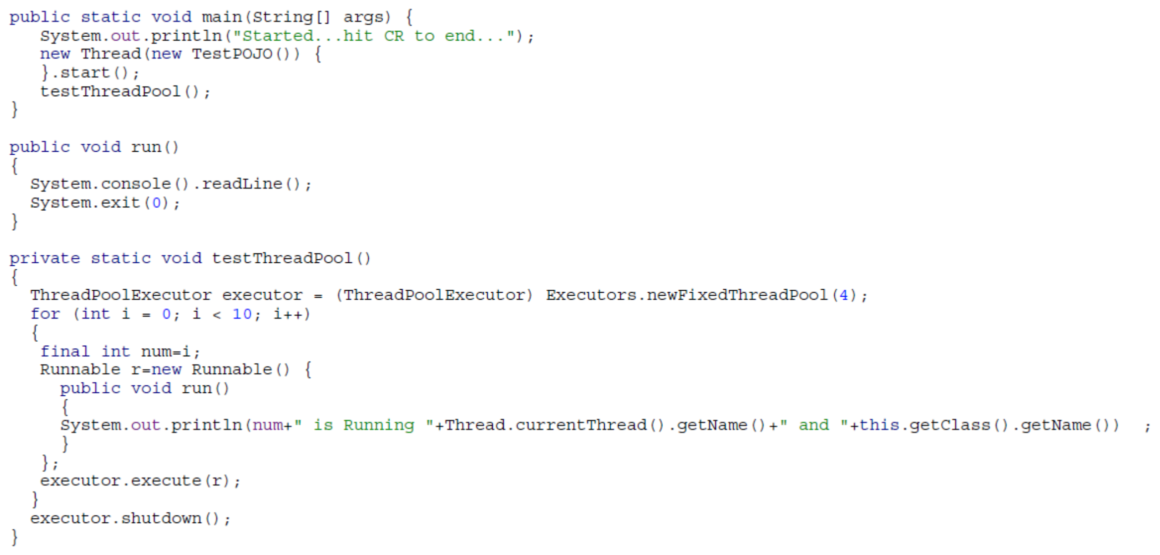
Figure 2 below illustrates an example call stack for the optimized transform described herein. In this example, there is a real class (not yet initiated) that may be timed for instrumentation. Normally redefineClasses would generate a transform at the end, but here there are no transformers, and as such it is not called.



```
Name: TestPOJO$2 Class: class TestPOJO$2 Loader: sun.misc.Launcher$AppClassLoader@18b4aac2
at sun.instrument.InstrumentationImpl.redefineClasses0(Native Method)
at sun.instrument.InstrumentationImpl.redefineClasses(InstrumentationImpl.java:170)
at boot.JavaAgentBoot.callTransformer(JavaAgentBoot.java:72)
at java.lang.ClassLoader.defineClass(ClassLoader.java:765)
at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:142)
at java.net.URLClassLoader.defineClass(URLClassLoader.java:467)
at java.net.URLClassLoader.access$100(URLClassLoader.java:73)
at java.net.URLClassLoader$1.run(URLClassLoader.java:368)
at java.net.URLClassLoader$1.run(URLClassLoader.java:362)
at java.security.AccessController.doPrivileged(Native Method)
at java.net.URLClassLoader.findClass(URLClassLoader.java:361)
at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:331)
at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
at TestPOJO.testThreadPool(TestPOJO.java:26)
at TestPOJO.main(TestPOJO.java:11)
```

Figure 2

Figure 3 below illustrates an example Plain Old Java Object (POJO). Here, the POJO creates one run() to wait for user input to shut down the JVM, and creates ten threads as part of a thread pool. Retransforming this class is not an option since these run() methods are already executed by the time the class loads and a retransform is subsequently issued, and as such the instrumentation would not be invoked.



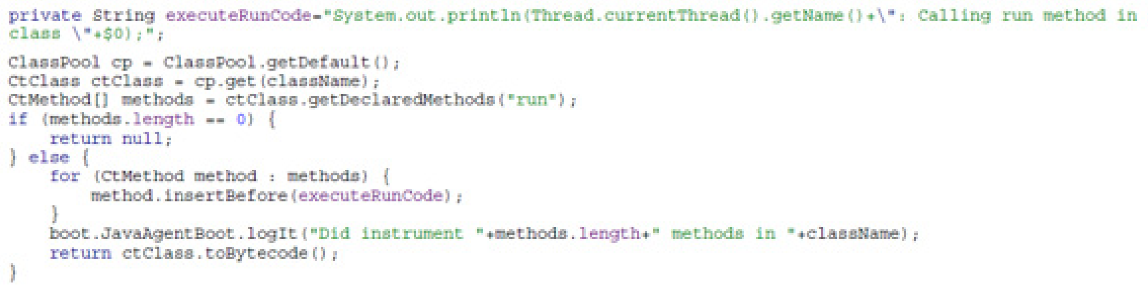
```
public static void main(String[] args) {
   System.out.println("Started...hit CR to end...");
   new Thread(new TestPOJO()) {
   }.start();
   testThreadPool();
}

public void run()
{
  System.console().readLine();
  System.exit(0);
}

private static void testThreadPool()
{
  ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newFixedThreadPool(4);
  for (int i = 0; i < 10; i++)
  {
   final int num=i;
   Runnable r=new Runnable() {
     public void run()
     {
     System.out.println(num+" is Running "+Thread.currentThread().getName()+" and "+this.getClass().getName())  ;
     }
   };
   executor.execute(r);
  }
  executor.shutdown();
}
```

Figure 3

Figure 4 below illustrates an example instrumentation for the run() classes.

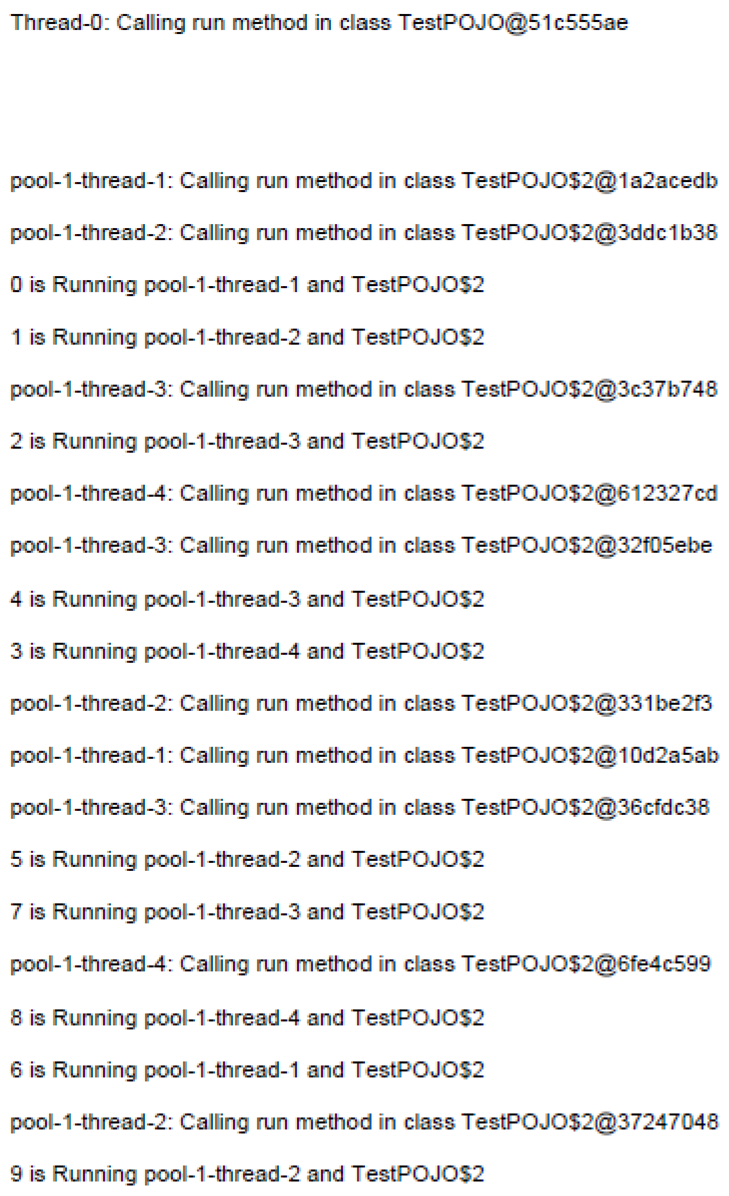


```
private String executeRunCode="System.out.println(Thread.currentThread().getName()+\": Calling run method in
class \"+$0);";

ClassPool cp = ClassPool.getDefault();
CtClass ctClass = cp.get(className);
CtMethod[] methods = ctClass.getDeclaredMethods("run");
if (methods.length == 0) {
    return null;
} else {
    for (CtMethod method : methods) {
        method.insertBefore(executeRunCode);
    }
    boot.JavaAgentBoot.logIt("Did instrument "+methods.length+" methods in "+className);
    return ctClass.toBytecode();
}
```

Figure 4

Figure 5 below illustrates example results.



```
Thread-0: Calling run method in class TestPOJO@51c555ae



pool-1-thread-1: Calling run method in class TestPOJO$2@1a2acedb

pool-1-thread-2: Calling run method in class TestPOJO$2@3ddc1b38

0 is Running pool-1-thread-1 and TestPOJO$2

1 is Running pool-1-thread-2 and TestPOJO$2

pool-1-thread-3: Calling run method in class TestPOJO$2@3c37b748

2 is Running pool-1-thread-3 and TestPOJO$2

pool-1-thread-4: Calling run method in class TestPOJO$2@612327cd

pool-1-thread-3: Calling run method in class TestPOJO$2@32f05ebe

4 is Running pool-1-thread-3 and TestPOJO$2

3 is Running pool-1-thread-4 and TestPOJO$2

pool-1-thread-2: Calling run method in class TestPOJO$2@331be2f3

pool-1-thread-1: Calling run method in class TestPOJO$2@10d2a5ab

pool-1-thread-3: Calling run method in class TestPOJO$2@36cfdc38

5 is Running pool-1-thread-2 and TestPOJO$2

7 is Running pool-1-thread-3 and TestPOJO$2

pool-1-thread-4: Calling run method in class TestPOJO$2@6fe4c599

8 is Running pool-1-thread-4 and TestPOJO$2

6 is Running pool-1-thread-1 and TestPOJO$2

pool-1-thread-2: Calling run method in class TestPOJO$2@37247048

9 is Running pool-1-thread-2 and TestPOJO$2
```

Figure 5

In summary, techniques are described herein for reducing overhead, risk, and code maintenance for Java instrumentation required to monitor customer-facing applications. These techniques avoid transforms and retransforms, thereby eliminating risk and overhead as there are no transformers registered with the JVM. This may use the same call mechanism as a typical transform, and therefore requires no code changes in instrumentation. The "dummy" transform may always obtain a real class (not a null) that has not been initialized. Thus, a user may add a static block or other one-time-only objects.

These techniques are also portable, because the only hook is in the core JVM classloader, and compact, because very little code is required for the agent itself.